Department of Informatics
University of Fribourg (Switzerland)

A Framework for Interactive Document Recognition

THESIS

submitted to the Faculty of Science of the
University of Fribourg (Switzerland)
in conformity with the requirements for the degree of
Doctor scientiarum informaticarum

submitted by

Oliver HITZ

of Untersiggenthal (AG)

Thesis no. 1488
Printed at Mécanographie, University of Fribourg
2005

Accepted by the Faculty of Science of the University of Fribourg (Switzerland) following the proposal of:

- Prof. Rolf INGOLD, Université de Fribourg, Switzerland, Thesis Director;
- Prof. Rémy MULLOT, Laboratoire PSI, Université de Rouen, France, Examinator;
- Prof. Béat HIRSBRUNNER, Université de Fribourg, Switzerland, Examinator.

October 6, 2005

The Thesis Director: The Dean:

[illegible] [illegible]

Prof. Rolf INGOLD Prof. Marco CELIO

Abstract

Document recognition is a research domain that doesn't lose its relevance even in a world where documents are increasingly often available in an electronic form. Whereas some years ago, the goal of document recognition was to convert documents from paper into an electronic form, the problem is shifted more and more from pure recognition towards document understanding.

This requires much more context knowledge—knowledge, that cannot be easily specified. The problem can be approached by considering the user as an important component of the recognition system. The user knows what he expects from the recognition system, why shouldn't this information be used? This is what interactive document recognition does. Interactive document recognition systems cooperate with the user and try to learn from him.

This thesis addresses the problem of interactivity in interactive document recognition systems. By using technology around the XML standards, and starting from an idea that is typically used for publishing content for the World Wide Web, a model for structuring interactive document recognition applications is developed. This model ensures a high reusability of program modules.

The feasibility of this model is demonstrated with a prototype that allows to graphically visualize document recognition data available in XML format. With a little bit more effort, the same data can be edited interactively.

The fact that the prototype has already been used in other research projects shows that the approach is very promising.

Zusammenfassung

Die Dokumenterkennung ist ein Forschungsgebiet, das auch mit der zunehmenden Verlagerung von Inhalten in elektronische Formate nicht an Bedeutung verliert. Ging es vor einigen Jahren noch in erster Linie darum, Dokumente in Papierform in eine elektronische Form zu bringen, verlagert sich das Problem immer mehr vom reinen Erkennen des Dokumentes hin zu einem Verstehen des Inhaltes.

Dazu wird immer mehr Kontextwissen benötigt—Wissen, das sich schlecht ein für allemal festlegen lässt. Dieses Problem kann angegangen werden, indem der Benutzer als wichtiger Bestandteil des Dokumenterkennungssystem gesehen wird. Der Benutzer weiss, was er vom Erkennungssystem erwartet, warum also kann nicht diese Information verwendet werden? Genau dies geschieht bei der interaktiven Dokumenterkennung. Interaktive Dokumenterkennungssysteme kooperieren mit dem Benutzer und versuchen, von ihm zu lernen.

Diese Dissertation behandelt das Problem der Interaktivität in solchen interaktiven Dokumenterkennungssystemen. Mit Hilfe von Technologie rund um den XML Standard und ausgehend von einer Idee, wie sie typischerweise bei der Publikation von Inhalten für das World Wide Web angewandt wird, wird ein Modell zur Strukturierung von Applikationen für die interaktive Dokumenterkennung entwickelt, welches eine hohe Wiederverwendbarkeit von Programmmodulen gewährleistet.

Die Machbarkeit dieses Modells wird mit einem Prototyp aufgezeigt, mit welchem beliebige, im XML-Format vorliegende Dokumenterkennungsdaten mit wenig Aufwand grafisch visualisiert, und—mit etwas mehr Aufwand—interaktiv bearbeitet werden können.

Die Tatsache, dass der Prototyp bereits in anderen Forschungsprojekten zum Einsatz gekommen ist, zeigt, dass der Ansatz sehr vielversprechend ist.

Acknowledgements

This work would not have been possible without the help of many people...

- My parents, who have always provided support for me during all the years of school and education. Vielen Dank!
- Rolf Ingold, my thesis advisor, who let me lead my work freely and without unnecessary constraints. He also did have a great deal of patience and confidence. Merci beaucoup!
- Rémy Mullot and Béat Hirsbrunner, who have accepted to participate in the jury for my thesis and comment on the work. Merci beaucoup!
- My research group colleagues Lyse Robadey and Maurizio Rigamonti. Lyse has been the first beta-tester of the xmillum prototype and has provided me with lots of useful feedback. Maurizio has extended the prototype and has with his input contributed to the third generation xmillum model described in this thesis. Merci! Grazie mille!
- All other colleagues and friends at the DIUF who were always available for profound technical discussions, non-technical chats and other crazy projects. At the risk of forgetting somebody (apologies), in no particular order: Simon, Sergio, Olivier, Darie, Daniel, Rolf, Folco, Lukas, Houda, Karim, Nicolas, Dijana, Alessio, Amine, Soraya, Anthony, Yvette, Marie-Claire, Marianne, Eliane... Danke! Merci!
- Last but not least, Caroline, for her tolerance and support during the final steps of this work. Thank you very much!

Contents

Chapter 1

Introduction

The document recognition problem—the task of reading a paper document using a computer in order to process it by further—has many different facets that have occupied the document recognition community for many years.

As some of the problems get solved, new challenges turn up. The better the recognition results get, the more people expect from them and the further they want to go. While in the beginning document recognition research was mainly focused on identifying printed characters in order to reduce the cost of typists, document recognition is used today to extract much more information than simply text. The structure of a document can be as important as the text itself, and also components such as images, figures or tables contain much useful information. Ideally, we would like to go as far as to understand the text in question to get an idea about its meaning.

The further we move towards recognizing the semantics, the more knowledge about the context is required. For recognizing printed characters, we might need to know the used font. For recognizing the structure of a document, however, we need to know much more information related to layout rules, to the document type and maybe even to the culture of the document's target audience. This knowledge is very difficult to express, and it cannot be done once and for all.

We believe that this difficulty can be overcome with interactive systems with the users as substantial components. The users know what results they expect from a document recognition process. The system provides to its users the knowledge they need to make informed and intelligent decisions, which can be used to generate the context knowledge necessary to make such decisions independently of the user at a later time.

This thesis presents a software framework that helps to create interactive systems for document recognition. It allows researchers and developers to refrain from some of the tasks connected to visualization and user interaction in order to concentrate on the more important issues related to document recognition.

1.1 Document Production and Recognition

The purpose of a document is to store information so that it can be exchanged between users. During the document production process, this information is put into the document so that a user can conveniently extract it at a later time. The traditional document production process involves four steps as illustrated in figure 1.1:

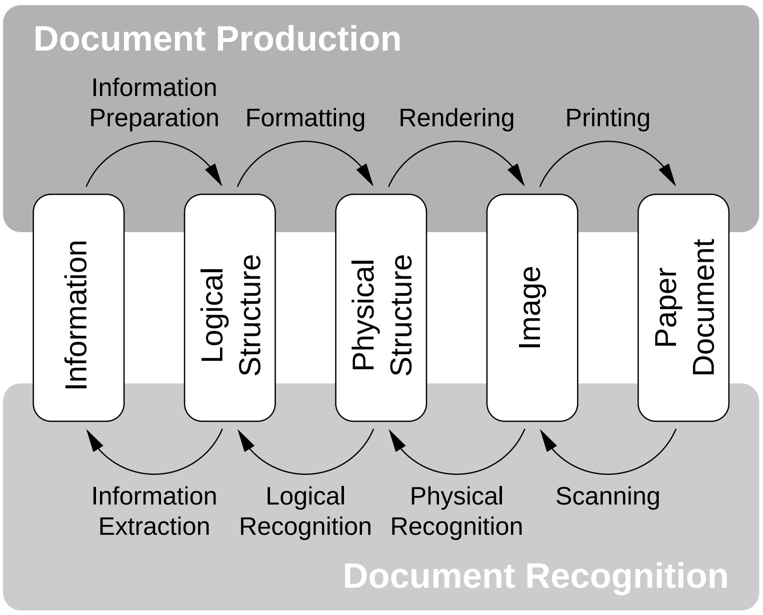


Figure 1.1: Document production and document recognition are two opposing processes.

Information Preparation The information is transformed to be used in a document. This transformation includes writing and editing text, drawing images etc. The result of this step is the *logical document structure*, a structure of information prepared for use in a document.

Formatting The formatting phase maps the logical structure onto a specific media type, resulting in a *physical structure*. This structure is constrained by the characteristics of the final media such as the two-dimensionality and the page size for paper documents.

Rendering During the rendering phase, the physical structure is transformed to an image consisting of pixels. This phase takes into account the resolution and the color space of the output device.

Printing The printing phase, finally, prints the image produced so far onto the final media.

Document recognition is the opposite process. Its goal is to restore the electronic form of a printed document so that information can be extracted or that the document can be reused again. The whole document recognition process comprises the following four steps:

Scanning The paper document is scanned. This step produces a document image, which is a simple matrix of pixels. Noise may be introduced in this step, due to dirt on the paper documents, dust on the scanner, irregularities in the scanner's optics or skew because of inaccurate paper handling. The scanned image therefore differs from the rendered image.

Physical Recognition The physical structure recognition phase basically combines individual pixels and regions to groups belonging together and attaches additional semantics in the form of labels to these groups. Immediate results of this step are regions of the image containing individual characters, words or text lines, or regions containing images, logos, rules etc. Data attached to the regions might be the text contained in textual regions (the result of an optical character recognition algorithm), the names and properties of the fonts in which the text is written or formatting properties such as the text justification. The result of the physical recognition is typically of hierarchical nature, often representing the top-down decomposition of a document.

Logical Recognition During the logical structure recognition, the individual building blocks of the physical structure are labeled and aggregated in a higher level structure that reflects the logical structure of the document. The actual form of the logical structure depends on the type of document and contains information on how the different building blocks of the document are related.

Information Extraction The information extraction phase consists in extracting useful information from the logical document structure. For many document recognition tasks, this step is straightforward provided that the logical structure allows to address the objects of interest directly. For other applications, however, the information extraction phase is much more complex. Examples of such applications are information indexing, building of knowledge bases or natural language processing.

Not all of these steps are always necessary. For example, if the goal of the document recognition application is to restore the electronic form of a document in order to reuse it, the information extraction can be skipped. The result of such an application is the logical structure.

1.2 Logical and Physical Document Structure

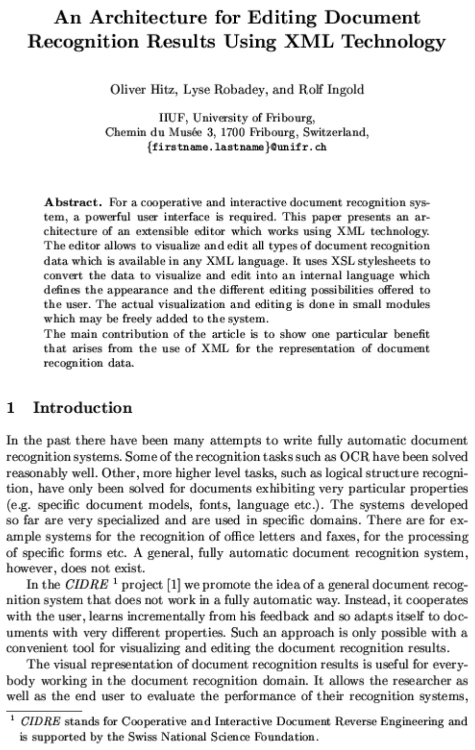
Two fundamental data structures used during the document recognition process have been mentioned: the logical and the physical document structure.

The logical document structure contains objects independent of formatting aspects. For example, consider the document image of a scientific article shown in figure 1.2. The corresponding logical structure is sketched in figure 1.3. This structure shows the objects the document is composed of, their role in the document (title, paragraph etc.) and how they are related (relations are hierarchical in this example, but they can be much more complex graphs in the general case). The logical structure does not specify any properties related to formatting. For example, there is no mention of font attributes, or of paragraph widths. Also the fact that the document title is printed on two lines is not reflected in the logical strucutre. This depends only on the page size.

The physical document structure contains descriptions of objects that can be found in the document image. Figure 1.4 shows part of the physical structure of the document image mentioned previously. The physical structure includes generic objects such as text blocks, images, rules including all their respective content and so on without any hints about the role of the objects in the document. However, it may include additional information:

- font attributes: font family, size, style
- text justification: left, right, centered etc.
- location on the document image: absolute bounding box coordinates (or relative coordinates for nested boxes) for rectangular objects or polygon vertices for free-form objects

Sometimes objects in the physical structure are grouped together to form higher level objects. For example, consecutive text lines written with the same font attributes may be grouped together as text blocks.



An Architecture for Editing Document Recognition Results Using XML Technology

Oliver Hitz, Lyse Robadey, and Rolf Ingold

IIUF, University of Fribourg,
Chemin du Musée 3, 1700 Fribourg, Switzerland,
{firstname.lastname}@unifr.ch

Abstract. For a cooperative and interactive document recognition system, a powerful user interface is required. This paper presents an architecture of an extensible editor which works using XML technology. The editor allows to visualize and edit all types of document recognition data which is available in any XML language. It uses XSL stylesheets to convert the data to visualize and edit into an internal language which defines the appearance and the different editing possibilities offered to the user. The actual visualization and editing is done in small modules which may be freely added to the system.
The main contribution of the article is to show one particular benefit that arises from the use of XML for the representation of document recognition data.

1 Introduction

In the past there have been many attempts to write fully automatic document recognition systems. Some of the recognition tasks such as OCR have been solved reasonably well. Other, more higher level tasks, such as logical structure recognition, have only been solved for documents exhibiting very particular properties (e.g. specific document models, fonts, language etc.). The systems developed so far are very specialized and are used in specific domains. There are for example systems for the recognition of office letters and faxes, for the processing of specific forms etc. A general, fully automatic document recognition system, however, does not exist.

In the *CIDRE* [1] project [1] we promote the idea of a general document recognition system that does not work in a fully automatic way. Instead, it cooperates with the user, learns incrementally from his feedback and so adapts itself to documents with very different properties. Such an approach is only possible with a convenient tool for visualizing and editing the document recognition results.

The visual representation of document recognition results is useful for everybody working in the document recognition domain. It allows the researcher as well as the end user to evaluate the performance of their recognition systems,

[1] *CIDRE* stands for Cooperative and Interactive Document Reverse Engineering and is supported by the Swiss National Science Foundation.

Figure 1.2: A document image of a scientific article.

1.3 Electronic Documents

Documents are not always printed, some remain in electronic form throughout their lifetime. The reason for this is that the electronic form offers possibilities a printed document does not. It is for example possible to quickly search even huge electronic documents or document collections for specific words or phrases. With paper documents, this is infeasible even for small documents. Electronic documents can also facilitate the navigation between parts of the document or between different documents, for example with hyper links.

It might seem that for such documents, the whole document recognition problem is solved. If documents are available electronically, why would we need to recognize them? And, as more and more documents get published in

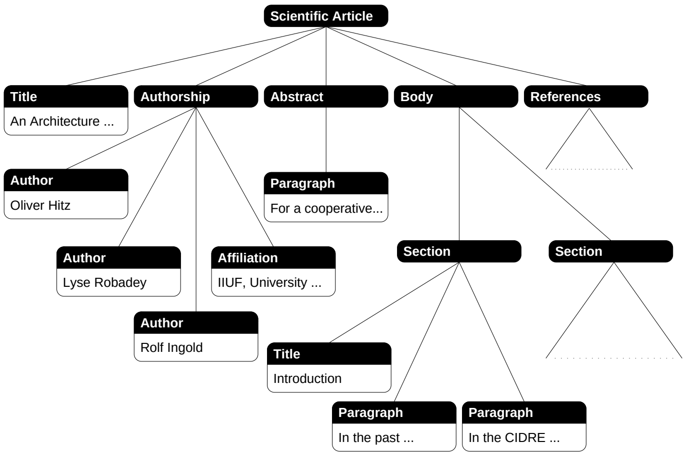


Figure 1.3: Logical structure of the document image of figure 1.2.

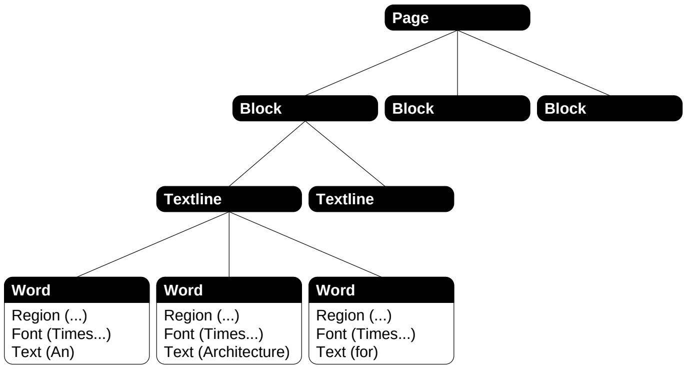


Figure 1.4: Physical structure of the document image of figure 1.2.

their electronic form, what will the use of document recognition be in future? The following two sections will give answers to these questions.

1.3.1 Paper Documents vs. Electronic Documents

There are many different types of electronic documents, each with advantages and disadvantages with respect to document recognition.

