Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

Technische Universität Wien

M A G I S T E R A R B E I T

Generating Structured Documents to Create Reports by Integrating Data from CMS/DMS and EAI Systems

Ausgeführt am Institut für

Softwaretechnik und Interaktive Systeme

der Technischen Universität Wien

unter Anleitung von Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
und Univ.Ass. Dipl.-Ing. Dr.techn. Alexander Schatten als verantwortlich
mitwirkendem Universitätsassistenten

durch

Peter Gerstbach, Bakk. rer. soc. oec.

http://www.gerstbach.at/

3. Mai 2006

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 3. Mai 2006 Peter Gerstbach

Acknowledgments

I would like to express my gratitude to the following people for their support and assistance in writing this thesis:

- my parents for a lifetime of love, support, encouragement, guidance and inspiration.
- my darling Ingrid, for being patient, providing constant support, and for proofreading my thesis.

Abstract

Structured documents are omnipresent in businesses as they contain vital information and may be created and processed automated. Apart from text parts and graphics they contain metadata commonly displayed using a characteristic layout. Some examples of structured documents include offers, contracts, business reports, letters, and catalogs. For creating such documents a system which automates this process is very helpful because uniform documents are filled with different data. The processing logic and the layout is defined only once at the beginning. Afterwards an automated process retrieves the data from external system and creates the documents according to the defined rules.

In this thesis existing software applications and technologies are analyzed that are able to create structured documents. Apart from business reporting systems, commonly used within databases, the main focus lies on XML-related systems, for example stylesheet designer and solutions for application integration (EAI). The use of XML enables new ways to create structured documents. Therefore document processing with XML is discussed separately and technologies such as XSLT and XSL-FO are described.

Furthermore the development of such a system is described supporting the integration of external data from CMS, DMS and EAI systems in addition to document creation. These data can be converted automated and repeatedly to appealing documents using templates. The deployment of standardized technologies like XSLT and XSL-FO enables a flexible and open transformation approach. Due to a graphical user interface the creation of such templates is simple and can be performed without specific knowledge about XML and XSLT.

Kurzfassung

Strukturierte Dokumente sind im Geschäftsleben allgegenwärtig, da sie wichtige Informationen enthalten und automatisiert erzeugt und weiter verarbeitet werden können. Neben den einzelnen Text- oder Grafik-Bestandteilen enthalten sie Metainformationen, die meist durch ein bestimmtes Layout dargestellt werden. Beispiele strukturierter Dokumente sind: Angebote, Verträge, Geschäftsberichte, Briefe oder Kataloge. Für solche Dokumente ist ein System, das deren automatisierte Erzeugung unterstützt, besonders hilfreich, weil viele gleichartige Dokumente mit unterschiedlichen Daten gefüllt werden. Die Verarbeitungslogik und das Layout wird einmal zu Beginn festgelegt, während ein automatisierter Prozess in Folge die benötigten Daten aus externen Systemen lädt und die Dokumente entsprechend den Regeln generiert.

Diese Diplomarbeit analysiert Softwaresysteme und Techniken, die strukturierte Dokumente erstellen und verarbeiten können. Neben Business Reporting Systeme, die meistens in Verbund mit Datenbanken angewendet werden, liegt der Schwerpunkt auf XML-bezogene Systeme wie beispielsweise Stylesheet-Designer oder Lösungen zur Anwendungsintegration (EAI). Die Etablierung von XML hat neue Möglichkeiten geschaffen, um strukturierte Dokumente zu erstellen. Deswegen wird das Thema der Dokumentverarbeitung mit XML gesondert behandelt und Techniken wie XSLT und XSL-FO beschrieben.

Weiters wird die Entwicklung eines XML-basierten Dokumentmanagement-Systems beschrieben, das neben der Dokumenterstellung auch die Einbindung von externen Daten aus CMS, DMS und EAI Systemen ermöglicht. Durch die Anfertigung von Dokumentvorlagen können diese Daten automatisiert und wiederholt zu ansprechenden Schriftstücken weiterverarbeitet werden. Der Einsatz von standardisierten Techniken wie XSLT und XSL-FO ermöglicht einen flexiblen und offenen Ansatz, um die Transformation durchzuführen. Da eine grafische Oberfläche die einfache Erstellung dieser Vorlagen ermöglicht, können jedoch auch Anwender ohne Kenntnisse von XML und XSLT das Programm benutzen.

Contents

1 Introduction to Enterprise Reporting and Document Processing

In the mid 20th century many countries passed from the industrial to the information era. In these years, the number of information workers (employees primarily handling information) surpassed the number of industrial workers [MS01, p. 4]. However, apart from simple telephones, *information technology* hardly existed.

Nowadays information is largely available electronically. Electronic data processing has accelerated and improved business, although not all promises told about the advantages of IT have become true. The integration of different data sources is still a challenge also there is an enduring need of making documents available on paper.

Reasons to print out documents or to create electronic documents with appealing layout are diverse. Still not all business transactions can be performed electronically due to legal restrictions. Contracts, invoices, reports are printed on paper although their content creation process is more and more automated. Aside from legal and technical argues, there are important social reasons to keep up with printed documents or at least nice looking electronic ones. E.g., people often prefer to own a physical item in place of a virtual one and like elaborate documents more than meaningful but unemotional ones. Furthermore electronic documents offer the possibility to provide always up-to-date versions which is especially important for reporting. Other features are imaginable for security-relevant areas, where the filtering of information or the use of cryptography increases privacy.

At present the IT industry offers different systems for storing and processing data and documents. While EAI (Enterprise Application Integration) systems are used to bring a set of enterprise applications and their data together, documents are mostly stored and retrieved using Document Management Systems (DMS) and Content Management Systems (CMS). DMS evolved from systems used to store images of paper documents, while CMS is a newly-made term, frequently used to define web applications used for managing websites and web content. Furthermore, the creation of documents is mostly done with another group of tools, ranging from word processing and office suites to desktop-publishing systems.

Business Cases In such a heterogeneous systems landscape many business cases tend to be inefficient. They are time expensive, error-prone and difficult to monitor. The following list shows some business cases where an integrated and automated approach may increase efficiency:

- **Capital Management - Stock and Funds Reports**: the process is triggered manually or time-based and causes information to be retrieved from financial systems (up-to date funds and stock data) and CRM systems (names, addresses). Later the document becomes generated and data from previous steps inserted. In accordance to the defined workflow the documents may then be reviewed (quality assurance), archived and delivered to the addressee.
- **Logistics - Contract Order Creation in Logistics**: this business case is initiated by the call center system or the CRM that take an order. Information is retrieved from cargo and ERP systems managing the disposition of vehicles, people and performing up-to-date calculations. After this step the document is created and processed according to the workflow.
- **Sales Offer**: the document author uses the editor to create a new sales offer with data input possibility. Parts of the document's content may be linked with CMS systems, such as terms and conditions. Up-to-date calculations are performed by ERP systems and customer data is retrieved using a CRM system. Multi-channel delivery may be used to create output in various formats.

The Missing Link In the early days of the information era, systems acted as "paperwork factories" to get employees paid, customers billed, products shipped, and son on. Later the focus shifted to producing reports for the management [MS01, p. 15]. To provide the information for knowledge-enriched reports a system like this needs to *integrate* different data sources (for example using an EAI system) and has to have the ability to *manage* documents. The implementation of such a system was the intend for starting this thesis.

Structured Documents with InStruct In 2004 the GFT Technologies' office in Vienna launched the *InStruct* project. InStruct is a document management system that facilitates the creation and management of structured documents. Beyond standard DMS-functionality such as versioning, search and retrieval and user management it combines DMS with EAI systems, provides means to support workflows and offers a visual editor to create structured documents enriched with data and formatting.

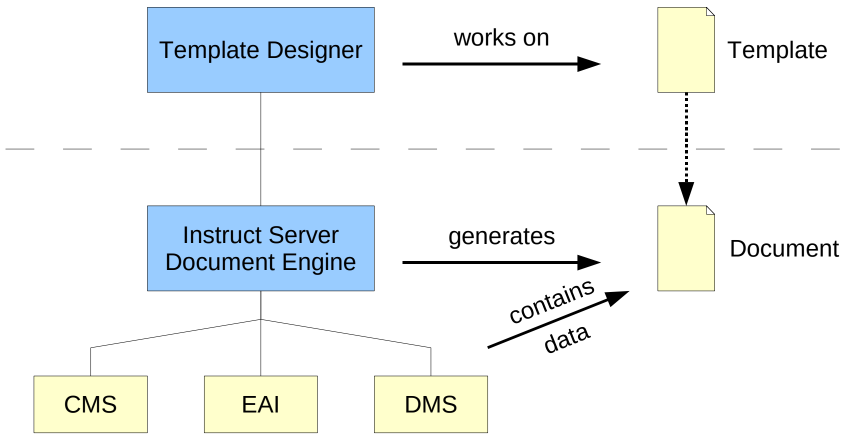


Figure 1.1: Instruct Overview

InStruct is served in two parts: the Template Designer and the Document Engine (see figure 1.1). The designer facilitates the creation of structured document templates by using a WYSIWYG like user interface. It allows to insert XML data from any external systems and supports some other standards related to CMS and DMS (e.g. WebDAV). The result of the designer is a *template* that may be rendered to a final *document* using real or test data. The *Document Engine* is used to mass-produce documents by using templates from the designer as input and including the necessary data from external systems. A SOAP interface allows to integrate the engine into a workflow management system.

Both, the designer and the engine use a generator component to create the output. This component, classified as *XML document processing*, uses XSLT and XSL-FO to produce PDF and other output formats.

The Structure of this Thesis This thesis covers aspects ranging from XML document processing to the implementation of InStruct's document generator components. The following chapter 2 (Reporting Business Data, page 5) provides an overview of related work including stylesheet generators and reporting tools. Knowing the potentials and limitations of these tools is crucial to understand the need for an enhanced and integrated document management system, as InStruct is aimed to be.

Today, XML is the tool of choice for many data-centric applications as well as for applications processing documents. On account of this, chapter 3 (Document Processing with XML, page 14) introduces into the concepts of XML document processing. Current and historic tools are discussed and usage examples are given. The main part of this chapter deals with the two technologies that are primarily used for InStruct's generator component, namely XSLT and XSL-FO. In document processing XML can be used as the connector between the word of data and the world of documents.

Chapter 4 (Implementing an XML-based Document Management System, page 55) describes the implementation of the InStruct software system. The idea of this part is to give a view on how XSLT and XSL-FO can be used in a real-life system, how the requirements influence the decision making and what kinds of problems have to be solved in practice. Chapter 5 (Conclusion, page 88) sums up the thesis.

2 Reporting Business Data

Together with the increased awareness of the problems associated with electronic documents, initiatives have been launched to tackle these issues. This chapters compares some existing products that have been evaluated at different stages of the InStruct project. The first section (2.1) describes some stylesheet designers for transforming XML content into arbitrary formats including HTML and PDF. These visual tools create XSLT stylesheets that can then be used to mass produce documents from any XML data.

The second section (2.2) is about enterprise reporting tools that emerged when databases have been started to be used as primary store for information. Currently these tools often lack functions to process XML data in an adequate way but on the contrary they are proved and tested. It is likely that these two groups of products (XML and database related) will merge together in future. Of course, the need to integrate different data sources and application is well known to the industry. This leads to EAI systems that are addressed in section 2.3. Such solutions mostly provide reporting tools as well as tools for handling XML and XSLT transformations.

2.1 Stylesheet Designer

This section describes some stylesheet designers that produce XSLT stylesheets to format XML content using XSL-FO. The underlying technologies for this transformation process are described in section 3.1.

2.1.1 Altova XML Suite

Altova is one of the leading providers of XML editors. The company started with XMLSpy, an XML developing environment that facilitates editing and working with XML. It provides different visual views on plain XML, XML Schema and other formats. A built-in XSLT processor can be used to test and debug newly created XSLT stylesheets. Beside XMLSpy, StyleVision and MapForce are two other XML-related products of Altova. StyleVision is an XSL stylesheet designer, while MapForce is a data-centric XML mapping tool that may be as well used to access databases, flat files or EDI data. It generates conversion code in XSLT, XQuery, Java, C++ or C#. This section focuses on StyleVision which overlaps with many topics in this thesis.

StyleVision offers two different windows for editing XSLT stylesheets. On the one hand a XML text editor exists that provides syntax highlighting and helpers for creating elements from the XSLT and XSL-FO namespace; on the other hand there is a design view that allows to design a stylesheet using drag and drop functions that create transformation code for HTML, Word/RTF and XSL-FO using a unified approach. The three different transformations can be run and previewed from within the program.

After a new document is created, StyleVision asks for the structure of the source file. This can be provided by using a sample XML, a DTD or an XML Schema. The structure of these files are then displayed on the left-hand-side of the window and can then be dragged from there to the editing view. Droping an item on the editing view forces one ore more `<xsl:for-each>` elements to be created on each of the three XSLT files for HTML, RTF and XSL-FO. The `match` attribute contains the name of the dropped element. The for-each instruction ensures that all elements with the same name are processed, whatever cardinality is specified in the describing grammar. The graphical representation of a for-each element is a tag-like symbol, shown in figure 2.1. If an object on the designer is selected, its properties (which correspond to FO properties) can be adjusted using the view on the bottom left.

The use of for-each constructs results in an XSLT file containing one big root template with all transformation code in it. The XSLT-aware power user can use special commands to convert a selected for-each element into an own template element, which may be re-used at different places in the stylesheet. However the designer view does not allow to edit the used XPath expressions by hand. The automatic creation of these XPath expressions sometimes failes. This results in a bad expressions which hast to be corrected by hand. Otherwise StyleVision recognizes a change of the underlying XML Schema after the time of editing. In this case it shows a pop-up window and marks all usages of non existent Elements.

To sum up, StyleVision offers a good designer for XSLT, although an XSLT expert is still likely to use a more or less simple XML editor.

2.1.2 XSLFast

XSLFast is a graphical editor for XSL-FO documents, developed by jCatalog Software AG. It uses a WYSIWYG like approach that facilitates the creation of XSL-FO documents in many aspects. XSLFast allows the integration of different XSL formatters, including Apache's FOP and RenderX's XEP. Similiar to Altova's product, XSLFast offers a structure view from which items can be dragged to the current working sheet. XSLFast supports both absolute and relative (floating) positioning, focing the absolute

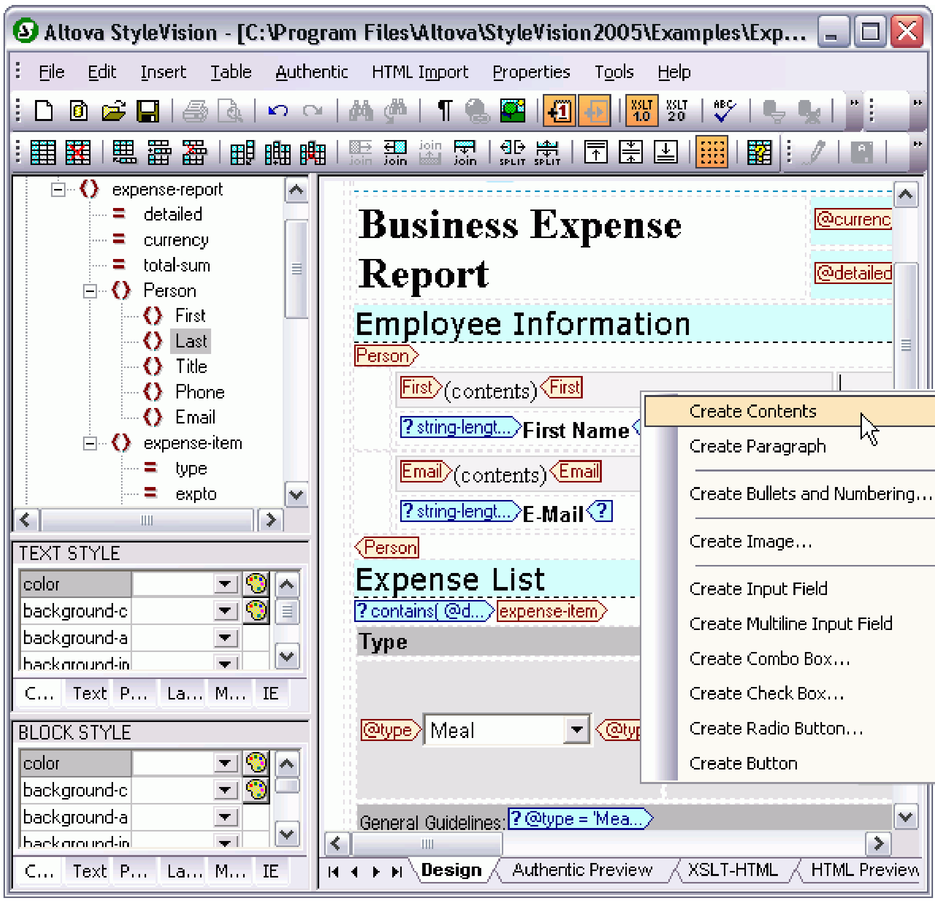


Figure 2.1: Screenshot of Altova's StyleVision
Source: [Alt]

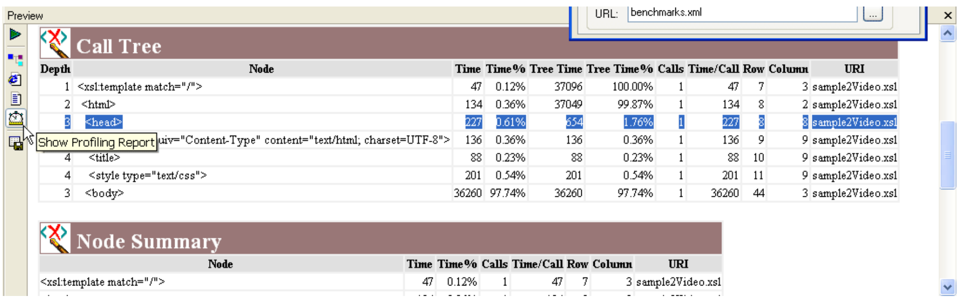
positioning. It enables the user to define pixel-perfect layouts. In contrast the program seems to have problems with floating blocks that are not placed in the resulting document as expected.

Although XSLFast focuses on XSL-FO, which perforce difficult handling, many tasks can be fulfilled quickly. Although some bugs and inconsistencies with the FO specifications lead to problems with the integration into other products.

2.1.3 Stylus Studio

Stylus Studio is an integrated development environment for XML. It is shipped with XML editors, XSL, XSLT and XQuery tools, a Schema editor and other tools.

The XSLT Editor is a text editor for XSLT files that facilitates programming XSLT code. It comes with some outstanding features that are explained here in short. The *refactoring* capability allows to extract named templates. This facilitates the creation of modular, re-usable stylesheet code. *Backmapping* is an interesting feature that eases troubleshotting: a click in the transformed document opens the part of the stylesheet that created the selected node. A *profiler* can be used to trace every detail of an XSLT transformation. It can be used to identify bottlenecks and analyze performance. In conjunction with the backmapping feature, good candidates for optimizations can easily be found with the profiler. Figure 2.2 shows a screenshot of the profiler window.



Call Tree

Depth	Node	Time	Time%	Tree Time	Tree Time%	Calls	Time/Call	Row	Column	URI
1	<xsl:template match="/">	47	0.12%	37096	100.00%	1	47	7	3	sample2Video.xsl
2	<html>	134	0.36%	37049	99.87%	1	134	8	2	sample2Video.xsl
3	<head>	227	0.61%	654	1.76%	1	227	8	8	sample2Video.xsl
	uiv="Content-Type" content="text/html; charset=UTF-8">	136	0.36%	136	0.36%	1	136	9	9	sample2Video.xsl
4	<title>	88	0.23%	88	0.23%	1	88	10	9	sample2Video.xsl
4	<style type="text/css">	201	0.54%	201	0.54%	1	201	11	9	sample2Video.xsl
3	<body>	36260	97.74%	36260	97.74%	1	36260	44	3	sample2Video.xsl

Node Summary

Node	Time	Time%	Calls	Time/Call	Row	Column	URI
<xsl:template match="/">	47	0.12%	1	47	7	3	sample2Video.xsl

Figure 2.2: Screenshot of Stylus Studio's Profiler
Source: [Pro]

The XSLT Designer is a WYSIWYG designer to create XSLT stylesheets. The user interface is comparable to Altova's StyleVision. The XML source tree window provides a visual representation of the input document being used. Its items are annotated with symbols and icons, while double clicking or drag and drop create new templates for the selected XML element. Backmapping and the profiler is available from the Designer view as well as from the XSLT editor view.

The combination of different editors using one single interface makes Stylus Studio a comprehensive XML IDE which is a good alternative to the market-leading Altova XML Suite.

2.1.4 VXT

VXT is a visual language for the specification of XML document transformations [CC02]. It so differs from the products in former sections, although VXT includes as well an interactive environment to create and edit transformations.

The approach introduces a set of visual elements that represent the structure of the source, the target and the transformations steps. The authors point out that all mature languages used to transform XML documents such as XSLT or other script languages make use of a textual syntax. Textual languages mostly use a two-dimensional syntax, but only the first dimension conveys semantics. The second dimension is limited to increase readability through indentation and new lines. In contrast, a graphical representation takes full advantage of both dimension resulting in a representation of structures that is often easier to grasp.

The visual language can be exported to two target languages: Circus [VDLP02] and XSLT (see as well section 3.1.5). The structure of the documents can be illustrated with a node-link and a tree-map representation that are more expressive than simple trees. The nodes in both representations convey information like node type, cardinality and the association to the parent node and the children.

The used transformation model is close to one found in XSLT: a transformation rule consists of a pattern and a template. If the pattern matches an element in the source XML, the template is created in the result document. Patterns that differ from patterns in XPath are indicated using transparency, different borders and colors. Transformation rules, like *deep copy* or *apply rules* are made visible using different icons and arrows.

Beside the visual language aspect, VXT is also an interactive environment in which transformations can be run and debugged.

2.2 Enterprise Reporting

The ever-increasing demand for high quality information drives the development of *reporting systems* that produce unified reports to gain different views on distributed data. This process, in business also termed *enterprise reporting*, includes querying different data sources and transforming the results to create human-readable reports.

Usually any structured data may be used as data sources. The most obvious and supported data sources are relational databases, XML data and those coming from enterprise information systems and data warehouses. The transformation step may include selecting only certain parts of the retrieved data, joining together multiple sources, translating coded data, deriving new calculated values, or grouping and summarizing parts of data. The created reports may be delivered using browsers, e-mail, file system storage or even fax and printers. Supported formats vary from PDF or Postscript to HTML, XML or application specific formats like Excel. Figure 2.3 shows a possible architecture of such a system.

Responding to the high demand for reporting tools the market offers products ranging from standalone systems to full fledged enterprise solutions. One of the most known

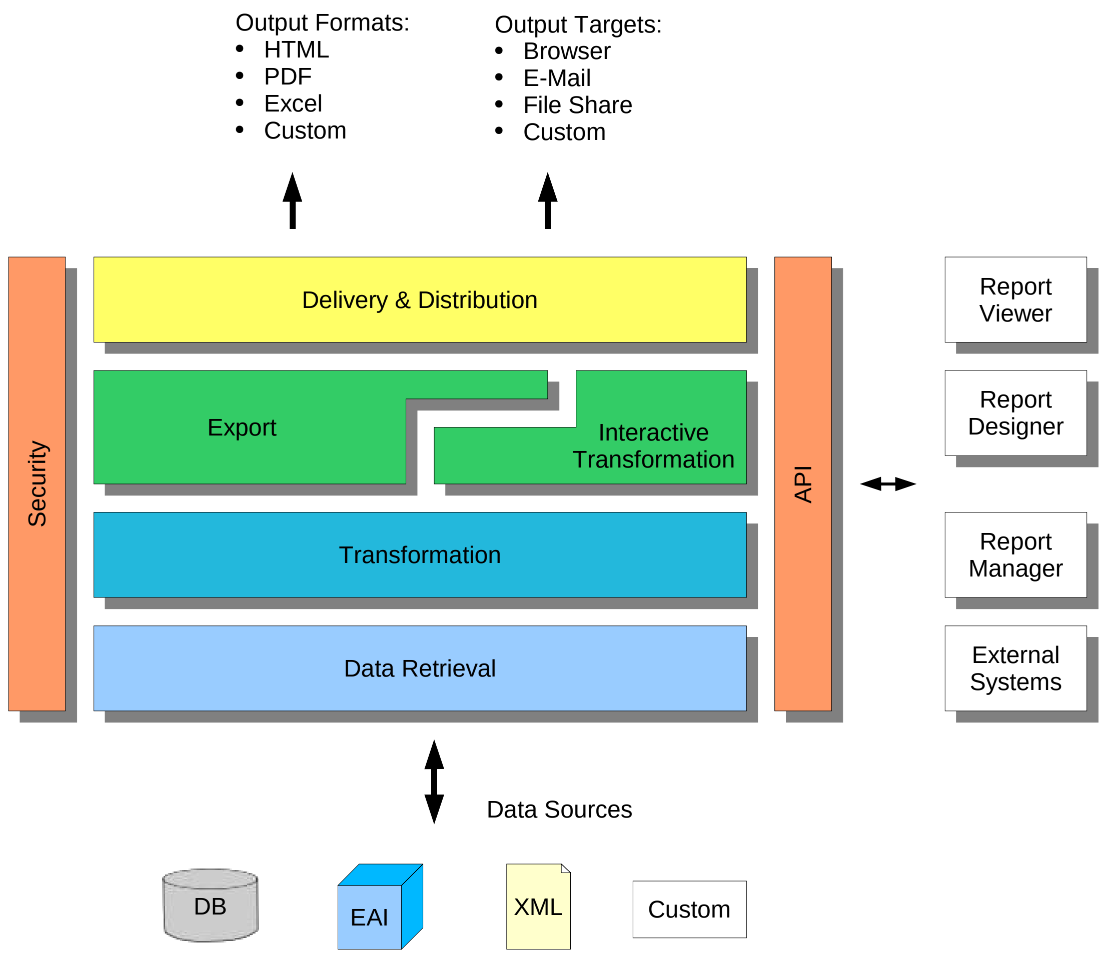


Figure 2.3: Reporting Architecture

reporting tools is *Crystal reports*[1] others are *Oracle Reports*[2], *Crystal Clear*[3], *JReports*[4], or the open source product *JasperReports*[5]. Beside these all-in-one solutions, others also allow the integration of the report engine in external applications using APIs. This enables the developing of reporting solutions that are independent of tool vendors. In [Veg01] a system like this is described. It uses JSP pages on top of a Tomcat server offering web-based access to corporate reports in an easy and expandable way.

[1]Crystal Reports: `http://www.businessobjects.com/products/reporting/crystalreports/`, see as well section 2.2.1

[2]Oracle Reports: `http://www.oracle.com/technology/products/reports/`

[3]Crystal Clear: `http://www.inetsoftware.de/products/crystalclear/`

[4]JReports: `http://www.jinfonet.com/`

[5]JasperReports: `http://jasperreports.sourceforge.net/`, see as well section 2.2.2

The following sections gives an overview of two reporting tools, namely *Crystal Reports* and *JasperReports*.

2.2.1 Crystal Reports

Crystal Reports is a Business Intelligence application created by Crystal Decisions (now owned by Business Objects) used to gather data and design reports. Supported data sources are native, ODBC, OLE DB, and JDBC connectivity to relational, OLAP, XML, legacy and Enterprise data sources.