The lowest form of electronic document is the document image. It does not contain any symbolic information for facilitating the recognition process. Document images that have not been produced by scanning but by rendering are called *synthetic* document images. Compared to scanned document images, such images are special because they do not contain the typical noise that usually gets added during the scanning process. Recognizing synthetic document images will produce more accurate results than scanned document images.

Due to the lack of advantages over paper documents and their size, electronic documents are rarely made available in image form. Some document formats represent a formatted physical structure. The rendering is delayed to the instant when the user wishes to consult the document. This type of documents require a special program to be installed on the user's computer to interpret the document and render its content depending on the physical characteristics of the user's output device. This results in different rendering results for different users. A very widespread example of such a format is Adobe's Portable Document Format (PDF) [37].

At least since the invention of the World Wide Web, also the document formatting phase is shifted towards the moment when the user wishes to consult the document. The language of the World Wide Web, HTML [74], was originally designed to hold the logical structure of documents so that the content could be formatted by the user agent (i.e. the web browser) depending on the capabilities of the user's equipment. However, many people and organizations publishing documents on the Web were not satisfied with the level of control they had on the appearance of their documents. This is why HTML has been subject to comprehensive modifications and extensions and offers many features for specifying the exact appearance of documents, making it a language for representing the physical document structure rather than the logical structure.

To remedy this mistake, Cascading Style Sheets [71] (CSS) were introduced. CSS allow to keep information related to the appearance of documents outside the document in question and thus separate content from presentation. This is a very important addition, but until today, many web authors still do not use them to their full extent, probably also because not all features of CSS are supported by all of today's browsers (not to speak of compatibility problems).

But the trend towards representing the logical structure of electronic documents continues. The XML family of technologies is a huge step forward into this direction. Without doubt, XML will replace HTML more and more. It allows to separate content and presentation, and therefore offers the best of the two worlds. The appearance of documents can be fully controlled while at the same time the content is described in a logical structure that has all the advantages of a high level structure (e.g. indexing, archiving).

1.3.2 From Document Recognition to Document Restructuring

With electronic documents, individual recognition steps can be replaced by simpler programs that extract the wanted information from the document files in question. However, the document recognition problem is far from going to be solved by electronic documents as new problems are introduced.

The numerous different formats of electronic documents is becoming a serious problem. Already now there are dozens of document file formats, all of them with different versions. It is easy to find documents that are a few years old on the Internet - the real challenge is to find the programs that are able to visualize them. Or, even worse, programs that converts documents into useful formats.

Such conversion problems might get solved by using document recognition techniques in future. If we assume that there exists a program for visualizing a given document, there is, on the one hand, a way to produce a document image. On the other hand, we have document recognition technology knowing how to deal with document images. The document image thus becomes a pivotal format to build a higher level structure (see 1.5).

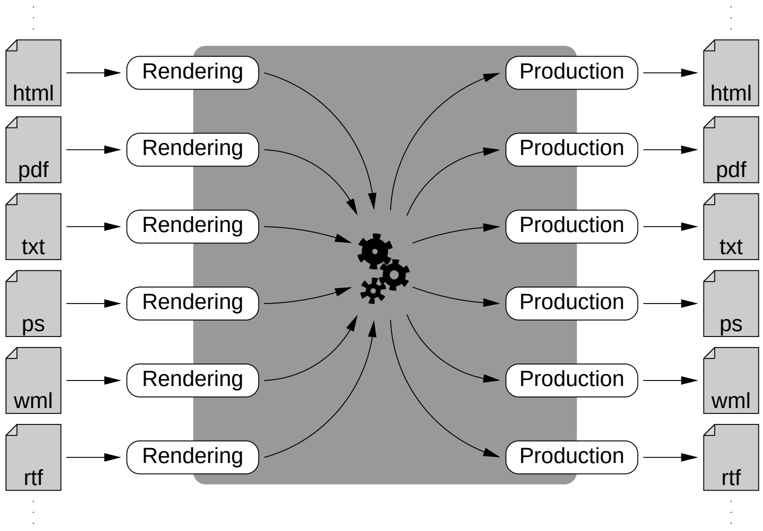


Figure 1.5: A document recognition engine used for document conversion: The document image is the pivotal form common to all document formats.

It is not necessary to go as far as to the document image in order to benefit from document recognition. Even if documents are described in a format such as XML, data sometimes needs to be transformed to fit into another structure. This leads us to the problem of document restructuring, which is something that is regularly done during the document recognition process. The same techniques that have been used for restructuring paper documents into electronic documents can be adapted and used for restructuring already structured documents.

1.4 Document Recognition Application Domains

Contrary to traditional document recognition systems, interactive recognition systems are designed to be more versatile and have therefore more application fields. In order for the reader to better understand what kind of document recognition system we mean by interactive document recognition system, the following sections present possible applications where such systems could be used.

1.4.1 Office Applications

In offices there are numerous possible application fields for document recognition systems. Some applications aim at the recovery of the electronic format of printed documents in order to be able to edit them, others target at the indexing and archiving of documents or the support of workflows [52] by automatically routing documents (e.g. incoming faxes or email) to the responsible officials. Common points among these applications are:

- variable physical structure
- variable logical structure
- variable document formats (paper, electronic formats)
- low rather than bulk volumes

Tuning document recognition systems is a task involving a lot of expert and domain-specific knowledge. Tuning an automatic document recognition system for office applications is therefore only profitable if a significant number of documents of the same type and format are to be recognized and if the document types can be clearly specified beforehand.

An interactive document recognition system could lower the cost of tuning by letting the end users show the system how to recognize an unknown document. The performance of the system should so gradually improve.

1.4.2 Postal Address Recognition

Postal address recognition [68, 70] has always been one of the main applications for document recognition systems. Some of the main characteristics of this application are:

- strict logical structure
- highly variable physical structure
- handwriting as well as machine print
- bulk volume, very high throughput (real-time)
- contextual constraints that can simplify the recognition

Since high throughput is required, fully automatic recognition is indispensable. However, an interactive system could be used in order to train the system off-line whenever addresses with yet unknown physical formats are met. An operator could show the system how to correctly recognize such addresses so that the system improves with use.

1.4.3 Check Reading

Check reading [44] is an application where the accuracy is critical. Because checks are dealing with money directly, errors are not acceptable. Other characteristics are:

- strict logical structure
- accuracy is very critical
- bulk volume, very high throughput (real-time)
- contextual constraints (amounts are typically written as numerals as well as written words and can thus be cross-checked)
- handwriting as well as machine print

The accuracy requirement is enforced by the fact that amounts are written as numerals as well as written words. If these two numbers differ, the check is rejected and needs to be processed manually by an operator.

1.4.4 Document Archiving

With the advent of the Internet, document archiving has gained a new meaning. The Internet makes huge archives of indexed documents at everybody's disposal. Documents that were previously only available as paper documents are being analyzed with document recognition techniques [9] and indexed. There are search engines for specific types of documents whose analysis goes far beyond simple text indexing (e.g. CiteSeer [1] [30] analyzes the citations of electronically accessible scientific literature). Typical characteristics of document archiving applications are as follows:

- variable logical structure
- variable physical structure
- bulk volume, but throughput not very important
- accuracy not critical

1.4.5 Web Document Analysis

The vast amount of information readily available in the Internet offers many possibilities not only for the average user. Questions such as: "What online shop offers product x at the best price?" can be answered by consolidating data from different online shops.

The predominant format for presenting information on the Internet is the Hypertext Markup Language, HTML. While the fact that is a textual format suggests that analyzing it should be easier than doing so with image formats, reality looks different. Even though HTML may contain text that does not need to be recognized with OCR techniques, the average web page is much more complex: HTML is a multimedia format combining text with images, which often contain important parts of the text. Moreover, much of the really useful information is often arranged in complex tabular structures. This makes it hard to distinguish real tables from tables that are used for reasons of design only [36].

Common characteristics of Web Document Analysis applications are:

- variable physical structure
- variable logical structure

[1] `http://citeseer.org`

1.4.6 Personal Use

Personal use of document recognition systems is restricted to few applications: recovery of the electronic format, archiving and indexing (e.g. invoices). The key characteristics are:

- variable physical structure
- variable logical structure
- variable document formats
- very low volume

Commercial OCR systems can solve most of the demand for commonly used document types. As soon as the documents get non-standard or more complex, document recognition systems need to be fine-tuned what makes them no longer feasible for personal use anymore.

1.5 Document Models

There are many different application domains for document recognition. The applications range from postal address reading, over the indexing of old paper documents to the recognition of complex technical documents containing heavily cross-referenced text as well as graphical elements. The requirements for these applications are also very varied. While some applications require extremely high throughput, others necessitate very high accuracy or high adaptability to varying document types.

All document recognition applications require knowledge about the types of document that they need to recognize. This knowledge is called *document model*. Document models are not always expressed explicitly. Algorithms that only work with one given type of documents have their document model embedded in the algorithm. Other algorithms have hard coded parameters (e.g. thresholds) that represent part of the document model. Some algorithms express document models explicitly. The document models can either be static and predefined (e.g. Tao Hu in [35]), or dynamic and evolving. An example of such an algorithm is 2(CREM) (presented in section 2.2.8).

Document models have the very unpleasant property to never be complete. No matter how much time has been invested for tuning an algorithm to a certain kind of documents, sooner or later a document that does not fit into the frame turns up and the document model needs to be adapted in order to also recognize this new document.

The creation and maintenance of document models is a very tedious and costly process. In traditional recognition systems, maintenance modifications of the document models have a big impact on the overall system and they generally require not only knowledge of the documents to recognize, but also knowledge of the specificities of the underlying algorithms. The goal of interactive document recognition system is to integrate the maintenance of document models directly into the document recognition system and by making it accessible to the non-expert user. Instead of creating a document model once and for all, the model evolves depending on the processed documents and the system improves with use.

Interactive document recognition does not fully exclude automatic recognition. When a document can be recognized by a system, there is no need for the user to step in. User intervention is only required when the system cannot decide, when it is uncertain (i.e. because of too low confidence values) or when detects contradictory results.

1.6 Automatic vs. Interactive Document Recognition

Every application domain where document recognition is used, has a special set of requirements. In some domains, high throughput or high accuracy is important, whereas in other domains the ability to adapt to variable documents is crucial.

For some of these domains, the recognition does not need to be fully automatic. If a document has to be recognized only to save somebody a few minutes of typing for example, this person can as well assist the system in finding the correct result instead of having it done completely automatically. These helping hints from the user can then be integrated into the system to be reused at a later time.

Systems working like this adapt to changing documents and are thus able to learn, as opposed to automatic systems in which the parameters of the individual recognition parameters are tuned by experts prior to operation. This tuning consists in adapting the system to the kind of documents that should be recognized. This is a very tedious and expensive task which, in order to yield useful results, requires many example documents covering all the different possibilities the system might encounter.

1.7 Web Based Document Recognition

Web based document recognition [63] is a future application inspired by the interactive document recognition idea. It lets users access remote document recognition services through the World Wide Web. In addition to the basic functionality offered by interactive document recognition systems, it provides means for sharing document models among different users.

If a document model for a given document type does not yet exist, a new model can be created by the user. Likewise, if a document model is not complete enough for a specific document, the user can extend it and give the result back to the other users, improving the system. Like this, users cooperate not only with the system, but also with the other users.

A web based document recognition system can be characterized primarily with the following properties:

- highly variable physical structure
- highly variable logical structure
- variable document formats

A web based document recognition system opens new questions unknown in traditional systems. First of all, since people work remotely, there is the problem of privacy. This problem can probably be solved with standard Web techniques (Secure Socket Layer SSL [28], or, the more recent Transport Layer Security, TLS [21]). Another, important problem is the security of models. Since users can extend models, there needs to be a mechanism that ensures that models converge towards something useful so that users cannot in some way pollute models with erroneous data or pull a model towards an unintended document type while degrading the recognition performance for the original document type. Such interesting questions will need to be addressed by future research in order for web based document recognition systems to become reality.

1.8 Goal of this Thesis

The goal of this thesis is to present xmillum, an approach and framework for structuring interactive document recognition systems. The approach is heavily focused on the use of XML as data representation format. This allows to use existing XML technology in a very advantageous way. The claims made in this thesis are backed with a complete working prototype.

The remainder of this thesis is structured as follows:

- Chapter 2 gives an overview of the state of the art in document recognition. The document recognition process is decomposed into basic recognition steps. Furthermore, the chapter presents several projects that are somehow related to this thesis.

- Chapter 3 presents the technology that was used to create our software framework as well as some technical design issues.

- Chapter 4 presents the requirements and goals that have driven the design and development of xmillum.

- Chapter 5 presents the idea behind the xmillum framework and shows how this idea has grown into a complete prototype for creating interactive document recognition applications. Furthermore, a generalization of the xmillum idea is proposed.

- Chapter 6 presents the xmillum prototype implemented. It shows what has been implemented and how it can be used to created interactive document recognition applications.

- Chapter 7 summarizes and concludes the thesis and gives directions for future developments.

Chapter 2

State of the Art in Document Recognition

2.1 Document Recognition Phases

We have seen that document recognition is the opposite process of the document production process. However, the document production process cannot be easily reversed. The recognition process is much more complex than the production process.

During the document production process, the information per elementary object decreases gradually. It starts from a very high level information representation and ends with pixels as elementary objects, where every pixel simply holds a value of black or white for binary documents, or a grayscale or color value for color documents. Reversing this process is non-trivial and involves many different techniques.

This section gives an overview of some of the most important methods, along with references to the corresponding research papers.

2.1.1 Image Preprocessing

The goal of the preprocessing phase is to get rid of all kinds of noise introduced in the scanning phase. Different types of noise may have been introduced into the image. There are various sources of noise: dust in the scanner, irregularities in the scanner's optics/mechanics/lighting, skewed documents or bent pages (typically when scanning book pages). A very complete analysis of different document degradation models is available in the thesis of Tapas Kanungo [42].

In order to improve the further recognition steps, this noise needs to be dealt with. Various image processing filters are normally applied to reduce the impact of noise. The exact filter that need to be applied depends entirely on the nature of the noise:

Image Despeckling removes unwanted black (or white) pixels from the document image and renders it smoother. Many filters, every one with its own advantages and disadvantages, exist for this purpose (blur filters, median filters etc.).

Image Deskewing is done using an estimation of the skew angle. Common estimation techniques are based on the hypothesis that text is aligned on straight, parallel lines. Henry Baird [6] and Wolfgang Postl [59] analyze projection profiles to exploit this hypothesis and Sargur Srihari and Venu Govindaraju [69] use the Hough transform [33].

A very thorough review of 23 papers related to document image deskewing has been written by Rolando Cattoni *et al.* [12].

Binarizing images is often necessary required because many of the document recognition algorithms work on binary images only. Because this process reduces information present in the image, it needs to be done carefully not to reduce too much information. The algorithms for binarizing range from thresholding with globally fixed thresholds (global to a whole page) to more sophisticated locally fixed thresholds. A survey of thresholding techniques has been written by Prasanna Sahoo [64].

2.1.2 Image Segmentation

A document image is a simple two-dimensional pixel matrix. The size of this matrix depends on the page size of the document and the scanning resolution. A problem when working with images is complexity. Since images are two-dimensional, algorithms working on images have in general a complexity that depends on the number of pixels (something such as $O(f(width \times height))$ where $f(N) = N$ or worse). Because of the tradeoff between computation cost and accuracy, we need to reduce computation cost in order to increase the accuracy. The goal of the image segmentation phase is to identify smaller regions of interest for the succeeding recognition steps. There are bottom-up as well as top-down approaches to solve this problem.

Connected Components Analysis groups neighboring black pixels together to form higher-level objects. The idea behind this method is that touching pixels belong to the same object. Unfortunately, some characters are composed of multiple connected components (e.g. the lowercase *i*), so that the connected components analysis results in an over-segmented image. On the other hand, characters may touch each other (especially with small print on scanned documents), resulting to under-segmentation. Connected components analysis is for example used by Lyse Robadey in [62] to segment newspaper document images according to predefined rules or by Antoine Azokly in [4].

Run Length Smoothing Algorithm (RLSA) presented by Kwan Wong *et al.* [84] smears closely located pixels. Combined with the connected components analysis, RLSA reduces the number of resulting regions. It is used to detect words, text lines and other objects. For instance, using a smearing distance (i.e. the minimum distance for two pixels to be connected together) smaller than the distance between words, but greater than the distance between characters of the same word, a word segmentation can be achieved.

Projection Profiles allow to find separations between objects by analyzing the regularity of peaks and valleys. This method is for example used by Sargur Srihari and Venu Govindaraju [69].

X-Y Cut described in George Nagy's paper [56] is a typical top-down method that cuts the document image recursively along separations between objects (as obtained by projection profiles for example). A multi-column document for example is first cut into individual columns which are then cut into single text lines.

Document Spectrum (docstrum) by Lawrence O'Gorman [58] characterizes the relative locations between neighboring connected components on the document image. This information is then used to cluster connected components to higher level objects.

2.1.3 Zone Classification

In the succeeding document recognition steps, the image zones resulting from the image segmentation phase are processed individually according to their type. The goal of the zone classification step is to classify the zone into different types: text blocks, tables, graphics, photographs etc.

Numerous approaches for classifying zones exist. From features extracted from the image zones, standard pattern recognition techniques are applied. Feature extraction consists in "extracting from the raw data the information which is most relevant for classification purposes" (Devijver and Kittler [20]).

Nearest Neighbors Classification is an ancient classification method first analyzed by Cover and Hart [14]. This classification technique is very simple, yet effective. It labels an object with the class that is most often represented among the *k* nearest neighbors of the object in the reference set.