The product includes a visual design environment that allows to format and design reports using a drag-and-drop interface and object-oriented explorers. Wizards and Experts simplify common used reporting tasks like connecting to data sources or selecting, grouping and sorting data. Due to the important requirement for creating consistent reports, customized templates can be designed and applied. Those templates specify standards in formatting and logic to ensure this consistency. Often used report elements and functions can be stored in a central repository for sharing, reuse, and single-point updating across multiple reports.

Reporting viewing and interaction is possible in different ways. Mostly static export formats include PDF, XML, HTML, RTF and Excel. The web publishing feature of the server edition allows interactive reports to be uploaded on the web with enabled security and scheduling features. With the help of interactive reports (client-side DHTML, ActiveX or Java components) data may be explored using drill down, scroll, export and print functions. On the server-side Crystal Reports allows to integrated the created reports into existing systems, applications and infrastructure. To accomplish this development kits are available for J2EE, .NET and COM.

2.2.2 Jasper Reports

JasperReports is a popular open source Java reporting tool using XML templates to generate reports that can be delivered onto the screen, to the printer or into PDF, HTML, Excel, CSV and XML files. It is entirely written in Java and can be used in a variety of Java enabled applications, including J2EE or Web applications, to generate dynamic content. Its main purpose is creating page oriented, ready to print documents in a simple and flexible way [Jas].

Although the copyrights of the code is owned by a company, JasperReports is still licensed under the LGPL (Lesser General Public License). This makes JasperReports an interesting candidate for integration into other applications. For example IBM covers in an article the generation of online reports using JasperReports and WebSphere Studio [Oli04]. The article describes a sample Web application using the reporting tool to create dynamic reports in HTML and PDF. The application is deployed to a WebSphere Application Server and uses DB2 UDB as an enterprise data repository.

2.3 Enterprise Application Integration

Enterprise application integration (EAI) solutions bring together a set of enterprise computer applications. An important part of EAI is data integration. With adapter help an integrative view on different interfaces and data sources is offered. Many EAI solutions support XSLT to manage the growing amount of XML data. The following section describes the *Inubit Business Integration Server*, an EAI related software application. Other products of this category include IBM's *WebSphere*, Microsoft's *Biztalk* or *Webmethods*.

2.3.1 Inubit Business Integration Server

The Business Integration Server from Inubit (Inubit IS)[6], a German supplier of standard software, offers a platform for business process integration. It allows the exchange of business data using the most diverse transfer protocols and data formats. It integrates business applications and brings them together with the IT application of other business partners. Inubit IS provides tools for Business Process Modeling (BPM), system integration (EAI), workflow management (WFM), reporting, and corporate performance management (CPM).

Inubit IS follows an integral XML-based approach and thus responds to the growing importance of XML in the world of business documents. Data loaded into the server are first transferred into an XML format before it is further processed. Contrariwise, if required, the XML data stream is recoded into the required target format (e.g. EDIFACT). This approaches permit the use of XSLT and other XML related technologies.

Business processes are described using UML-compliant activity diagrams. The technical realization is accomplished using so-called *technical workflows* that automate business processes on the basis of XML. A technical workflow consists of defined modules, connected to each other, which represent the data flow and processing logic. One module is the *XSLT Converter* that falls into the category of *Data Converters*. Other data converters access relational databases, mainframes, ERP systems, and Web Services. The

[6]Inubit: http://www.inubit.com/

XSLT Converter performs an XSLT transformation, while the applied stylesheet may be edited directly in the application.

In the stylesheet editor XML sample files or XSD Schema files may be used to define the source and target format for the transformation. The stylesheet can be edited using a simple text editor, an external stylesheet editor or with a graphical tool. The tool allows using drag and drop to assign source elements to the desired target elements. However, the view is centric to the XSLT syntax and therefore does not support a non XSLT-aware user.

Beside the integration aspect the Inubit IS also offers a report generator for graphical visualization used to create reports for *business activity monitoring* (BAM) or external systems, e.g. for invoice processing.

2.3.2 Output Management

Output management solutions, also referred to as *document automation* solutions are also known as low-cost EAI systems. In the very beginning such tools were used to deliver customized report printouts across an enterprise. Today they rather handle multichannel delivery of information to users and applications [Rap01]. They supply a single point of control for enterprise documents ranging from creation to final disposition.

Output Management solutions uses one centralized document repository that eases document related tasks. They usually provide strong content extraction and formatting capabilities and modules to connect to other systems for application integration. Common supported destinations vary from print and fax to email and web. Solutions typically include functions for routing and tracking the output job, transformations, reporting of job status events, and redelivery to alternate devices. The benefits from using document automation are a reduction of paper and postal charges, increase in quality of service, and the transition to more flexible business processes. Today output management solutions combine the print related processes with online processes and offer access through web clients. The shift from print to online-services lead to a paperless office in future.

Systems from this category that have been evaluated at GFT are Xultation Suite[7], Scriptura XBOS[8], ISIS Papyrus[9] and Invaris[10].

[7]Metafocus: `http://www.metafocus.no/en/`
[8]Scriptura Inventive Designers: `http://www.inventivedesigners.com/scriptura/`
[9]ISIS Papyros: `http://www.isis-papyrus.com/`
[10]Invaris Solutions: `http://www.invaris.com/`

3 Document Processing with XML

A document is described as a unit of *recorded information structured for human consumption* [Lev91]. The world of information is growing and documents are a major part of it. Document processing means using new technologies to create and edit/modify documents. It functions include *the capability to create, edit, merge, and format documents* [Ope98]. With the help of a software system considerable time saving economies can be achieved. On the one hand data must not be re-entered anymore resulting in an improvement of speed and a decrease of errors. On the other hand a software may assist the author with functions like layout and spell checking.

Currently, there are various different document processing systems existing. Most companies use office suites to facilitate the work with simple stand-alone documents. When demands on layout increase, desktop publishing tools are offering good solutions. They increase the capabilities but introduce concepts that many users are not familiar with. In this case the creation of documents is often outsourced to professionals. In case of documents, that make heavy use of cross-references and need access to external data, the IT industry offers many powerful but expensive document processing systems which need to be integrated in the company's workflow. Another feature of these systems is the efficient organization of documents which becomes more and more important for large companies.

Recent developments have shown that the technology for document processing is widening. Until recently, the technology has been limited mostly to generate, print, and transport text documents preciser and faster. Now we can see a shift to better computer-based information management that includes document processing. These advances include digital image processing, hypertext, multimedia documents and improved concepts for retrieving information and data [MS01, p. 421].

Since 1998, when the final *XML* recommendation has been published, *document processing* is tightly related to this technology. XML, the general-purpose markup language, fits perfectly for to overcome many problems associated with documents and its managing.

Although computers have been used for many years for document processing, many problems regarding the format chaos and *interoperability* are still existing. XML allows to create special-purpose markup languages that may be targeted at a specific domain but still remaining processable by standard XML tools. Examples of such document markup languages are given in section 3.2 including *DocBook*, the *Open Document Format* and others. The transformation of XML documents may be done with XSLT combined with XSL-FO; these tools and others are discussed in section 3.1.

One important aspect of many existing markup languages is their heavy use of *generic markup*, in contrary to *visual markup*. Generic markup uses *metadata* ("data about data") to describe the semantics of the content which reveals more information about it. By using elements describing structure, it is possible to separate content from style. This change in attitude can be seen on the evolution of HTML. Previous versions of the hypertext markup language contained many elements referring to layout concepts (e.g. the `<font>` element) while the upcoming XHTML 2 will introduce new techniques for structuring content (e.g. the `<section>` element) [Bra05]. Another example is the replacement of the element `<b>` (standing for bold) in favor of the element `<strong>` used to mark a portion of text as important.

Nowadays many word processors use *hard-formatting* for annotating content with style. Changing the style of such a document means spending much time on a job which could be done in minutes if the the style was separated from content. This separation of concerns often implies to move away from WYSIWYG tools and to force the user to think in terms of semantics than in terms of formattings.

Generic markup has two major advantages: it allows intelligent search algorithms to query data in respect to metadata and it causes complete independence of the final electronic medium. For example, if a document must be made available on print and on the web, it is important that a simple transformation from the source to different target formats is possible. XML is cost beneficial for cases where more than one media is involved (statistically for greater than 1.6 media) [Paw02]. If the same document is processed many times with only small changes or if the document is part of a periodic publication, the look and feel becomes recognizable and thus important. This fact is often utilized by companies to promote their corporate identity. Given the example of a newspaper or a magazine a change in style always attracts the attention of the reader. Therefore supporting a constant layout and style is fundamental to document processing.

The use of XML may also decrease transformation costs at the initial stage of the publishing process. If the source material is available electronically but in different formats (e.g. databases, proprietary formats), the use of XML document processing may cause advantages. In this case no human effort is needed to transform the source to the target format, where it can be used as input for the next processing stage. Summing up, XML document processing is especially suitable if different medias are involved and repeatability is an issue.

3.1 Document Processing Technologies

This part introduces some technologies commonly used for document processing with XML. The first subsection introduces low-level approaches for XML processing. The second subsection covers *DSSSL*, a language for specifying stylesheet for SGML documents and CSS, a massively used language in web publishing. DSSSL can be seen as the predecessor of the nowadays most often used technologies: XSLT and XSL-FO. Their importance is the reason why most of this section is dedicated to these two specifications. The final subsection after the XSL-Family deals with streaming transformations, an alternative to XSLT for large data applications.

Beside the technologies quoted above there are as well other tools like TeX and LaTeX that can perform similar tasks but are not covered here. These mature tools have a big community and are capable of producing high quality documents. There are also some well known desktop publishing packages like Adobe InDesign and Frame Maker. However, these tools are lacking standardization and often do not support XML very well.

3.1.1 Low-Level

When writing about XML processing, the two widely used APIs, SAX and DOM need to be mentioned. They are low-level APIs which are easy to understand, providing direct access to XML data. The *Simple API for XML* (*SAX*) is a widely adopted API for XML, and is a "de facto" standard, implemented in many programming languages. SAX allows to implement an event-based transformation process that allocates less system resources. It is used for serial processing, mostly reading. When using SAX to implement a parser, the programmer has to provide callback-methods which are invoked by the parser during the traversal of the XML document. However, by calling such methods it is also possible to use SAX to create XML documents.

The *Document Object Model* (*DOM*) is a W3C recommendation that provides ways for manipulating HTML and XML documents. Initially, web browsers implemented various interfaces to manipulate HTML using JavaScript. To increase interoperability the W3C developed DOM which ended in a well-established interface. With a DOM-parser an in-memory tree of an XML documents can be created. After parsing it is possible to change nodes or create new ones as needed. Hence, DOM is best used for applications where the document elements have to be randomly accessed and manipulated.

Compared to more sophisticated concepts (like XSLT) the two low-level APIs requires a minor learning effort but reusability is often limited. Although template-like processing as in XSLT is possible in SAX, applications are usually targeted at specific XML instances. However, low-level APIs often provide a basis for higher-level languages or may be used to build rich applications like the publishing framework Apache Cocoon.

3.1.2 DSSSL

Before the first recommendation of XML has been published by the W3C the *Standard Generalized Markup Language* (SGML) was a well established metalanguage in which markup languages for documents could be defined. During the whole history of SGML great effort has been put into defining a style language capable of formatting Markup for screen and print.

The first partially successful attempt was FOSI (*Formatting Output Specification*) deployed by the United States Department of Defense. At the same time DSSSL, the *Document Style Semantics and Specification Language*, has been developed. DSSSL is an ISO/IEC standard and is based on a subset of the Scheme programming language, a dialect of Lisp. DSSSL contains both a transformation language and a style language that makes it an extremely powerful but also complicated language. Hence a cut-down version of DSSSL has been introduced in 1995 for the Internet: DSSSL-o (for DSSSL-online) [NH01]. Given that XML is a true subset of SGML, DSSSL can also be used to transform and format XML documents.

DSSSL enables formatting and other processing specifications to be associated with elements to produce and layout a document. During the DSSSL transformation process formatting information may be added to the result of the transformation [ISO96]. Figure 3.1 shows the conceptual model of DSSSL and illustrates this whole process.

The transformation part of DSSSL allows combining structures, creating new elements and associating new descriptions with particular sequences of content. The latter means that a sequence of elements in the source document may trigger the association of different formatting characteristics. For example, a paragraph following a warning may be required to be presented differently from all other paragraphs [ISO96].

Regarding to the style language the basic modules of an DSSSL stylesheet are so called *flow objects* like `paragraph`, `line-field`, `character`, `external-graphic`, or `box`. Every flow object has its own properties that controls the final layout. Flow objects support bi-directional writing modes, which implies that feeds are specified with `start` and `end`

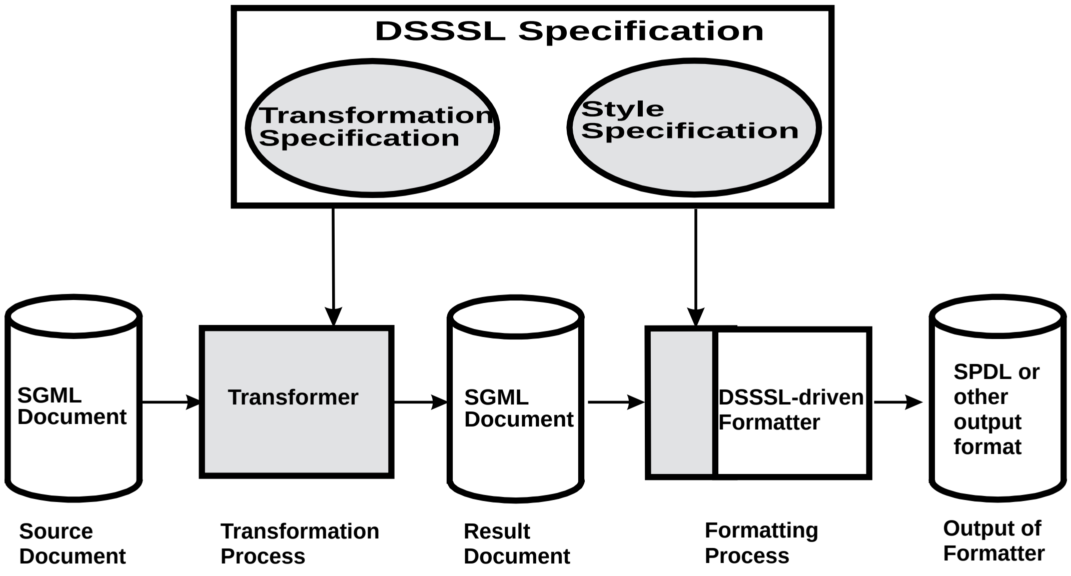


Figure 3.1: DSSSL Conceptual Model
Source: [ISO96, p. 11]

values instead of `left` and `right`.

Listing 3.1 shows a DSSSL example converting a `H1` element into a `paragraph` with increased font size, bold and uppercase letters and with spaces before and after.

Listing 3.1: DSSSL Example

```
(element H1
    (make paragraph
        font-size: (* 1.2 (inherited-font-size))
        space-before: 2em
        space-after: 1em
        font-weight: 'bold
␣␣␣␣␣␣␣␣(uppercase)␣))
```

A DSSSL-Stylesheet can be processed by a DSSSL engine, for example James Clark's free Jade[1]. Jade supports RTF, XML, TeX, MIF and SGML output.

[1]Jade: http://jclark.com/jade/

3.1.3 CSS

Cascading Stylesheets (CSS) is a stylesheet language allowing to describe the presentation of structured documents. The most common use of CSS is in conjunction with HTML. The first CSS recommendation was published by the W3C in 1996, followed by *CSS 2* in 1998. Currently *CSS 3* is under development.

In the beginnings of HTML in the early 1990s various browsers developed their own style language used to customize the appearance of web documents. Originally style sheets were primarily aimed at the end-user who could decide about the appearance of a web document. Later the stylistic capabilities of HTML grew and style sheets became less important. In 1994 the concept of Cascading Stylesheets was proposed by Håkon Wium Lie. The concept of *cascading* referred to the capability for a document's style to be inherited from other style sheets. As a next step, the W3C organized the ongoing development of CSS and in 1996 the first recommendation was published. Nevertheless it took more than three years before any web browser implemented CSS1 completely. The adoption of the new style language was difficult because many implementations remained inconsistent and faulty up to the present [Wik05, Cascading Style Sheets]. With CSS 2 and the upcoming CSS 3 capabilities of the style language increased and allowed web developers to describe the presentation of their sites fine-grained.

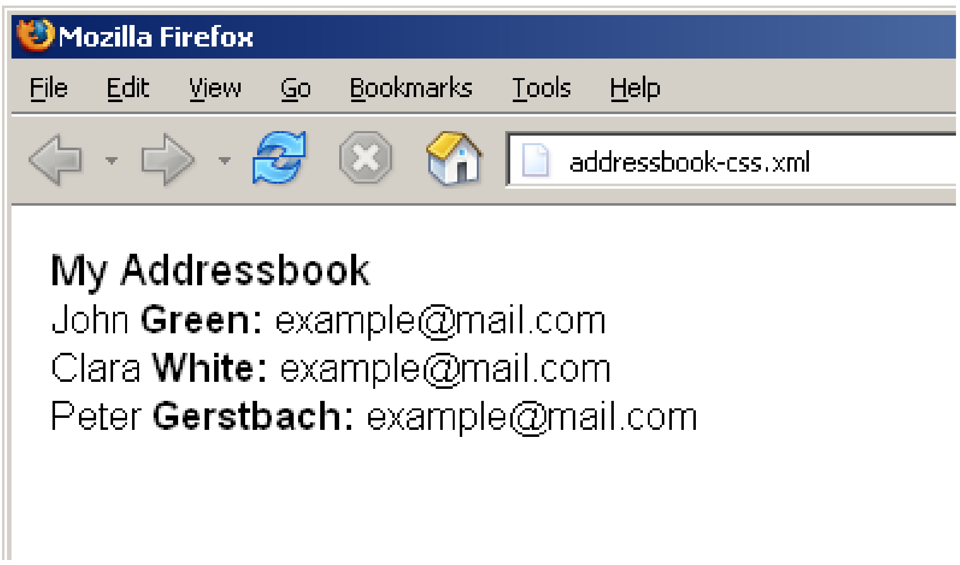
Although CSS is primarily used to format HTML or XHTML it is also capable of formatting plain XML. Listing 3.2 shows a css file that can be used to define the presentation of an addressbook file (the structure of this example is introduced in appendix A.1). In the XML file the stylesheet can be added by using the processing instruction:

```
<?xml-stylesheet href="addressbook.css" type="text/css"?>
```

The result in a web browser may look like figure 3.2.

Listing 3.2: CSS stylesheet for the addressbook example

```
addressbook {
  display:block;
  font-family:sans-serif;
  padding:1em;
}
addressbook:before {
  content:attr(name);
  font-size:large;
}
person {
   display:block;
}
surname {
   font-weight:bold;
}
surname:after {
   content:": "
}
gender {
   display:none;
}
```



My Addressbook
John **Green:** example@mail.com
Clara **White:** example@mail.com
Peter **Gerstbach:** example@mail.com

Figure 3.2: Addressbook Example: XML with CSS style

A CSS style sheet contains a list of *rules*. Each rule consists of a *selector* and a *declaration block*. A declaration block consists of a semicolon-separated list of *declarations*. Each declaration consists of a *property* and a *value* separated by a colon:

```
selector { property:value; property:value; ... }
```

CSS information can be provided by the author using an *external* CSS file, an *internal* style section, or *inline* definitions overriding the general style. Additionally a user or the user agent may also specify styles. If more than one rule applies to a structured element the style information of each rule is *cascaded* according to its weight.

Beside the concept of cascading the inheritance is another important concept of CSS. CSS defines some properties that are inherited by the children of an element in the document tree. This concept is demonstrated in the following line containing a `h1` heading element with an emphasizing element inside:

```
<h1>This is an <em>important</em> heading!</h1>
```

If a color is assigned to the heading the emphasized ”important” will inherit the color of the parent element. Together with the concept of cascading the formatting of structured documents with CSS is easy to learn but still powerful.

3.1.4 The XSL Family

XSL, which stands for *Extensible Stlyesheet Language*, is a family of W3C recommendations for defining XML document transformation and presentation. It consists of three parts [W3C05a]: first *XSL Transformations* (XSLT) a language for transforming XML, second *XML Path Language* (XPath) an expression language to access or refer parts of an XML document, and third *XSL Formatting Objects* (XSL-FO) an XML vocabulary for specifying formatting semantics.

All three parts can be set together to create a PDF document out of an XML document. Therefore an XSLT stylesheet transforms the XML to an FO document, which can then be rendered to PDF. Nevertheless XSLT is helpful on its own because it provides a way to transform one XML document into another, using a powerful language. The following sections covers XSLT (including XPath) and XSL-FO.

At the beginning of XSL there was a proposal submitted to the World Wide Web Consortium (W3C) in August 1997. The W3C published this note which defined XSL as a stylesheet language for the Web community to provide functionality beyond CSS [W3C97]. According to this note XSL is based on DSSSL and shares its fundamental design principles and processing model. XSL supports all functionalities of CSS so that a translation from CSS to XSL is possible. In contrary to DSSSL and CSS, XSL is expressed in XML syntax. All these amendments result in stylesheets that should be clear and easy to create.

While XML's popularity grew, the W3C set up a working group in January 1998 to organize and accelerate the development of the stylesheet language. During that time, the first implementations have been published. Soon it became clear that the transformation part of the language could be of importance in other areas too. For example, a direct transformation from XML to HTML was an important field. That is why the working group decided to split up the two parts into two specifications, namely XSLT and XSL-FO. In November 1999 the first *XSLT* recommendation was published, followed by the *XSL* recommendation (with the formatting part) in October 2001.

Section 3.1.5 covers the XSLT recommendation while XSL-FO is addressed in the succeeding section 3.1.6. Currently the W3C already works on new versions of both specifications. Some aspects of these upcoming versions are discussed at the end of each section.

3.1.5 XSLT

XSLT, standing for *Extensible Stylesheet Language: Transformations*, is a language for transforming XML documents [W3C01]. In most cases it is used to convert one XML document into another. However, it is also possible to transform an XML document into HTML or any other text-based format like comma-separated values or Java code. The concepts of XSLT are explained in detail in the first section. Section 3.1.5.2 introduces usage scenarios and shows which XSLT programming styles relates to them. Soon after the publication of the first XSLT recommendation, the first processors have been available. Some of them are addressed in section 3.1.5.3 and put into categories. Section 3.1.5.4 summarizes how the performance of transformations can be optimized, followed by an outlook about the future directions of this recommendation.

3.1.5.1 Concepts of XSLT

XSLT is a sophisticated but powerful language to transform XML documents. Compared to low-level XML processing technologies like SAX or DOM, XSLT is a high-level declarative programming language. Thus, it is more complicated and induces a higher learning effort, especially for programmers used to object oriented and procedural development. However, XSLT offers many advantages. For example, XSLT drives the separation of the transformation process in multiple stages. If one input format needs to be transformed into multiple output formats the different transformation stages help to increase reuseability and allow the development in teams.

This section describes the XSLT language in the current version 1.0 of the recommendation. If some parts are related to the upcoming version 2.0 it is explicitly stated.

An XSLT processor takes a stylesheet and a XML source document as input and transforms it to a result document as shown in figure 3.3. Internally the processor constructs a *tree* from the XML source and applies the transformation on it. The result of this transformation is called the *result tree*. The concept of the tree is the same as defined in *XPath* and consists of nodes corresponding to the components of the XML document.

XSLT is a high-level *declarative* language free of side effects. It enables a programmer

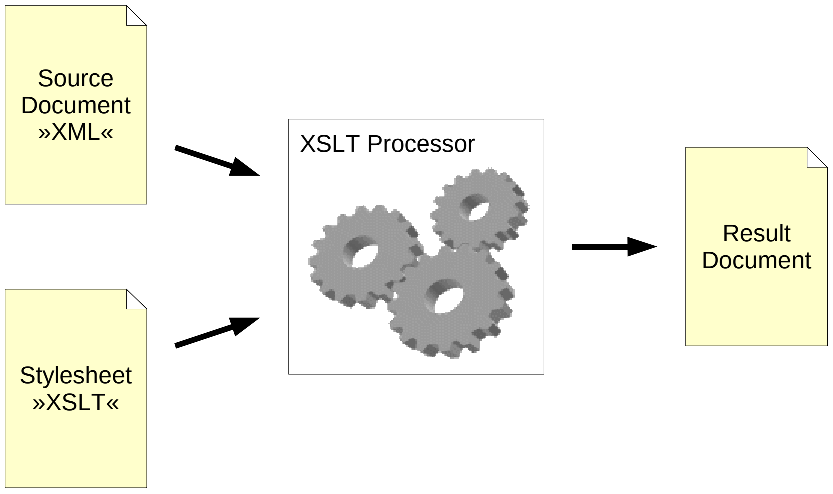


Figure 3.3: XSLT core process

to write a stylesheet that can be successfully applied to many different but similar XML documents. In XSLT the transformation can be expressed as a set of template rules. These template rules define the output that is generated if particular patterns occur in the input. During the *transformation process* the processor creates the output according to these template rules.

Template Rules While other programming languages modularize source code by using classes, methods or procedures, XSLT is based on template rules. Each template rule is expressed as an `<xsl:template>` element. This element may have a `match` attribute which value is called a pattern. A pattern is a subset of an *XPath* expression and determines the nodes of the source tree *matched* by the template rule.

For example, the element `<xsl:template match="/">` matches the root node of the source tree. A template with the pattern `book` matches a `<book>` element, and the pattern `book[@isbn="052101624X"]` matches a `<book>` element whose `isbn` attribute satisfies the given condition.

By starting an XSLT transformation, the processor first parses the stylesheet and creates an internal tree representation of it. In the second step it searches for a template which matches the document node of the source tree. Finding no suitable template, a built-in template is used. If the right template has been chosen, the processor executes its content, a *sequence constructor*.

In addition to the `match` attribute a `mode` can be defined for each template rule. When using *mode processing*, the current mode may be specified and then only the rules having this mode are processed. This feature is useful if the same node in the source tree are processed more than once, in different ways.

Sequence Constructor A sequence of XSLT *instructions*, *literal result elements* and *text nodes* inside a template rule is called a *sequence constructor*. It can be seen as the equivalent in XSLT to a block in a block-structured programming language like C or Java [Kay04b]. By evaluating a sequence constructor, the result is (mostly) a sequence of items.

Sequence constructors can be nested: given a template rules, looking like this:

```
<xsl:template match="/addressbook">
   <html>
      <head>
         <title>My Addressbook</title>
      </head>
      <body>
         <xsl:apply-templates/>
      </body>
   </html>
</xsl:template>
```

The template rule contains a sequence constructor consisting the *literal result element* `<html>` that copies itself to the output. Inside the `html` element, there is a nested sequence constructor, containing the `<head>` and the `<body>` element, that become evaluated in the same manner. The `<title>` element, that is nested inside the `<head>` contains the string "My Addressbook" that is called a *text node*; it gets copied to the output as well. The `<body>` element holds an *instruction* that becomes evaluated: this `<xsl:apply-templates/>` instruction causes that each child of the current node is selected and the matching template rule is evaluated.

Items inside a Stylesheet This section gives a closer look at the elements that may occur in a stylesheet. The following categorization of the elements is determined by the location, where they can be placed in the stylesheet.

- **The root element** of a stylesheet is the element `<xsl:stylesheet>`. A synonym for this is the element `<xsl:transform>`. Both elements are equivalent relating to usage and effect.

- **Top-level elements** are immediate children of the `<xsl:stylesheet>` element. If their namespace is the XSLT namespace they are also called *declarations*. These elements mostly set options, regarding the whole stylesheet (`<xsl:output>`, `<xsl:character-map>`, or `<xsl:key>` ...) or are used to structure and modularize the stylesheet (`<xsl:template>`, `<xsl:import>, or <xsl:include/>`). Another kind of top-level elements are *user-defined data elements*. They have namespaces outside the XSLT namespace and are often applied to control behavior that is *implementation-defined*.

- **Instructions** are the elements setting up a sequence constructor, hence they may all be children of a `<xsl:template>` element or other elements that can contain a sequence constructor. There are three different kinds of instructions: *XSLT instructions*, *extension elements* and *sequence constructors*.

  - **XSLT Instructions** are elements in the XSLT namespace like `<xsl:for-each>` or `xsl:value-of`. They are used for specific actions like iterating over other instructions or constructing nodes in the output.
  - **Extension Elements** are instructions defined by the vendor or the user. Their namespaces have to be defined as an attribute of the `<xsl:stylesheet>` element. For example, an extension may be used to define a `sql-query` element that returns the result of a query on a relational database.
  - **Literal Result Elements** are elements within a sequence constructor that are neither part of the XSLT namespace nor declared as extension. Such elements are copied to the result tree. This facilitates the creation of elements in the output by just writing them down in the stylesheet.

As a whole, XSLT defines about 50 declarations and instructions. The following list offers some of the most important elements, grouped into six categories [2].

- **Elements defining the structure of the stylesheet:** `<xsl:stylesheet>`, `<xsl:include>`, `<xsl:import>`. The element `stylesheet` is the root element of a stylesheet, the two other elements allow to unitize the stylesheet by including/importing parts of the stylesheet from different locations.

- **Elements related to template rules:** `<xsl:template>`, `<xsl:apply-templates>`, `<xsl:call-template>`. These elements admit to define and call template rules that can be used to modularize a stylesheet. Noted as well in the paragraph about template rules above.

[2]compare a similar grouping in [Kay00, p. 151]

- **Elements used to create or copy nodes:** `<xsl:element>`, `<xsl:attribute>`, `<xsl:text>`, `<xsl:value-of>`, `<xsl:copy>`, `<xsl:copy-of>`. They can be used to create new nodes in the output tree and to copy values and complete nodes from the input to the output tree.

- **Elements used to define program flow:** `<xsl:if>`, `<xsl:choose>`, `<xsl:when>`, `<xsl:otherwise>`, `<xsl:for-each>`. These elements define sequence constructors, that are used for conditional processing and iteration. Elements inside the if condition are evaluated only if some given condition is fulfilled. The `for-each` element is similiar to `apply-templates` and selects sequence of nodes and performs the processing on them.

- **Elements used to define variables and parameters:** `<xsl:variable>`, `<xsl:param>`. Variables and Parameters are equivalent to those in other programming languages, with the only respect that their value cannot be changed after it has been assigned during declaration.

- **Elements used to control the output:** `<xsl:output>`, `<xsl:result-document>`[3]. The format of the output (doctype, media-type, ..) can be configured and multiple result-documents may be created.

Variables, Parameters and Side Effects XSLT allows global and local variables to be defined. However, the use of variables is quite different to other programming languages. By defining a variable a name and a value have to be set. As the type system of XSLT is defined by the XPath specification, the type of a variable is as well defined by XPath. Hence, it can consist of a number, strings or even nodes. After the variable has been defined, it can be subsequently referenced in an XPath expression.

There is one notable difference of variables, compared to conventional programming languages: the value of a variable cannot be changed, once it has been set. The reason lies in the design of the language: as already stated XSLT is free of side effects. If a procedure of a programming language changes the environment (for example change the value of a global variable) it is said to have side effects. If a procedure has side effects it becomes important how often and in what order a procedure is called. Whereas XSLT was designed to be a declarative language without side effects. This means that applying a template never changes the environment. It is up to a processor to determine the best order which makes optimization possible. For example it would be possible to run transformations in different threads or to choose an order of processing that is more

[3]This instruction is new in XSLT 2.0.

memory efficient. When talking about variables this implies that it is not possible to change the value of a variable after it has been declared and initialized.

Parameters are like variables except that they are used to supply values from outside, when the transformation is invoked, or when applying a template or a function.

3.1.5.2 Processing Styles

Push vs. Pull Processing An often cited characteristic feature of XSLT is that programmers may choose between *pull* and *push processing style*. In many known programming languages processing is controlled by loops. In XSLT loops are possible as well, but very often a different approach is preferred.

In a traditional programming language iteration is implemented using a loop. For example, given a list of books (refer to appendix A.2), where each book has a title, an author, and a description, a loop is used to iterate over all books to output them as HTML. In XSLT an `<xsl:for-each>` element performs this iteration, using only one root template that contains the following snippet:

Listing 3.3: books.xslt using pull processing

```
      <h1>Books</h1>
      <xsl:for-each select="books">
        <xsl:for-each select="book">
          <h2>
            <xsl:value-of select="title"/>
          </h2>
          <p>
            <xsl:text>by </xsl:text>
            <xsl:value-of select="author"/>
          </p>
```

The *pull* processing style is about selecting nodes explicitly and deciding what to do with them. Usually this style is easy to understand for XSLT beginners. While most think this is a good approach, there is another implementation possible, that may be better. When using the classic *push* processing style, a template has to be written for each kind of node that can be found in the input. Given the book example, there are templates for the title, the author and the other elements. Those templates are called by the book template:

Listing 3.4: books.xslt using push processing

```
  <xsl:template match="book">
```

```
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="author">
    <p>
      <xsl:text>by </xsl:text>
      <xsl:value-of select="."/>
    </p>
  </xsl:template>
  <xsl:template match="title">
    <h2>
      <xsl:value-of select="."/>
    </h2>
  </xsl:template>
```

Compared to each other, both transformation have the same results: a list of the books and their description. Which approach is best suited for a given case is a matter of the match between the source and the result. This was well noted by Jeni Tennison in [Ten01]:

> *If the result follows the structure of the source, then a push method is more natural—the source drives the process. If the result has a substantially different structure from the source, then a pull method is more natural—the result drives the process.*

A good example for the first approach, where the result follows the structure of the source, is a text-document: paragraphs from the input are transformed to paragraphs in the output (for example HTML), some formatting is added, or some special elements in the input are replaced by pre-defined icons. In this case the push approach is the way to go.

A business report is a good example for the second approach: the input is a data-oriented XML file that contains sales data, the output is a document that is used for a presentation of the data. The resulting structure of the document is totally different from the data. Hence a pull approach is better suited.

Of course, not every case can be put into one of these two categories. Often a document contains both data- and document-oriented XML; this would result into a stylesheet that uses both the pull and the push style. And finally, the pull method is needed when the source must be present in the result more than once: an article might be formatted using the push approach, while the table of contents at the beginning of the document is generated using the pull approach.

XSLT Instruction vs. XPath Expression In some cases an XSLT instruction can be replaced by an appropriate XPath expression or vice versa. Bongers [Bon04, p. 75] demonstrates how this can be accomplished with an XSLT 2.0 processor. The following two examples can be applied on the addressbook example (see appendix A.1).

The XSLT-oriented approach uses a template to output the name and e-mail address of a person using a template and value-of instructions:

```
<xsl:template match="person">
<xsl:value-of select="forename"/>
<xsl:text> </xsl:text>
<xsl:value-of select="surname"/>
<xsl:text> </xsl:text>
<xsl:value-of select="e-mail"/>
</xsl:template>
```

The output of this part of the transformation for a person might look like this:

```
John Green example@mail.com
```

The XPath-oriented approach uses new *sequence expressions*, introduced in XPath 2.0. With the XPath expression shown above all parts of the sequence are separated with a whitespace. The result of the transformation is the same as with the XSLT-oriented approach.

```
<xsl:template match="person">
<xsl:value-of select="(forename, surname, e-mail)"/>
</xsl:template>
```

Another possibility to replace XSLT with an XPath 2.0 expression arises when evaluating conditional processing [Bon04, p. 123]. This example outputs the correct salutation, dependent from the gender of a person. In XSLT 1.0 this can be done with the following choose instruction:

```
<xsl:choose>
<xsl:when test="gender='male'">Mr.</xsl:when>
<xsl:otherwise>Ms.</xsl:otherwise>
</xsl:choose>
```

XPath 2.0 supports expression that may contain if-then-else parts as followed:

```
<xsl:value-of select="if (gender='male') then 'Mr.' else 'Ms.'"/>
```

Other possibilities where different XSLT instructions or XPath expressions produce similar results are shown in section 3.1.5.4 about performance optimizations. The performance tips reveal that choosing between these two approaches often affects performance.

3.1.5.3 Processors

The software responsible for transforming source trees into result trees using an XSLT stylesheet is referred to as the processor [W3C05c]. This chapter makes an attempt to categorize processors and contains information about the two prevalent implementations, Xalan and Saxon.

XSLT processors can be categorized in many different ways. For example this can be based on their internal way of processing, their capabilities or their intended audience.

When using XSLT 2.0, processors can be easily divided into two groups, which are defined in the specification:

- A *basic XSLT processor* is an XSLT processor that implements XSLT 2.0 with the exception of constructs related to schema processing.
- A *schema-aware XSLT processor* covers the full specification, including all schema related constructs.

Currently, Saxon is one of the rare complete implementations of XSLT 2.0. A free version (a basic XSLT processor) is available for download at SourceForge[4]. A schema-aware version is available on a commercial license.

Interpreting and Compiling Processors Another distinction may be drawn between *interpreting* and *compiling* processors. A compiling processor generates executable code from an XSLT to speed up the process of transformation. Most of the available processors are interpreting processors. An interpreting processor parses and interprets the XSLT and applies it on the XML file.

In contrast to the above mentioned a compiling processor takes a stylesheet as input and generates executable code as its output. In case of Java this is of course not machine code but Java bytecode. This bytecode can then be executed by a Java Virtual Machine to start the transformation. The generated class file is commonly called a *translet*.

The first compiling XSLT processor was *XSLTC* developed by Sun Microsystems. It was donated to the Apache Open Source Foundation and is now a component of

[4]Saxon-B at SourceForge: http://saxon.sourceforge.net/

the Xalan-J2 project. The Apache community has agreed to make XSLTC the default processor for developing XSLT 2.0 and in Java 1.5 XSLTC was chosen as the default transformer [Suna].

The question whether to compile or not is an ancient question of computer science. This is often referred to as the *Compile Conjecture* and depends on many factors like the kind of project [Pun03]. When speaking about XSLT processors, the claim is, that such a translet is both smaller and faster. The purpose of this is to make it ideal for small devices with limited memory and slow processors. In practice XSLT's translets are in fact smaller: the class-file itself is about two times smaller than the stylesheet. The library, which is needed to execute the class-file, is also noticeable smaller than the comparable library for an interpreting processor. By comparing speed, the run-time transformation performance is about the same as an interpreting processor. At first glance this might be unexpected, but in fact Sun has proved right that a good interpreter using techniques such as just-in-time compilation is able to be as fast as traditional compiled code [Kay00, p. 840]. So the only relevant advantage of translets are that no XSLT file is needed to be parsed. Most of the work (building the tree, sorting and serializing) is exactly the same whether the stylesheet becomes compiled or not [Kay01]. Other advantages are that translets can be moved between machines or downloaded and that the source code of the stylesheet is hidden.

At XML Europe 2003 Jacek R. Ambroziak officially released his compiling XSLT processor *Gregor*. Ambroziak, who was the architect of Sun's XSLTC processor, started with *Gregor* a second approach to build a high performance compiling processor. Currently version 1.1. can be obtained. In [Amb03] Ambroziak outlines that a fast XSLT processor is a key factor to real-time XML processing. For example a web portal, where the information requested is natively available as XML, may use *Gregor* with its algorithmic optimizations to achieve a low latency. However, code optimizations as discussed in 3.1.5.4 or caching strategies as implemented in Cocoon may yield a similar result.

Hardware Accelerators Another kind of processors are hardware accelerators. According to [Sch05] XML based traffic on network will grow up close to 50% in 2008. To accomplish this increase in traffic and workload, hardware accelerators for XML processing including XSLT may alleviate the problem.

Current hardware solutions from DataPower[5] and Sarvega[6] use a rack-mountable network device to accelerator common types of XML processing. These systems sup-

[5]DataPower XA35 XML Accelerator: http://www.datapower.com/products/xa35.html
[6]Sarvega Speedway XSLT Accelerator: http://www.sarvega.com/xml-processing.html

port XSLT transformations, schema validation and SOAP routing. In the box a XSLT stylesheet is converted to machine code optimized for the built in processor.

Xalan *Xalan* is a widely-used opensource XSLT processor developed by the Apache Software Foundation. Both a C++ and also a Java versions are available. This section mainly focuses on the current Xalan-Java version 2. Xalan is possibly the most used XSLT processor, since it is shipped with Java 1.4 and has been set up as the default transformer of the JAXP implementation [Sunb]. Xalan also contains the compiling XSLT processor XSLTC, which is the default transformer in Java 1.5.

Xalan implements TrAX, the Java *Transformation API for XML* which is part of JAXP. It may be configured to work with any XML parser that implements JAXP. It can process Stream, SAX or DOM input, and output to a Stream, SAX or DOM. Xalan also supports Java and scripting language extensions, that allows calls to a procedural languages from within the stylesheet.

Currently Xalan does not support XSLT 2.0. Although there is a XSLT 2.0 branch in apache's CVS, it has not been altered for almost 2 years. However, as already stated, the compatibility notes of Java 1.5 let assume that XSLTC will be the default processor for developing XSLT 2.0 support [Suna].

Saxon Saxon is an XSLT and XQuery processor for Java developed by Michael Kay[7]. Saxon comes in different versions, as explained below.

Saxon 6.5.3 is a mature and stable implementation of XSLT 1.0. Additionally many of the features that were defined in the XSLT 1.1 working draft are implemented as well. Saxon implements the JAXP interfaces and can process SAX and DOM inputs. It supports *EXSLT*[8] extension functions and extension elements to be used across different XSLT processors; additionally Saxon provides some own extensions.

The current Saxon 8.x releases support the XPath 2.0 and XSLT 2.0 specifications, that are at the moment under development by the W3C. Saxon 8 is available in two versions: Saxon-B is a non-schema aware open-source processor; Saxon-SA is based on Saxon-B, additionally providing schema-aware processing and available as a commercial product.

[7]Michael Kay is the editor of the W3C's XSLT 2.0 recommendation
[8]EXSLT homepage: http://www.exslt.org/

3.1.5.4 Performance

This section tells about solutions for high performance XSLT processing. According to Moore's Law the complexity of an integrated circuit will double in about 18 months [Wik05, Moore's Law]. Although exponentially improved hardware does not necessarily imply exponentially improved speed, the performance increase is definitely considerable. Nevertheless the demand for high performance XML processing doubles every year or even faster. Because of this gap performance optimization of XML processing is a very important issue today.

As already mentioned before, XSLT is a high-level declarative language. Because of concepts like template rules the traditional sequential programming becomes obsolete. But a poorly optimized XSLT processor, or (even more widespread) poorly programmed stylesheets cause considerable performance bottlenecks.

Before thinking about details of performance and optimization some best practices about performance should be considered. Of course these best practices apply as well to any other technology. In a first step the performance requirements should be defined. Then, a prototype often helps to evaluate the risk. The get correct data, where the bottlenecks are located, measurements should be made at every stage of development. Finally, code should only be optimized if it gives measurable benefits.

Before deciding about the processor used, benchmarks usually help to measure performance of different approaches. Well-established benchmarks include XSLTMark from DataPower[9] and the XSLT Benchmark from Sarvega[10]. Although benchmarks mostly try to be neutral it is important to measure the performance with data, representative of the real workload. The results should be measured, saved and compared with respect to the tips above. Tracing facilities[11] are helpful to investigate which instructions are executed most often and in what time. This helps finding bottlenecks which should be optimized at first.

Efficient use of XSLT For a simple transformation the performance depends largely on the cost of parsing the input, building the tree and the output. For this reason big effort should be put into this part before thinking about writing efficient XSLT code. The following paragraphs summarize some performance tips from Michael Kay and others from the XSL mailing List [Mul] and the XSLT FAQs [K$^+$05] maintained by Dave Pawson.

[9]XSLTMark: `http://www.datapower.com/xmldev/xsltmark.html`
[10]XSLT Benchmark: `http://www.sarvega.com/xslt-benchmark.php`
[11]For example Saxon provides a tracing facility with the `-T` option.

Usually an XSLT processor builds a tree of the input documents at the very beginning - that is why very large XSLT transformations should be avoided if possible. Splitting a document into smaller pieces may also improve performance. The limit depends on memory size and can be considered at about 10MB. However there exists some powerful XSLT processors which support streaming mode, so they can even handle great amounts of data in reasonable time. The same matter applies to the output document: Serializing the result is a time-consuming process. The use of imports, if possible, is recommended. For example, if you're generating HTML, the use CSS makes the document considerably smaller.

From the technical aspect of the matter the source document should be supplied as a SAX or byte stream, not as a DOM. Since a DOM parser builds a memory-exhaustive tree of the source document, SAX should be always preferred to DOM. Most processors use an internal tree model, which represents the stylesheet; a DOM parser would build a tree that has to be reread by the processor to build its own tree.

Caching is another important approach to speed up transformations. The XSLT processor and the stylesheet should be kept in memory between runs. This eliminates high startup costs. Some processors even support compiling or serializing the stylesheet. XSLT APIs, like the *Java Transformation API for XML* (TrAX), provide ways to construct a processor instance which can be applied multiple times on different documents without having to re-load the stylesheet. If you use the same source document repeatedly, it should be kept in memory too. Although this situation is not as common as using a stylesheet repeatedly it can improve the transformation considerably. Of course this also applies to the result document: instead of doing the same transformation twice, the results should be cached.

Splitting complex transformations into several stages may also increase performance. Very complex transformations are difficult to understand; hence, using a pipeline is recommended. This increases modularity, maintainability and reusability and makes performance optimization possible. A pipeline also allows to configure the process more precisely. By using the result of a transformation as input of another transformation or another process serializing the document and reparsing it should be avoided in either case.

Writing efficient XSLT Although usually more time is spent on building the tree than on navigating it, there are some very expensive operations to be avoided if possible. Of course, every XSLT processor is trying to optimize queries but compared to a database query engine, those optimizations are of an entirely different art. A database query engine can optimize by rearranging queries to exploit persistent indexes. An in-memory processor does not have this opportunity because the only indexes available are those that are constructed transiently for the duration of a transformation or query [Kay04c].

The following list shows some XPath and XSLT-related hints extracted from the XSL mailing List [Mul] and the XSLT FAQs [K+05], which should help to write efficient XSLT.

Some XPath expressions and XSLT instructions should be used with care, including `//item`, `<xsl:number>`, and `count()`. The expression `//` is a shortcut for `/descendant-or-self::node()/`. For example `//item` will select any `item` element in the document. To accomplish this task the processor has to traverse the whole document which is very expensive. Therefore the use of `//` should be avoided in favor of a more explicit path. The count function returns the number of nodes in the argument node-set. With the number instruction a similar task can be achieved. Both instructions may need to access many nodes and thus should be reviewed carefully. Often the same result can be yielded using the `position()` function.

A node-set should not be evaluated more than once. Instead it should be saved in a variable. This is very important in particular when sorting nodes - a very expensive operation. In this case the result should be saved in a temporary tree (known as result tree fragment in XSLT 1.0). If some nodes need to be evaluated often, keys should be used. The `<xsl:key>` element allows to declare a named key, that can be referenced with the `key()` function in expressions and patterns. Most processors make use of an index or a hash table to offer fast access to the defined nodes.

When using the preceding[-sibling] or following[-sibling] axes caution is advised: These XPath expressions may also access many nodes. They should be avoided or optimized by using variables and keys. Another risky XPath expressions are the ones that contains the equals operator to compare non-singular node-sets; to process such a operations many nodes must be traversed. To output the text value of a simple PCDATA element, `<xsl:value-of>` should be used in preference to `<xsl:apply-templates>`. Although the built-in template for text nodes performs an implicit value-of, an explicit call may enhance performance.

3.1.5.5 Outlook

Soon after the publication of the XSL 1.0 recommendation W3C started working on a successor. The first working draft was version 1.1 but soon it became clear that larger, strategic decisions are needed which will one day result in XSLT 2.0. The current candidate recommendation of XSLT 2.0 is from November 3, 2005 and has the following main enhancements to the 1.0 specification [W3C05c][Kay04b]:

- There are many new features introduced that have been missing in the current specification. The most important ones are a built-in ability for grouping (this is similar to grouping in SQL) and some new functions for handling of strings and numerics including regular expressions.
- XML Schema: an XSLT 2 processor can now be *schema aware*. This means that information from a schema, to which the source or the result document validates, can be used in processing. For example this makes it possible to sort dates and other types that have a natural order which differs from simple strings or numbers.
- XSLT 2.0 is developed alongside XPath 2.0, so all modifications in XPath are also relevant to XSLT 2.0.
- With a new instruction multiple output documents are now possible.
- Temporary trees allows it to use a tree, which was created while processing, to be used as input in another part of processing.

Currently, there are already some implementations of the current XSLT 2.0 candidate recommendation.

In [Kay04a] Michael Kay illustrates that the new features introduced in XSLT 2.0 extend the scope of the language. The combination of the grouping feature, the new string manipulations, and regular expressions, make transformations possible which take data from legacy ASCII-based formats to standardized XML vocabulary (*up-conversion*).

With the `unparsed-text()` function it is possible to load text from a resource identified by an URI. The new functions and instructions for regular expressions now enables string manipulations that are known from Perl and similar languages; and the new grouping instruction allows to add an additional layer of hierarchy in the result tree that is not present in the source tree. Although there exists workarounds (like the *Muenchian grouping*) for some grouping problems the new instruction provides a simple way defining grouping criteria. For up-conversion this is of relevance because grouping allows to replace the implicit structure in texts (detected by string manipulations) with explicit markup.

3.1.6 XSL-FO

The Extensible Stylesheet Language (XSL) [...] is a language for expressing stylesheets. Designers can use an XSL stylesheet *to express their intensions how structured content*

(XML) *should be presented; that is, how the source content should be styled, laid out, and paginated onto some presentation medium, such as a window in a Web browser or a hand-held device, or a set of physical pages in a catalog, report, pamphlet, or book.* [W3C01]

The quotation from the XSL specification above shows that XSL is targeted at a very broad area of applications. XSL consists of two parts: a language for transforming XML documents and an XML vocabulary for specifying formatting semantics. The transformation part (XSLT) is described in section 3.1.5, the formatting part (XSL-FO, while FO stand for *Formatting Objects*) is the topic of this section.

In short XSL-FO is just another XML based markup language: a document layout language. Nevertheless in conjunction with XSLT it is much more than this. XSL-FO provides means to define fine-grained formattings, but the actual content may come from arbitrary structured XML. While XSLT performs the transformation of the source document to XSL-FO, an FO processor intereprets these results.

According to [Paw02] the following cases show where XSL-FO is a good choice for the document production process (extract):

- The source XML is valid to a schema (or DTD) that changes not at all or slowly.
- The document format is easily repeatable.
- Automation is desirable.
- Human checking of the final form is not essential and does not add process value.

The first point is mainly required by XSLT. A meaningful transformation is only possible if the source document structure is known in advance. It is almost impossible to write a XSLT stylesheet if the source is only well-formed but not valid to a schema. In addition to validity, the source should also be well structured. The more variable content is allowed the more the stylesheet has to "guess".

One main advantage of XML document processing is automation. XSLT and XSL-FO supports automation at best. It is even usual that no human checking is performed on the final form. This makes XSL-FO very useful in an automated process. A drawback of XSL might arise by using many different charsets and fonts. A XSL formatters must be configured to work with fonts that are not initially supported. Otherwise the output will have glyphs missing.

3.1.6.1 Concepts of XSL-FO

As already stated XSL-FO is often used in conjunction with XSLT. The diagram of XSL conceptual model (figure 3.4) shows a source tree (XML), transformed in a result tree (actually an XSL-FO document), itself rendered by the XSL formatting on devices including printers, a cell phone and a Web browser [W3C01]. XSL-FO is often used in this two stage process. However, the use of a formatter on a cell phone is currently out of focus because such a transformation process is far too resource intensive for a small device.

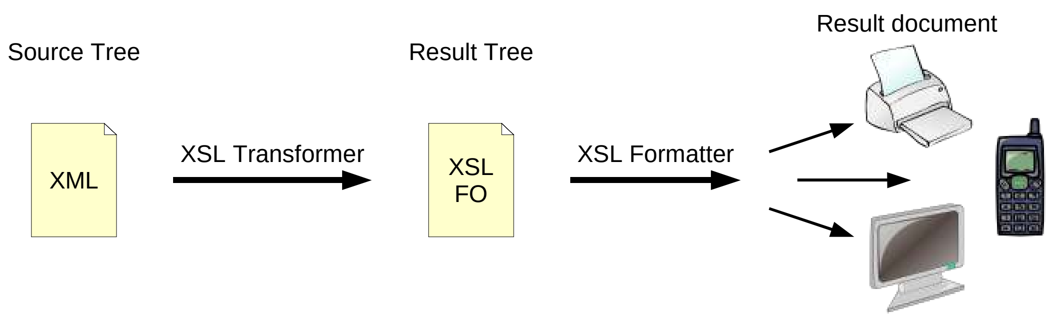


Figure 3.4: XSL Two Processes: Transformation & Formatting
Source: [W3C01]

Any source tree that an *XSLT processor* can process may be used as input. Usually this is an XML file that validates against a known schema. This file will be transformed to an FO document by applying a stylesheet on it. The stylesheet was created to work with any XML file that conforms to the schema. At the first stage it creates the elements from the FO namespace using literal result elements and other XSLT instructions. The second stage of the process is performed by the *XSL Formatter*. The formatter renders the FO file to a result document. Most formatters use Adobe's Portable Document Format (PDF) as the main output format; other formats include Postscript or Microsoft's Rich Text Format (RTF).

An FO document is an XML vocabulary made up of *Formatting Objects* which are annotated with *Formatting Properties*. For example, a *Block* is a Formatting Object commonly used for formatting paragraphs, headlines or captions. The element `<fo:block>` is used to define a Block. It can have many Formatting Properties, for example `font-size`, with absolute values like `14pt` or relative ones like `120%`. An example of such a Block definition that defines a formatting property using an attribute looks like this:

```
<fo:block font-size="14pt">Some content.</fo:block>
```

FO capabilities XSL-FO is a very powerful and feature-full formatting language. It allows to define *page layouts* that differs according to well defined rules. For example it is possible to use different page layouts on odd and even page respectively. Beside the already explained blocks it supports graphics, inlines, tables and lists which can be formatted using a huge set of formatting properties. With their help it is possible to specify absolute positioning, borders and margins, backgrounds, fonts, hyphenation, text alignment and wrapping and many other things.

Beside the features, well known to most end users, FO also supports many advanced features. For example an FO document may specify a writing-mode like right-to-left (e.g. used in Farsi, Arabic and Hebrew texts) and top-to-bottom (e.g. traditional Chinese and Japanese texts).

In this context the naming of the elements becomes important. Instead of naming the regions of a page header, body, and footer, FO calls them *region-before*, *region-body* and *region-start* to make the name independent from the writing-mode. FO defines at least five regions which may be used to put content in and which layout can be defined by the page layout. Figure 3.5 shows the page model of an FO document containing the five regions.