Bayesian Classification as presented in Duda and Hart [24] assumes that the classifiction problem is posed in probabilistic terms and that the relevant parameters such as the a-priori distribution are known. Under these circumstances Bayesian classification is proved to be optimal.

Decision Trees are hierarchically structured decision rules. In [66], Sivaramakrishnan *et al.* classify zones into one of nine different classes using a decision tree.

Neural Network is an information-processing paradigm inspired by the way the parallel structure of the mammalian brain works. A neural network is composed of a large number of simple interconnected processing elements that tied together with weighted connections. A learning phase is required to adjust these weights systematically in order to get a working neural network.

Le *et al.* [49] compare four different neural network models for classifying zones into text and non-text.

2.1.4 Optical Character Recognition

OCR is without doubt the best investigated and also best resolved document recognition task. The goal of OCR is to recognize the characters on the document image. In other words, OCR converts an image to ASCII or UNICODE.

Good overviews of OCR have been written by George Nagy [55] and by Shunji Mori *et al.* [54].

Template Matching is probably the most obvious character classification approach. The classification features are the pixels themselves. Characters to classify are matched to known prototype templates, which may or may not be weighted. This technique is easy to understand, but is very sensitive already to minor differences between the templates and the characters to recognize. Examples of systems using template matching are presented by Rolf Ingold [40] and Gary Kopec [46]. George Nagy and Yihong Xu [57] present a system for extracting prototypes from unsegmented text images.

Statistical Classification works by extracting numeric features from the character bitmaps or from transformed versions of the bitmaps. These features are then classified using standard classification techniques (nearest neighbors, neural networks etc.). By selecting the features carefully, classifiers can be made invariant with respect to deformations, character size, font style or to the font family.

Structural Classification techniques work on structural character features, defined in terms of the topology of strokes, holes, concavities, stroke junctions etc. For example, the character "A" is composed of three straight strokes with three junction points. At the heart of structural classification systems is a rule base which does the actual classification. Because the construction of a good rule-base can be very time-consuming, several people have tried to use structural features together with statistical methods (Henry Baird [5]).

Linguistic Information such as n-grams, language or writing style can be introduced to improve recognition accuracy, often as a post processing step. Such information extends the perception from local properties to contextual features, but requires additional knowledge about the content. Linguistic information is generally used in conjunction with other techniques to enhance the recognition accuracy. In [18], Andreas Dengel *et al.* present various techniques for improving OCR results.

Multiple Classifiers are frequently applied to overcome limitations of single classifiers. By comparing the results of different classification methods, possible problems can eliminated to improve the recognition accuracy (Louisa Lam *et al.* [47]).

2.1.5 Optical Font Recognition

Font attributes are a useful hint to determine the role of a text object. When the font attributes are detected prior to character recognition, i.e. *a priori* font recognition in contrast to *a posteriori* font recognition, this knowledge can even be helpful for the character classification. Knowing the font attributes of a text object allows to choose the set of features that best matches this font.

Statistical Approaches are used for *a priori* font recognition systems. Various features extracted from the bitmaps of a text line or text block can be used along with classification algorithms. The *ApOFIS* system by Abdelwahab Zramdini [85] uses a Bayesian classifier to recognize the fonts, Min-Chul Jung *et al.*'s [41] system works with a neural network.

Template Matching can be applied when the text is known. This approach is very similar to character recognition by template matching. By comparing the text image with a rendered version of the text in all available fonts (or better: in the most probable fonts), the correct font can be detected. This has been proposed by Frédéric Bapst in [7].

2.1.6 Logical Structure Recognition

By combining the results of the preceding recognition steps, the logical structure recognition can be done. The logical structure depends strongly on the type of document. The structure of a scientific article, for example, is completely different of the structure of a newspaper. Whereas the former is basically a sequence of paragraphs, the later is a set of different kinds of articles. There is not a universal logical structure, it is therefore not possible to create a logical structure recognition system that works for all kinds of documents.

Rule-Based Systems contain precise descriptions of the mapping between the physical and the logical structure of documents. Manual creation of rules requires a lot of expert knowledge and is a very costly process (Tao Hu and Rolf Ingold [34]). Ideally, rules should be generated interactively and iteratively, as presented by Lyse Robadey [61].

Statistical Methods create rules from statistics gathered from example documents. Statistical methods are more flexible than purely rule-based systems since they allow easier modification of the rules. By presenting new example documents to the system, the statistics can be modified and the rules change. Because statistical methods work with probabilities, they are not well suited for situations where many special cases and exceptions need to be handled. A logical structure recognition system using probabilities is presented by Rolf Brugger [11].

2.2 Related Systems

The following sections present projects that are somehow related to this thesis, namely interactive environments and interactive document recognition systems.

2.2.1 CIDRE

The *Cooperative and Interactive Document Reverse Engineering* project (CIDRE), which is best described in Frédéric Bapst's thesis [7] tries to revaluate the role of the user in the document recognition process.

Bapst shows several aspects of the interactive document recognition problem: mechanisms to drive the dialog between human and machine, data management issues, a system design, and a cost model to measure the performance of interactive document recognition systems.

Thanks to an approach based on the multi-agent paradigm, the designed system is both modular and decentralized. But since this work dates from a time where standardization (for example XML) was not as advanced as today, the Web did not offer the same possibilities as it does nowadays and platform-independent languages like Java were not yet as widespread as today, the effort for create a test framework was much more low-level at that time. For creating a prototype, much work was invested for coupling low-level libraries handling communication between processes distributed on different machines and high-level programming languages for quick creation of graphical user interfaces.

2.2.2 Handwriting Understanding Environment (HUE)

Chris Cracknell's Handwriting Understanding Environment (HUE) [16] is a Tcl/Tk framework supporting the rapid development and reuse of handwriting and document analysis systems. It is open source and free software and runs on Unix as well as on Windows platforms.

HUE provides an extensive collection of components and data types programmed in C++ that can be combined with the Tcl/Tk scripting language for rapid development. This two-level programming model where a low-level language is coupled with a high-level scripting language yields fast and flexible programs.

The HUE component library contains a full palette of low-level image processing components. There are components for image input/output, image filtering, color mapping and common operations such as rotation, cropping, resizing and so on. Other components feature histogram operations to with projection profiles or color histograms. Binary connected components can be extracted and processed further. An OCR engine based on an n-tuple classifier is included as well.

HUE can be extended with additional components and data types.

2.2.3 TrueViz

Tapas Kanungo's TrueViz [43] is a ground-truth and metadata editing and visualization tool. It is mainly directed towards multilingual OCR. As such, it addresses two issues:

- it provides a public domain annotation tool that can be used by everybody
- it defines an XML-based format for representing groundtruth.

TrueViz manages groundtruth for the physical document structure mainly and offers the basic entities page, zone, line, word and character along with all the necessary attributes for further characterizing these objects (location on the document image, font, language, text alignment etc.).

TrueViz is developed in Java and therefore platform independent. Its user interface shows two views: a WYSIWYG view where the document entities are laid over the document image, and a tree view that shows the hierarchical structure of the data shown in the WYSIWYG view. A nice feature of TrueViz is the representation of the logical reading order. Arrows between the individual entities show the reading order.

The textual groundtruth is represented in Unicode and is therefore multilingual. In addition to visualizing this groundtruth correctly, TrueViz allows the user to edit multilingual groundtruth using alternative keyboard mappings. It also features a search facility that is helpful for correcting groundtruth.

The hierarchical data structure is read from and writtten to XML files. These files exactly represent the hierarchy of the document along with the attributes.

2.2.4 WISDOM++

WISDOM++ [2] by Oronzo Altamura *et al.* is a complete document recognition system encompassing all the steps necessary for transforming a scanned paper document into a format suitable for viewing with an Internet browser. The goal of the system is to produce a web-accessible format that is as close to the initial document as possible.

In WISDOM++, this is done in a five step process:

Document analysis involves preprocessing of the document image in order to detect the skew angle and to eliminate noise. The image is then segmented into basic blocks by using the Run Length Smooting Algorithm (RLSA) with adaptive smooting thresholds. Subsequently, the resulting blocks are classified using built decision trees automatically built from a set of training samples. The decision trees can be modified interactively at classification-time, allow thus incremental learning. WISDOM++ distinguishes the following types of blocks: text block, horizontal and vertical line, picture and graphics. In a further layout analysis step, the blocks are arranged in a hierarchy using a bottom-up algorithm.

Document classification consists in associating a document class to the document. This mapping is done with a rule base built automatically from training samples. The rule base is created by the administrator of the system and out of the control of the actual user.

Document understanding is similar to document classification, with the difference that the mapping is done with all the individual page layout components instead of the page itself. In other words, the basic building blocks of the document are associated to logical objects.

Text recognition is then applied to selected objects resulting from the document understanding process. Depending on the type of document and the kind of application, it is not necessary to extract the text from all objects. If the user wants to recognize scientific articles, he may not be interested in the page headers and footers, for example.

Text transformation is the last step performed by WISDOM++. The data resulting from the previous steps are now transformed to XML. All elements in this XML structure have no style-related attributes (i.e. no font, alignment or position information), such information is entirely stored in external Cascading Stylesheets (CSS) which are only used for presenting the documents. The CSS files are not created automatically, but WISDOM++ assists the user in defining CSS files for every class of documents. The XML files can then be transformed to HTML or other viewable formats relatively easy using XSLT stylesheets.

WISDOM++ is targeted towards an end user. It has multiuser capability, with a per-user rule base. A special privileged user is the administrator who is responsible for initially training the system. WISDOM++ runs on Windows platforms only and it requires a special version of the MS-Access database installed, as well as an external OCR. The system is freely available for research, but without source code what makes it not possible to be extended.

2.2.5 Illuminator

Illuminator is an application for viewing and editing files in DAFS format (see section 3.1.2). It offers access to all of the features of the DAFS format through a graphical user interface. It is an X11 application and runs on Unix systems only.

Illuminator has different modes for visualizing data:

Image mode is a WYSIWYG-style visualization mode. It shows the document image with overlaid entities. New entities can be created and existing entities can be modified.

Out of context is a view that extracts all objects of a given type from the document puts them onto a separate page. This allows the rapid generation of reference sets by putting for example all a's on one page, b's on another etc.

Flagged objects is a variation of the *out of context view* that extracts all flagged objects and puts them onto a separate page. One application of this view is the correction of OCR results. All objects with a confidence value below a specific threshold are put onto a separate page where they can be corrected.

Text is a simple view showing only the concatenated textual content of all objects in a document. This visualization mode is useful when the document image is no longer of interest but only the recognized content.

Illuminator is not a document recognition system itself, but it could be part of an interactive recognition system. With the proper configuration, it could be used to visualize and edit specific stages in the document recognition process. However, since Illuminator relies on a non-standard format, it is not an ideal candidate.

2.2.6 Style-Directed Document Recognition

Lawrence Spitz's style-directed document recognition [67] system is an end-user oriented system. Document models are constructed interactively and are called *styles* in this system. Styles are stored in XML files.

A style is a graph of rectangular zones. This graph is constructed manually by the user. He selects the regions of interest on a document image and labels then with logical names. The coordinates of the resulting zones are attributed with three values: *absolute* for coordinates that are constant values for the corresponding document class, *relative* for coordinates that are relative to the coordinates of some other objects (e.g. a paragraph that follows a title) and *variable* for coordinates whose values are unpredictable. Using graph isomorphism, the resulting graph is then used for mapping a segmented document to a specific document class and to recognize the logical names of the individual segments.

2.2.7 Generalized n-Grams

Rolf Brugger's Generalized n-Grams [11] is an technique for recognizing document structures using probabilities. It is based on n-grams, a natural language model that assumes that only the previous $n-1$ words in a sentence affect the probability of the next word. The generalized n-grams model extends this idea from the linear structure of sentences to the hierarchical structures of documents.

In the generalized n-grams model, a document structure is represented using probabilities of local tree node patterns. The model supports interactive incremental learning and is therefore well suited for interactive document recognition systems.

2.2.8 *2(CREM)*

Lyse Robadey's *2(CREM)* [61] (Configuration REcognition Model for Complex Reverse Engingeering Methods), is a typical classification technique for use in interactive document recognition systems. It allows the user to incrementally and interactively build document models.

2(CREM) is very flexible and can be used for different classification tasks. In the project described in [61], it has been used for recognizing newspapers, which are documents of very high complexity. *2(CREM)* has been used for classifying line segments, classifying frames, merging text lines to blocks and for logical labeling of blocks.

The technique uses a two-part model:

Static Model describes the characteristics that are of interest for the classification. Characteristics include object attributes (size, typographic information etc.), attributes of the neighbors and relations between objects. This information is used during the feature extraction phase which produces so called *configurations*.

Dynamic Model contains reference configurations also called *patterns* with a subset of the characteristics defined in the static model.

The static model depends on the nature of the objects to be classified and is built prior to classifying anything. For the two applications logical labeling of blocks and merging of text lines into blocks, the static model is completely different. The logical labeling application has blocks as its central object of interest, whereas for the lines merging problem the object of interest is a couple of neighboring text lines. In this case the system does not classify the text lines themselves but rather classifies the relation between the text lines into the two classes *join* or *separated*. It is then up to an additional program to actually join the text lines.

The dynamic model is constructed by the user. At the beginning, it is completely empty and the system is unable to classify anything. The user then starts to classify objects manually. The corresponding configurations are recorded in the dynamic model, making them *patterns* of the model and increasing the knowledge of the system. Instead of plainly storing new configurations, the system re-evaluates the model with every new addition in order to make it general enough so that it is able to cope with configurations that do not exactly correspond to patterns, but that are only similar. This is done by not taking into account all characteristics of a configuration but only a subset of them. In the case of conflicts, that is, when a configuration can be associated to multiple classes, user interaction is required and the model is re-evaluated with the new data.

At the moment, *2(CREM)* does not use any statistical information. That is, it does not count how often a specific configuration occurs or how often it is a specific characteristic that decides between two classes.

2.2.9 OfficeMAID

The OfficeMAID system, developed by Andreas Dengel's research group and described in [17], is designed for processing business documents (letters, orders, invoices etc.). It supports automatic mail delivery and can be integrated into a workflow management system.

OfficeMAID processes incoming documents in several steps. After extracting the physical document objects, a geometric decision tree is used to generate hypotheses concerning the location of logical objects. The hypotheses are then evaluated according to a statistical database for office mail and their validity regarding manually fixed global consistency rules is checked.

The method allows to incrementally modify the geometric decision tree and therefore to adapt to changing situations or environments.

2.2.10 *smartFIX*

Made by the same research group as OfficeMAID, *smartFIX* [19]—short for *smart F*or *I*nformation e*X*traction—is an ambitious interactive system. It is a commercial end-user oriented system designed towards assisting users in extracting data from paper documents.

smartFIX is a general system which was at the origin designed for German health insurance companies to extract billing information from medical bills. Since every medical bill needs to be inspected by a human operator, the goal of *smartFIX* is not to completely replace the operator, but to assist him. It cannot cope with all the different medial bill formats and bill types, but allows to recognize parts of these bills what can substantially reduce the human operator's workload. Even with no recognition results at all, its user interface can facilitate the processing of bills.

Besides standard low-level document recognition components, *smartFIX* uses specific high-level domain knowledge in order to achieve its goal. This knowledge can include for example the format of product codes (or diagnose codes in the health insurance domain), layout information (bills from certain domains may have special characteristics), customer databases or other external data sources that can give hints for the recognition process.

2.2.11 Qgar

Qgar, a free software reincarnation[1] of the ISADORA platform [23] by Philippe Dosch *et al.*, is dedicated to the analysis of technical drawings. It is composed of three components:

QgarLib is the core of the Qgar platform. A comprehensive library of C++ classes with basic processing methods and utilities that are the building blocks of document analysis applications. It contains class hierarchies for the following facilities: graphical objects (segments, arcs, rectangles etc), histograms, images (binary, gray level, float etc.), Qgar-specific data structures and file formats, convolution masks.

QgarApps is a collection of basic applications built using the QgarLib classes.

QgarGui is a graphical user interface that allows to control all the QgarApps applications interactively and to visualize and modify the results.

Applications inside QgarApps are "batch" processing modules that take several input parameters and generate output. In the current version, they do not allow interaction with the user during processing, but this might get added in a future version. When invoked with a special command line parameter, QgarApps applications display the required parameters in a machine-readable form. This output is used by QgarGui to create a user interface where the user can conveniently set all the parameters.

2.2.12 ACTI_VA

In his ACTI_VA project [65], Youssouf Saidali studies knowledge modeling and acquisition in the document image analysis domain. The project aims at making document image analysis knowledge easier accessible to the non-expert user.

[1]available at `http://www.qgar.org`

The ACTI_VA prototype is a Java application composed of a dynamic interaction module, a graphical user interface and a library of document recognition processing modules. Knowledge is represented in the form of scenarios that are constructed interactively by experts using a Wizard-like user interface. These scenarios can then be exploited by non-experts.

2.2.13 UIML

UIML [1] stands for User Interface Markup Language and is an XML language for defining user interfaces. This language is not directly related to document recognition, but nonetheless this thesis shares some of its ideas.

The goal of UIML is to provide an appliance-independent user interface language, i.e. a language that allows to simultaneously describe user interfaces for use on computers, mobile phones, handhelds and possibly other appliances. It should clearly separate a program's internal logic from the user interface, which is not always the case in today's programs.

A user interface described in UIML is a set of user interface elements, each with appliance-independent data and links to other interface elements with which it exchanges data. Depending on the appliance in question and the user category, these elements may be organized differently.

Appliance-dependent features of the user interface can be found in a separate section of the description. This section represents a sort of stylesheet that maps the appliance-independent elements to the appliance in question. An application running on a personal computer supporting Java may for example map button objects to the Java class implementing the buttons.

The interface communicates with the backend through events. Another section in the user interface description details the different events that the interface generates. These events may not only be sent to the backend, but they can also be used for communication between interface elements. For example, the backend may not be interested in events generated by a scrollbar that moves the items of a list. Such events can be handled by the user interface independently.

Chapter 3

Technology

The goal of the present chapter is to show technological aspects and design decisions that have been considered for this thesis.

The chapter starts with a study of different formats for the representation of document recognition data. Several well-known formats are presented and their advantages and disadvantages are compared. Then, a brief introduction to one of the cornerstones of this thesis, the XML family of technologies, is given. Finally, the chapter concludes with a short presentation of Java, the programming language that has not only been used for implementing the prototype but whose properties have also influenced significant design decisions.

3.1 Data Representation Formats

The data representation format is an important issue for a reusable and modular document recognition framework. Reuse takes not only place at the level of software components but also at the level of the produced data. Results produced by a recognition system need to be reused by other parts of the system.

In order to decide on a data representation format, a trade-off between a number of decision criteria needs to be found:

Complexity: A format that is too simple is not able to represent the data we would like to store. On the other hand, a very complex format may involve a non-negligible overhead, not in the performance sense but more in the sense of the ease of implementation and maintenance of the programs that are supposed to work with the data.

Software Support: The software support of a given format consists of programming languages, libraries, APIs and tools that support it. The more programming languages exist that support the format, the more flexible is the developer to choose the language that best suits his needs. Equally, the more tools that exist, the easier it is to work with the format without having to develop these tools every time some special processing of the data in question is required.

Extensibility: In order to guarantee that the format best suits the current and the future needs of its users, it has to be extensible. Data that is not of importance at the moment and is therefore not integrated in the format, may become more important in a future generation of the system, what causes an extension of the format. A format that is too restricted might not allow to insert future data items.

Along with the extensibility criterion, versioning becomes inevitable. The format needs to provide means for discriminating between different concurrent versions.

Compactness: The size of the produced files may not seem very important for many applications, but as soon as data needs to be archived, this issue can get more important. Some formats may be more compressible than others.

Also, some formats refer to image data outside the actual file, what can be an advantage if multiple files reference the same image. On the other hand, it can get more difficult if the data from one recognition task is distributed among different files (i.e. data files and image file).

Legal Issues: Last but not least, legal issues can influence the choice of a representation format. Although mostly incomprehensible from a purely technical standpoint, some algorithms might be patented by companies. This was shown by the legal dispute concerning the GIF format and, more recently, concerning the JPEG format. Choosing a 100% open format can help to avoid future problems.

A document recognition system produces various kinds of data: document image, physical and logical document structure, document models, statistical data etc. Whereas there are many standard file formats for image data (e.g. TIFF or JBIG), there are not agreed upon standards for the representation of the other types of data.

Our goal is to find a data representation format for the various sorts of non-image data. It may allow the representation of the document image also, but this is not our main concern. The following sections introduce several data formats commonly used with documents or in the document recognition domain.

3.1.1 Ad-hoc formats

Ad-hoc formats are formats that are created out of an immediate need. A freshly implemented algorithms produces a new kind of data that needs to be represented somehow. Such formats are in general not very complex, but they do not have strong software support since data is not shared between many tools. They often require converters in order to convert data into more standard formats that can be used by other parts of the system. Also, ad-hoc formats are mostly poorly structured and not easily extensible.

3.1.2 Document Recognition Oriented Formats

DAFS

The Document Attribute Format Specification [60] (DAFS) is a format specifically developed for document recognition systems. It is a result of the Document Image Understanding (DIMUND) project funded by ARPA and it is intended to be a standard for the representation of document images and the corresponding intermediate and final recognition results. More on the design of DAFS can be read in the paper from Dori et al.[22].

A DAFS document is composed of *entities* such as "document", "chapter", "block". These entities can be fit into hierarchical relationships. For instance, a "document" can contain "chapter" entities and a "chapter" can contain "block" entities. Associated to entities are *properties* that describe their characteristics. Users can extend DAFS by defining their own entities and properties.

A special characteristic of DAFS is the fact that an entity can be a child of multiple parent entities. Such *shared* entities allow several concurrent hierarchies to be represented in the same data structure. DAFS is therefore not limited to the physical document structure, but can at the same time also represent the logical structure. Another use of these shared entities is the special *OR*-type relationship that can be used to represent alternative possibilities of results originating from different recognition algorithms or results where a recognition system is not 100% sure (e.g. an OCR algorithm that is not sure if a given character is "l" or "1").

DAFS is implemented as a link library usable from C or C++ programs without further work. In order to use the format from other programming languages, much more work needs to be done.

Since DAFS stores the image and the recognition data in one data file, DAFS is a binary format. A textual ASCII or UNICODE version of the format is mentioned in the documentation, but it does not seem to be implemented in the current version.

Further software support is provided with the Illuminator (see 2.2.5) application, as well as some small utilities that mainly extract text nodes from DAFS documents. Illuminator is a powerful viewer/editor for DAFS.

3.1.3 General Data Representation Formats

SGML

The Standard Generalized Markup Language [25] (SGML) is a standard for creating structured, interchangeable documents. It allows to specify the structure of documents using a special grammar called a Document Type Definition (DTD). Documents can then be verified to check if they conform to a given grammar.

SGML has a strong software support. There are public domain parsers that facilitate the integration of SGML into many programming languages and tools that allow you to do further processing with SGML data.

Of particular interest is the Document Style Semantics and Specification Language [27] (DSSSL), which is a stylesheet language for transforming and formatting SGML data. DSSSL is composed of two languages: a transformation language that transforms SGML data conforming to one DTD into SGML data conforming to another DTD, and a style language controlling aspects of the formatting process. Figure 3.1 illustrates this idea which will be exploited further in this thesis.

SGML is flexible and extensible. New versions of a given data structure can be created by modifying the DTD. The example that best illustrates this is the language that has driven the development of the World Wide Web, the Hypertext Markup Language (HTML), an instance of an SGML language. Since its first version, HTML has evolved significantly in order to satisfy the requirements of the World Wide Web.

Despite all these positive points, SGML is a complex language. Designing new document types requires a profound knowledge of SGML and developing SGML applications is even more complicated. This is one of the main reasons why there has never been a real breakthrough. Nowadays, SGML gets largely replaced by XML, because XML is much simpler and more intuitive, a consequence of which is that there are much more tools available that support XML than SGML.

As far as document recognition is concerned, SGML has been used in the

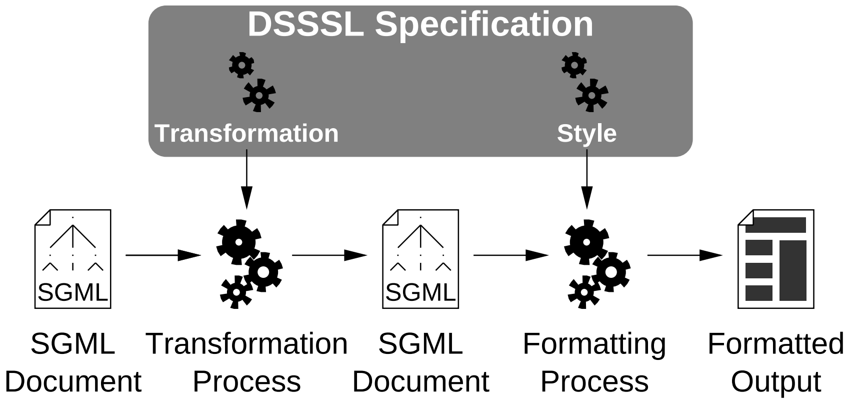


Figure 3.1: DSSSL, the Document Style Semantics and Stylesheet Language

PRASAD system [50] by Lefèvre *et al.* The Office Document Image description Language [51] (ODIL) used in this system is an SGML language to represent the physical format of documents.

XML

The Extensible Markup Language [73] (XML) is not just another markup language. It is the language that is supposed to replace SGML and become the standard format for data representation and exchange. Looking at the vast number of applications using XML and the quantity of implementations, it seems that this goal is not too ambitious. At the origin, the notion XML meant the language itself, but more and more XML is used to address the family of technologies around it.

The software support for XML is enormous:

Standards for Specific Domains facilitate the exchange of data between applications, organizations or services of the common domain by defining common languages. For example XHTML, an HTML variant with XML constraints (i.e. parseable by an XML parser), SVG (see section 3.1.4) for representing scalable vector graphics, MathML for representing mathematical formulae, SMIL for multimedia documents, DocBook for book content etc.

General Standards around XML define how common tasks and problems are solved. For instance, XSLT defines a language for transforming one XML structure into another structure, XPath specifies a language to address specific parts of an XML structure with simple expressions and Namespaces define how documents can be included in documents even if they follow other grammars.

Programming Standards define how programs can interact with XML documents. The Document Object Model (DOM), for example, defines an object oriented API for accessing an XML structure. It defines the names and types of the objects as well as all the methods available. A programmer familiar with DOM programming in C++ can fully re-use his knowledge of the API in another object oriented programming language.

General Tools are of course also available. Implementations of parsers are available for all major programming languages, there are XML editors, XML presentation tools, converters from and to many proprietary formats etc. In short, there is a lot of software available to make the life of not only developers, but also of users easier.

3.1.4 Presentation Oriented Formats

Rich Text Format

The goal of the Rich Text Format (RTF) is the exchange of formatted text between different applications. It is a simple textual format that can be generated by most popular desktop word processors. RTF has a limited number of features, is not extensible, but sufficient for daily use.

PostScript/PDF

PostScript [38] is not simply a document format, but a full featured programming language. It is optimized for printing graphics and text, making it a page description language. The main purpose of PostScript was to provide a language for describing pages in an output device independent manner. In order to render a PostScript document, the PostScript program needs to be executed.

The Portable Document Format [39] (PDF) is inspired by PostScript. It is no longer a programming language, but a real descriptive page layout language. The goal of PDF is to provide a format which can be used for online viewing as well as for printing. Online viewing and navigating inside PDF documents is facilitated by hyperlinks (internal and external), bookmarks, page thumbnails, forms and annotations.

SVG

Scalable Vector Graphics [79] (SVG) is an XML language for describing two-dimensional vector and mixed vector/raster graphics. The fact that SVG is an XML language means that it is XML format, but with a special, predefined grammar called SVG. SVG files can therefore be parsed and processed just like any other XML file.

SVG is a relatively low-level presentation language. It is closely inspired by the PostScript and PDF formats and offers similar rendering possibilities.

SVG also supports an event model and the notion of scripts, what makes it possible to introduce interaction. Both the event and scripting model are similar to the models used in HTML. To every SVG object, attributes can be attached that describe the behaviour of the object when a given event occurs. The event action itself is implemented using a scripting language such as ECMAScript [3] (the standardized version of the JavaScript language).

Since a document recognition system tries to reverse the document formatting process, the data will be closely related to SVG data at some stage during the recognition. Using SVG to represent such data allows to benefit from all the existing software for working with SVG (editors and renderers). However, since SVG is low-level, it is not suited for representing high-level results. Also, the huge number of visual attributes makes SVG not a suitable format for representing intermediate data. Otherwise, every program that further processes the intermediate data needs to handle the full range of SVG attributes, what renders the task much more complex. Instead, a subset of SVG or even a completely different language inspired by SVG (that can be transformed to SVG easily for visualization) should be used in order to keep things simple.

XSL-FO

The Formatting Objects [78] (XSL-FO) is an XML language for describing paginated documents. Being an XML language, it benefits from the same advantages as SVG.

XSL-FO is a very robust and rigorous specification covering all aspects of paginated documents. However, this makes it also a very lenghty specification, which is why there are not many software packages that fully implement it. For instance, the Apache project's quite advanced FOP (Formatting Objects Processor)[1] does still not implement all aspects of XSL-FO.

[1] http://xml.apache.org/fop/index.html

3.1.5 Logical Structure Oriented Formats

ODA

The Office Document Architecture [26] (ODA) is an ISO standard designed to facilitate the exchange of office documents. The following is a very simplified description of ODA (the official standard comprises 14 volumes!).

ODA documents may contain both a logical as well as a physical structure. The logical structure, in ODA terms called the *specific logical structure*, is a hierarchical object model. The root node is called the *document logical root*, and its descendants are either *composite logical objects* or *basic logical objects*, optionally labeled with a name (the equivalent of tag names in markup languages). The composite logical objects themselves contain other composite logical objects or basic logical objects. The basic logical objects may contain zero or more *content portions*, which are the parts where the actual information is stored. This information may contain text, graphics (raster and vector formats).

The physical structure is called the *specific layout structure*. It contains the following physical objects: page set, composite page, basic page, frame and block. Content portions already seen in the logical structure are associated to these physical objects.

Depending on how an ODA document is described, it belongs to one of three document classes:

Formatted Document Class for documents that are described in terms of the physical structure only,

Processable Document Class for documents that are described in terms of the logical structure only,

Formatted-Processable Document Class for documents that are described in terms of both structures.

The grammar of an ODA document can also be described. This structure is called the *generic logical structure*, a mechanism which is much like the DTDs of markup languages.

ODA is without doubt very complete for office documents. However, it is a very heavy standard, making it difficult to have applications supporting it.

DocBook

DocBook [83] is a language for describing the structure of books, papers and technical documentation in particular using SGML or XML (there exist two versions of DocBook). The DocBook DTD defines about 300 tags, making it a complete and robust specification. The DocBook DTD can be easily extended for new kinds of document types not supported. DocBook documents can be formatted using freely available DSSSL and XSLT stylesheets in order to produce HTML, PDF or RTF documents.

Because DocBook documents are XML or SGML documents, they benefit from all the general XML/SGML tools available. Reading or writing DocBook documents from programs is not more difficult than reading or writing XML/SGML and requires no other software.

LaTeX

Leslie Lamport's LaTeX [48] is a structured, high-level language for preparing documents. It is based on TeX, Donald E. Knuth's typesetting system "for the creation of beautiful books–and especially for books that contain a lot of mathematics" [45]. LaTeXis used by many scientists around the world for writing scientific publications[2].

The LaTeXapproach to typesetting is *logical design*. The goal of this language is to relieve writers from having to visually design their documents with WYSIWYG systems. Rather than concentrating on the visual appearance of their text, writers should concentrate on the content. The visual design is done entirely by the LaTeXcompiler.

LaTeXis used for writing documents only. It is in not used as an exchange format nor as a format processed by programs other than the LaTeXcompiler. Only very specialized software exists for working with LaTeXfiles.

One such very popular program is BibTeX. It allows authors to manage their bibliographical references in separate files and to reference them in a variety of styles if needed. This encourages authors to create a database of references that can then be used without the hassle of copying and pasting text and risking to introduce errors.

3.2 The XML Family of Technologies

XML is the most promising data format of the formats presented in the previous sections. Its use is one of the key design decisions for the software framework presented in this thesis.

The term XML has different facets. At the beginning, XML was simply considered a markup language for documents containing structured information. However, since XML defines neither semantics nor a specific tag set, it is in reality a meta-language for describing markup languages. In other words, it

[2]This thesis is written in LaTeXas well.

allows to define tags and the structural relationships between them. Nowadays, the term XML more and more denotes the whole family of emerging technologies that are related to XML.

The following couple of sections present the technologies that were used for this thesis.

3.2.1 The XML Language

The XML language is inspired by SGML, but it is stricter and therefore easier to understand than SGML. Roughly speaking, an XML document consists of a prolog and an element. Every element may contain other nested elements, forming a hierarchical structure.

To illustrate the simplicity of XML, consider the XML document in listing 3.1, a note sent from Jane to John. The prolog is on the first two lines. It simply specifies that this is an XML 1.0 file and that the text contained is coded in UNICODE (UTF-8). Furthermore, it defines that the *<note>* tag respects the grammar described in the Document Type Definition (DTD) *note.dtd*.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE note SYSTEM "note.dtd">
3 <note priority="high">
4   <from>Jane</from>
5   <to>John</to>
6   <subject>Reminder</subject>
7   <body>Do not forget this weekend.</body>
8 </note>
```

Listing 3.1: A sample XML document.

On line 3, the actual data starts. The top element is the *<note>* element. It is further characterized by the attribute *priority*, stating that this is urgent. The note contains everything on lines 4 to 7, until the closing tag *</note>* on line 8. The closing tag is one of the main characteristics of XML making it more strict and easier to parse than SGML or HTML. Every element needs to be closed with a closing tag. Empty elements such as an empty note *<note></note>* may be abbreviated as *<note/>*.

The XML document in listing 3.1 is a *valid* document. The reference to the DTD on line 2 allows the parser to verify the grammar while parsing the document. An application can so ensure that the data it is about to read conforms to the desired grammar. Further validations inside the application are so no longer necessary.

Without line 2, the document would still count as an XML file. However, it would be just a *well-formed* document rather than a valid one, because it respects the basic XML grammar.

A Document Type Definition (DTD) is formalized similar to SGML. Listing 3.2 presents the DTD for our <*note*> document. As can be seen, a DTD is not an XML document itself. In addition, the DTD mechanism is very limited and only offers very basic grammar descriptions. The XML Schema standards [80, 81, 82] try to overcome these often critizised shortcomings with a much more powerful and complete language for describing grammars. XML Schema offers support for data types, namespaces and subclassing—features that are missing in the DTD mechanism.