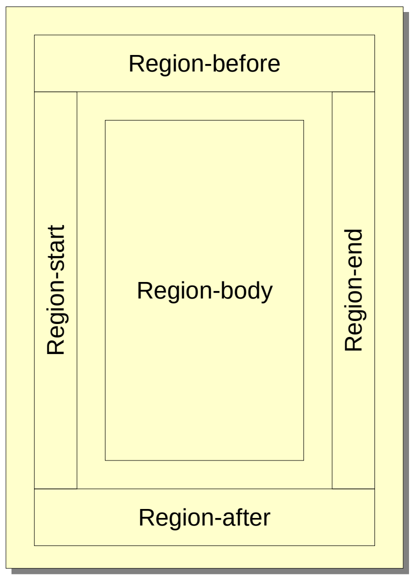


Figure 3.5: XSL Page Model

Formatting Objects An XSL-FO document contains a tree of *formatting objects*. Those formatting objects can be categorized into three types:

- The skeletal structure of an FO document is made of *declarations, pagination and layout formatting objects*. They are used to define the layout of the pages and how the regions of the pages should be filled.
- *Block-level formatting objects* represent smaller parts of a document like paragraphs, lists, or tables.
- *Inline-level formatting objects* resided inside of blocks and are most commonly used to format a portion of text.

The appearance of a formatting object is specified by *formatting properties*. Every formatting object has its own set of supported formatting properties.

The root node of an FO document is the `<fo:root>` element. It contains layout definitions and a page sequence list. The former can be used to define paper sizes and margins of the regions of a page. The latter contains `<fo:page-sequence>` elements used to split up a document into parts with different page layouts. A page sequence contains a `<fo:flow>` or a `<fo:static-content>` element for each region of the page. Flows are used to define content *flow*ing over many pages, static-content is repeated on every page and is typically used for headers and footers.

Block-level formatting objects include paragraphs, lists and tables. They represent smaller parts of a document. Blocks can be nested and therefore allow to structure a document. The most simple block in FO is the `<fo:block>` element. It may contain `CDATA`-sections (characters) making up the text of a paragraph and other block-level elements or inline-level elements. A block always starts a new line.

Inline-level formatting objects may be placed only inside a block-level object and cannot contain block-level elements. They do not start a new line and therefore they are used to format portions of text. For example inlines can be used to format a word in bold letters or to place an image next to some text.

Formatting Properties The structure of a document is based on the formatting objects it contains. The actual formatting of these objects are determined by the *formatting properties* of each formatting object. The specification lists the set of supported properties for each formatting object. E.g, the `<fo:block>` element may have font properties (`font-family`, `font-size`, `font-weight`, ...), border properties (`border-bottom`, `border-top`, ...) and many others.

XSL's property system is similar to CSS's. Actually CSS is often cited from within the FO specification: both specifications share concepts and many formatting properties have the same name and the same meaning as in CSS. While in HTML CSS properties are defined inside the `style` element using a comma-separated list XSL-FO uses standard XML attributes for them.

The following paragraphs summarize the two most important concepts needed to be known regarding property refinement:

The property system supports *inheritance*. Many but not all properties are inheritable. This means that those inheritable properties are propagated down the formatting object tree from a parent to each child: an `<fo:inline>` element inside an `<fo:block>` element inherits the value of each inheritable property (e.g. font-size) from its parent—unless it is not set directly on the inline element. Inheritance has the advantage that a child must not declare again all properties already declared on the parent, or one of it's ancestors.

Some property values use relative types. During the refinement process the actual values of those relative values must be computed. The font-size, for example, may be specified using the value `120%`; this percentage value relates to the font-size value of the parent object. If the parent has a font size of `10pt` the value of the child's font-size property is computed to `12pt`.

Areas As already stated in the previous section an XSL-FO document contains a tree of formatting objects. This tree serves as input to a formatter. From this input a formatter generates an ordered tree of *Areas*, the *Area-Tree*. One formatting object might produce more than one area. For example, a block element produces two areas if it is split over a page boundary (see figure 3.6).

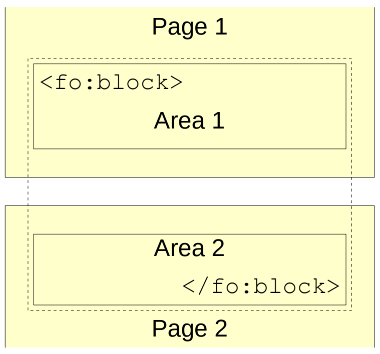


Figure 3.6: A Block split over a Page Boundary

In principal FO specifies two different area types:

- *Inline-Areas* correspond to chunks inside a block, for example characters and inline images.
- *Block-Areas* correspond to paragraphs, lists and tables.

Two more area types are defined that correspond to a single glyph (*Glyph-area*) and a line inside a block (*Line-area*). After this there are some other areas related to the page layout.

A formatting object may have properties that specify its formatting. What a property is for a formatting object, is a *trait* for an area. In most cases properties and traits correspond one-to-one because normally one area corresponds to one formatting object. Figure 3.7 summarizes the XSL process for a sample formatting object block and two attributes: objectification turns the `block` XML element to the *block* formatting object and its attributes into properties. After refinement properties become traits (units are normalized) and then a block area is created for the block formatting object [W3C01].

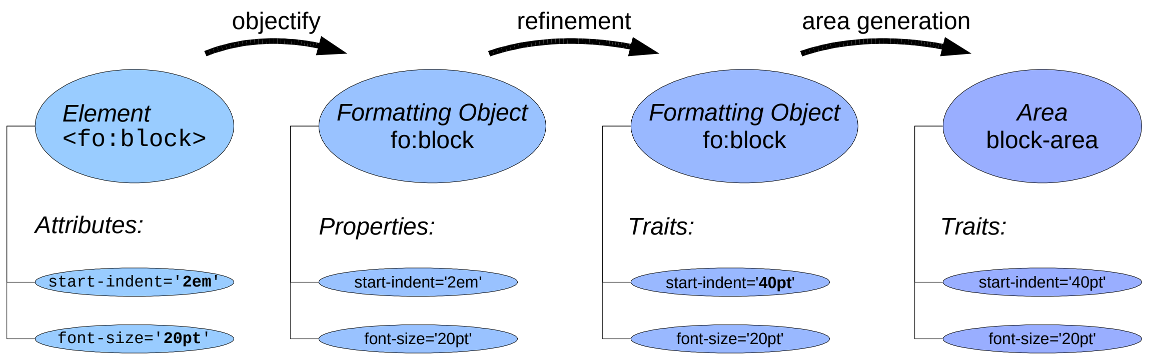


Figure 3.7: FO-to-Area Process

3.1.6.2 Formatters

This sections focuses on three implementations of XSL-FO formatters, namely Apache FOP, RenderX XEP and Antenna House XSLFormatter.

Apache FOP According to Apache FOP website, *FOP is the world's first print formatter driven by XSL formatting objects* [Apa]. FOP is written in Java and released as open source software under the Apache License. Several output formats are supported (e.g. PDF, PS, PCL, SVG) and others can be added using plug-ins. The primary output target is PDF. The latest version of FOP is 0.20.5 and was released in July 2003. After a complete redesign addressing the design issues for layout and performance, the FOP team is currently working on stable release for FOP 1.0. The current version of this branch is 0.92 beta and already implements many new features compared to the older branch.

RenderX XEP XEP is a commercial XSL-FO and SVG processor written in Java. In the current version 4.4 it supports the output formats PDF and Postscript. A free personal edition is available for private use. RenderX also provides a connectivity kit for an easy integration of the XEP engine in other applications like jEdit, Ant, Cocoon, Java Servlets and J2EE applications.

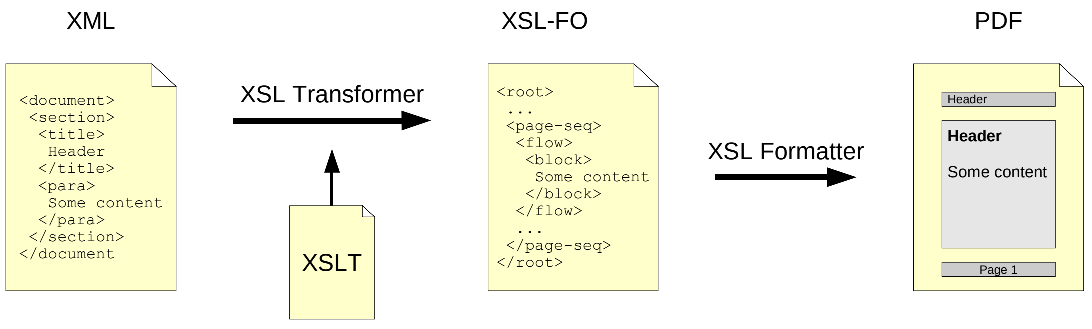
Antenna House XSLFormatter XSLFormatter V3.3 is a commercial XSL-FO formatter that supports the PDF output format. It includes support to render SVG and WMF/EMF vector graphics and MathML. XSLFormatter provides interfaces for .NET, COM, Java, C++/C and for SOAP/HTTP Web Services.

XSL-FO standard compliance, as recommended by the W3C, is classified into three categories: Basic, Extended and Complete. However, FOP's support of FO is limited (version 0.20.5). Most basic formatting objects and properties are supported, but a significant number is missing. In addition some elements from the extended and complete sets are also implemented. XEP supports nearly all basic elements as well as a large number of extended and complete elements. The feature set of XSLFormatter is comparable to those of XEP, with each formatter supporting a few features the other does not support.

3.1.6.3 Example

Figure 3.8 illustrates a simple example how XSL-FO can be used in conjunction with XSLT to create a nice looking PDF document. An XML file that conforms to some *document* vocabulary is used as input. Inside of a root `<document>` element multiple `<section>` elements with titles can be placed. The sections contains the actual text wrapped with `<para>` elements that represents paragraphs.

Then a stylesheet is applied to the XML file. The root template of the stylesheet creates the `<fo:root>` element and all the page-masters. Another template creates page-sequences whenever a `<section>` element in the input XML is encountered. For the content of the section an `<fo:flow>` is created that contains the title of the section and the following paragraphs. In addition to the flow an `<fo:static-content>` object is created that becomes the page header. It contains the title of the section as well. Another template transforms the `<para>` elements into `<fo:block>` elements.



```
<document>
 <section>
  <title>
   Header
  </title>
  <para>
   Some content
  </para>
 </section>
</document
```

```
<root>
 ...
 <page-seq>
  <flow>
   <block>
    Some content
   </block>
  </flow>
  ...
 </page-seq>
</root>
```

Figure 3.8: XSL Two Process: Transformation & Formatting — Example

The source code producing an output as shown in figure 3.8 is available in appendix A.3.

3.1.6.4 Outlook

In December 2003 the first working draft of a revised version of XSL, version 1.1, was published by the W3C. The current working draft is from July 2005. XSL 1.1 introduces new functionality to support change marks, indexes, multiple flows, and bookmarks. Some existing functionality has been extended to support partial sums, and page number referencing [W3C05b].

These new features are especially useful for business-type documents, like invoices, purchase orders or marketing materials. XSL 1.0 supports markers that can be retrieved to calculate, for example, subtotals. Since XSL 1.1 markers can be retrieved inside of table headers or footers. This makes it possible to create subtotals of a table at the end of a page or to create tables that contain ”Continued on next page” captions.

In XSL 1.0 it is only possible to obtain the total amount of pages, if an object on the last page is known, that can be referenced. XSL 1.1 simplifies this with a new property that can be used as well, when it is not known which object is the last one. Another important feature are multiple flows. In XSL 1.0 a page-sequence can only contain one flow. In XSL 1.1 it is now possible to have several flows. This can be used for example to layout magazines, where several articles are starting on the first page, and every article is continued on a later page. Multiple flows can also be used to accomplish more complex page layouts, like wrapping text around images [Bal04].

Of course the working draft is still subject to change but it is expected to be taken through the W3C Recommendation track process soon. Some implementations of XSL already support some of the new features that will be introduced in the upcoming version.

3.1.7 Streaming Transformations for XML

Streaming Transformations for XML (STX) is a one-pass transformation language for XML documents. As such it is intended as an alternative to XSLT providing high-speed and low memory consumption. Unlike XSLT there is no need to construct an in-memory tree of the XML document which makes STX suitable for very large data and resource constrained applications [BBC+].

The requirements leading to the current STX specification are discussed in [Bec04]. The most important requirement is the stream-oriented architecture. The source documents are supplied in the form of streams of XML events, for example using SAX. Using a stream-oriented approach decreases memory consumption because not the whole document must be present in memory. However, the power of serial transformations are limited. So the language offers the possibility to process arbitrary transformations - at the expense of memory. STX code is is very similar to XSLT: it is XML based and many instructions that have counterparts in XSLT have the same local-name. The specification is platform independent and compatible to the W3C's XML family specifications. Currently there exists two implementations: the Java-based *Joost* and the perl implementation *XML::STX*.

3.2 Document Processing Systems and Formats

As described in the preceding section, there are many technologies waiting for deployment. This section presents some approaches using the quoted technologies. Most technologies for document processing are domain-independent. Because knowledge representation is strongly related to domains, corporate cultures and communities, the systems introduced in this section are mainly targeted at specific domains. Most attempts to define a universal DTD ended up either unused or unfinished [DPS01].

DocBook, for example, tries to fulfill the need for a vocabulary in which to write computer hardware and software documentation. The *Text Encoding Initiative* focuses on literary and linguistic texts for online research, teaching, and preservation. Furthermore, current office suites also support XML formats, like *WordprocessingML* and *OpenDocument*. Their intent is to introduce a interchangeable format that simplifies interchange of documents, extraction of data and integration within external systems. All these approaches have in common that they use XML to describe the structure of documents. They offer schemas describing the vocabulary and tools to work with documents.

The document processing formats of this chapter serve as perfect examples of a document-oriented use of XSLT. A well structured XML file is transformed to an XSL-FO document (and thus PDF in subsequent steps) with the use of a complex and customizable stylesheet. This approach is a good example for *separation of concerns* (SoC) which is commonly used in XML based publishing systems (cp. Apache Cocoon). The main purpose of SoC is to separate content, logic, and style. At the beginning of the web a HTML page always intermingled content and style. For example, content was described using headings and paragraphs: `<h1>...</h1>...<p>...`; while style was defined using attributes or style-related elements: `<font size="+1">...</font>`. This lead to a tight contract between those two concerns: a change in the style affected all pages and adding new content often broke the style otherwise. The document-oriented use of XSLT is one possibility to overcome these problemes: All concerns are isolated which increases maintainability and reuseablity. The source XML file contains the content while the style is stored only in the XSLT stylesheet. During the transformation process the stylesheet is applied to the XML file and the final document can be created. This concept has many benefits because it allows using different stylesheets: one for XSL-FO, a second for HTML, a third for WML or even one for the format of a word prosessor. On the other side the same stylesheets can be applied to many different XML documents and must not be aware about the changing content of the input.

3.2.1 DocBook

DocBook is a schema providing a system for writing structured documents using SGML or XML. It is maintained by the *DocBook Technical Comittee* of the *Organization for the Advancement of Structured Information Standards* (OASIS). DocBook is in particular dedicated to books and other documentation about computer hard- and software but it is not limited to this domain [EN02]. DocBook 4.3 is the current version of DocBook. It is published as an DTD for XML and a DTD for SGML. Additionally, there is an unofficial W3C XML Schema and a Relax NG version of DocBook available [OAS04].

DocBook is a very complex schema with more than 400 elements allowing to define meta data as well as structured content. Listing 3.5 shows a very simple DocBook example that contains a book. In the `<bookinof>` element some metadata about the book and the author is provided. A chapter begins with a title, a section and a paragraph. Many elements like a paragraph can contain mixed content thus text intermingled with elements like `<emphasis>`, `code` or `<quote>` that allows to structure the text fine-grained.

Listing 3.5: A simple DocBook file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD␣DocBook␣XML␣V4.3//EN"
               "http://www.oasis-open.org/docbook/xml/4.3/
                  docbookx.dtd">
<book>
  <bookinfo>
    <title>Book Title</title>
    <author>
      <firstname>Peter</firstname>
      <surname>Gerstbach</surname>
    </author>
  </bookinfo>
  <chapter>
    <title>Chapter Title</title>
    <sect1>
      <title>Section Title</title>
      <para>Some <emphasis>important</emphasis> paragraph.</para
        >
    </sect1>
  </chapter>
</book>
```

Since DocBook is especially targeted at writers of technical documents about computers there are many elements for this domain. They range from programming language terms (`classname`, `methodname`, `parameter`) to terms describing graphical user interfaces (`guibutton`, `guilabel`, `guimenu`). Of course there are also elements suitable for any domain to define lists, citations, footnotes or hypertext links.

DocBook XSL Stylesheets Norman Walsh, who takes the chair of the DocBook Technical Committee, developed a set of XSL stylesheets that generate PDF/print documents or Web/HTML content out of DocBook compliant documents [Wal05]. Beside the XSL stylesheets there are other publishing tools for DocBook like DSSSL stylesheets or perl scripts. Beyond PDF and HTML other supported target formats are Microsoft HTML Help, UNIX man pages, TeX/LaTeX and OpenOffice.

The current release (1.68.1) contains stylesheets to generate XSL-FO documents. They may be used to generate PDF documents in a second transformation step. Since they

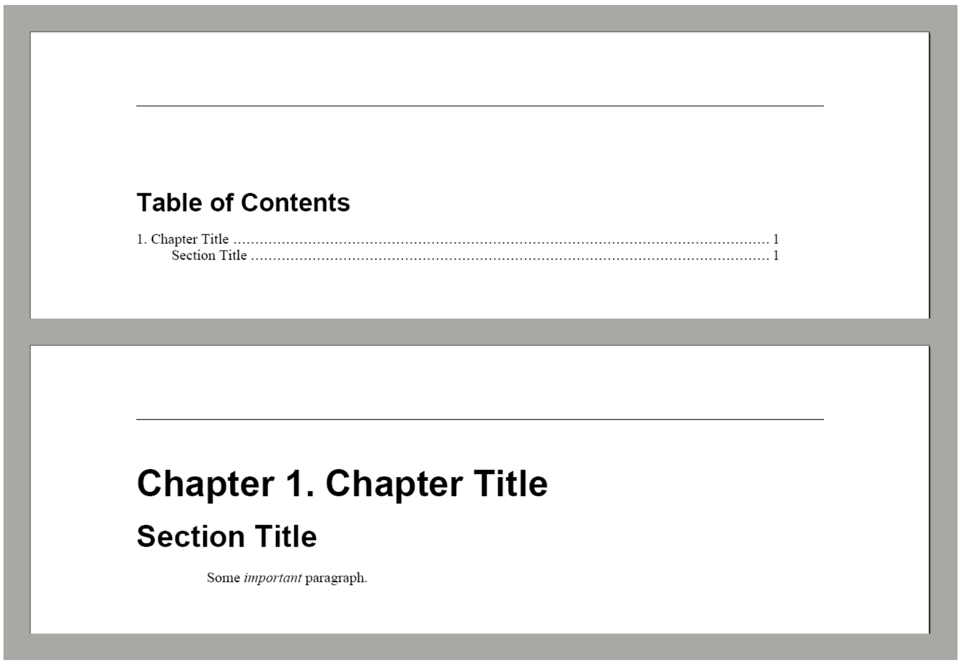


Table of Contents

1. Chapter Title 1
    Section Title 1

Chapter 1. Chapter Title

Section Title

Some *important* paragraph.

Figure 3.9: PDF generated by DocBook-Stylesheets

must be able to handle a huge amount of elements defined in DocBook the transformation process of a simple document is even complex. However, one gets satisfied with an appealing PDF document, containing a title page, a table of contents with hyperlinks and content pages with page numbers (see figure 3.9). The real power of DocBook becomes visible, by transforming one single input document into multiple output documents. This simplifies publishing on different media in an elegant way.

3.2.2 TEI

The acronym TEI stands for *Text Encoding Initiative* and is a consortium which maintains and develops a standard for the representation of literary and linguistic texts in digital form. The initiative was launched in 1987 and enhanced to an independent membership consortium, hosted by academic institutions in the US and in Europe.

The TEI's Guidelines for *Electronic Text Encoding and Interchange* were first published in 1994, followed by new editions in 1999 and 2002. Currently, they define some 400 textual components which can be expressed using SGML or XML markup. The TEI scheme is available as a modular DTD that may be customized for particular research or production environments. A very popular subset is *TEI Lite*, a small part of the whole TEI system, selected to include the most commonly used elements. The next major release of TEI will be version P5, a complete revision of the original with a number of fundamental changes.

A TEI-conformant text contains a header and text. The header may store the full bibliographic description of an electronic file, the relationship between an electronic text and its source, non-bibliographic aspects and the revision history for a file. The text can have an optional front or back matter. Among this is the body of the text located, grouped in the case of a composite text.

A prose text sometimes contains a series of paragraphs (marked up using the `<p>` element) or it may be grouped using chapters, sections, etc. (element `<div>`). Some other tags provide the encoding of structure and semantics of verse and performance texts (speakers, stage directions, etc.). In addition TEI supports many ongoing concepts like page and line numbers, highlighting of text passages, notes, cross references, lists and tables.

Similar to DocBook, TEI offers as well XSLT stylesheets to transform TEI XML files[12]. Supported target formats are (X)HTML, Latex and XSL Formatting Objects although the stylesheets concentrate on the elements found in TEI Lite. With the help of stylesheets it is also possible to import and export TEI documents to OpenOffice and Microsoft Word[13].

3.2.3 OpenDocument

The OpenDocument format, short for the OASIS Open Document Format for Office Applications, is a XML-based file format specification for office applications developed by the *Organization for the Advancement of Structured Information Standards* (OASIS). It supports documents containing text, spreadsheets, charts, and graphical documents, and is intended to provide an alternative to proprietary document formats. OpenDocument has its origins in the OpenOffice.org XML file format initially developed by Sun Microsystems. In December 2002 the Technical Committee for OpenDocument was founded, followed by a first draft in March 2003. OpenDocument was approved as an OASIS Standard in May 2005 [OAS05]. The OpenDocument specification is available for free download and use.

A number of office suite applications currently support OpenDocument including the most prominent ones: OpenOffice and KOffice. Up to now, Microsoft has never stated officially to support OpenDocument in its Office suite. However, there already exists some conversion tools used as plugins to read the OpenDocument format in Microsoft Office. Since OpenDocument aims at guaranteeing long-term access to office documents

[12]TEI XSLT stylesheets: `http://tei.sourceforge.net/`
[13]TEI-Specific tools: `http://www.tei-c.org/Software/`

without legal or technical barriers, it has quickly become an important topic for governments. For example, the US State Massachusetts endorses the OpenDocument format for its public records and the European Union proposes the format as an international ISO/IEC standard.

Listing 3.6: OpenDocument Example

```
 1 <office:document-content>
 2   <office:automatic-styles>
 3     <style:style style:name="T1" style:family="text">
 4       <style:text-properties fo:font-style="italic"/>
 5     </style:style>
 6   </office:automatic-styles>
 7   <office:body>
 8     <office:text>
 9       <text:p text:style-name="Standard">An <text:span
            text:style-name="T1">important</text:span> paragraph.</
            text:p>
10     </office:text>
11   </office:body>
12 </office:document-content>
```

Listing 3.6 shows how the paragraph "An *important* paragraph." is represented in OpenDocument. The text of the paragraph is placed in the `<text:p>` element at line 9, where the italic word is wrapped within the `<text:span>` element having a `style-name` property that references a style defined at line 3. In OpenDocument, the XML code from the listing above is packed with the name content.xml in an ZIP archive with a structure known from JAR (Java Archive) packages. The archive may contain other files like meta-data, styles, and images.

3.2.4 WordprocessingML

WordprocessingML (also known as WordML) is Microsoft's XML schema introduced with the release of Microsoft Office Word 2003. It is a lossless format containing all information Word needs to re-open a document. It contains the text, formatting, styles, metadata, images, etc. that is also available in the traditional *.doc* format. The new XML-based format has several advantages: it allows the creation of Microsoft Word documents with standard XML tools; it simplifies retrieval of portions of text, and the conversion between Word and other formats is much simpler. Many XSLT stylesheets for transforming WordprocessingML can be found on the internet proving the fact.

However, the WordprocessingML schema is far away from being elegant. It was not designed from scratch for the purpose of creating documents in XML markup. Instead, it is an unveiling of the internal structures that have been existing in Microsoft Word for years [LLR04]. For example, WordprocessingML does not make use of *mixed content* that is common in the world of markup. Next, properties are set using empty sub-elements instead of attributes; and the text embedded hierarchically using headings is flattened into a sequence of paragraphs. Listing 3.7 shows how the paragraph "An *important* paragraph." is represented in WordprocessingML. The empty element `<w:i/>` indicates how *italic* text is defined.

Listing 3.7: WordprocessingML Example

```
<w:p>
  <w:r>
    <w:t>An </w:t>
  </w:r>
  <w:r>
    <w:rPr>
      <w:i/>
    </w:rPr>
    <w:t>important</w:t>
  </w:r>
  <w:r>
    <w:t> paragraph.</w:t>
  </w:r>
</w:p>
```

Although WordprocessingML's differs in terms of purpose and elegance, there are existing some transformation approaches for it. For example there are XSLT stylesheets available tranforming FO to WordML and the other way round. Microsoft offers a WordML-to-HTML stylesheet and the *Word XML Software Developer Kit* contains a tool[14] that turns Microsoft Word into a lightweight stylesheet editor: given an XML file that becomes formatted using Microsoft word, this tool generates an XSLT stylesheet from it that will apply the formattings on similar XML files. The result document is a WordprocessingML file ready to be opened in Microsoft Word.

[14]XSLT Inference Tool: https://partner.microsoft.com/global/competency/iwsolutions/40021644

3.2.5 DITA

The Darwin Information Typing Architecture (DITA) is an XML-based architecture for authoring, producing, and delivering technical information. The architecture and the DTD was designed by a cross-company workgroup available at IBM's developerWorks Web site beginning in 1999. After significant updates it outgrowed its prototype stage. In June 2005 DITA version 1.0 was approved as an OASIS Standard.

DITA is an architecture for creating topic-oriented, information-typed content ready to be reused in a variety of ways. As such it differs significantly from DocBook although they play in the same domain. While DocBook was designed for single technical narratives, DITA focuses on discrete technical topics collected into information sets:

- A *topic* is a chunk of information about a subject. A DITA-topic is a container for a body and any number of nesting topics. In detail, it consists of a title, a body, an optional short description, a prolog and metadata. Listing 3.8 contains a sample of a topic.
- The *information type* describes the content of a topic. DITA has three types of this: a generic topic, tasks, and reference topics. Additional information types may be derived from these three basic types using specialization shown in figure 3.10. Different information types usually support different kind of content.

Listing 3.8: DITA topic sample [DPH01]

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="ss/dit2htm.xsl" type="text/xsl"?>
<topic id="sample1">
  <title>Title for the topic</title>
  <body>
    <p>A topic may contain nearly any combination of text
       elements, including lists...</p>
    <ul>
      <li>List item.</li>
      <li>List item.</li>
    </ul>
    <p>definition lists...</p>
    <dl>
      <dlentry>
        <dt>Term</dt>
        <dd>definition</dd>
      </dlentry>
    </dl>
    <p>and so on.</p>
  </body>
  <concept id="minicncpt">
    <title>This is a concept in the "include␣zone"</title>
    <conbody>
      <p>Roses are red.</p>
    </conbody>
  </concept>
</topic>
```

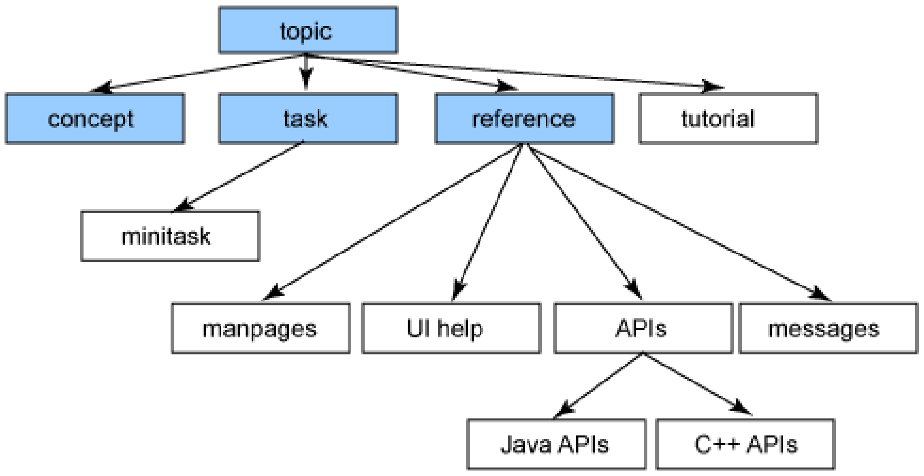


Figure 3.10: DITA Information Types
Source: [Pri01]

DITA provides both, a DTD and a XML Schema representation of the architecture. An implementation of the specification is available: the DITA Open Toolkit[15] transforms DITA content (for example topics) into deliverable formats. Therefore it uses XSLT stylesheets to convert topics into XHTML or XSL-FO. A set of topics may be turned into HTML Help, Java Help, and Eclipse Help. The invocation of the publishing process may be executed by an Ant script.

The transformation of a set of documents from one XML type to another using DITA is discussed in [Les01]. The author uses an automated process to transform documentation of an Apache project (Apache XML) to DITA XML. A Servlet is then used

[15]DITA Open Toolkit: http://dita-ot.sourceforge.net/

to dynamically transform DITA XML to HTML and a XSL-FO formatter is used to generate documentation in PDF format.

4 Implementing an XML-based Document Management System

Up to now this thesis has discussed systems and technologies for reporting and XML document processing. In chapter 2 some tools used for reporting business are discussed. Among these tools some are based on stylesheet generation and processing, working directly with XML input files. Other tools from the enterprise reporting and application integration domain provide similar features but are often not specialized in XML processing. Chapter 3 describes document processing with XML. It points out the advantages that can be yielded if a XML-centric approach of generating documents is selected. Then some XML technologies are discussed, ranging from low-level APIs like SAX and DOM to high-level languages such as XSLT.

In the following the implementation of a document management system is described that is able to generate reports using enterprise data sources. During the development of the application, solutions introduced in the former chapters of this thesis has been applied. Hence, this chapter provides an example how a reporting system may be designed and developed that uses different XML technologies to generate reports.

Roughly speaking, the application consists of a *Template Designer* and a *Document Engine* (refer to the following section 4.1 for an introduction to InStruct). The application uses XSLT to transform an input XML document to the desired output format. For this reason, the Template Designer is comparable to the stylesheet designers introduced in chapter 2.1. With the assistance of the designer it is possible to create a template (in fact an XSLT stylesheet) that transforms an XML document to a well-designed PDF document. However, the application supports as well other data sources that make it comparable to enterprise reporting systems from chapter 2.2 and EAI systems from chapter 2.3.

The second component, the Document Engine, may be used to mass-produces documents. As already stated, the document generation process uses various XML document processing technologies introduced in chapter 3.1, for example XSLT as the transformation language, XSL-FO to describe the desired document, and SAX to serialize XML in a fast and memory-efficient way. During the design phase of the application some document processing systems and formats (see Chapter 3.2) like DocBook, OpenDocument and DITA has been reviewed and checked for feasibility.

On the following pages the implementation of the InStruct-application is described in deep. The discussed solutions and technologies from the former chapters are picked up and referred accordingly. The focus lies especially on InStruct's *document generation* component developed by the author. The other parts of the system like the Template Designer were mainly contributed by other team members and are covered only in brief.

4.1 InStruct Overview

InStruct is a document management system fully implemented in Java that facilitates the creation and management of structured documents. The project has been started in 2004 at GFT Technologies Office in Vienna. Beyond standard DMS-functionality such as versioning, search and retrieval and user management it combines DMS with EAI system, provides workflow support and offers a visual editor to create structured documents enriched with data and formatting.

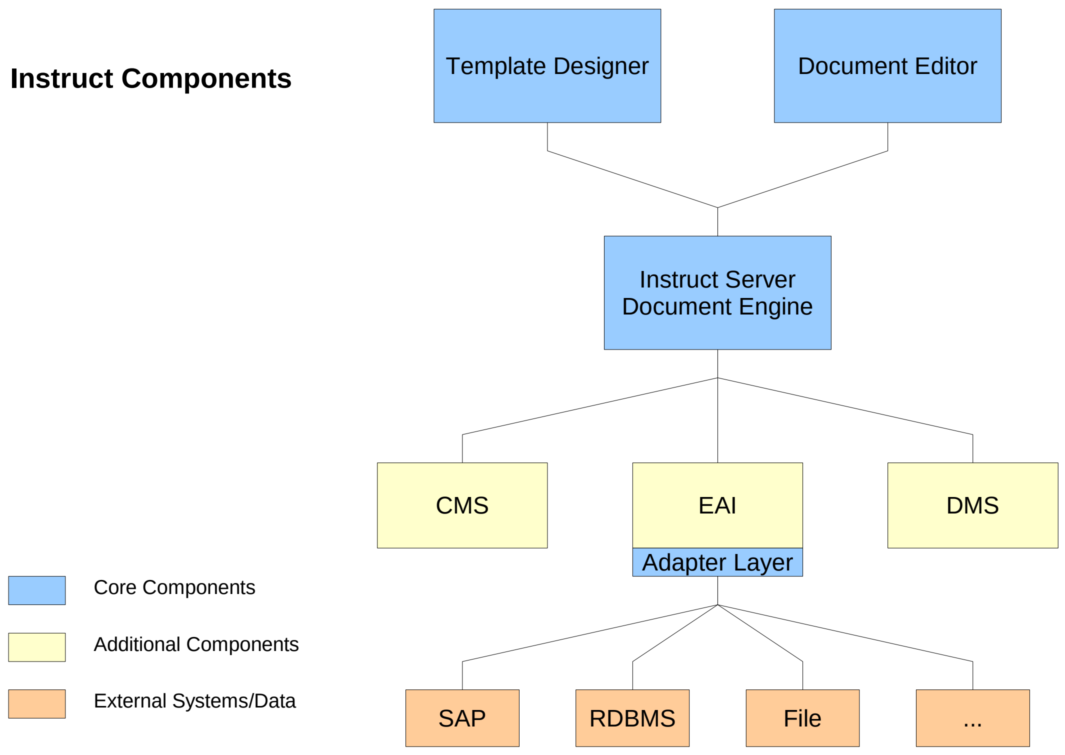


Figure 4.1: Instruct Components

Instruct consists of the following core components (see figure 4.1): *Template Designer*, *Document Editor*, *Document Engine*, and an *Adapter Layer*. Apart from these components there are several essential additional components which make InStruct an enterprise solution with a multitude of functionalities. These additional components are a *Document repository and management system* (DMS), a *Content repository and management system* (CMS), and a *Workflow engine* (connected using EAI). The core components have been built by the project team, whereas for the additional components existing software systems have been chosen.

The **Template Designer** is a graphical user interface helping to build the document template in a WYSIWYG-like way. A user can create a document template, set the layout and define data mappings. The intended user of the Template Designer is an advanced end-user, knowing the data sources and their relations and being able to create a well structured document template. This template will be used later to generate a document. In the **Document Editor** a user works with such a document: an instance of a template. The user can add data to and change documents. In special cases the adaptation of layout data is permitted. To use the Document no in-deep knowledge of the consuming data is needed.

The **Document Engine** serves several purposes:

- It generates the target documents based on the defined document template and the corresponding dynamic data.
- It manages the communication to additional components such as the CMS, DMS and EAI.
- It is the central hub for communication to all clients and manages the complete set of necessary metadata.

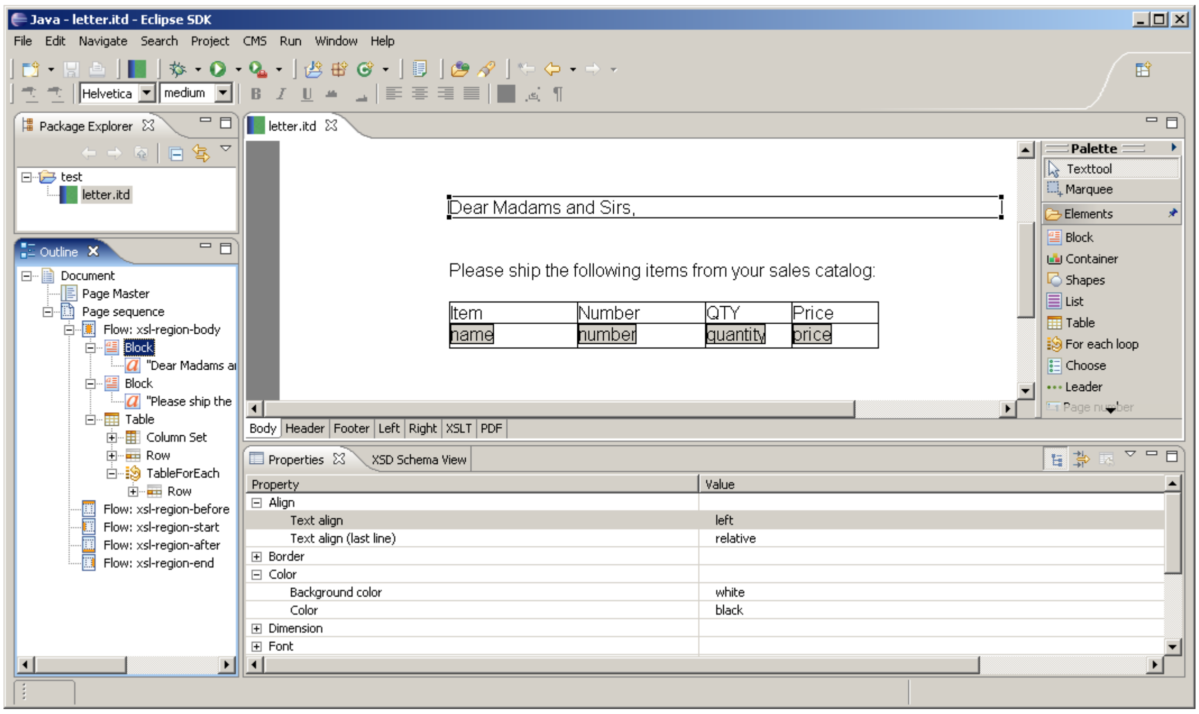
From within the Designer it is possible to connect to a DMS (for example using WebDAV), browse and search in it and display properties. Document templates and documents may be stored to and retrieved from the system. Apart from DMS InStruct also supports CMS using the Java Content Repository API (JCR). This allows the storing and retrieving of parts of content (for example terms and conditions) with or without formattings. Instruct EAI features are based on the Java Connection Architecture (JCA). They allow to retrieve data from external systems, for example XML schemes, mapping information and example or real data sets. All three types of systems may also be accessed from within the Document Engine that executes the document generation process.

4.2 The Template Designer

The Template Designer is a Java based stand-alone application. It was implemented on the basis of the Eclipse Rich Client Platform. This platform provides a minimal set of plug-ins to build a Java-based rich client application that can compete with native applications on any platform. It contains the Eclipse Runtime, the Standard Widget Toolkit, JFace and the Workbench. The WYSIWYG-like editor of the Template Designer uses the Graphical Editing Framework (GEF) that allows developers to create a rich graphical editor from an existing application model. It employs an MVC (model-view-controller) architecture which enables changes to be applied to the model from the view. The design of InStruct's model is described in section 4.6.1.

Figure 4.2 shows a screenshot of the Template Designer. From the pallet on the right-hand-side model elements like text blocks, lists, tables or images may be dragged onto the editor view in the center of the window. The template in the screenshot represents a letter for a purchase order. It has a table with cells containing text boxes grayed out. These boxes reference data of XML files using XPath expressions. A template may be associated with multiple XML Schema files describing the structure of XML files. These XML files store the information that gets included when the final document is generated from the template. The lower part of the window shows the properties of the currently selected item in the editor. This property view allows to set formattings like text alignment, colors, fonts, etc. The outline view on the left-hand-side displays the structure (*outline*) of a template that is currently open in the editor. The use of a tree structure provides a better view of the template compared to the 2-dimensional editor. For example, it indicates the structure of the second row of the table, that is wrapped inside a *ForEach* model element. This element ensures that the final document contains a row for each purchase order item in an underlying XML file.

To insert data from XML documents first an XML Schema file must be assigned to the template for each XML file to be used. Then various model elements representing XSLT instructions may be included in the template. The *ForEach* model element representing the XSLT instruction `for-each` has been already described, other model elements are *ValueOf* (the grayed out boxes) and *Choose*. The XPath expressions used by these elements may either be defined by hand or automatically by using drag and drop. Therefore each assigned XML Schema file is outlined as a tree of items in the *XSD Schema View*. From there an item can be dropped onto the editor resulting in the



Dear Madams and Sirs,

Please ship the following items from your sales catalog:

Item	Number	QTY	Price
name	number	quantity	price

Property	Value
Align	
Text align	left
Text align (last line)	relative
Border	
Color	
Background color	white
Color	black
Dimension	
Font	

Figure 4.2: Screenshot of Instruct Template Designer

appropriate XPath expression for this item to be generated. Afterward such expressions can be edited using an XPath Editor. It provides a list of possible XPath functions and operators and performs a syntax check of the expression. During document generation the XPath expressions are evaluated by resolving the data from XML documents.

4.3 The Document Engine

The transformation process, from the stylesheet to the final document, might be either executed on the client using the Template Designer or on the server using the Document Engine. The Designer permits a quick preview of the template at design time, the Engine is used in production to mass-produce documents. It provides the following features:

- **Define Workflows:** The Document Engine allows to define a workflow for a given job. This enables the system to automate for example standardized letters and reports including triggers, data capture and final output generation. Scenarios with user interaction (notifications, activation) are as well possible.

- **Data Capture:** Data can either be captured actively by the Document Engine (pull mode) or submitted by an external application (push mode), e.g. an EAI system.

- **Document Generation:** This covers the full document generation process from the stylesheet to the final document including the integration of data.
- **Syndication:** It includes the generation of the media-specific documents and the publishing in various formats (print, e-mails, server uploads, etc.).

The Document Engine is written in Java. For connectivity the open source enterprise service bus (ESB) messaging framework *Mule*[1] is used. Supported interfaces include Web Services, JMS, HTTP, SMTP and many others. The functionality of the Document Engine can thus integrated into other systems using one of these interfaces. However, the document engine can be installed in several ways:

- As a stand-alone application
- As a stand-alone application with an application server providing its functionality as a web service
- Using a message driven approach (JMS listener with attached message driven beans)
- In a service oriented architecture where the single services can be embedded in any J2EE compliant application server

Purchase Order Example. In the appendix the InStruct system is illustrated using a simple purchase order example (section A.4 on page 95).

4.4 The Document Generation Process

InStruct's *document generation process* is the main part of this implementation chapter. The process is based on XSLT and is explained in detail in the following paragraph and in figure 4.3.

As already presented, the Template Designer is a visual tool facilitating the creation of document templates. The user can drag new elements from the palette and drop them onto the editor view. These new elements may be static elements, like text boxes and images, or dynamic elements that represent loops and conditions and define the mapping to data sources. All changes on these elements are stored in the *document model.* From

[1]Mule: http://mule.codehaus.org/

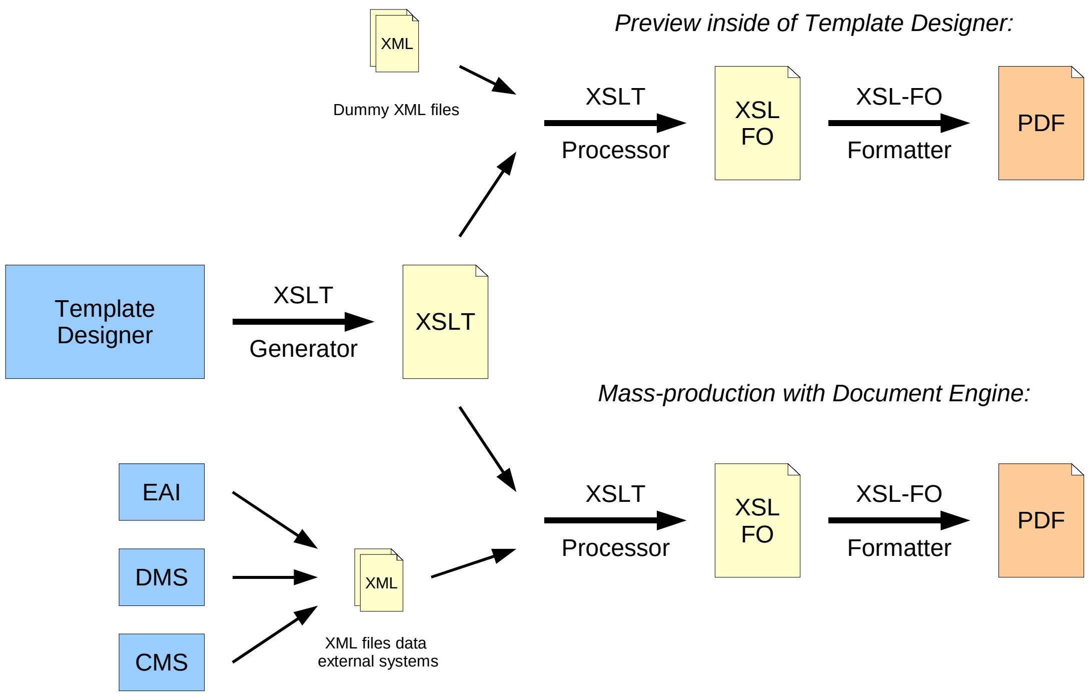


Figure 4.3: Document Generation Process

this document model an XSLT file can be generated by using the *XSL Generator*. The generated XSLT stylesheet is going to be used internally by the preview component of the designer and externally by the document engine. With the preview component the user checks how the generated document looks like. In this case an *XSL-FO formatter* takes the stylesheet and applies it to a sample XML file. Inside, the formatter calls an *XSLT processor* to transform the XML file to an XSL-FO file; this will be rendered to the output format—usually Adobe's *Portable Document Format* (PDF). The PDF will then be presented to the user. If the user is satisfied with the results the stylesheet will be exported to the document engine. The document engine performs the same steps as the preview component, with the only difference that the real data sources are queried instead of using a sample XML file. These transformation steps on the document engine can be invoked manually, by using a Web Service, messaging or other interfaces. Such mass produced PDFs can be printed, sent by e-mail, or archived afterwards.

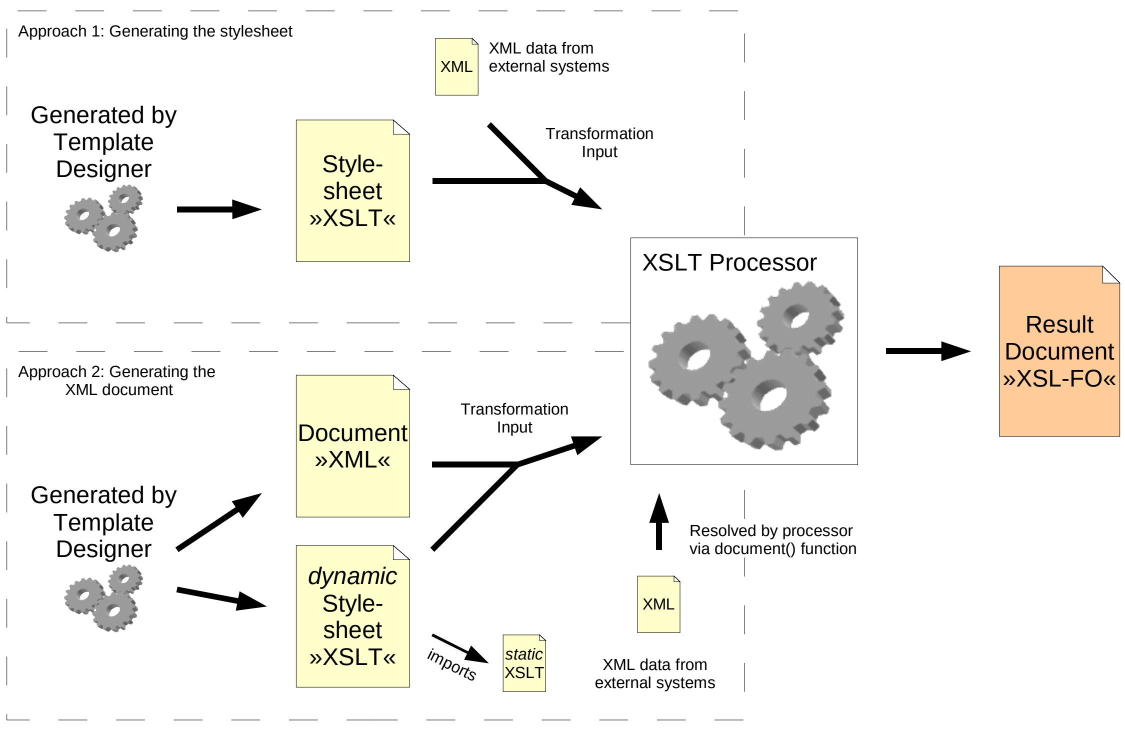
The implementation of the generator and the requirements that influenced the design of components dependent on the generation process are covered in-deep in section 4.6.

4.5 Generator Prototypes

The aim of the document generation process is to transform the input (which may consist of one or more XML files) into the output (usually PDF). At first two prototypes with different design-aproaches of this process have been developed to ensure feasibility. In the following sections each approach is discussed in detail and at the end a conclusion is drawn.

The main point is *where* to store *what* information: There is "static" information (for example a paragraph of text), there are "dynamic" elements (like loops and conditions), and there are elements including external data. All these types of information may be either stored in the stylesheet or in the source XML file. The two approaches differ in this matter. In the following each approach is examined focusing on how it incorporates the concepts of separation of concerns (cp. section 3.2).

4.5.1 Generating the Stylesheet



Approach 1: Generating the stylesheet
XML
XML data from external systems
Generated by Template Designer
Style-sheet »XSLT«
Transformation Input
XSLT Processor
Result Document »XSL-FO«
Approach 2: Generating the XML document
Document »XML«
Transformation Input
Generated by Template Designer
dynamic Style-sheet »XSLT«
Resolved by processor via document() function
XML
imports
static XSLT
XML data from external systems

Figure 4.4: Two Generator Prototypes with different Design Approaches

On the upper part of figure 4.4 the first approach is illustrated. The XSLT stylesheet is generated by the template designer according to the user's definitions made in the graphical user interface. It contains static elements like simple paragraphs and dynamic elements referring to XML data from external systems. With these two parts the transformation is executed: The stylesheet is applied to the XML file resulting in an FO document. If the processor encounters static elements it copies it to the output; by encounting dynamic elements the actions are performed and, for example, data from the external XML is queried.

This approach seems to be straightforward and easy but under the hood the structure of the generated stylesheet is not as perfect as it seems. In a typical document-oriented stylesheet the use of XSLT realizes many benefits. However, in this case a data-oriented XML is transformed to a document-oriented FO and some of these benefits disappear.

To investigate the structure of the stylesheet, it helps to take a closer look at an example. Please refer to appendix A.1 for a short introduction to the addressbok-example. Based on this XML file the following sentence could be the head of a letter designed by the template designer:

Dear Mr Green,

or, writing to a woman

Dear Ms White,

This very simple sentence already contains the three major concern areas of a typical document. The following list shows how the information is distributed among stylesheet and XML file.

- **Content**: Fixed content must be stored in the stylesheet because it is common to all documents and the information can not be found in the XML data file. In the letter example above the fixed content is the word *Dear* and the text of the letter. Both is placed inside a `fo:block` element.

- **Logic**: Logic must be stored in the stylesheet as well. In the example above the system has to decide about using *Mr* or *Ms* depending on if the letter will be written to a man or a woman. This can be done with the XSLT instruction`<xsl:if>` that reads the `<gender>` element from the input XML document. The surname of the person is inserted using an `<xsl:value-of>` instruction.

- **Style**: The salutation should be formatted in *Helvetica 12pt*; this style information is stored in the stylesheet using formatting properties of the `<fo:block>` element.

Listing 4.1: XSLT for the sample letter, Approach I

```
      <xsl:for-each select="addressbook/person">
        <fo:block break-before="page" font-family="Helvetica
           " font-size="12pt">
          <xsl:text>Dear </xsl:text>
          <xsl:if test="gender='male'">
            <xsl:text>Mr </xsl:text>
          </xsl:if>
          <xsl:if test="gender='female'">
            <xsl:text>Ms </xsl:text>
          </xsl:if>
          <xsl:value-of select="surname"/>
          <xsl:text>,</xsl:text>
        </fo:block>
        <fo:block font-family="Helvetica" font-size="12pt">
           Here follows the text of the letter.</fo:block>
      </xsl:for-each>
```

The list above shows that all information is stored in the stylesheet. From view of separation this can be considered as a disadvantage of this approach. Additionally, the stylesheet uses the pull programming style and contains only one big template where all XSLT instructions are embedded. However, the XSLT is intuitive and easy to write because it is a fairly straightforward approach. Leading graphical XSLT stylesheet designers are also using this approach.

4.5.2 Generating the XML Document

The second prototype implements another design approach. It is illustrated in the lower part of figure 4.4. Again the XML input file is coming up from an external system. A second XML file contains the structure and the content of the document, hence, called the *document-XML file*. Listing 4.2 shows such a document-XML for the letter example.

It consists of two parts: the `<styles>` element contains all styles defined for this specific document; the `<content>` element holds the actual content plus the logic elements. A style has a unique name, some properties (the same as defined in FO), and may inherit properties from another style. The content may consist of elements representing paragraphs, lists, tables, images and other block- and inline-level elements. Also there are logic elements like `<for-each>` and `<value-of>` standing for the corresponding XSL elements.