```
<!ELEMENT note from to subject body>
<!ATTLIST note priority (low|middle|high) #REQUIRED>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Listing 3.2: The Document Type Definition (DTD) for the *note* document.

This was a very brief overview of the XML language. It is sufficient to illustrate the basic concepts of the format, what is probably one of the main reasons of the impact of XML.

3.2.2 DOM vs. SAX

The two most commonly used standards for accessing the contents of XML files are the Document Object Model [72] (DOM) and the *Simple API for XML* [10] (SAX).[3] Even if xmillum does not use SAX, it is presented here to contrast the characteristics of DOM.

DOM specifies a platform- and programming-language-independent interface for accessing and navigating through the hierarchical structure of an XML document from an object oriented programming language. Figure 3.2 illustrates the DOM. It operates on a tree representing the XML document in question. Using the standardized API, applications can access and modify the contents of the tree. The DOM tree typically remains in memory while it is worked on and it is only written to an XML file when it is no longer needed.

[3]In fact, of the two only DOM is a real standard in terms of a standardization organization, while SAX is a widely adopted *de-facto* standard.

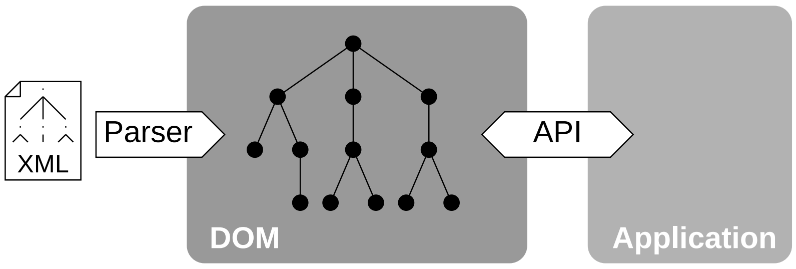


Figure 3.2: DOM provides an API for accessing and navigating through XML documents.

SAX is event-based rather than tree-based like DOM. An application using SAX registers call-back functions within the SAX API. These functions are then called by the parser while the document is parsed, as sketched in figure 3.3. SAX does not automatically create an object model, this task is up to the application.

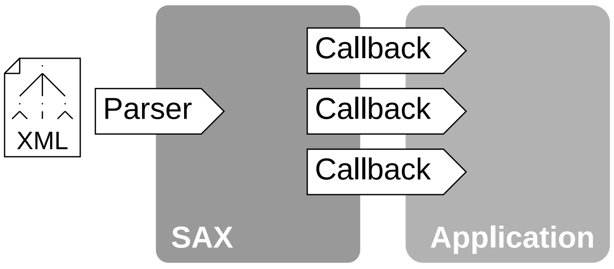


Figure 3.3: SAX is an event-based API for accessing XML documents.

SAX is very well suited for accessing data structures sequentially, but not for navigating in complex data structures. Because the document is not hold in memory, it requires less memory and has a smaller overhead. Modifications to a document are not possible with SAX, they are entirely up to the application.

3.2.3 XSL: Presentation of Data using XSLT and XSL-FO

XML documents hold content. In order to present this content on computer screens, on paper etc., presentation rules are required. Such rules, also known as stylesheets, describe how data inside the XML documents is presented.

With the Extensible Stylesheet Language (XSL), the presentation of XML documents is done in two steps, similar to SGML (see section 3.1.3). In the first step, the source XML structure to present is transformed to a structure containing formatting properties. Only in the second step this structure is then presented.

The transformation of an arbitrary XML structure is done using XSL Transformations [77] (XSLT). Transformation rules in XSLT stylesheets (which are themselves XML files) are composed of patterns and templates and define how a source structure is transformed. Figure 3.4 illustrates the XSLT process: patterns are matched recursively against the elements in the source document whereupon the corresponding templates are instantiated to create parts of the resulting document.

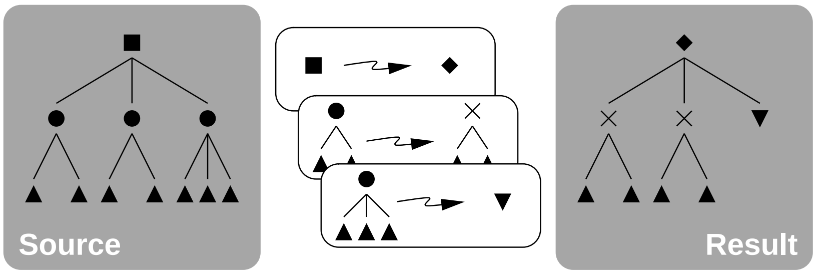


Figure 3.4: XSLT transforms one XML structure into another XML structure using transformation rules.

XSLT allows very rich transformations to be made. The structure of the resulting document can be completely different from the source document. Patterns can match on element types, on attribute values and on more complex structures specified by relations between elements and attribute values. In addition to the pattern-matching mechanism, XSLT also offers the basic control structures known from imperative programming languages (such as branches, loops, functions), as well as functions for sorting data or for assigning numeric labels. It is for example possible to create tables of contents using XSLT stylesheets.

The Formatting Objects [78] (XSL-FO) define the XML vocabulary for specifying the formatting properties. In other words, they characterize the basic building blocks for presenting documents. Examples of the concepts defined by XSL-FO are text blocks, font styles, colors, text formatting attributes etc. Special characteristics of output devices such as pagination for paper documents and scrolling for web-style documents are also taken into account.

3.2.4 XPath

The XML Path Language [76] (XPath) is extensively used in XSLT. It allows to address parts of an XML document using a syntax which is (in its basics) very similar to the paths in a file system. XPath also offers basic functionality for manipulating strings.

XPath expressions are represented by non-XML character strings. They can be relative to a *context node*, which can be seen as the current node in the hierarchical document structure, much like the current working directory in a file system:

- `a` selects the context node's <*a*> children;
- `a/b` selects the <*b*> children of the <*a*> children;
- `a[4]` selects the fourth <*a*> element of the context node;
- `@n` selects the attribute *n* of the context node.

Along the same lines, XPath allows to address absolute locations:

- `/a` selects all <*a*> elements which are direct children of the root node (i.e. the document);
- `//a` selects all <*a*> elements that are descendants of the root node — in other words this selects all <*a*> elements in the document.

The whole power of XPath can be demonstrated the the following expressions, where elements are addressed not only by their location in the XML structure, but also by their content or their own structure:

- `a[@n="val"]` selects all <*a*> elements whose *n* attribute is equal to *val*;
- `a[@n and @m]` selects all <*a*> elements that have both an *n* attribute and an *m* attribute.

3.2.5 XML Namespaces

Some XML documents contain elements from different languages. For example, an XSLT stylesheet contains two types of elements, commands for the XSLT engine and data that is produced. As long as all the element names are distinct, this is not a problem, but as soon as two element names are identical, we may run into a problem.

XML Namespaces [75] solve this problem with a simple method that qualifies element and attribute names by associating them with namespaces. These namespaces are identified by Universal Resource Identifiers [8] (URIs).