Listing 4.2: Document-XML for the sample letter, Approach II

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <styles>
    <style name="standard">
      <property name="font-family" value="Helvetica"/>
      <property name="font-size" value="12pt"/>
    </style>
    <style name="break" inherits="standard">
      <property name="break-before" value="page"/>
    </style>
  </styles>
  <content>
    <for-each id="person">
      <p style="break">Dear <value id="salutation"/>
        <value id="name"/>,</p>
      <p style="standard">Here follows the text of the letter.</
         p>
    </for-each>
  </content>
</document>
```

After saving the template the generator creates both, the XSLT file and the document-XML file; together they are holding all information needed for the transformation. During the transformation the XSLT processor reads the document-XML and applies the stylesheet to it. The stylesheet consists of two parts: a static part which must not be generated and a dynamic part which is generated. The static part is referenced from the generated stylesheet via an `import` instruction.

The static part contains the instructions that convert the nodes of the document into FO nodes. For each element in the source file (e.g. `<p>`) a template with a corresponding `match` attribute (`match="p"`) exists. For example a `<p>` element is transformed to a `<fo:block>` element and an `<em>` element to an `<fo:inline>` with appropriate formattings added. This fixed stylesheet containing all these template rules can be used for all documents.

The dynamic part of the stylesheet comprises all instructions that handle logic. E.g., the document-XML contains a `<value>` element transformed to *Mr* respectively *Ms.* Listing 4.3 lists the template responsible for the salutation. The `data` parameter is needed because nodes from two different XML files must be read. The current node comes from the document-XML which contains the `<value>` element. The nodes from the data-XML are first created using the `document()` function. After this they are passed to the template using the `data` parameter. The parameter now contains the `<gender>` element used for deciding which salutation should be used.

Listing 4.3: XSLT for the sample letter, Approach II

```
<xsl:template match="value[@id='salutation']">
  <xsl:param name="data" select="''"/>
  <xsl:for-each select="$data">
    <xsl:if test="gender='male'">
      <xsl:text>Mr </xsl:text>
    </xsl:if>
    <xsl:if test="gender='female'">
      <xsl:text>Ms </xsl:text>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
```

The following list sums up how the main concerns are distributed among the files:

- **Content**: Fixed content is stored in the document-XML using a well defined schema. In the letter example above the fixed content is the word *Dear* and the text of the letter. Both texts are placed inside the `p` element.

- **Logic**: Logic must be stored in the dynamic part of the stylesheet because XSLT does not allow to build full XPath expressions with variables. In the letter example the salutation and the name of the person is the logic part.

- **Style**: The style information is stored in the document-XML namely in the `styles` element at the beginning of the document. Each `style` element then has some properties which represents the formatting properties to be used in the formatting objects.

The second approach better succeeds in separating all three concerns. It also uses a more advanced XSLT programming style. Holding the content in the XML file makes the use of the push model (see 3.1.5.2) possible. Another advantage is about serialization and roundtripping which does not relate directly to XSLT. The model only exists in memory and must be serialized at the end of each editing session. By using approach II it can be serialized into the same XML which will then be input for the XSLT tranformation process. By starting a new session the XML file can be parsed back, so the model is available again in memory.

As a disadvantage may be seen that during the generation process all the information must be splitted into structure which is stored in the XML document, and logic stored in an XSLT stylesheet. The implementation of this splitting is time-consuming and yields no direct benefit. Linking the logic (stored in the stylesheet) and the data it operates on (stored in both document-XML and data-XML) is complicated and inflexible. Next the document-XML file must be valid against an XML Schema. This schema is rather complex because it contains elements for representing both structure and formatting. This would lead to numerous elements converted to FO elements one-to-one. The main loss results from using two input XML files. Although XSLT allows the use of multiple input files using the `document()` function—the handling of such templates is complicated (see listing 4.3).

4.5.3 Conclusion

For design and architecture purposes a clean separation of concerns is desirable. In a scenario like DocBook—where XSLT is just used to do formatting and restructuring—this yields to an increase in maintainability. In the case of a data-to-document transformation, where the XSLT is primarily responsible to perform logic instructions, separation of concerns is hard to implement and does not necessarily increase clarity.

In the case of InStruct the XSLT stylesheet is generated and must not be edited afterwards by humans. In addition, each designed template results in one stylesheet that can be invoked multiple times by the document engine. So there is no imperative to separate content, logic and style. A flexible solution is far more important. For that reasons the first approach was selected, to meet best the requirements.

4.6 Implementation of the Generator

The purpose of the generator is to create an XSL stylesheet out of the document model that was produced by a visual tool. In short the process connects four artifacts:

- The **Document Model** is the in-memory-representation of the template, constructed by the Template Designer. The *XSL Generator* converts this model into the *Stylesheet Model*.
- The **Stylesheet Model** is an in-memory-representation of the XSL stylesheet containing both objects representing elements from the XSLT namespace and the

FO namespace. The *XSL Marshaller* then serializes the Stylesheet Model into the *Stylesheet*.

- In production the XSL **Stylesheet** can be used by the *Document Engine* to mass-produce documents or it is used in the Template Designer to generate a preview of the document. In the end the final artifact is the *PDF Document*.
- The **PDF Document** is generated by an *XSL-formatter* using the generated stylesheet from the former step and the XML files as input.

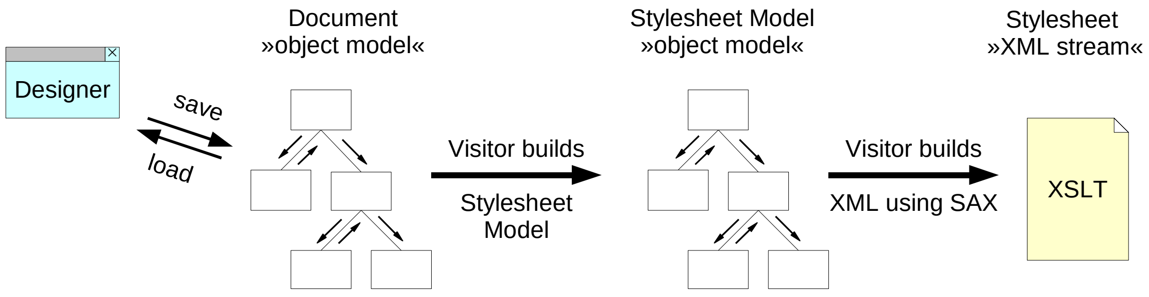


Figure 4.5: Stylesheet Generation

Figure 4.5 shows the two generation steps from the internal *Document Model* to the *XSL stylesheet*. The following sections describe the process and the artifacts in detail.

4.6.1 Document Model

The document model is an important design artifact of InStruct. It represents the template document that can be constructed with the Template Designer. It serves multiple purposes:

- for the GUI it functions as the model of the model-view-controller (MVC) architecture,
- the XSL Generator uses the model as input and creates the XSL stylesheet from it, and
- by saving the application has to make the model persistent to be able to restore it in subsequent sessions.

Those purposes led to some important requirements:

- First, the model has to be independent from the view so that it can satisfy the MVC paradigm. Additional views can only be implemented easily if the model is separated from all views.
- Second, it must contain all information that is needed for the XSL stylesheet generation. Since FO is used in the stylesheet, the model should be similar expressive than FO.
- Third, to support persistence it has to be possible to mark some parts of the model as persistent while others remain transient. To increase robustness the serialized data should also remain parseable even if the underlying model has changed due to updates.

MVC Architecture The *Model-View-Controller* architecture separates the application's data model user interface and control logic into three distinct components. The model is the domain-specific representation of the information on which the application operates [Fow02, p. 56]. One main benefit of the MVC architecture is that the model remains stable (as long as the modeled domains remains the same) whereas the view usually undergoes frequent change. In case of InStruct the modeled domain is XSL and the model's complexity strongly depends on the features the editor should support.

Input for Generator. The second requirement is that the model must contain all information needed for the XSL stylesheet generation. XSL-FO in version 1.0 is already a very complex specification. Although not all XSL-FO features have to be implemented from the very beginning the system design has to be very generic. Thus it should be able to add some feature in a later version without changing the whole system. The most important concepts of FO must be supported by the document model.

This leads to a model comparable to the elements defined in XSL-FO and XSLT: there are block-level objects like `Text`, `List` and `Table`; there are inline-level objects like `Image` and `Inline`; and there are so-called *logic objects* like `LogicForEach`, `LogicIf` and `LogicValueOf` allowing to define containers that are later transformed to the appropriate XSLT instructions.

Another important part of the model is about properties. The FO specification defines a huge amount of *Formatting Properties* that determine the final appearance of the document. The template designer supports most of them. A formatting property is far more complex than a simple list of attributes: there are different types of properties that belong to different groups, the fine-grained inheritance system and the possibility to define compound properties. Instruct's style and property system is outlined more precisely in section 4.6.1.

Persistence. The model should support persistence so that it can be restored in a subsequent session without loss of data. For this persistence three different approaches have been evaluated: *Native Java Serialization*, *XML Data Binding* and Java *Long-Term Persistence for JavaBeans*.

Native Java Serialization is a common serialization technique available in any Java version. It is easy to implement but has some major drawbacks: the biggest problem with native Java serialization is that deserialization fails if the underlying classes have changed. This causes compatibility problems when product updates occur. In addition the serialized data is binary, and hence it is not very well suited for persistence over a long period.

XML Data Binding provides a way to convert objects to XML and back (see as well 4.6.2). Although there are existing many tools for XML Data Binding with Java none of them fulfilled all requirements. One problem with XML data binding is that most of the tools generate Java classes out of XML Schema and not in reverse. In the case of InStruct, another problem had been, that these generated classes can not be changed afterwards. This implies that the classes can be used as data containers only but not to implement business logic.

Java Long-Term Persistence for JavaBeans is a Java Specification Request of the Java Community Process released in May 2002 [Jav02]. Although it is primarily targeted at IDE vendors to provide a standard way to archive JavaBeans in long-term it can be used with all bean-like Java objects. A stable implementation can be found in the Java 2 Standard edition since version 1.4. With the class `XMLEncoder` objects can be made persistent using a standardized XML format. The class `XMLDecoder` may then be used to parse this XML and reconstruct the objects. `BeanInfo` classes are provided to define in detail which attributes and methods should be used by the persistence system. *Java Long-Term Persistence* was chosen for the implementation because it proved to be stable and flexible enough to fulfill all requirements for persisting the document model.

UML Class Diagram

Based on the requirements the document model was designed with the knowledge about formatting objects in mind (left-hand-side of figure 4.6). Base class of almost all classes in the model is the abstract class `ModelElement`. It has an association with the Style class storing formatting properties, offering some methods common to all model elements and implementing the observer pattern to notify listeners about changes of properties. Concrete subclasses of ModelElement are Document (the root object), PageMaster that contains a page layout, PageSequence, and Flow.

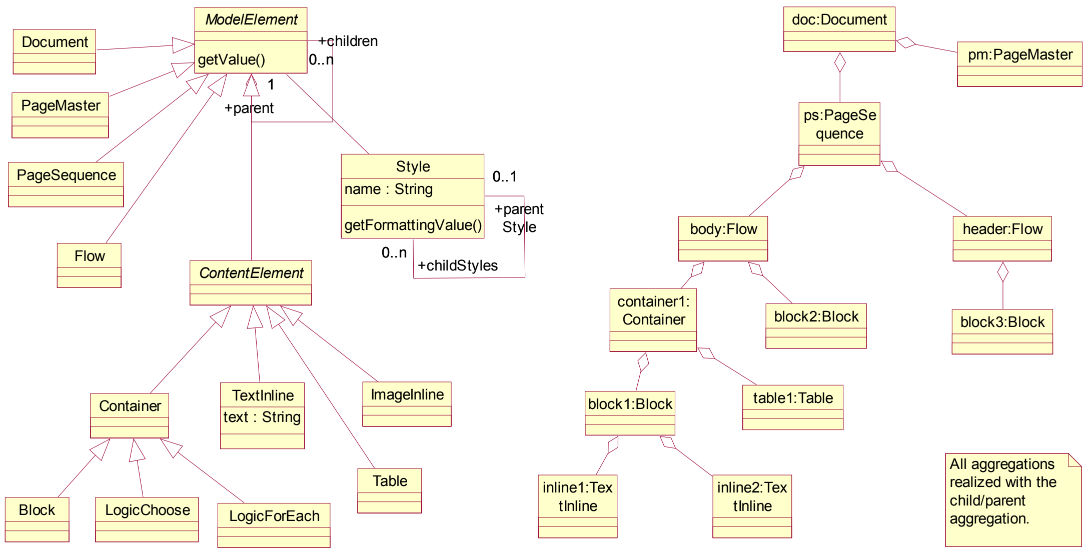


Figure 4.6: UML Class Diagram and Instance Diagram “Document Model”

Another abstract class is ContentElement inherited from ModelElement. It is the base class for all classes that are visible on the user interface. Example of subclasses are Block, TextInline, or Table that represent the formatting objects fo:block, fo:inline, and fo:table.

One of the most notable associations is the parent/children aggregation of the class ModelElement. With this aggregation the hierarchy of instances may be built as shown on the right-hand-side of figure 4.6: doc (class Document) is the root object of the model owning a page sequence and a body. The body has two children, container1 and block2. The former has again two children, block1 and table1 and so on. The aggregation is navigable in both directions, that enables each model element to find its parent.

Styles and Properties

XSL-FO supports a very complex property system (see section 3.1.6.1). The most important parts of this concept are implemented in Instruct's model. Additionally a style concept is implemented allowing to reuse styles for model components. Figure 4.7 shows the parts of the UML class diagram relevant to the style concept.

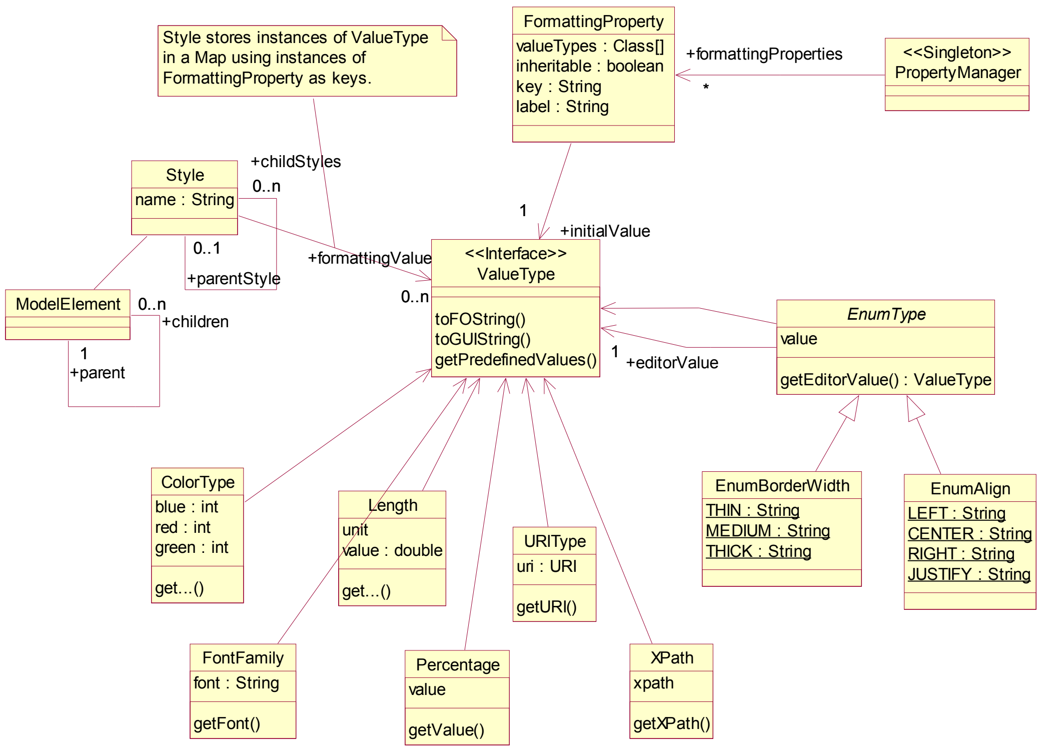


Figure 4.7: UML Class Diagram “Property and Style System”

As already shown in the main UML diagram a `ModelElement` has an association to a `Style`. In this context a style is used to store the information about the appearance of an object. A `Style` may have a parent from which it inherits the appearance information. For example this makes it possible to define a style named *important* that inherits all values from a style named *standard* (with a font *Times 12pt* but colors the text in red.

At system startup the `PropertyManager` (a Singleton) creates for each *Formatting Property* defined in XSL-FO one instance of the class `FormattingProperty` that represents it. This class owns a unique key, an internationalized label and a list of value classes used for this Formatting Property. A value class is a class that implements the `ValueType` interface, for example `FontFamily`, `Length`, or `Percentage`. These classes correspond to the datatypes specified in the XSL-FO specification.

Now turning back to styles: a `Style` stores its appearance information in an associative array. As key it uses the *key* attribute of the instance standing for the formatting property. For the value it uses an instance of a value class. The code snippet in listing 4.4 shows how a style with a font size of 12pt may be created.

Listing 4.4: Creating Styles and Properties

```
Style style = new Style();
ValueType length = new Length("12pt");
style.setFormattingValue("font-size", length);
```

The inheritance hierarchy of a `Style` is fully transparent to a model element. The model element calls the `getFormattingValue()` method of its style and gets a `ValueType` object returned. If this value was specified on the style or one of its parents does not make a difference for the calling model element. Additionally to the inheritance of styles properties can be inherited as well. This behavior is specified in XSL-FO and is implemented by `getValue()` method in the `ModelElement` class.

Figure 4.8 is an activity diagram showing how formatting values are resolved. In a first step the model Element tries to get the value of a specified property from its style. If the style does not return any value the parent model is consulted. If a style is asked for a property it first looks inside its map, if a value for the desired property is found there. If nothing is found the style asks its parent style. If all steps do not return any values the *initial value* for this property (as defined by XSL-FO) is returned.

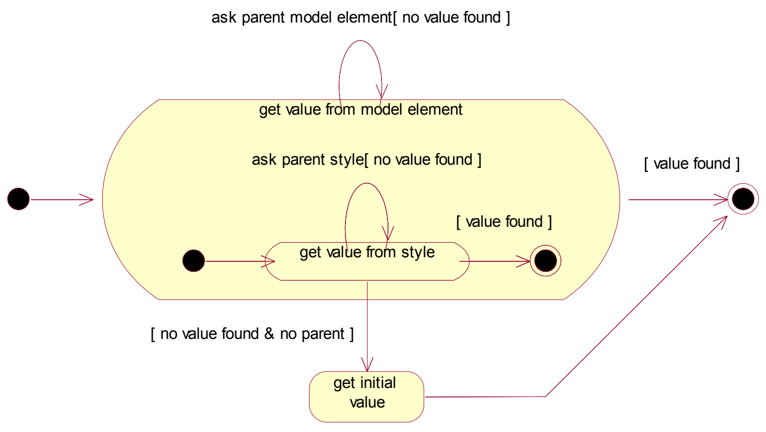

ask parent model element[no value found]
get value from model element
ask parent style[no value found]
[value found]
[value found]
get value from style
[no value found & no parent]
get initial value

Figure 4.8: Resolving Inherited Property Values

4.6.2 Stylesheet Model

The generation process causes the Document Model to be converted to the *Stylesheet Model*. This step creates an in-memory representation of the XSL stylesheet. The

Stylesheet Model is independent from the Document Model and focuses only on XSL and XSLT. Its design is driven by the following requirements:

- **Completeness.** It must be possible to create all elements needed in the final stylesheet. These are elements from the XSLT namespace, XSLT extensions, and all used literal result elements: XSL-FO and SVG.
- **Simple Use.** The objects should be intuitive to use. They should have constructors and getters/setters for their attributes to facilitate programming.
- **Marshalling.** It must be able to marshal the XSLT model to XML in order to produce the XSLT stylesheet.
- **Memory usage** should not be too exhaustive.
- **Incremental Updates.** It should be possible to update only parts of the XSLT model, if a part of the underlying document model is changing. This should increase performance which is an important requirement for the whole generation process.

The following sections describe what has been done finding the best model for XSL and XSLT representation. First some existing open source projects were analyzed if their internal models may be of use for the project. Then some code generators based on XML Data Binding were investigated. After this it became apparent that both ways did not meet the requirements a new model was written.

Evaluation of Existing Models

The open source community provides many XSLT processors that each comes with its own data model. The most known open source processors include Apache's *Xalan* and Michael Kay's *Saxon*. The model of both processors were considered as a basis for the generator's stylesheet model.

Apache Xalan. The source of Xalan-J of the current version 2.6 was inspected. The objects representing the XSLT elements are contained in the package `org.apache.xalan.-templates`. For each XSLT element exists an equivalent object. All the objects have well defined methods. For example the class `ElemForEach` represents the `xsl:for-each` element and has a `setSelect()` method that takes an `XPath` object as input.

Saxon. The Saxon-model (version 8.4) stores its elements in the package `net.sf.-saxon.style`. The model is backed by a complex system of inherited classes and interfaces. The class `XSLForEach` provides a public generic method for setting attributes. The attribute that stores the `select` value is declared private and therefore cannot be accessed from outside. In exchange each class provides a `validate()` method allowing to check if the object is valid to the XSLT specification.

Both XSLT processors, Xalan and Saxon, parse existing XSLT stylesheets and build up some internal model. Nevertheless both do not offer the capability to use their internal model from outside and to serialize it back to XSLT documents. Another problem is the lacking possibility to integrate literal-result elements (in this case XSL-FO elements) easily. That is why these two attempts have been dropped.

Apache FOP. Currently FOP is the only open source XSL-FO formatter. The latest maintenance release is 0.20.5 from mid 2003. It was tested if it possible to use FOP's model as a basis for generating the elements from the XSL-FO namespace. FOP has similar drawbacks as Xalan and Saxon. Especially, it lacks integration and serialization features. Additionally the source code is hardly documented which makes any modifications difficult.

Evaluation of XML Data Binding Tools

Another attempt was to use a Data Binding Tool. *XML Data Binding* provides a way to convert objects both to XML and reverse. Most Data Binding frameworks generate classes and interfaces from an XML Schema or DTD. These classes can then be instantiated. The frameworks allow to *marshal* those objects to XML and to *unmarshal* XML back to objects. The produced XML is valid against the defined schema and usually the objects can be validated against the same schema in-memory [Ger04].

This section demonstrates the attempts using the Data Binding Tools *JAXB*[2], XML-Beans[3] and Castor[4].

Schema from the XSLT 2.0 Draft The current XSLT 2.0 working draft contains a non-normative XML Schema describing the structure of an XSLT stylesheet. Although

[2]JAXB (Java Architecture for XML Binding) implementation from sun, version 1.0.4: `http://java.sun.com/webservices/jwsdp/`

[3]Apache XMLBeans, version 1.0.3: `http://xmlbeans.apache.org/`, originally developed by BEA Systems

[4]The Castor Project, version 0.9.6: `http://www.castor.org/`

it uses many advanced XML Schema constructs it only describes the most important constraints of an XSLT stylesheet.

Unfortunately, all three tools failed to work with the given XML Schema as input. JAXB was able to generate some classes (although with warnings) but terminated in a `NullPointerException` while running a simple marshalling task. With Castor and XMLBeans it was even not possible to generate the classes. Both complained about errors in the XML Schema. Further investigations showed up that XMLBeans' schema validator reports an error in the schema that is actually correct. In this case XMLBeans tripped over a special part of the schema specification allowing a complex type with simple content to be derived by restriction from a complex type with mixed content [Kay05]. Drawing a conclusion, the XML schema published in the working draft can not be used as a basis for the XSLT model. Currently, it seems that it is more a good quality check for schema processors.

RelaxNG schema from Norman Walsh Another attempt was to use the RelaxNG schema for XSLT 2.0 by Norman Walsh [Wal04] written to provide a grammar-checker for the text editor *Emacs*. Sun's JAXB implementation has a experimental support for RelaxNG. First, the RelaxNG compact syntax had to be converted into the XML syntax. After some binding customizations JAXB was able to generate classes based on the schema. Soon it became clear that the generated classes may be very useful to check the grammar of an XSLT 2.0 file in Emacs but not to build an XSLT model. JAXB maps every XSLT element to an interface with a nested object containing the actual data. The resulting object model is far away from being elegant.

The experiences gained using Norman Walsh's RelaxNG schema demonstrates that data binding tools currently often does not produce code that meets high demands. Another general problem with XSLT and the data binding approach is about *literal result elements*. Literal result elements are elements inside the stylesheet which are no XSLT instructions and thus get copied as-is to the result document. Because of literal result elements a stylesheet does not consist of elements from the XSLT namespace only. This hast to be addressed in a schema for XSLT. It also implies a data binding approach being able to handle arbitrary elements in a usable way.

Realization

The two preceding sections illustrate that both existing models from open source projects and XML data binding-approaches did not fulfill the requirements. As a result a new object model has been developed from scratch. Because no complicated features like validation are needed the Stylesheet Model is based on a few simple classes.

Classes representing formatting objects implement the abstract class `Formatting-Object`. It provides a map and accessor-methods that offer storing and reading formatting properties. A descendant of this class is the class `FormattingObjectWithContent` used for all formatting objects that can have (mostly) arbitrary content. It implements the Interface `ObjectWithContent` and its sole method `getContent()` returning a list with all children of an instance. For objects representing elements from the XSLT namespace an analog implementation exists. Both implementations use a special list allocated not until an element is inserted.

A convenient usage of most FO classes are realized with constructors and getters/setters for often used properties. The XSLT classes provide getters/setters for each attribute. Classes representing XSLT elements with mandatory attributes (such as the `test` attribute of the `<xsl:if>` element) do not provide a default constructor but a constructor with parameters for the mandatory attributes. However, beyond this, no further validation is implemented.

4.6.3 Transformation from Document to the Stylesheet Model

As a first step of the generation process the Document Model is transformed to the Stylesheet Model. The result is not already the final serialized stylesheet but only an in-memory representation. The Stylesheet Generator takes the Document Model as input and generates this in-memory representation of the XSL stylesheet: an instance of the Stylesheet Model (see preceding section 4.6.2). Since the Document Model was designed with XSL-FO in mind many document model elements are already similar to XSLT and XSL-FO elements. So they can be converted to their corresponding elements of the Stylesheet Model.

The Stylesheet Generator uses the *Visitor Pattern* [GHJV95, p. 331] to create the stylesheet objects. This behavioral pattern allows *an operation to be performed on the elements of an object structure*. Additionally it *lets you define a new operation without changing the classes of the elements on which it operates*. In the case of InStruct those elements are objects of the Document Model which should remain independent from the Stylesheet Generator. For each object in the Document Model there exists a `visit` method in the `StylesheetVisitor` class accepting a special object from the Document Model. If invoked the method constructs the Stylesheet Model instance. For example, the signature of the visit method for a *block* may look like this:

```
public void visitBlock(Block block)
```

During the transformation process the visitor comes to a `Block` object and calls this method that creates an instance of the `FOBlock` class.

The drawback of the visitor pattern is that each visited objects (elements of the Document Model) and also each new constructed objects (elements from the Stylesheet model) must "know" their parents and children.

An example: a `Block` may have children (e.g. a `TextInline`) and can itself be placed inside a `Container`. The corresponding objects from the XSL model are `FOBlock` containing `FOInline` objects and can itself be placed inside an `FOBlockContainer`. When the `visitBlock` method is invoked a new `FOBlock` object is constructed and must be added as a chil to an already existent `FOBlockContainer`; then the visit method is called iteratively for all the Block's children.

In the method `visitBlock(Block block)` we know the block itself (given as parameter), and of course the newly created `FOBlock` instance; thus we are able to iterate over the block's children. Nevertheless, the parent of the newly created `FOBlock` is missing. It was created in a preceding invocation of another visit method. To be able to get the instance of the needed container the visit method is extended with another parameter that contains a reference to an object used as the parent of the newly created object. Now it follows the extended signature of the visit method for a *block*:

```
public void visitBlock(Block block, Object parent)
```

The parameter `block` is the instance of the `Block` element visited. The parameter `parent` is an object of the Stylesheet Model to which the newly created object is added. When invoked the `visitBlock()` method constructs a new `FOBlock` object that is added to the given parent.

The use of the `parent` parameter makes incremental generation of the Stylesheet possible: in this case only the changed parts have to be re-generated, while all unmodified object remain untouched. The incremental approach improves the performance of the generation process.

The Stylesheet Generator implements the Visitor pattern using a reflection mechanism similar to [Blo00]. The implementation uses reflection to call the appropriate visit method instead of calling the `accept(Visitor)` method as proposed by [GHJV95]. This has the advantage that the elements on which the operation is performed have no visitor-specific methods at all. A disadvantage of this more flexible approach is caused by the use of reflection: renaming a class in the document model breaks the link between this class and the visit method. This also applies to automatic *class name refactoring*.

For each class of the document model the *Reflective Visitor* has a corresponding visit method as already explained: given the class `Block` the visitor implements the method `visitBlock(Block block)`. Additionally there is a new method `visit(Object obj, Object parent)` (see listing 4.5). A call of this method starts the conversion: it uses reflection to gain the class name of the given object `obj`, in this example `Block`. Then it creates a `Method` object for the visitor's method with the name `visitBlock` and invokes it. The `visitBlock` method constructs a `FOBlock` object later serialized to the XSL-FO element `fo:block`.

Listing 4.5: Reflective Visitor's `visit` Method