For example, the document shown in listing 3.3 contains a note whose body is expressed in XHTML. Line 2 defines the (fictitious) URI of the default namespace. This namespace applies to all elements and attributes that are not prefixed. Line 3 defines that all elements and attributes prefixed by `xhtml:` are from the XHTML namespace. The <*body*> element in line 8 is therefore clearly distinct from the <*body*> element in line 13 even though they have the same element name. Without XML Namespaces, the XML parser would not be able to distinguish these two elements.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <note xmlns="http://mynotes.com/notes"
3        xmlns:xhtml="http://www.w3.org/1999/xhtml"
4        priority="high">
5    <from>Jane</from>
6    <to>John</to>
7    <subject>Surprise</subject>
8    <body>
9      <xhtml:html>
10       <xhtml:head>
11         <xhtml:title>Happy Birthday!</xhtml:title>
12       </xhtml:head>
13       <xhtml:body>
14         <xhtml:p>Happy Birthday, John!</xhtml:p>
15       </xhtml:body>
16     </xhtml:html>
17   </body>
18 </note>
```

Listing 3.3: A document with two namespaces.

XML namespaces are very simple, but they offer very interesting possibilities. It is for example possible to create XSLT stylesheets that generate other XSLT stylesheets or, in other words, XSLT stylesheets that contain XSLT commands that need to be handled as data and XSLT commands that need to be interpreted. Without the namespace mechanism, this would not be possible.

3.3 Java

Apart from general requirements such as efficient performance and a large library of generic tools and data structures that needs to be available, we have chosen the programming language with the following requirements in mind:

Dynamic Binding We want to construct our applications at runtime, not at compile time.

Advanced XML Support The technologies we want to use need to be fully supported.

Platform Independence Although from the point of view of many computer users, everything seems to concentrate around Microsoft Windows, we need a platform independent framework in order to address a public as broad as possible.

GUI Toolkit In order to create interactive applications, GUI creation needs to be facilitated.

We have chosen the Java programming language to implement xmillum. Java is an object oriented programming language. Its machine language is platform independent, i.e. the binary data produced by a Java compiler runs on different platforms.

Independence is achieved with a platform dependent virtual machine (the Java virtual machine, or JVM) that interprets the instructions produced by the Java compiler (also known as the Java bytecode) on the host platform. Due to the additional interpretation phase, Java programs are in general slower than native programs and they require more memory than native programs. However, thanks to recent advances with *just in time* compiling (i.e. the translation of bytecode to native code prior to execution), the performance gap between native code and bytecode is becoming less important.

Despite the performance and memory drawbacks, Java offers many advantages. With automatic memory allocation and garbage collection, multi-thread support, as well as a standard library with an extensive selection of useful basic tools ranging from I/O facilities, distributed computation support and standard data structures to full GUI toolkits, Java helps to cut down development time compared to many other languages.

Unlike traditional programming languages, the different parts of a program are linked together only at runtime. This makes it straightforward to add plugins to programs, a technique that is extensively used in xmillum. In the remainder of this text, plugins are referenced by the fully qualified name of the Java class implementing a particular functionality, i.e. *package.Plugin*.

Chapter 4

Requirements and Goals of xmillum

xmillum—short for *XML Illuminator*—is the main outcome of this thesis. It is a framework designed for rapidly visualizing document recognition data and for creating interactive document recognition applications.

Using a framework is fundamentally different from using a toolkit or software library: "When you use a toolkit, you write the main body of the application and call the code you want to reuse. When you use a framework, you reuse the main body and write the code *it* calls." Gamma *et al.* [29]. This chapter introduces the requirements of interactive document recognition applications and expresses the goals we would like to satisfy with our framework.

4.1 Requirements

The requirements for an interactive document recognition system depend on a variety of factors. One of the key factors is the target user. We will analyze the requirements by separating the users into three classes: end users, system integrators and researchers.

4.1.1 End User

The end user uses the system to solve an immediate recognition task. He does not know anything about the underlying algorithms and techniques and is only interested in the result of the recognition task. Depending on the context, some end users use the system casually, while others use it more regularly.

Apart from the requirement that the system has to work and produce a useful result (this can be seen as the very basic requirement for all document recognition systems), the casual end user is first of all interested in an easy to use system. Since he does not know anything about the underlying algorithms and does not use the system frequently, it is important that the learning time is very short. The user has a document and wants it recognized. In order to guarantee this, the complexity of the system needs to be hidden behind a convenient user interface.

The document image, being generally a two-dimensional area[1], is probably best suited as user interface. The image is enriched with visual features that allow the user to quickly discriminate the different kinds of information: object types, confidence values, possible conflicts etc. Visual feature include first of all colors, drawing style, shapes and borders.

As soon as higher-level recognition tasks are performed, a WYSIWYG [2] view is no longer sufficient. Other views are required. Many documents can be structured hierarchically. A view to represent this kind of structure is therefore required.

If there are different views, it is also necessary to take into account the relations between them. Some data can be represented more easily in one view than in another, but nevertheless the user needs to know the objects that correspond to a given object in the other view. For example, an object that has been correctly identified in the document image may be at a wrong position in the hierarchy. Let's suppose that the WYSIWYG-view does only represent the hierarchy and that operations such as moving an object around in the tree need to be done in a specialized tree-view. Now, the system can greatly help the user by scrolling to the corresponding object in the tree-view as soon as the user clicks on the object he wants to modify in the WYSIWYG-view. Synchronized views are therefore important components for creating convenient recognition systems.

Last but not least, editing operations are required. For the end user, these operations don't need be too general, but more tailored to the application in question. Having in mind that it is the system that needs to cooperate with the user, and not the user that has to do the work, results in operations that are much simpler and easier to understand. For instance, instead of asking the user to correct the result of a segmentation step by moving and resizing zones on the document image, the system might just ask the user to select the

[1]"Generally", because hyper links or other multimedia extensions in electronic documents may be seen as another dimension added to documents.

[2]*WYSIWYG* is an acronym for *What you see is what you get* and is used for document production systems (e.g. word processors) where the document appears exactly the same way on the computer screen as when it is printed on paper.

blocks that were incorrectly segmented. The system can then try to segment the corresponding blocks differently or with other parameters. In a second step, or, if the system is really unable to segment the document properly, the user could be asked to do the actual work.

4.1.2 System Integrator

The system integrator installs and fine-tunes a system for a recognition application. He knows about the type of documents that will be recognized and can therefore pre-tune the system for this type of documents in order to simplify the end user's life. He has limited knowledge of the underlying algorithms, but knows about the global architecture and the possible parameters of the system.

The most important requirement for the system integrator is modularity. A modular document recognition system allows him to construct a document recognition system for a given recognition task by plugging together different components responsible for the recognition. It also allows him to create systems for different kinds of end users depending on their level of knowledge. The power user can be expected to delve much deeper into the recognition parameters whereas the casual end user does not at all wish to be confronted with the internals of the system.

4.1.3 Researcher

A researcher uses a document recognition system not in the same way as a user or an integrator. He does not use it for real recognition tasks, but merely uses the services the system already offers to verify the feasibility of new ideas, to monitor the performance and to develop new algorithms. From a researcher's point of view, the document recognition system therefore becomes a framework.

The most important requirement for a document recognition framework from the researcher's point of view is the capability to rapidly integrate new algorithms and tools. This allows the verification of new ideas without having to bother with too many related details.

To guarantee this capability, the framework needs a flexible component system. By plugging together components, a researcher working on a specific part of a document recognition system can create a system that produces the input to his algorithms. For instance, somebody working on a new algorithm for high-level logical structure analysis usually needs data from the lower-level recognition stages in order to verify the new algorithm. By reusing already existing components, this data can be produced efficiently, resulting in more time that can be spent for concentrating on the actual algorithm.

Another feature that can facilitate the rapid testing of new ideas is a powerful and easy to use visualization. Results from document recognition algorithms are, most of the time, strongly connected to a specific document image and can therefore be best analyzed visually. By providing a flexible visualization system, results from a new algorithm can be quickly displayed in a framework in order to check if the produced results are useful or not. If the visualization is not completely passive and can be augmented with interactive features as well, the analysis can even be more fruitful.

Components in a framework exchange data. In order to have the greatest possible compatibility among the components, a standard data representation format is required. With a format that is too restricted, the system can greatly assist the individual components by offering implementations of commonly used functionality, but on the other hand, the format may not be rich enough to represent all the data items required by the components. With a format, that is too open, too much work is left for the component developers what slows down development. A tradeoff between these two extrema is therefore necessary.

4.2 Goals of xmillum

The xmillum framework serves as a user interface to document recognition systems producing XML data. It has access to the XML data and can communicate with the recognition process.

The four main goals of xmillum are discussed in the following sections.

4.2.1 Visualization

The well-known saying that "an image is worth 1000 words" also holds in the document recognition domain. Data produced by a recognition process contains low-level information that may be useful for the system integrator or the researcher. A data visualization tool can greatly improve the usefulness of this data and even make it accessible for the end user.

Our framework provides support for conveniently visualizing data produced by a document recognition system (figure 4.1). It allows to emphasize parts that are of particular interest or that meet certain criteria using visual features. This allows the user to distinguish between the different data objects. Different views allow the user to see the information from different points of view. A WYSIWYG view helps to visually correlate the data with the document image.

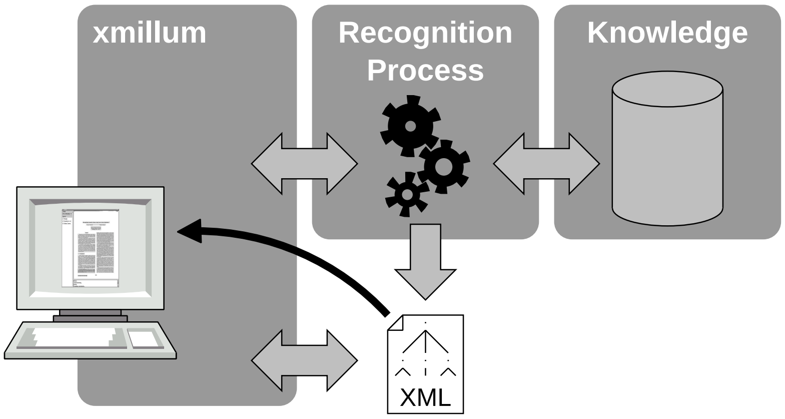


Figure 4.1: Visualization of XML data.

4.2.2 Validation

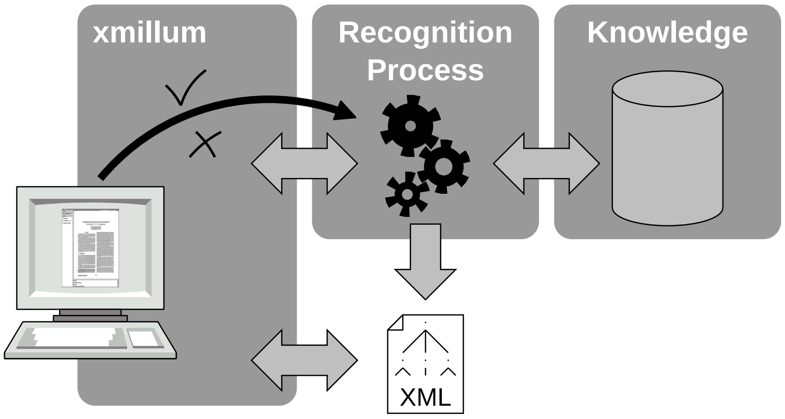


Figure 4.2: Validation, a simple form of interactivity.

Visualizing data allows users to quickly spot incorrect results and mark them. Marking these mistakes is a simple interaction scheme that generates valuable information. This information can be used to evaluate the recognition performance, to compare different recognition systems or to improve the recognition performance by tuning the underlying models, as illustrated in figure 4.2.

Validation, the process of labeling objects with *correct* and *wrong* labels, is one possible interactive application that is facilitated with our framework.

4.2.3 Correction

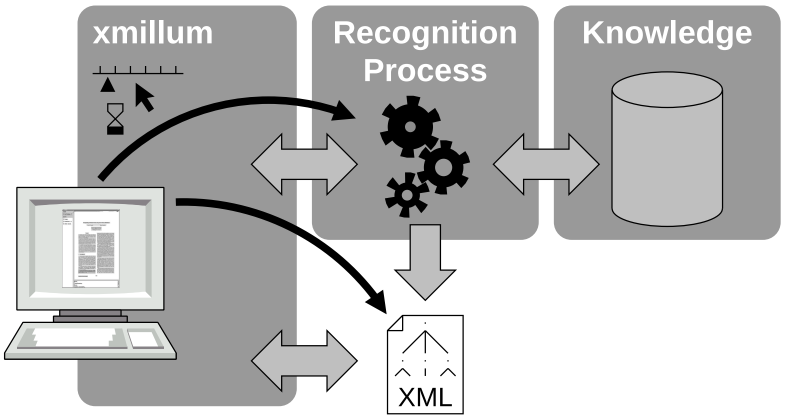


Figure 4.3: The goal of correction is not only to correct data, but to give the recognition system hints for improvement.

A more advanced interaction scheme involves manual correction of mistakes made by a document recognition system, as sketched in figure 4.3.

Manual correction requires much more complex interaction primitives than the validation. Whereas the validation can be done using a simple *correct*—*wrong* labeling, the correction depends heavily on the kind of errors to correct. To correct the result of a segmentation algorithm, a tool to split and merge objects is required. Likewise, to correct OCR results, a text entry tool is necessary.

The changes made are then stored in the document. Additionally, they can be reported to the recognition process in question, that can then use this information to adapt its underlying models in order to improve the recognition performance of future recognition tasks. This is the topic of the next section.

4.2.4 Learning

In order to produce useful results, a document recognition system needs to be trained. This training is typically done using initial learning and incremental learning, as illustrated in figure 4.4.

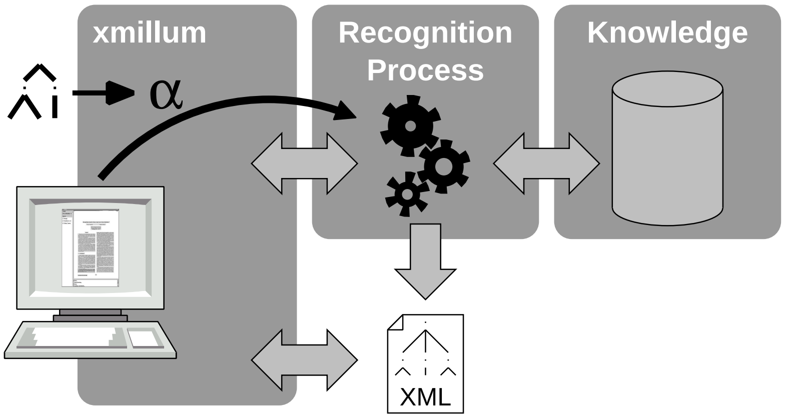


Figure 4.4: Facilitating the learning process has a direct impact on the performance of recognition systems.

During initial learning, the system is presented learning data, so called *ground truth*, that typically consists of objects that have been labeled by an operator and that are expected to be correct. The creation of ground truth is a very expensive process. A convenient user interface that assists the operator in the labeling task, or even a semi-automatic system where some of the labeling is done by the system, can greatly facilitate the process and results in more or better ground truth. For many recognition algorithms, it is a known fact that the more ground truth is available, the better the algorithms perform. Facilitating the creation of ground truth has therefore a direct influence on the recognition performance.

Humans are allowed to make mistakes, but we are all expected to learn from them. The same is also true for a document recognition system. This is where the incremental learning comes in. As already introduced in the previous section, the correction information can not only be used to correct the results at hand but also the results of future recognition tasks. By adapting the underlying models accordingly, the system improves its recognition accuracy. In a perfect world, with a non-ambiguous problem and the perfect recognition algorithm, a document recognition system should so converge towards a system that makes no more mistakes at all. In practice, however, the system will converge towards a stable state which minimizes the errors. In pattern recognition, there is always an unreducible error rate.

Chapter 5

The Design of xmillum

The initial motivation for working on xmillum has been the lack of a standard visualization tool for document recognition data. When this project was started, the only sufficiently general tool to be used for this purpose was Illuminator (see section 2.2.5). Illuminator, however, suffered from the major disadvantage that it only allowed to visualize data available in the proprietary DAFS format. Furthermore, extending it was a serious undertaking.

This chapter presents how xmillum has grown from a simple initial idea for visualizing document recognition results to a full visualization and editing framework with a working prototype. It also sketches how the idea can be carried further to make xmillum even better.

5.1 Generation 1: The Fundamental Idea

The design of xmillum is inspired by the basic idea of modern web publishing frameworks: the separation of content and presentation. In web publishing, this separation is to take into account the special characteristics of the different output devices (e.g. the display size, number of colors, available bandwidth, computation power etc.). The content is transformed using presentation rules for the various different presentations, suitable for use on output devices with varying properties, as shown in figure 5.1. With this approach, the same data is reused in different forms what avoids inconsistencies because of redundant data.

In the document recognition domain, we are not primarily concerned with different output devices, but nevertheless our requirements are comparable:

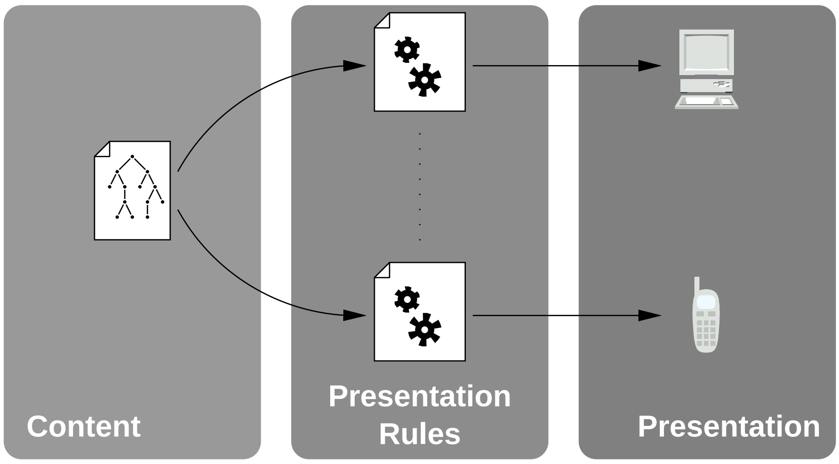


Figure 5.1: xmillum borrows from modern web publishing, where content and presentation is separated.

Different applications require that the same data is processed in various different ways.

Different classes of users all have their very own requirements on what kind of data they want to visualize and how they want it presented.

Different types of data need to be handled. The various document recognition phases in section 2.1 show that there is a great diversity of data that is processed and that results in a document recognition system.

The very fundamental idea of xmillum is to transform all data with XSLT transformations prior to processing it. Instead of having one application for every type of data, we have one general application and one set of transformation rules, depending on the actual format of the input data.

5.1.1 Advantages

The only restriction that is imposed by this approach is that the data to be processed is available in XML format. Given that this is only a syntactic restriction rather than a structural one (since there is no predefined XML markup language the input data has to satisfy), and given the fact that XML has become a de-facto standard for representing data, this means that the input data format is virtually unrestricted.

The approach has numerous advantages with respect to traditional methods where a predefined data structure on the data is imposed:

"Natural" data structures Every recognition application produces data structured in a way that best suits its needs. There is no need for a recognition application to take into account the requirements of a visualization/editing framework. Even data produced by legacy applications can be used this way (supposed that the data exists in XML format or that it can be easily converted to XML format).

Multiple views By providing different transformation rules, different views can be created. Rules can emphasize some data items, filter others or completely rearrange objects.

One single data source All applications and users operate on the same data. There is no need for converting and re-converting data.

5.1.2 First Feasibility Tests and Results

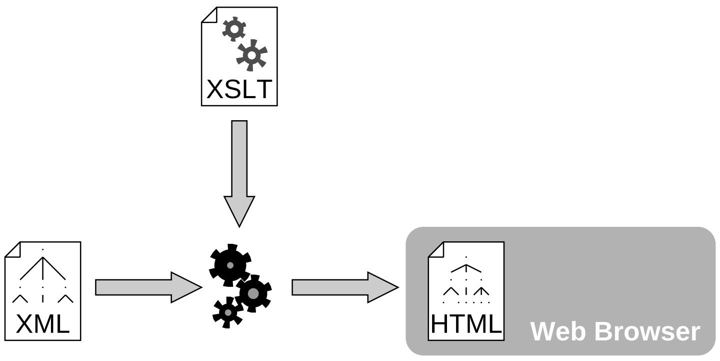


Figure 5.2: First tests have been done using standard software.

The initial basic idea has been verified with standard software. A simple XSLT processor has been used to transform document recognition data to HTML[1], which has then been visualized in a standard web browser (see figure 5.2).

[1]Even though HTML is not an XML language, XSLT defines the HTML output format. What is done in this case is that XSLT produces XHTML and then strips the resulting document down to HTML. This means that e.g. *<BR/>* will be stripped to *<BR>*, what makes it no longer XML compliant.

This test has been done mainly with document segmentation data. The goal was to visualize the document image with overlaid transparent regions representing the different document objects.

Using a very simple input data structure that consisted of simple rectangular blocks represented by their absolute coordinates, writing an XSLT stylesheet was a straightforward task.

However, the major flaw of this approach became apparent very quickly: It is limited to the possibilities of HTML. At the time these tests were done, web browsers were not as powerful as they are today. For instance, they did not work very reliably with absolute positions (which was a very critical requirement for visualizing bounding boxes on top of some other image). Furthermore, transparent objects were not supported.

In order to overcome this flaw, the xmillum idea has evolved to the next generation.

5.2 Generation 2: A Custom Application

A straightforward solution to overcome the problems encountered when doing the first tests was to create a dedicated markup language and a corresponding viewer that satisfied our requirements.

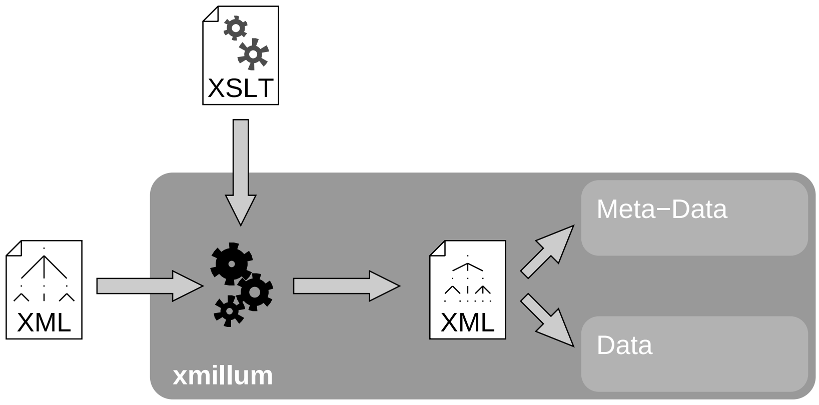


Figure 5.3: The XSLT transformation produces data as well as meta data.

In order not to restrict this language too much, we have decided not to predefine any visualization primitives at all. This may sound odd, but it is precisely one of the strong points of xmillum. Visualization primitives may be added to the system in the form of plugins. This makes xmillum flexible and extensible.

The general functioning of the second generation xmillum design is comparable to the first generation. As illustrated in figure 5.3, the input data is again transformed using XSLT. This time, however, an XML data structure satisfying our own markup language is generated.

This markup language contains two types of information:

Meta data describing how the different components of the xmillum framework are composed to form an application;

Data describing the actual data to be visualized.

5.2.1 Meta Data

The meta data describes how the data we want to be visualized is presented. The whole visualization is centered around a WYSIWYG-style view in which the data is rendered. The meta-data concerning this view is specified using the following three concepts:

Layers allow to group the data to visualize into two-dimensional collections of visual objects. All layers are stacked and can be switched on and off independently by the user.

Styles define a set of attributes for drawing an object. Attributes concern color, transparency settings, font configurations etc. Styles are referenced to by objects, which are explained next.

Objects specify what plugins to use for visualizing one specific type of data item using any of the defined styles. Examples for objects are rectangular blocks, images loaded from a file, text areas and so on.

The WYSIWYG-style view is very useful where we have two-dimensional data that we would like to lay over a document image. However, sometimes it is useful to have other, alternative views of the data. This is why we have come up with a fourth concept:

Tools are plugins that have access to the entire internal data structures of xmillum, and that can implement numerous functionality such as alternative views of the data, interfaces to other programs, user interaction etc.

A simple example of how these concepts are used more concretely will be given in section 5.2.3.

5.2.2 Plugins in xmillum

Generally speaking, plugins are program modules that can be used by the xmillum application in order to provide functionality which is missing in xmillum. The code of plugins is not integrated in xmillum, it just gets called from xmillum whenever the functionality it offers is required.

In order for plugins to work, there needs to be a definition of the interface between the main program and the plugins. In the object oriented paradigm, this can be done in a straightforward manner using classes and subclasses. The base classes define the names of methods along with their arguments and return types and the subclasses define the actual implementations of these methods. This provides the main program with a well-defined interface.

The way the second generation xmillum design has been presented, it is known only what plugins to use once the XSLT transformation has been applied and the result is known. We therefore need to load the plugins at runtime.[2]

Throughout the design of xmillum, we have tried to delegate functionality into plugins whenever this was making sense. This is for instance true for the visualization objects (i.e. the *object* concept from the previous section), which define how some data item is drawn. Having delegated this functionality into a plugin means that it is possible to add completely new graphical objects without changing the main application. This is a very important requirement for an extensible framework.

Listing 5.1 shows an excerpt of how a plugin is registered as an xmillum tool. The Java code for the plugin is sketched in listing 5.2. In this example, the interface the plugin has to respect is not specified in a *class*, but in a Java *interface*, a special kind of base class. This interface specifies that a tool needs to implement the two methods *activateTool* and *deactivateTool*. xmillum then calls these two methods where the tool can do whatever it wishes. The tool typically installs itself in the internal data structures of xmillum, which are given in the form of the *BrowserContext* object.

5.2.3 A Simple Visualization Scenario

In order to see how the concepts from the previous paragraphs interact, let us suppose that we want to visualize document segmentation data such as shown in listing 5.3. This listing shows us that we know the file name of the document image, as well as the bounding boxes of text blocks and text lines.

[2]In Java, this is easily accomplished using the Reflection API. In traditional (mostly compiled) languages, functionality can often only be added to a running program in the form of libraries. This is usually more difficult and trickier than how it is done in Java. Of course, nowadays many

```
<xmi:document>
  ...
  <xmi:tool class="my.project.MyTool"/>
  ...
</xmi:document>
```

Listing 5.1: The plugin *my.project.MyTool* is registered as an xmillumtool.

```
package my.project;

import org.w3c.dom.Element;

import iiuf.xmillum.Tool;
import iiuf.xmillum.BrowserContext;

public class MyTool
  implements Tool
{
  public void activateTool(BrowserContext c, Element e)
  {
    ...
  }

  public void deactivateTool()
  {
    ...
  }
}
```

Listing 5.2: A tool has to implement the *iiuf.xmillum.Tool* interface, which specifies that two methods need to be present.

Our goal is to lay all the data items over the document image in order to visually verify the validity of the data. This is a very common task for people working with segmentation algorithms. While it does not allow for a precise qualitative measurement of the accuracy of the results, it allows to quickly check if the calculated segementation data appears correct.

We want text lines and text blocks to be shown in different styles. Listing 5.4

modern programming languages offer functionality such as Java does.

```
<document image="1-4-a-1.tif">
  ...
  <block x="679" y="154" w="2581" h="324">
    <line x="679" y="154" w="2530" h="144"/>
    <line x="682" y="310" w="2181" h="81"/>
    ...
  </block>
  ...
</document>
```

Listing 5.3: The data structure we want to visualize contains a variety of data.

```
<xmi:style name="textblock-style">
  <param name="foreground" value="green"/>
  <param name="transparency" value="0.4"/>
  <param name="fill" value="true"/>
</xmi:style>

<xmi:style name="textline-style">
  <param name="foreground" value="red"/>
  <param name="transparency" value="0.4"/>
  <param name="fill" value="true"/>
</xmi:style>
```

Listing 5.4: We need two styles for visualizing the text blocks and text lines.

shows the definition of two styles.

Next, we need to define how the different data objects will be visualized. Since we have three different sorts of objects to visualize, this will give us three objects, as listing 5.5 shows. Note that text blocks and text lines are visualized with the same plugin, but using a different style.

Finally, we need to put create an XSLT transformation that transforms our input document structure into a structure xmillum can handle. This can be accomplished with listing 5.6. Our three types of data objects are put in three layers the user can switch on and off individually in order to verify the validity of every layer of information with respect to the document image.

```
<xmi:object name="image"
            class="iiuf.xmillum.displayable.Image"/>

<xmi:object name="textblock"
            class="iiuf.xmillum.displayable.Block">
  <param name="style" value="textblock-style"/>
</xmi:object>

<xmi:object name="textline"
            class="iiuf.xmillum.displayable.Block">
  <param name="style" value="textline-style"/>
</xmi:object>
```

Listing 5.5: These three objects are used for visualizing the three types of data objects.

5.2.4 From Visualization to Active Interaction

Now that we have a simple way to visualize data, we would like to interact with it actively, rather than just looking at it. Several modes of interaction can be imagined, as already mentioned in sections 4.2.2 to 4.2.4.

The separation of content and presentation, the basic concept of the xmillum idea, is very useful for visualization. However, the fact that the user interface operates primarily with a transformed version of the data, introduces an important new challenge when it comes to modifying the data. The goal of data modification is generally to modify the original data, not the transformed one, so the two data structures need to be connected somehow.

To illustrate this, consider a user who wants to modify the attributes of an object. He selects this object and modifies the attributes. The selection of the object and the modification have taken place at the user-interface level, in the transformed data structure, but they need to be propagated back to the original XML data (see figure 5.4).

The ideal solution to this problem would be to reverse the transformation and to transform the modified data back to its original form. Unfortunately, this approach is not as straightforward as it might seem. Reversing an XSLT transformation is only possible if the transformation which converted the original structure into the internal structure is a bijection. In reality, this is rarely the case[3] and imposing such a constraint would be too restricting.

[3]A very common scenario for such transformations is to filter some information because it is not relevant to the application at hand. Such a transformation is not a bijection.

```
<xsl:stylesheet version="1.0">
  <xsl:template match="document">
    <xmi:document>
      ...
      Styles
      ...
      Objects
      ...
      <xmi:layer name="Document Image">
        <image src="{@image}"/>
      </xmi:layer>

      <xmi:layer name="Text Blocks">
        <xsl:for-each select=".//block">
          <textblock x="{@x}" y="{@y}" w="{@w}" h="{@h}"/>
        </xsl:for-each>
      </xmi:layer>

      <xmi:layer name="Text Lines">
        <xsl:for-each select=".//line">
          <textline x="{@x}" y="{@y}" w="{@w}" h="{@h}"/>
        </xsl:for-each>
      </xmi:layer>
    </xmi:document>
  </xsl:template>
</xsl:stylesheet>
```

Listing 5.6: This XSLT transformation excerpt transforms the input structure into an xmillum structure with three layers.

Another elegant solution is the definition of a language to describe the modifications to the original structure. This information is included directly into the internal data structure using the XSLT transformation. This approach can also be seen as some sort of “incremental reverse transformations”, where the individual modifications are back-propagated to the original XML structure. The modified XML structure is then re-transformed according the XSLT stylesheet.

Our prototype uses the latter solution, but it does not define an own language. Rather, the programs performing the modifications to the original XML structure are encapsulated in plugins which get called whenever a modification takes place. These plugins are called *Handlers*. Handler perform a modification and then trigger a re-transformation if this is necessary.

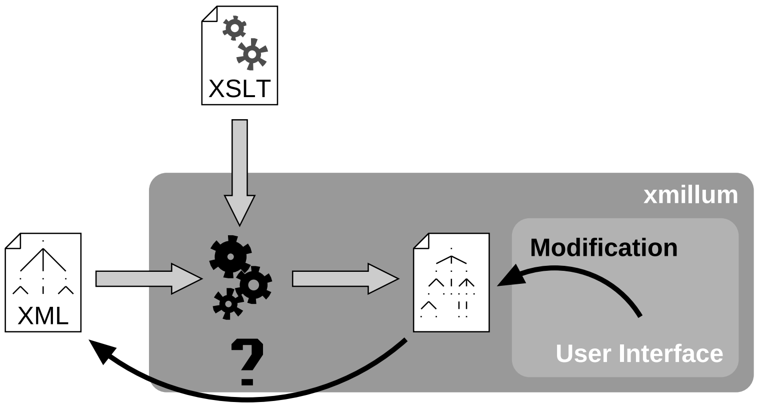


Figure 5.4: Modifications take place at the user-interface level. They need to be propagated to the original XML data.

Listing 5.7 shows how a handler for deleting text blocks can be registered. It specifies that whenever the user *clicks* on an object of type *textblock*, this object should be deleted. Listing 5.8 finally shows the code of the handler itself.

Note the use of the *ref* attribute. When xmillum loads the original data structure, it inserts a unique attribute *tmp:refvalue* in every element. The value of this attribute is then placed in the *ref* attribute of every generated *textblock* and serves so as a pointer to the original data structure. This is how xmillum knows what element belongs to the object on which the user has clicked. Listing 5.8 also shows the call to *retransform()*, where the original data structure is retransformed using the XSLT transformation.

5.2.5 Bundling xmillum Document Recognition Applications

The term *application* has already been mentioned before when we have introduced meta data. In the context of xmillum, an application is a particular configuration of xmillum and associated plugins. It consists of an XSLT transformation (i.e. a stylesheet) and a set of plugins that are referenced in the meta data produced by the transformation. An application can be used to solve one particular type of problem with one particular type of data structure.

Since plugins can be arbitrarily complex, there can be quite a lot of individual components forming an application. This is especially true for the Java programming language. There are quickly hundreds of files forming an application since every class is stored in one file.

Bundling all components together can be very helpful for keeping things

```
<xmi:document>
  ...
  <xmi:handler name="delete" class="my.project.Delete"/>
  ...
  <xmi:object name="textblock" class="my.project.Block">
    <param name="style" value="default-style"/>
    <param name="click" value="delete"/>
  </xmi:object>
  ...
  <xmi:layer name="My Layer">
    <xsl:for-each select=".//block">
      <textblock ··· ref="{@tmp:refvalue}"/>
    </xsl:for-each>
  </xmi:layer>
</xmi:document>
```

Listing 5.7: Upon a *click* on an object of type *block*, the handler *my.project.Delete* is called.

tidy and well organized. In addition, it also facilitates and even encourages the exchange of such applications. Instead of having users share hundreds or thousands of files along with a documentation on how and where to install, one file containing the whole lot can be shared.

In Java, such packaging can be done analogous to the Web Application Archives (WAR) [15], where web services are bundled together in JAR files (*J*ava *AR*chive) in order to facilitate application deployment.

Although exactly the same mechanism could be used for xmillum, this is not implemented in our prototype.

5.2.6 The xmillum Prototype

Using the concepts of the second generation xmillum design a prototype has been implemented. The prototype has been programmed in the Java programming language, mainly because of the powerful dynamic binding feature that allows plugins to be created very easily.

The prototype has shown that the model is feasible. An in-depth presentation will be given in chapter 6.

```java
package my.project;
...
import iiuf.xmillum.ActionHandler;
...
public class Delete
  extends ActionHandler
{
  public void init(BrowserContext c, Element e)
  {
  }

  public void handle(ActionHandlerParam param)
  {
    BrowserContext context = param.getContext();

    String ref = param.getElement().getAttribute("ref");
    Element source = context.getSourceElementByReference(ref);
    source.getParentNode().removeChild(source);

    // Now, retransform the original data structure
    context.retransform();
  }
}
```

Listing 5.8: The *Delete* handler removes an entire subtree from the original data structure.

5.2.7 Benefits and Drawbacks

The second generation xmillum design solves the problems presented in section 5.1.2, while retaining all the advantages of the original idea.

Using xmillum, it is possible to quickly and easily visualize document various kinds of document recognition data. For most applications, the stylesheets necessary to transform the document recognition data into xmillum's internal format are not very complex (as long as the input data is not complex). The resulting visualizations are quite comfortable, thanks to the graphical attributes that can be specified as styles.

In addition to solving the visualization problem, the second generation xmillum design generalizes the idea and makes it possible to actively interact with the data by modifying it. This broadens the horizon for xmillum considerably. It is no longer simply a visualization tool, but can be used as a fully-fledged customizable framework for interactive document recognition.

However, nothing is perfect. The active interaction extension has revealed several design flaws that ought to be improved. These drawbacks of the second generation design are as follows:

Meta data produced by transformation As shown in figure 5.3, there is only one XSLT transformation that produces both the meta data as well as the document data. The meta data is thus only known once the transformation has been applied. More important, once data is modified by the user and the transformation is reapplied, nothing guarantees that the meta data stays the same; the whole application can change. From an implementation's point of view, this renders matters much more complex than if meta data was static.

Centered around a WYSIWYG view The entire design of xmillum is centered around a browser-style WYSIWYG view. This influenced all design decisions and made the whole implementation heavily biased towards this view. Among others, aspects concerning the user interface are deeply nested in the program. Similarly, all possible events the user can trigger in the WYSIWYG view (in order to get interactivity), are predefined. It is not possible to add new types of events without changing the xmillum source code, something that we wanted to avoid by using plugins. It would have been better if there was a general backend and several views, the browser-style being view one of them. Views could then be especially tailored to specific applications whenever special requirements are requested.

To summarize, the xmillum idea has resulted in a quite powerful prototype that is actively used by different people. However, at the same time it also showed the possibility to develop the idea further.

The next section sketches the third generation of the xmillum design that generalizes the framework and solves the discussed problems.

5.3 Generation 3: Generalizing the xmillum Idea

The goal of this section is to provide the next generation of the xmillum design that takes into account the lessons learned so far. It does not abandon the original ideas of xmillum, the use of XML and XSLT, and the rule that functionality is to be delegated into external plugins whenever possible.

In fact, the main design principle of the third generation xmillum design is that the framework should only define the most important concepts and leave the rest up to the plugins. xmillum should only provide the glue that holds all together, but the actual work should be done entirely in the plugins. This design ensures a very really general model that can be used for a wide range of applications.

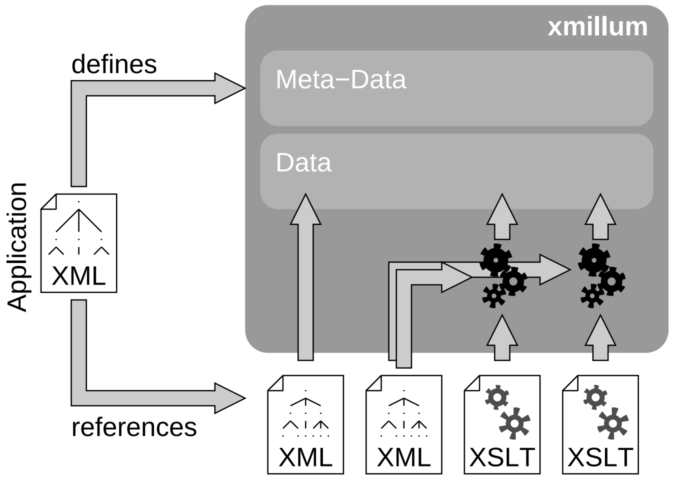


Figure 5.5: Only data is transformed, meta data is statically defined in an application description.

An illustration of the third generation xmillum model is shown in figure 5.5. It shows the most obvious changes with respect to the second generation xmillum design: The whole application is now described in an application description structure. This structure defines the meta data statically—it is no longer transformed using XSLT. What is transformed is the actual data. There can be multiple data sources, each one with its own transformation. It is also possible to use data as-is, without any transformation.

5.3.1 Concepts

Third generation xmillum applications are created with the three following concepts:

Data repositories allow the system to access XML data. This data typically comes from a file or another data repository and can be processed with an XSLT transformation.

Types define the elementary data objects processed by this application and the operations that can be applied to these objects.

Views implement the user interfaces. A view fetches the data to represent from a data repository, visualizes it and gives users access to the operations defined in the types. Every view defines its own language for the data to represent. It is up to the data repository in question to deliver the data in the requested structure.

These concepts are presented more thoroughly in the next couple of sections.

5.3.2 Data Repositories

All XML data processed by the system is accessed through data repositories. The data repository defines where the data it offers comes from and what can be done with it.

There are different types of data repositories that can be classified according to the following criteria:

Direct vs. indirect data source Data repositories with direct access to the data source typically operate directly on the file containing the XML data. Data repositories with indirect access to the data source, on the other hand, take their input from other data repositories.

Transformation vs. pass-through One of the fundamental ideas of xmillum is that data can be transformed using XSLT transformations in order to convert it into a structure suitable for xmillum. This is where transforming data repositories enter into play. Such data repositories are indirect data source repositories that convert data from other data repositories into another format. This transformation is typically done using XSLT transformations.

Read-only vs. read-write Some data repositories may offer read-only access whereas others allow data to be modified. Such read-write data repositories are always repositories with a direct data source, because only direct data source repositories have a direct link to the location where the modified data is ultimately stored.

Every data repository keeps a list of other objects that depend upon it. Such objects can be other data repositories that transform the data, or views that require the data for visualizing it. As soon as data changes, all the dependent objects are notified so that they can be updated.

5.3.3 Data Types

The data types define the elementary objects an application processes. This information is needed as soon as data modification is needed. All modification operations take place through plugins that are defined here.

A modification plugin itself does not know anything about the user interface that invokes the modification. It only knows how the document structure is changed once the operation is invoked. It is then up to the view invoking the modification to provide the user with an appropriate user interface to make the operation as convenient as possible.

Of course, data objects can only be modified if the data repositories allow read-write access. As per our definition, such data repositories always have a direct data source. This ensures that the changes can actually be stored somewhere. It also helps to solve the problem stated in section 5.2.4 (i.e. how are changes carried back to the original data structure?) by excluding such cases altogether. Modification always take place at the original data structure. Using the dependencies between the different data repositories, all dependent objects can then be notified that data has changed and views can be updated accordingly.

5.3.4 Views

Contrary to the second generation xmillum design, there is no central predefined WYSIWYG view. Similarly, the xmillum model does not define the notion of user interface details such as styles, graphical objects or user interaction facilities.

All such details are delegated to *view* plugins. In order to define a view, the following three types of information are required:

Configuration data defines in a plugin-specific way what the view looks like. This is where matters such as styles and graphical objects, as seen in the second generation xmillum design are described. The structure of this definition depends entirely on the plugin in question and is in no way predefined by the xmillum model.

Data repository references define what data the plugin visualizes. The plugin fetches this data from the respective data repositories and reacts if the data contained in a data repository changes (e.g. after data has been modified by the user).

When the system is started up, a view subscribes to all the data repositories it requires access to and subsequently receives all the data contained in these repositories. If data is changed, all views subscribed to it will be notified accordingly.

Operation mappings map the facilities provided by the user interface to the operations defined in the data types. These user interface facilities can be arbitrarily complex, ranging from simple mouse clicks to complex split and merge functions. All depends on the functionality the user interface wants to provide to the user. A general view may provide simple, not so convenient functions, whereas a view with a very specific goal may offer the same facilities much more conveniently, but less general.

Using these ingredients, all of the second generation's WYSIWYG view can be done using a plugin this way, without embedding it statically into the model.

5.3.5 A Markup Language for Applications

The glue that holds the different components forming an application together is the application description. As with all the work done around xmillum, this is done in form of an XML markup language that describes all the components involved in the system along with their dependencies.

In accordance with the main design goal behind the third generation of xmillum, the application description language only specifies what cannot be delegated to the plugins. This keeps the system general and does not place unnecessary restrictions on the plugins. However, this also reduces the influence of xmillum on the individual components.

This section presents the different parts of the application description language using excerpts from fictitious applications[4].

Plugin Declaration

Central to the application description language is the <*implementation*> tag. It is the tag that is used by xmillum to reference a plugin. A sample plugin reference is given in listing 5.9. It states that the plugin is located in the Java class *org.xmillum.data.XMLFile*. The class is set up using the content of the <*setup*> element. Since every plugin has different setup requirements, the structure of this element is not specified and thus not fixed by the application description language.

[4]Please note that there is no prototype of the third generation xmillum design. The plugins mentioned in this section thus do not exist.