```java
public void visit(Object o, Object parent) {
    // This strips off the package information giving us just the class name
    String fullClassName = o.getClass().getName();
    String className = fullClassName.substring(fullClassName.lastIndexOf('.') + 1);
    String methodName = "visit" + className;
    // Now we try to invoke the method visit
    try {
        // Get the method visitFoo(Foo foo)
        Method m = getClass().getMethod(methodName, new Class[] { o.getClass(), Object.class });
        // Try to invoke visitFoo(Foo foo)
        m.invoke(this, new Object[] { o, parent });
    } catch (Exception e) {
        // Handle exceptions
    }
}
```

In general, the entry point of the generation is an instance of the `Document` class, the root class of the Document Model. For the document object the root object of the stylesheet (class `XSLStylesheet`) and the root template (class `XSLTemplate`) (with a `match` attribute that has the value "/") are generated. All children of the document object are then converted to their XSL representation iteratively: it has children that represent the FO page-sequences. For each child the `visit` Method is called. The page-sequences have children on its own, that are as well visited. This iterative process is repeated until a leaf-node is found. This assures that the whole tree of the document model is traversed and converted to the Stylesheet Model.

The resulting stylesheet has only one root template. Whenever there is a `LogicFor-Each` object in the document tree an XSL for-each statement is generated. This XSLT programming-style is also known as the *pull*-style. Table 4.1 shows how the objects of the document model are represented in XSLT or XSL-FO respectively.

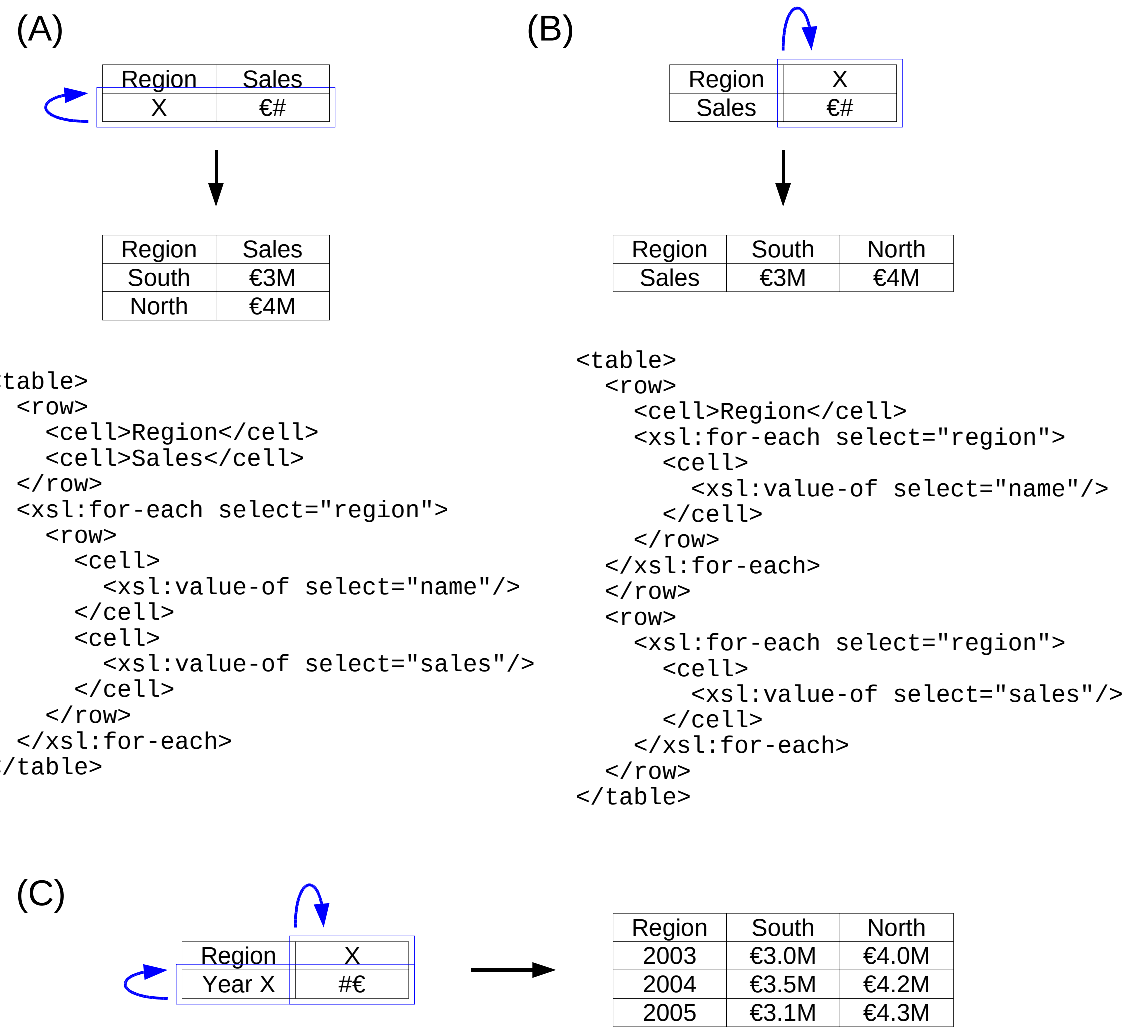
Document model	**XSL-FO/XSLT**
Document	`xsl:stylesheet`, `xsl:template` ("/")
PageMaster*	`fo:layout-master-set`, `fo:simple-page-master`, `fo:page-sequence-master`
PageSequence*	`fo:page-sequence`
Flow*	`fo:flow`, `fo:static-content`
Block, TextInline	`fo:block`, `fo:inline`
Table	`fo:table`
Style, FormattingValue	Attributes (Formatting Properties) of each Formatting Object
LogicForEach	`xsl:for-each`
LogicChoose	`xsl:choose`
LogicValueOf	`xsl:value-of`
...	...

Table 4.1: Mapping between the Document Model and XSL

Mapping Abnormalities: Tables The table above shows that most Document Model elements have obvious representations in XSLT or XSL: a `Flow` is converted to `FOFlow`, a `Block` to `FOBlock` and so on. However some elements must be treated separately. This section tells how tables with a dynamic number of rows and columns respectively are handled.

To understand the problem one needs to know how tables must be defined in XSL-FO. Usually a table defines a number of rows which themselves have a number of cells. The columns evolve implicit from this definition (however it is possible to define rows to be able to specify formatting properties on them). This row-centric structure has a major impact on how loops on tables have to be implemented.

Figure 4.9 shows how three possible ways of surrounding table rows or columns with a for-each instruction. In part *(A)* on the left a table is shown containing a list of regions and their sales: each region is placed in one row. A for-each instruction is used to iterate over all `<region>` elements in an XML file. The source code in the middle left shows how



(A)

Region	Sales
X	€#

Region	Sales
South	€3M
North	€4M

```
<table>
  <row>
    <cell>Region</cell>
    <cell>Sales</cell>
  </row>
  <xsl:for-each select="region">
    <row>
      <cell>
        <xsl:value-of select="name"/>
      </cell>
      <cell>
        <xsl:value-of select="sales"/>
      </cell>
    </row>
  </xsl:for-each>
</table>
```

(B)

Region	X
Sales	€#

Region	South	North
Sales	€3M	€4M

```
<table>
  <row>
    <cell>Region</cell>
    <xsl:for-each select="region">
      <cell>
        <xsl:value-of select="name"/>
      </cell>
    </row>
  </xsl:for-each>
  </row>
  <row>
    <xsl:for-each select="region">
      <cell>
        <xsl:value-of select="sales"/>
      </cell>
    </xsl:for-each>
  </row>
</table>
```

(C)

Region	X
Year X	#€

Region	South	North
2003	€3.0M	€4.0M
2004	€3.5M	€4.2M
2005	€3.1M	€4.3M

Figure 4.9: Tables and For-Each

this can be implemented in XSLT. Note that the names and structure of the `fo:table` are shortened (and thus not correct) for the sake of brevity. It shows a row wrapped by an `<xsl:for-each>` instruction. The row contains cells that output the name of the region and its sales.

In part *B* the same table is given, but with its axes swapped. Now there exists a column for each region and its sales. Due to the row-centric structure of tables the implementation for this table is a bit more complex. Yet there are two for-each instructions: one for the first row containing the names of the regions; another one for the second row that contains the sales.

Of course a combined version of the table with both a dynamic number of rows *and* columns is possible. This can be used to create a table offering the sales of different regions in different years (see part *(C)* in the figure).

4.6.4 Stylesheet Serialization

In the former step the Document Model was transformed to the Stylesheet Model. The *Stylesheet Serialization* now serializes the Stylesheet Model to XML fed as input for the XSLT transformer. This step can be compared to the marshalling step in an XML data binding environment or the serialization of a DOM tree.

For the serialization of the Stylesheet Model a visitor pattern is used again. For performance reasons a SAX-based approach was selected. A DOM-based solution would mean building just another expensive object tree in memory. Instead the Stylesheet Model is serialized using an implementation of a SAX `XMLReader` as proposed in [Sun04, p. 272]. With a SAX `XMLReader` an arbitrary data structure can be converted to XML using XSLT. This concept works as follows: At first, a visitor is written reading the data from the model and generating SAX events, like it would be done by a real parser. Then this SAX "parser" is used to contruct a `SAXSource` for the transformation:

```
SAXVisitorReader reader = new SAXVisitorReader(stylesheet);
Source xslt = new SAXSource(reader, null);
```

This source contains the final XSLT stylesheet. If this stylesheet is needed for a transformation it may be directly used to create a XSLT transformer object to make the transformation. If the stylesheet has to be saved an identity transformation may be used to serialize the source to a file. In this case the source is wired to a result object using a transformer to execute the conversion.

The UML sequence diagram in figure 4.10 points out how the reader and the visitor generate the SAX events. First, the `SAXVisitorReader` constructs a new `SAXVisitor` object and passes the current `ContentHandler`. The `ContentHandler` is provided by the SAX-enabled XSLT processor that performs the transformation. The reader then begins the XML document by calling the `startDocument()` method. The subsequent processing is then passed to the visitor. The Visitor calls recursively all visit methods for the Stylesheet Model's elements. These visit methods call the appropriate methods of the `ContentHandler`, for example `startElement()`. If the visitor has finished its processing the reader finalizes the XML document by calling the `endDocument()` method of the `ContentHandler`.

It is important to note that this sequence is executed not until the XSLT transformation actually starts: at first the source object is created using a `SAXSource` linked to

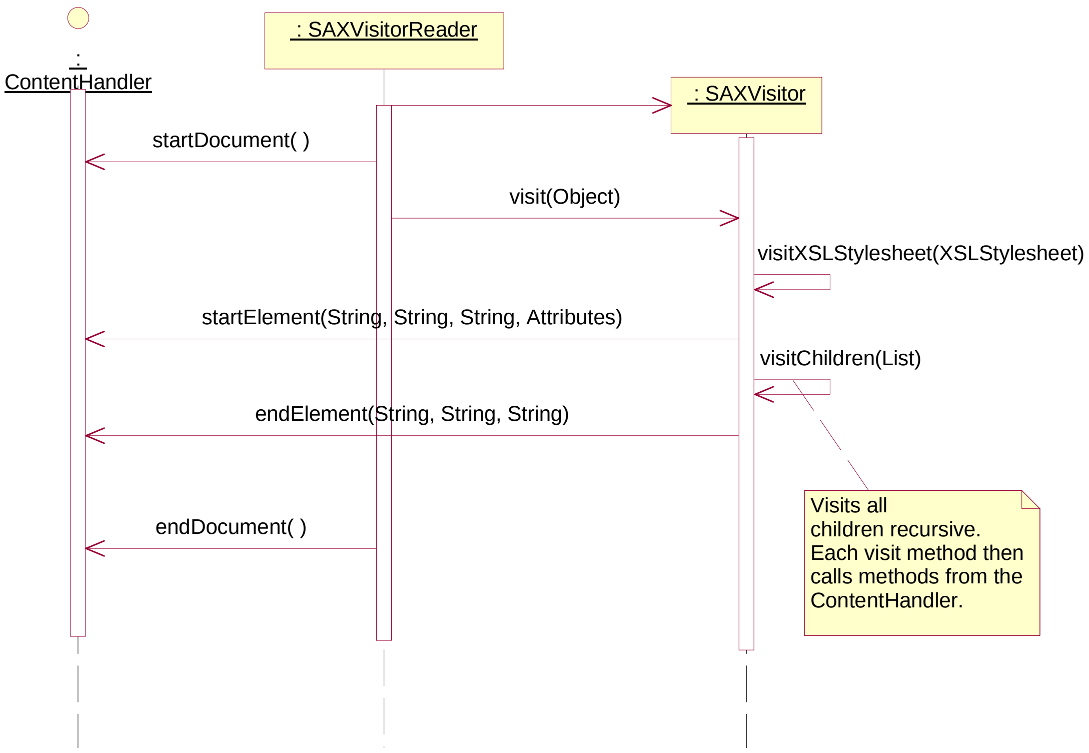

: ContentHandler
: SAXVisitorReader
: SAXVisitor
startDocument( )
visit(Object)
visitXSLStylesheet(XSLStylesheet)
startElement(String, String, String, Attributes)
visitChildren(List)
endElement(String, String, String)
Visits all children recursive. Each visit method then calls methods from the ContentHandler.
endDocument( )

Figure 4.10: UML Sequence Diagram "Stylesheet Serialization"

the reader. By starting the transformation the source is read and the process described above is triggered. This kind of processing is typical for SAX applications using passive handlers.

4.6.5 Final Transformation Process

The final transformation process wires the stylesheet with the input XMLs and produces the result document. This transformation is done by an XSLT processor and an XSL-FO formatter. In most cases the final output will be Adobe's Portable Document Format (PDF) but also other formats, like Rich Text Format, are possible if they are supported by the formatter. This section describes the transformation process, its execution and the way the XML data is resolved.

The Java API contains the *Transformation API for XML* (TrAX), used to transform XML in various ways. Most often it is implemented by XSLT processors to provide a simple way of integrating a processor into a Java application. If the API is utilized it

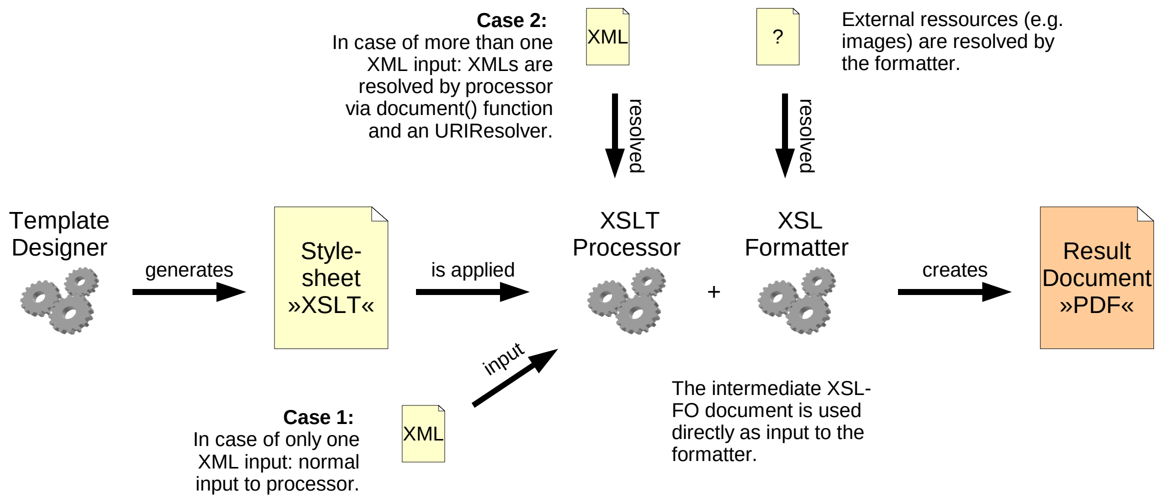

Case 2:
In case of more than one XML input: XMLs are resolved by processor via document() function and an URIResolver.
XML
resolved
?
resolved
External ressources (e.g. images) are resolved by the formatter.
Template Designer
generates
Style-sheet »XSLT«
is applied
XSLT Processor
+
XSL Formatter
creates
Result Document »PDF«
input
Case 1:
In case of only one XML input: normal input to processor.
XML
The intermediate XSL-FO document is used directly as input to the formatter.

Figure 4.11: Instruct's Transformation Process

is possible to change the processor at configuration time or at runtime. The transformation API can deal with many possible combinations of inputs and outputs, and is not specialized for any of the given types. The five most important TrAX-classes and interfaces to be used in InStruct's transformation process are:

- `TransformerFactory`: This is the vendor-neutral interface to the processor.
- `Transformer`: An instance of this class is created by the factory and performs the transformation.
- `Source`: This interface contains the information needed to act as the input of a transformation. It is used for both, the XML data and the stylesheet. There are implementations for DOM, SAX and stream inputs.
- `Result`: This interface contains the information needed to act as the output of a transformation. There are as well implementations for DOM, SAX and streams.
- `URIResolver`: An implementation of this interface is called by a processor to resolve URIs used in `document()`, `<xsl:import>`, and `<xsl:include>` and returns a `Source` object.

The transformation process is described in figure 4.11. The process starts with a `Source` object containing the stylesheet. The generation of this stylesheet either happens at transformation time or it has been already finished at a preceding serialization step. An implementation of the `TransformerFactory` class now creates a `Transformer` instance using the stylesheet. Depending on the number of XML input files there are two different cases how the input for the `Transformer` is processed:

- **Case 1:** There is only one XML input file available that is provided as a `Source` object. This is the normal case for XSLT transformations: the transformation will be started by applying the stylesheet on this single XML source.
- **Case 2:** There are two or more XML input files to be processed. They all have to be provided as `Source` objects and they must be made available to the processor. Since a normal transformation supports only one XML input the stylesheet has to instruct the processor to load each XML using the `document()` function. In this case the processor calls the `resolve` method of its `URIResolver` to access the document.

The `Transformer` uses the source objects and the optional `URIResolver` to perform the XSLT transformation resulting in an XSL-FO tree. Typically this result is directly used as input for the XSL formatter that creates the final document, for example a PDF. Instruct uses Saxon-B to process the XSLT 2.0-compliant stylesheets and RenderX XEP as the XSL-FO-Formattter.

The Resolver Concept

The answer to the problem of using more than one XML file in the transformation process is the topic of this section. Generally a transformation uses *one* stylesheet to transform *one* XML file. If data from more than one XML file is involved at least two different solutions to this problem are possible:

- All XML files may be stored in one big file like using a container. For example, the root element may be `<container>` with a sequence of `<item>` elements containing the root elements of each single XML file.
- The use of XSLT's `document` function and an optional use of a *resolver* to load the appropriate files at run-time. This is the recommended approach for this problem and is used in this project.

XSLT's document function allows the access to XML documents identified by a URI. This URI might be an absolute URL (typically one that starts with “http:” or “file:”), a relative one (interpreted relative to the URI of the document), or a globally unique URI (like one that identifies a book by its ISBN, for example: `urn:isbn:1570629641`).

An XSLT processor is usually able to interpret the familiar URLs using the http, ftp or file protocol. If a specific processor is not able to interpret a URI provided using the `document` function many APIs allow to define external resolver that may be written to accomplish this task. This permits the use of any URI to resolve a specific XML document. In Java this can be done with the interface `URIResolver` and its method `resolve(String href, String base)`. An implementation of this interface can be used during a transformation. If the processor encounters a `document` function it calls the `resolve` method and provides the URI and the optional base-URI as parameters of the method. It is the task of the implementation to search and to return the XML source referenced by the URI.

InStruct uses this concept to resolve any XML documents in cases of templates using more than one XML document. The decision which XML document will be returned is made at runtime depending on the document generation workflow. The XML might be read from the local filesystem, or come from a DMS or EAI system. Instruct uses *opaque* URIs starting with the scheme `instruct:` followed by the scheme-specific-part containing particulars sufficient to identify the correct XML instance: the name of the root element and its namespace. During the transformation the resolve method is called and the XML matching the root element is returned.

Dynamic Stylesheets

The resolver concept for loading XMLs is one possibility to use a resolver. Another application of resolvers is the loading of stylesheets. In Java a `URIResolver` is not only called by the processor if a `document` function is encountered but also on the XSLT instructions `xsl:import` and `xsl:include` that are used to reference another stylesheet. Typically these instructions modularize a stylesheet and increase readability and reuseability of parts of stylesheets.

In combination with a resolver an include or import instruction can be used to create stylesheets dynamically. By loading the stylesheet prior to start of transformation, the `resolve` method is called to turn a URI used in includes or imports into an XML `Source` object. Hence, it is possible to generate the stylesheet contained in the object at run-time. InStruct uses this concept to update templates by loading parts of a stylesheet from a content management system (CMS). Therefore a text or a component, for example the general terms and conditions of a company, first has to be stored in the CMS using XSLT/XSL-FO markup. This component can then be included into the stylesheet at run-time and allows to use up-to-date texts and components without even touching the template.

5 Conclusion

This final chapter summarizes the work presented in this thesis. In addition, opportunities for future research in this area are pointed out and briefly discussed.

The main purpose of this thesis has been to examine how structured documents can be created by combining CMS, DMS and EAI systems. The first part of this thesis examines some systems offering solutions in this area. Stylesheet Designer for XSLT are shipped already with excellent functions while having a integration deficiency. The creation of stylesheets using WYSIWYG is simple but it is not a subject where the input data come from and how this data flow may be automated. Enterprise reporting systems focuses mainly on this topic. Data from existing systems are queried and the results are prepared to create reports of them. Since those tools emerged from the database domain they do not support XML data very well. However, EAI systems concentrate especially on XML technologies leading to a good support for XML processing tasks including the import of data from external systems but CMS and DMS tools are often neglected.

The subsequent part covers technologies to be used in the XML document processing domain. Beside outgrown and minor powerful languages like DSSSL and CSS the main focus is lain on XSLT and XSL-FO that can be used in conjunction to transform and format XML. However it is essential to learn these complex languages prior to be able to exploit their power. The thesis also introduces some implementations of the two specifications proved to be stable for deployment. However there is still work to do in the field of formatting. The usage of XML document processing technologies is demonstrated with some XML based formats. DocBook, TEI and DITA can be best processed using XSLT and XSL-FO. They reveal the power of XML as things are now. Obviously OpenDocument and WordML are mostly used within word processors. XML tools able to work with documents from word processors will probably gain importance in future.

The final part of this thesis describes the implementation of InStruct, a system intended to help creating structured documents while integrating data from CMS, DMS and EAI systems. As such it offers a solution not provided by any existing application. The author developed parts of the system including the document model and the document generation component. Finally, the document model is used as the "M" of the MVC implementation of the designer GUI, as input for the generation of the stylesheet and as the basis for persistence. The document generation component first generates a XSLT stylesheet as designed by the user. Then it transforms multiple XML files with the help of the stylesheet to a final result document in PDF and other formats. Flexible resolvers allow the handling of multiple XML files and the use of dynamic stylesheets with automated update features.

It might be interesting to compare the final software to the products, processes and technologies from the chapters one to three. First, to demonstrate that InStruct is able to fulfill the requirements needed to support the business cases from chapter one (stock and funds reports in capital management, contract order creation in logistics, and sales offers): Using InStruct the process might be triggered manually or time-based. On the one hand data can be retrieved from external systems using an EAI adapter and on the other hand resources can be included from content management systems. Then, up-to-date calculations can be performed and finally multi-channel delivery is used to create various output formats.

By comparing InStruct to the products presented in chapter two many commonalities can be found. InStruct has some features of a stylesheet designer: it supports XML input and generates an XSLT stylesheet to transform the input to the desired output. Additionally, it offers many ongoing features traditionally found only in EAI tools like integration with content and document management systems. This points out that InStruct fits perfectly into an existing EAI solution to take over document-related tasks.

Chapter three introduces the concept of XML document processing, InStructs is making use of. The process starts with an arbitrary XML file. It is transformed using a stylesheet that applies logic and layout. The process is executed by an engine, supporting monitoring and management of the workflow. At the end of the process the output is created which might be a binary format or XML again. The implementation uses XML Schema to describe the input data, relies on XSLT and XPath for the transformation, XSL-FO for the formatting and uses a SAX-based approach to meet the requirements in terms of throughput.

5.1 Future Work

There are still many possibilities for the improvement of the current system. The most important ones to be tackled in the future are:

- **Enhancement of dynamic stylesheets:** The current Saxon release includes extension functions to compile and execute stylesheets which can be constructed dynamically or loaded from an external source. These extensions may be used for the adaption of transformation process to support updates of the stylesheet at time of execution.

- **Standardization:** The current system uses the JAXP API for all XSLT transformations making it possible to replace the processor easily. At the moment no related interface for XSL-FO formatters is available. If such an interface (for example JAXG [Mär05]) becomes established the implementation may yield profit.
- **Evaluation of a XSLT-centric stylesheet generation process:** After simplifying the XML format used for model persistence, an XSLT-centric approach to generate the final XSLT stylesheet would be possible. However, the benefits of such a bootstrapping method should be evaluated in advance.
- **XSLT Mapper:** Due to the generic architecture of the XSLT generation components the creation of an XSLT mapping tool for XML-to-XML transformations is feasible. Such a transformation would fit perfectly into the process from the EAI system to the final document.

A Examples and Listings

A.1 Addressbook Example

This thesis uses a recurring example of a simple addressbook. Listing A.1 shows an XML file representing an address book. The root element is `<addressbook>` consisting of multiple `<person>` elements. In addition to the person's forename and surname the gender and an e-mail address may be specified.

Listing A.1: A simple addressbook XML file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<addressbook xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:noNamespaceSchemaLocation="addressbook.xsd" name
   ="My␣Addressbook">
  <person id="john">
    <forename>John</forename>
    <surname>Green</surname>
    <gender>male</gender>
    <e-mail>example@mail.com</e-mail>
  </person>
  <person id="clara">
    <forename>Clara</forename>
    <surname>White</surname>
    <gender>female</gender>
    <e-mail>example@mail.com</e-mail>
  </person>
  <person id="peter">
    <forename>Peter</forename>
    <surname>Gerstbach</surname>
    <gender>male</gender>
    <e-mail>example@mail.com</e-mail>
  </person>
</addressbook>
```

A.2 Books Example

Another example is an XML file that stores information about books. The root element is `books` containing `<book>` elements. Each book specifies a title, an author, an ISBN number, a publisher and a description of the book. The description may contain elements from the XHTML namespace. Listing A.2 shows such a file.

Listing A.2: A simple books XML file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="books-push.xslt"?>
<books xmlns:xhtml="http://www.w3.org/1999/xhtml" xmlns:xsi="
   http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="books.xsd">
  <book>
    <title>The Art of Peace</title>
    <author>Morihei Ueshiba</author>
    <ISBN>1570629641</ISBN>
    <publisher>Shambhala</publisher>
    <description>Description of this book with optional <
       xhtml:em>HTML</xhtml:em> elements.</description>
  </book>
  <book>
    <title>Hagakure: The Book of the Samurai</title>
    <author>Yamamoto Tsunetomo</author>
    <ISBN>4770029160</ISBN>
    <publisher>Kodansha International</publisher>
    <description>Description of this book with optional <
       xhtml:em>HTML</xhtml:em> elements.</description>
  </book>
</books>
```

A.3 XSL-FO Example

The following listings contain the source code of the example in the XSL-FO chapter on page 43. The subsequent figure shows the result of the transformation rendered to PDF.

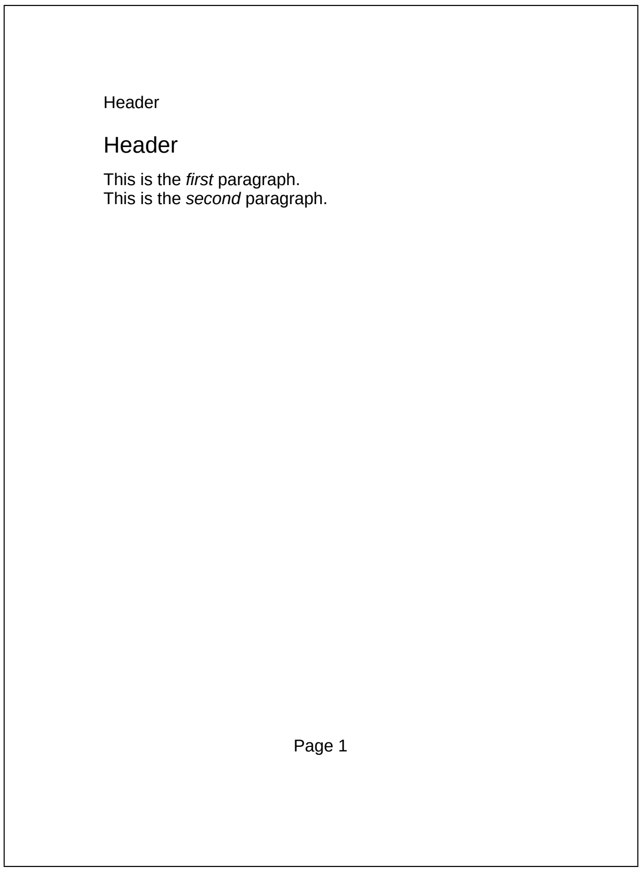
Listing A.3: XSL-FO Example: XML Source File

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <section>
    <title>Header</title>
    <para>This is the <em>first</em> paragraph.</para>
    <para>This is the <em>second</em> paragraph.</para>
  </section>
</document>
```

Listing A.4: XSL-FO Example: XSLT Stylesheet Producing XSL-FO Output

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/
   XSL/Transform" xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <xsl:output version="1.0" encoding="UTF-8" indent="no" omit-
     xml-declaration="no" media-type="text/html"/>
  <xsl:template match="/">
    <fo:root>
      <fo:layout-master-set>
        <fo:simple-page-master master-name="default-page" page-
           height="210mm" page-width="148mm" margin="2cm">
          <fo:region-before extent="1cm"/>
          <fo:region-after extent="1cm"/>
          <fo:region-body margin-top="1cm" margin-bottom="1cm"/>
        </fo:simple-page-master>
      </fo:layout-master-set>
      <xsl:apply-templates select="document/section"/>
    </fo:root>
  </xsl:template>
  <xsl:template match="section">
    <fo:page-sequence master-reference="default-page" initial-
       page-number="1" format="1">
      <fo:static-content flow-name="xsl-region-before">
```

```
        <fo:block>
          <xsl:value-of select="title"/>
        </fo:block>
      </fo:static-content>
      <fo:static-content flow-name="xsl-region-after">
        <fo:block text-align="center">Page <fo:page-number/>
        </fo:block>
      </fo:static-content>
      <fo:flow flow-name="xsl-region-body">
        <fo:block font-size="16pt" space-after="0.5em">
          <xsl:value-of select="title"/>
        </fo:block>
        <xsl:apply-templates/>
      </fo:flow>
    </fo:page-sequence>
  </xsl:template>
  <xsl:template match="para">
    <fo:block>
      <xsl:apply-templates/>
    </fo:block>
  </xsl:template>
  <xsl:template match="em">
    <fo:inline font-style="italic">
      <xsl:apply-templates/>
    </fo:inline>
  </xsl:template>
</xsl:stylesheet>
```


Header

Header

This is the *first* paragraph.
This is the *second* paragraph.

Page 1

Figure A.1: XSL-FO Example: Result of the Transformation Rendered to PDF

A.4 Purchase Order Example

The following example illustrates InStruct using a simple purchase order example: a customer orders some items using an online shop and receives a notification of the ordered items.

Listing A.5: The Purchase Order XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:noNamespaceSchemaLocation="po.xsd" orderDate="
   2006-04-01">
  <shipTo>
    <name>Peter Gerstbach</name>
    <street>Some Street 1</street>
    <zip>1130</zip>
    <city>Vienna</city>
    <country>Austria</country>
  </shipTo>
  <billTo>
    <name>GFT Technologies GmbH</name>
    <street>Canovagasse 7/2</street>
    <zip>1010</zip>
    <city>Vienna</city>
    <country>Austria</country>
  </billTo>
  <items>
    <item partNum="101-AA">
      <productName>Western Digital Raptor 150GB SATA</
         productName>
      <quantity>3</quantity>
      <price currency="EUR">500</price>
      <shipDate>2006-04-08</shipDate>
    </item>
    <item partNum="345-CD">
      <productName>Corsair DIMM 2GB PC2-667 DDR2</productName>
      <quantity>2</quantity>
      <price currency="EUR">250</price>
      <shipDate>2006-04-08</shipDate>
    </item>
    <item partNum="123-CA">
      <productName>NEC PlasmaSync 61"</productName>
␣␣␣␣␣␣<quantity>1</quantity>
␣␣␣␣␣␣<price␣currency="EUR">14000</price>
␣␣␣␣␣␣<shipDate>2006-04-10</shipDate>
␣␣␣␣</item>
␣␣</items>
</purchaseOrder>
```

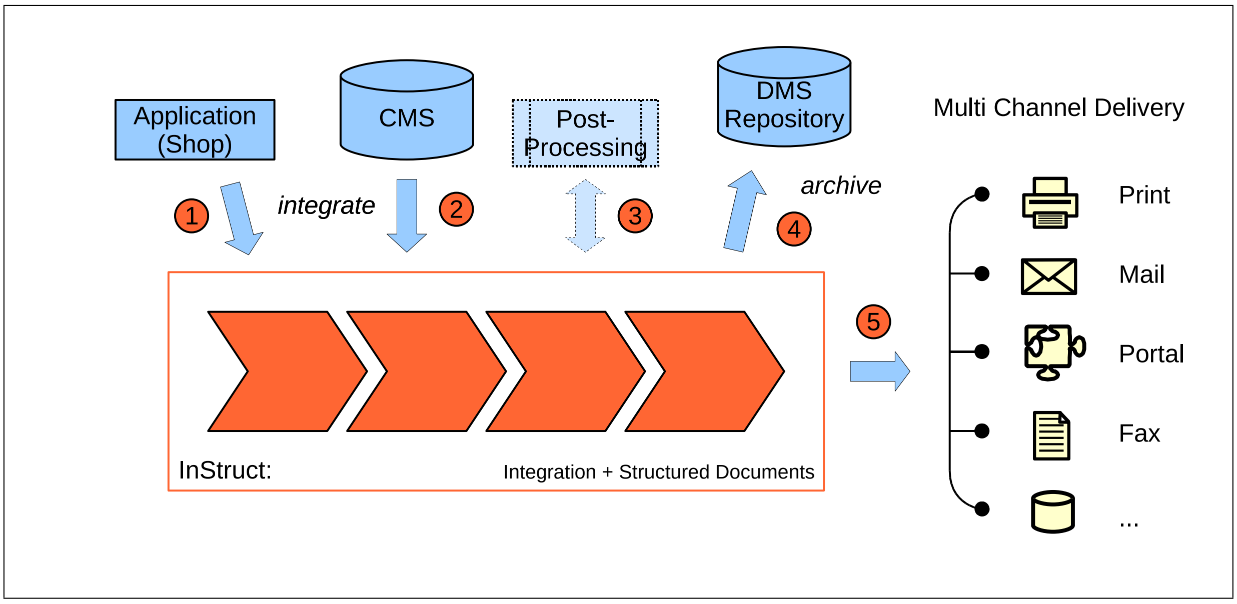


Figure A.2: A typical InStruct process including data integration and document generation.

1. The XML file from the external application (the shop system) is transformed using the XSLT stylesheet.
2. Larger units of text (for example the general terms and conditions) may be loaded from a CMS.
3. The workflow may also include a manual post processing of the document.
4. Other resources like images my be loaded from a DMS. The DMS can also be used as final archiving system.
5. Finally, the Document Generator performs multi-channel delivery.

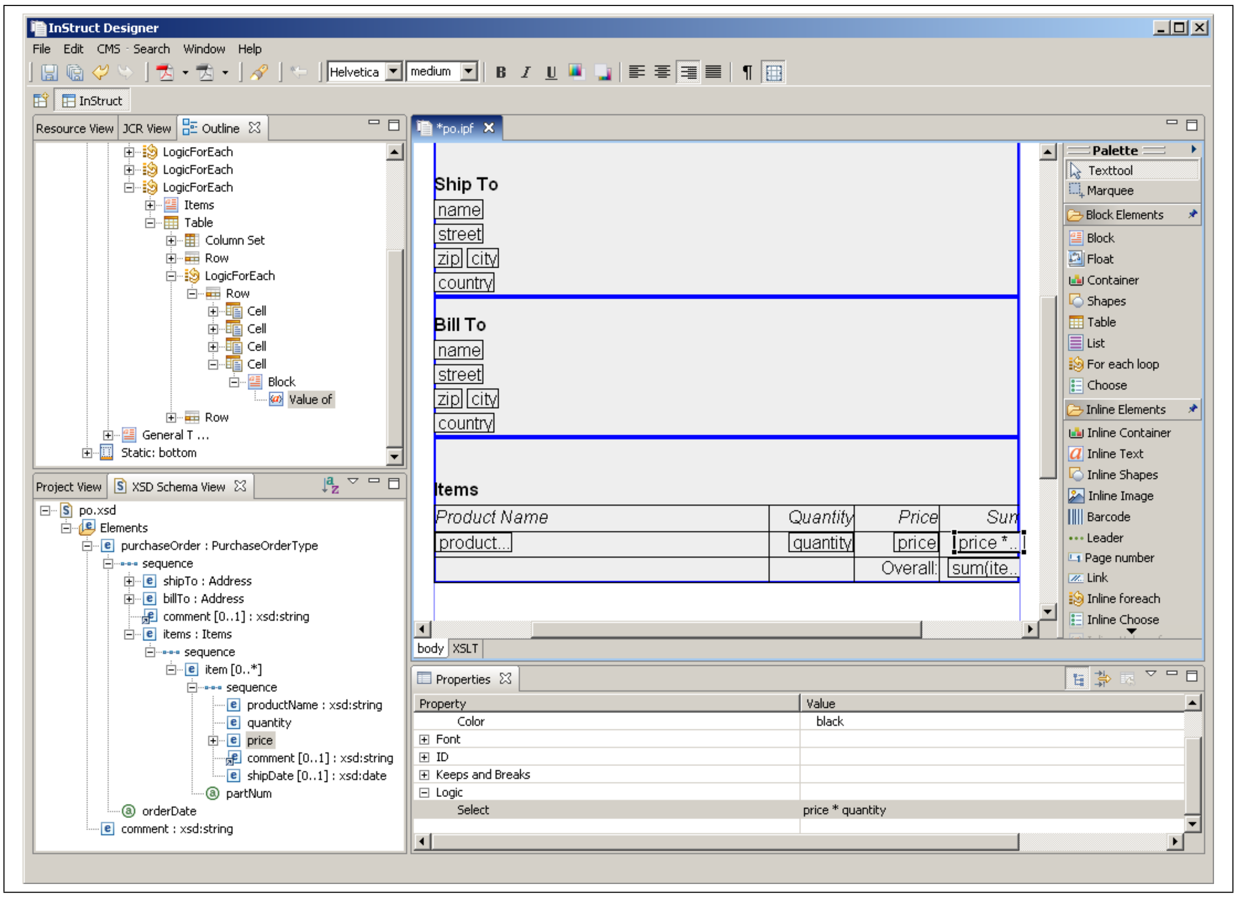
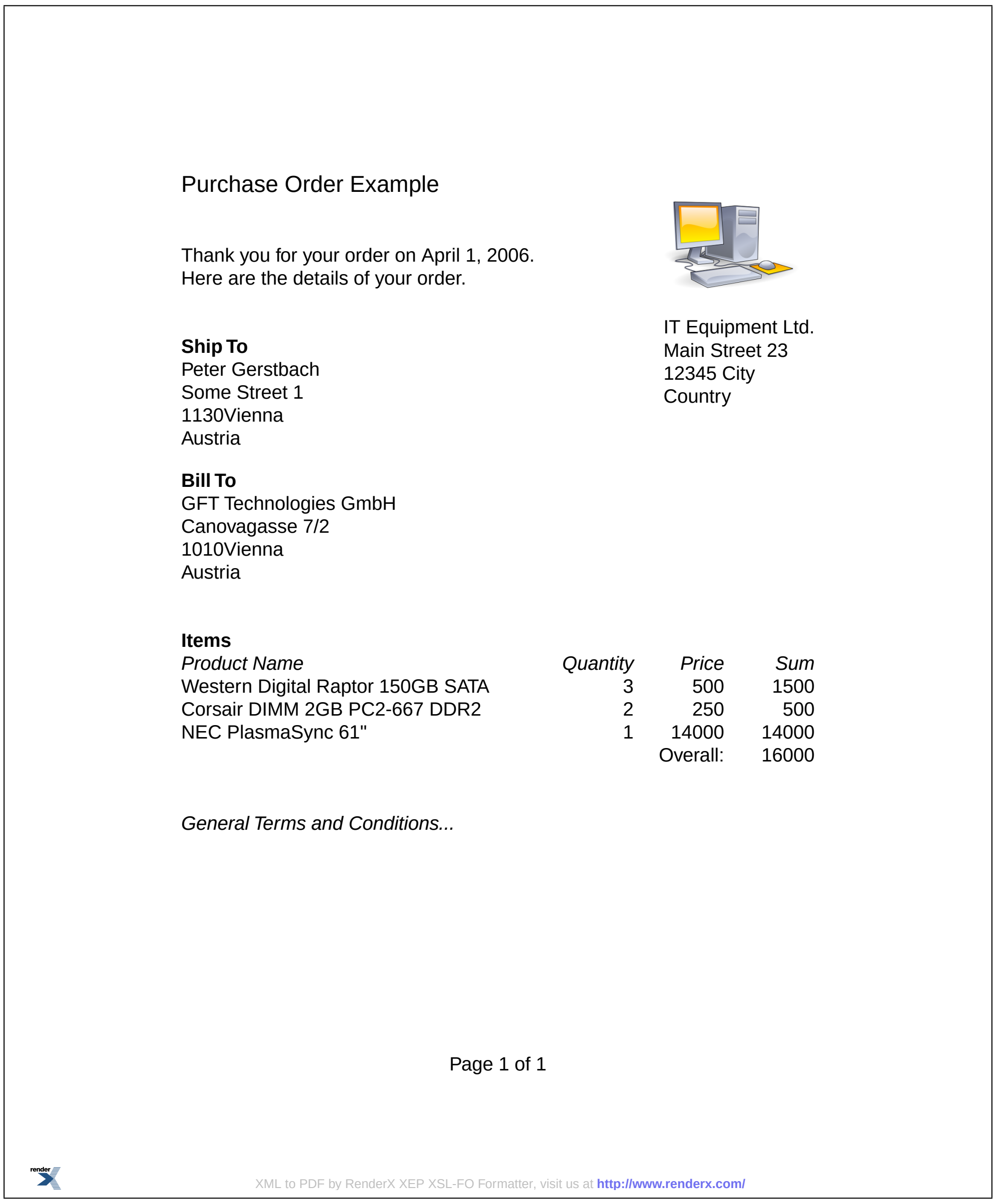


Figure A.3: Screenshot of the InStruct Designer.
The Screenshot shows the Designer while editing the purchase order stylesheet. The *Outline View* at the top left corner shows the structure of the stylesheet. The *Schema View* at the bottom left is a graphical representation of the XML Schema describing the XML. The main View in the center shows the WYSIWYG-editor for the stylesheet. The *Property View* at the bottom allows to define the behaviour and formatting of the elements including XPath expressions.



Purchase Order Example

Thank you for your order on April 1, 2006.
Here are the details of your order.

IT Equipment Ltd.
Main Street 23
12345 City
Country

Ship To
Peter Gerstbach
Some Street 1
1130Vienna
Austria

Bill To
GFT Technologies GmbH
Canovagasse 7/2
1010Vienna
Austria

Items

Product Name	*Quantity*	*Price*	*Sum*
Western Digital Raptor 150GB SATA	3	500	1500
Corsair DIMM 2GB PC2-667 DDR2	2	250	500
NEC PlasmaSync 61"	1	14000	14000
		Overall:	16000

General Terms and Conditions...

Page 1 of 1

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at **http://www.renderx.com/**

Figure A.4: The final purchase order document generated by the *Document Generator*. Most data is retrieved from the XML file (listing A.5). Larger units of text (the general terms and conditions) are loaded from a CMS. Calculations (e.g. the sums on the right) are performed using XPath expressions. Special XSL-FO expressions allow to output page-related data ("Page X of Y"). Dates may be formatted using the current locale.

List of Figures

List of Tables

Listings

Bibliography

[Alt] Altova. Altova stylevision. Website: http://www.altova.com/products_xsl.html.

[Amb03] Jacek Ambroziak. Introducing gregor 1.0 xml transformation framework: Enabling fast xslt transforms on critical paths. In *Proceedings of the XML Europe 2003 Conference*, London, England, 2003. http://www.idealliance.org/papers/dx_xmle03/papers/03-03-07/03-03-07.html.

[Apa] The Apache Software Foundation. Apache fop. Website: http://xmlgraphics.apache.org/fop/.

[Bal04] Klaas Bals. Using xsl-fo 1.1 for business-type documents. In *Proceedings of the XML Europe 2004 Conference*, Amsterdam, The Netherlands, 2004. http://www.idealliance.org/papers/dx_xmle04/papers/04-02-01/04-02-01.html.

[BBC$^+$] Oliver Becker, Paul R. Brown, Petr Cimprich, Christian Nentwich, and Tolja Zubow. Streaming transformations for xml. Website: http://stx.sourceforge.net/.

[Bec04] Oliver Becker. *Serielle Transformationen von XML*. Dissertation, Humboldt-Universität Berlin, 2004. http://edoc.hu-berlin.de/dissertationen/becker-oliver-2004-11-26/PDF/Becker.pdf.

[Blo00] Jeremy Blosser. Java tip 98: Reflect on the visitor design pattern. Website: http://www.javaworld.com/javaworld/javatips/jw-javatip98.html, July 2000.

[Bon04] Frank Bongers. *XSLT 2.0 – Das umfassende Handbuch*. Galileo Press GmbH, 1st edition, 2004.

[Bra05] Herbert Braun. Das neue Web. *c't*, 15:172–178, 2005. Heise Zeitschriften Verlag.

[CC02] Gerardo Canfora and Luigi Cerulo. A visual approach to define xml to fo transformations. In *SEKE '02: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 563–570, Ischia, Italy, 2002. ACM Press.

[DPH01] Don Day, Michael Priestley, and Gretchen Hargis. Frequently asked questions about the darwin information typing architecture, March 2001. Website: `http://www-128.ibm.com/developerworks/xml/library/x-dita3/`.

[DPS01] Don Day, Michael Priestley, and David Schell. Introduction to the darwin information typing architecture, March 2001. Website: `http://www-128.ibm.com/developerworks/xml/library/x-dita1/`.

[EN02] Elke and Michael Niedermair. *XML für Print und Screen*. Franzis' Verlag GmbH, 2002.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[Ger04] Peter Gerstbach. XML Data Binding. Bakkalaureatsarbeit, Vienna University of Technology, 2004. `http://www.gerstbach.at/2004/XMLDataBinding/`.

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

[ISO96] ISO/IEC. *Document Style Semantics and Specification Language (DSSSL)*, 1996. ISO/IEC Standard: 10179:1996(E).

[Jas] JasperSoft Corporation. Jasperreports. Website: `http://jasperreports.sourceforge.net/`.

[Jav02] Java Community Process. *Long-Term Persistence for JavaBeans Specification*, May 2002. Technical Specification: `http://www.jcp.org/en/jsr/detail?id=057`.

[K$^+$05] Michael Kay et al. XSL FAQ, Performance. Website: `http://www.dpawson.co.uk/xsl/sect4/N9883.html`, 2005.

[Kay00] Michael Kay. *XSLT Programmer's Reference*. Wrox Press Ltd., Birmingham, UK, 2000.

[Kay01] Michael Kay. XSL List: translets vs. templates. Mailing list archive: http://www.biglist.com/lists/xsl-list/archives/200112/msg00614.html, December 2001.

[Kay04a] Michael Kay. Up-conversion using xslt 2.0. In *Proceedings of the XML 2004 Conference*, Washington, D.C., U.S.A., 2004. http://www.idealliance.org/proceedings/xml04/abstracts/paper111.html.

[Kay04b] Michael Kay. *XSLT 2.0 Programmer's Reference*. Wrox Press Ltd., 3rd edition, 2004.

[Kay04c] Michael Kay. XSLT and XPath Optimization. In *Proceedings of the XML Europe Conference*, Amsterdam, The Netherlands, 2004. http://www.idealliance.org/papers/dx_xmle04/papers/02-03-02/02-03-02.html.

[Kay05] Michael Kay. XSL List: Bug in schema for xslt 2.0? Mailing list archive: http://biglist.com/lists/xsl-list/archives/200502/msg00745.html, February 2005.

[Les01] Donald M. Leslie. Transforming documentation from the xml doctypes used for the apache website to dita. In *SIGDOC '01: Proceedings of the 19th Annual International Conference on Computer Documentation*, pages 157–164, Sante Fe, New Mexico, USA, 2001. ACM Press.

[Lev91] Roger E. Levien. The civilizing currency: documents and their revolutionary technologies. *Technology 2001: The Future of Computing and Communications*, pages 205–239, 1991. MIT Press, Cambridge, MA, USA.

[LLR04] Simon St. Laurent, Evan Lenz, and Mary Mc Rae. *Office 2003 XML*. O'Reilly Media, Inc., 2004.

[Mär05] Jeremias Märki. JAXG - java api for xml graphics processing, 2005. Website: http://jeremias-maerki.ch/dev/jaxg/.

[MS01] Barbara C. McNurlin and Ralph H. Sprague. *Information Systems Management in Practice*. Prentice Hall, 5th edition, 2001.

[Mul] Mulberry Technologies. XSL-List – open forum on xsl. Website: http://www.mulberrytech.com/xsl/xsl-list/.

[NH01] Simon North and Paul Hermans. *Sams Teach Yourself XML in 21 Days*. Pearson Professional Education, 2001.

[OAS04] OASIS DocBook Technical Committee. *DocBook Schemas*, October 2004. Technical Specification: http://www.docbook.org/oasis/.

[OAS05] Oasis open document format for office applications (opendocument) tc faq, 2005. Website: http://www.oasis-open.org/committees/office/faq.php.

[Oli04] Ricardo Olivieri. Generating online reports using jasperreports and websphere studio. Website: http://www-128.ibm.com/developerworks/websphere/library/techarticles/0411_olivieri/0411_olivieri.html, November 2004.

[Ope98] The Open Group. *Data Interchange Services*, 1998. Website: http://www.opengroup.org/architecture/togaf7-doc/arch/p3/trm/tx/tx_d_int.htm.

[Paw02] Dave Pawson. *XSL-FO*. O'Reilly, Cambridge, MA, USA, August 2002.

[Pri01] Michael Priestley. Specializing topic types in dita, March 2001. Website: http://www-128.ibm.com/developerworks/xml/library/x-dita2/.

[Pro] Progress Software Corporation. Stylus studio xslt profiler. Website: http://www.stylusstudio.com/xslt_profiler.html.

[Pun03] Steve Punte. Fast XSLT. Website: http://www.xml.com/pub/a/2003/04/02/xsltc.html, April 2003.

[Rap01] Lowell Rapaport. Transactions: output management becomes low-cost EAI. *Transform Magazine*, 10(10):17–20, 2001. CMP Media, Inc., USA.

[Sch05] Ronald Schmelzer. Solving the very large messaging problem in the enterprise. Technical report, ZapThink, February 2005. Website: http://www.zapthink.com/report.html?id=WP-0137).

[Suna] Sun Microsystems. *Java 2 Platform Standard Edition 5.0 Compatibility with Previous Releases*. Website: http://java.sun.com/j2se/1.5.0/compatibility.html.

[Sunb] Sun Microsystems. *Release Notes for Java 2 SDK*. Website: http://java.sun.com/j2se/1.4.2/relnotes.html#JAXP_xslt.

[Sun04] Sun Microsystems. *The J2EE 1.4 Tutorial*, December 2004. Technical Documentation: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/J2EETutorial.pdf.

[Ten01] Jeni Tennison. XSL List: Rescuing xslt from niche status. Mailing list archive: http://www.biglist.com/lists/xsl-list/archives/200102/msg01143.html, February 2001.

[VDLP02] Jean-Yves Vion-Dury, Veronika Lux, and Emmanuel Pietriga. Experimenting with the circus language for xml modeling and transformation. In *DocEng '02: Proceedings of the 2002 ACM Symposium on Document Engineering*, pages 82–87, McLean, Virginia, USA, 2002. ACM Press.

[Veg01] Carlos Alonso Vega. Java and reports, some solutions for the past, present and future. In *SIGUCCS '01: Proceedings of the 29th Annual ACM SIGUCCS Conference on User Services*, pages 275–278, Portland, Oregon, USA, 2001. ACM Press.

[W3C97] W3C World Wide Web Consortium. *A Proposal for XSL*, August 1997. W3C Note: http://www.w3.org/TR/NOTE-XSL.html.

[W3C01] W3C World Wide Web Consortium. *Extensible Stylesheet Language (XSL)*, October 2001. W3C Recommendation: http://www.w3.org/TR/xsl/.

[W3C05a] The Extensible Stylesheet Language family (XSL), August 2005. Website: http://www.w3.org/Style/XSL/.

[W3C05b] W3C World Wide Web Consortium. *Extensible Stylesheet Language (XSL) Version 1.1*, July 2005. W3C Working Draft: http://www.w3.org/TR/2005/WD-xsl11-20050728/.

[W3C05c] W3C World Wide Web Consortium. *XSL Transformations (XSLT) Version 2.0*, November 2005. W3C Candidate Recommendation: http://www.w3.org/TR/2005/CR-xslt20-20051103/.

[Wal04] Norman Walsh. Validating XSLT 2.0. Website: http://norman.walsh.name/2004/07/25/xslt20, July 2004.

[Wal05] Norman Walsh. Docbook xsl stylesheets. Website: http://wiki.docbook.org/topic/DocBookXslStylesheets, October 2005.

[Wik05] Wikimedia Foundation Inc. Wikipedia, the free encyclopedia. Website: `http://en.wikipedia.org/`, 2005.