```
<implementation>
  <java-class>
    org.xmillum.data.XMLFile
  </java-class>
  <setup>
    <file source="segmentation-file"/>
  </setup>
</implementation>
```

Listing 5.9: The <*implementation*> element designates the plugin where the implementation is delegated to.

Data Repositories

```
<data-repository name="segmentation-data">
  <data-source name="segmentation-file">
    <physical-file/>
  </data-source>
  <implementation>
    <java-class>
      org.xmillum.data.XMLFile
    </java-class>
    <setup>
      <source name="segmentation-file"/>
    </setup>
  </implementation>
  <validation>
    <schema>
      segmentation.xsd
    </schema>
  </validation>
</data-repository>
```

Listing 5.10: The <*data-repository*> element describes a data repository by giving a reference to the implementation and several setup parameters.

The first integral part of an application description is the definition of the data repositories. A data repository is defined using the <*data-repository*> element. Listing 5.10 illustrates a sample physical file data repository. It first defines that the data-repository fetches its data from a physical file. Next, since the actual work in xmillum is delegated to plugins whenever possible, a reference to the implementation of this data repository is given. This reference is the <*implementation*> element we have presented above. As you can see, the <*setup*> element references the *segmentation-file* data source. It would be possible to place information about data sources directly into the <*setup*> element, but this would deter xmillum to properly keep track of the dependencies (because the contents of <*setup*> is not defined by xmillum).

The data repository is terminated with a <*validation*> element, which is some sort of post-condition for the data repository. This information can be used by xmillum to verify if the data used in the system is valid. If the validity condition is not met, erroneous data has been injected or there is a bug in the system. In our example, the output has to satisfy the XML Schema *segmentation.xsd*.

Data Types

```
<data-type name="text-line">
  <operation name="split">
    <implementation>
      ...
    </implementation>
  </operation>
  <operation name="merge">
    <implementation>
      ...
    </implementation>
  </operation>
  ...
</data-type>
```

Listing 5.11: The <*data-type*> element describes a type and the operations that can be applied to it.

Next in the application description we have the definition of the data types. As already mentioned, the data type definition primarily defines the operations that are possible on a specific type of data.

Listing 5.11 shows a sample data type for a text line. We define two operations on this data type, split and merge. Both operations are implemented by plugins. Our declaration does neither define how the operations are invoked nor what parameters are passed to them, this information will be given when the views are defined.

Views

Now that data repositories and data types are defined, we can go about visualizing them. This is done using the <*view*> element. A sample view is sketched in listing 5.12.

The <*data-source*> tags specify that the view uses the two data repositories *lines-data* and *blocks-data* as input. This information serves as a static definition of a dependency between data repositories and views.

The view itself is implemented in the Java class *org.xmillum.view.BlockBrowser*, which is once again declared using the <*implementation*> tag. The implementation-specific <*setup*> tag defines that the information in our view is presented in two layers with two different styles. Every layer presents one of the data repositories defined previously.

The important thing to notice here is that everything specific to the visual appearance is placed inside the view. Layers and styles, two integral parts of the second generation xmillum, are now defined in the <*setup*> element of *this* particular view rather than on a global level. Also, the notion of *layer* and *style* is wholly unknown to xmillum—it is the view implementation that knows about these concepts.

The <*facility*> tag eventually brings us to the mapping between the user interface facilities and the data manipulation operations. In the example, we assume that the view provides us with a facility for splitting rectangular blocks along a horizontal or vertical split axis the user can choose interactively. If this facility is invoked on an object of type *text-line*, the operation *split* gets called. The <*param*> tags define the parameters that are passed to this operation.

In order to split an object horizontally or vertically, two parameters are required: the direction (horizontal, vertical) as well as the position where the cut should be made. Since these parameters are chosen by the user through the facility, they cannot be specified statically. Instead, they are referenced using the *ref* attribute. The values *facility.direction* and *facility.splitpoint* are therefore simply pointers to the information.

At this point, we will refrain from specifying an entire language for passing the parameters to the operations. This language can be arbitrarily complex, but we are confident that one doesn't need to go that far. Assuming that the facilities are generally quite tightly coupled with the operations they are designed for, simple pointers to data provided by the facility should be sufficient for most applications.

5.3.6 Too General?

The third generation xmillum model is *very* general. Using the right plugins, virtually every kind of data can be treated in almost every imaginable manner.

In fact, contains almost no features specifically for the initial problem it was designed for, document recognition. All the document recognition specific issues enter into play only as as soon as the plugins will be designed. In this respect, the third generation xmillum does not solve any immediate document recognition problems, however, it suggests a method for arranging the different components of the system in order to make them reusable and the system extensible.

To summarize, even if this proposed model may seem too general, we feel confident that this is a very promising approach for designing interactive document recognition applications.

```
<view name="view-1">
  <data-source name="lines">
    <data-repository name="lines-data"/>
  </data-source>
  <data-source name="blocks">
    <data-repository name="blocks-data"/>
  </data-source>
  <implementation>
    <java-class>
      org.xmillum.view.BlockBrowser
    </java-class>
    <setup>
      <style name="yellow">
        <param name="foreground" value="#ffff00"/>
        <param name="transparency" value="0.4"/>
        <param name="fill" value="true"/>
      </style>
      <style name="blue">
        <param name="foreground" value="#0000ff"/>
        <param name="transparency" value="0.4"/>
        <param name="fill" value="true"/>
      </style>
      <layer name="text-lines">
        <layer-data name="lines"/>
        <style name="yellow"/>
      </layer>
      <layer name="text-blocks">
        <layer-data name="blocks"/>
        <style name="blue"/>
      </layer>
    </setup>
  </implementation>
  <facility name="split">
    <operation name="split" type="text-line">
      <param name="direction" ref="facility.direction">
      <param name="splitpoint" ref="facility.splitpoint">
    </operation>
  </facilities>
</view>
```

Listing 5.12: A fictitious view that visualizes data from two data repositories on two layers.

Chapter 6

The xmillum Prototype

What was in the beginning a simple application with a custom markup language for visualizing document recognition data more easily than it was possible with straight HTML, has rather quickly evolved into something much more powerful resulting in more and more ideas for xmillum. The second generation xmillum design has been influenced by the prototype in an incremental manner: The more features were added to the prototype, the more ideas emerged.

This chapter starts with two simple hands-on applications created using the xmillum prototype. These applications demonstrate how easily document recognition data can be visualized and interacted with using our xmillumprototype. All the implemented components of the prototype are then presented more in detail.

6.1 Two Real-World xmillum Applications

For a newspaper recognition project of our research group (see [61] for more details) one task consisted in segmenting the document image into different components. Figure 6.1 shows a sample newspaper document image used for this project and an excerpt of the XML data generated by the segmentation process.

As we can see, the XML data contains several types of nested rectangular regions identified by their location and size (actually some types shown in the following list are not in the excerpt shown in figure 6.1, but they are present in our XML data nonetheless):

- <*image*> regions that contain image data

```
<document image="1-4-a-1.tif">
 <thread x="7" y="3" w="3915" h="3"/>
 :
 <block x="679" y="154" w="2581" h="324">
  <line x="679" y="154" w="2581" h="324">
   <sign x="679" y="157" w="175" h="266"/>
   <sign x="861" y="235" w="134" h="190"/>
   <sign x="1014" y="235" w="143" h="189"/>
   <sign x="1228" y="154" w="247" h="266"/>
   :
  </line>
 </block>
 :
</document>
```

Figure 6.1: A newspaper document image and the result of applying a segmentation algorithm.

- *<thread>* horizontal and vertical rules used to delimit columns and articles
- *<frame>* regions that are completely surrounded by frames
- *<block>* lines of text grouped together
- *<line>* sequence of characters representing a line of text
- *<sign>* individual characters

Using this data and our xmillum prototype, we will now create two applications that solve the following problems:

Visualization In order to see how well the segmentation process worked, we will visualize the data so that we can easily see if it actually corresponds to the document image or if the segmentation is flawed.

Modification If the segmentation data has errors, why not offer the user a possibility for correcting them? The second application shows how under-segmented blocks can be further split.

6.1.1 Data Visualization

The three key concepts of the second generation xmillum design, i.e. styles, objects and layers have already been introduced in section 5.2.1. We will now show how these concepts are used in for our visualization example.

The skeleton of the XSLT stylesheet [1] we are going to create is shown in listing 6.1. The semantics of this skeleton are very simple: whenever an element called <*document*> is met in the input document, this element is replaced with everything that is between the <*xsl:template*> tags. Since our input document is a <*document*> it will immediately match this template. We will now fill in the styles, the objects and the layers we need.

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

<xsl:stylesheet version="1.0"
 xmlns:xmi="http://xmillum.sourceforge.net"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="document">
  <xmi:document>

⋮

  </xmi:document>
 </xsl:template>
</xsl:stylesheet>
```

Listing 6.1: Skeleton of our XSLT stylesheet.

Let's assume that we are not interested in visualizing the individual characters, but that we would like to see all other five types of objects (i.e. images, threads, frames, blocks and lines), each of them drawn with a distinct color. We will therefore disregard the <*sign*> elements.

Since we have five different objects, we introduce five styles with different colors, such as the one for threads shown in listing 6.2.

Next, we need to define the objects that use these styles. We also need an object that allows us to visualize the document image. Listing 6.3 illustrates how this is done. It shows the definition of an object for visualizing document images and one of the five objects for visualizing our rectangular regions. This is done using two different plugins, that are indicated using their Java classes.

[1] For the complete XSLT stylesheet see appendix A.

```
  <xmi:style name="thread-style">
   <param name="foreground" value="yellow"/>
   <param name="transparency" value="0.4"/>
   <param name="fill" value="true"/>
  </xmi:style>
```

Listing 6.2: We need to define five styles such as this one.

```
  <xmi:object name="image" class="iiuf.xmillum.displayable.Image"/>

  <xmi:object name="thread-block" class="iiuf.xmillum.displayable.Block">
   <param name="style" value="thread-style"/>
  </xmi:object>
```

Listing 6.3: We define an object for visualizing the document image as well as five objects for visualizing the five types of rectangular regions.

Note that one of these plugins, *iiuf.xmillum.displayable.Block*, requires the *style* parameter, whereas the other one (*iiuf.xmillum.displayable.Image*) does not need any initialization parameters.

Up to now, we were declaring the part of the internal data structure that was called *meta data* in the previous chapter. As a final step, we now need to create our layers and insert the data that corresponds to the elements in the input data structure.

Since we would like to be able to switch all types of objects on and off individually, we arrange all data in layers—one layer per data type.

The first layer is straightforward. It contains only the document image. The layer is called *Document Image*. The name of the document image is in the *image* attribute of the *<document>* element of the input data structure. The complete code for this layer is shown in listing 6.4. XSLT replaces the expression *{@image}* with the attribute *image* of the current context element, which is the *<document>* element.

The output of the other layers requires a little bit more knowledge of XSLT. We need to iterate over all the *<block>* elements of the document. This can be done using the XSLT's *<xsl:for-each>* construct. We select all *<block>* descendants (direct *and* indirect children) and iterate through them, as shown in listing 6.5. This produces one *<text-block>* element for each *<block>* of the source document.

```
<xmi:layer name="Document Image">
 <image src="{@image}"/>
</xmi:layer>
```

Listing 6.4: The *image* object visualizes an image.

```
<xmi:layer name="Text Blocks">
 <xsl:for-each select=".//block">
  <text-block x="{@x}" y="{@y}" w="{@w}" h="{@h}"/>
 </xsl:for-each>
</xmi:layer>
```

Listing 6.5: This loop also takes into account indirect <*block*> children of the current element.

For every type of object, we need a loop such as the one in listing 6.5. The resulting application visualizes our data as presented in figure 6.2.

6.1.2 Correction of Under-segmented Regions

In order to correct under-segmented regions, we need an event handler that allows us to split regions. In the prototype, we have implemented such an event handler in *iiuf.xmillum.handlers.Split*.

The complete XSLT stylesheet can be found in appendix B. The overall structure stays the same as for the visualization application.

What is added is the declaration of our event handler. This is illustrated in listing 6.6. Here, we associate the handler name *split* to the Java class *iiuf.xmillum.handlers.Split*. Since our event handler does not need any initialization parameters, the <*xmi:handler*> element is empty. Other handlers may require parameters, but the handler in question does not.

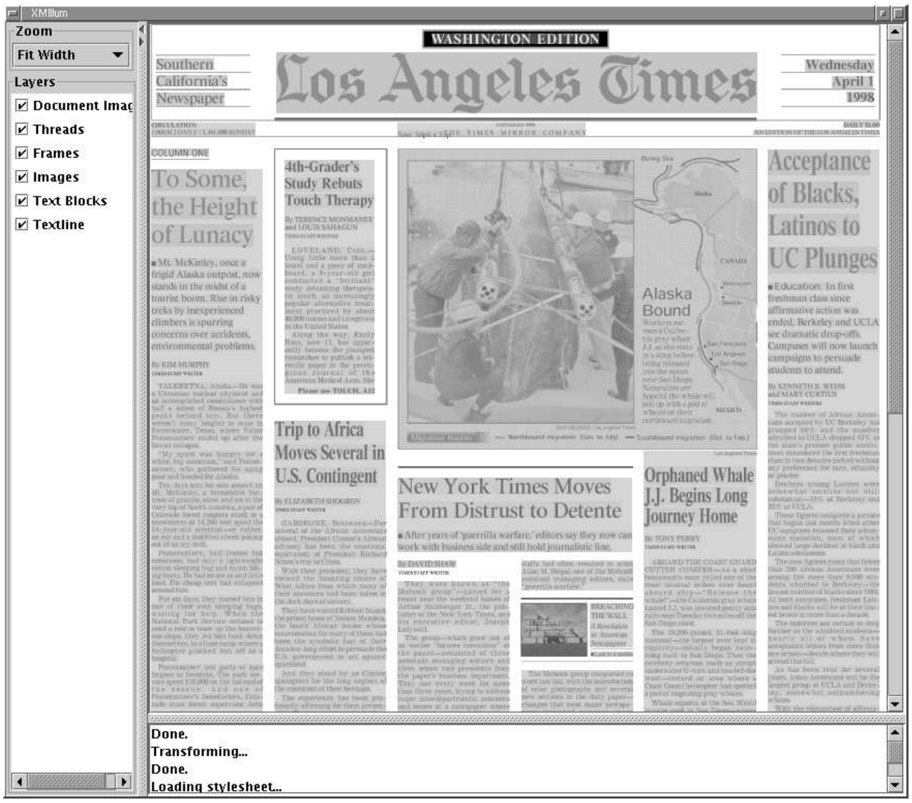
```

<xmi:handler name="split" class="iiuf.xmillum.handlers.Split"/>

```

Listing 6.6: Our event handler is declared.

This handler we have just declared now needs to be called from our objects.


XMIllum
Zoom
Fit Width
Layers
Document Imag
Threads
Frames
Images
Text Blocks
Textline
WASHINGTON EDITION
Los Angeles Times
To Some, the Height of Lunacy
4th-Grader's Study Rebuts Touch Therapy
Trip to Africa Moves Several in U.S. Contingent
New York Times Moves From Distrust to Detente
Orphaned Whale J.J. Begins Long Journey Home
Acceptance of Blacks, Latinos to UC Plunges
Alaska Bound

```
Done.
Transforming...
Done.
Loading stylesheet...
```

Figure 6.2: Using the discussed XSLT stylesheet, xmillum visualizes our data.

Since we would like to see where a block is split, it is not sufficient to invoke the split action when the mouse button is pressed on the object. This is why the handler provides not only one operation, but three of them:

split Split the object at the current location.

turn Change between a horizontal and a vertical split.

show Show using a horizontal or vertical line (depending on the current setting), where a split would be made.

Listing 6.7 shows how these operations are called for the different types of events: When the mouse pointer moves *over* an object, the *show* operation is called, indicating where a split would be made. When the left mouse button is clicked, the current object is split and when the right mouse button is clicked, the direction of the split is changed. Our event handler needs to be specified on all objects we want to be able to split.

```
   <xmi:object name="frame-block" class="iiuf.xmillum.displayable.Block">
    <param name="style" value="frame-style"/>

    <param name="click1" value="split" opt="split"/>
    <param name="click3" value="split" opt="turn"/>
    <param name="over" value="split" opt="show"/>
   </xmi:object>
```

Listing 6.7: The event handler is called for three possible operations.

Finally, a last change concerns the data in the layers: Since we do not want our handler to split the transformed object, we need to associate to every element the corresponding element in the original input data. This is done using the *ref* attribute shown in listing 6.8. The temporary *tmp:refvalue* attribute is a unique identifier that has been embedded by xmillum, as explained in section 5.2.4.

```
   <xmi:layer name="Threads">
    <xsl:for-each select=".//thread">
     <thread-block x="{@x}" y="{@y}" w="{@w}" h="{@h}" ref="{@tmp:refvalue}"/>
    </xsl:for-each>
   </xmi:layer>
```

Listing 6.8: The *ref* attribute contains a value that uniquely identifies the element that generated every <*thread-block*>.

6.2 Components of the xmillum Prototype

The xmillum prototype is driven by the result of an XSL transformation. This data structure has to comply to a particular format, part of which has already been introduced in section 6.1.

An outline of this internal data structure is sketched in listing 6.9. All elements are defined in the xmillum namespace[2].

[2]The namespace is `http://xmillum.sourceforge.net`.

```
<?xml version="1.0" encoding="UTF-8"?>

<xmi:document classpath="..."
              xmlns:xmi="http://xmillum.sourceforge.net">
  ··· Styles ···
  ··· Flag Sets ···
  ··· Handlers ···
  ··· Objects ···
  ··· Tools ···
  ··· Layers ···
</xmi:document>
```

Listing 6.9: xmillum's internal structure.

The following sections show how each of the individual components of xmillum's internal data structure look like and how they are connected. For the sake of textual coherence, the components are not described in the order in which they appear in the internal data structure, but in an order which makes most sense for the reader.

6.2.1 Layers

Document recognition data is often arranged in layers. The bottom layer is the document image, onto which other layers are stacked, as illustrated in figure 6.3.

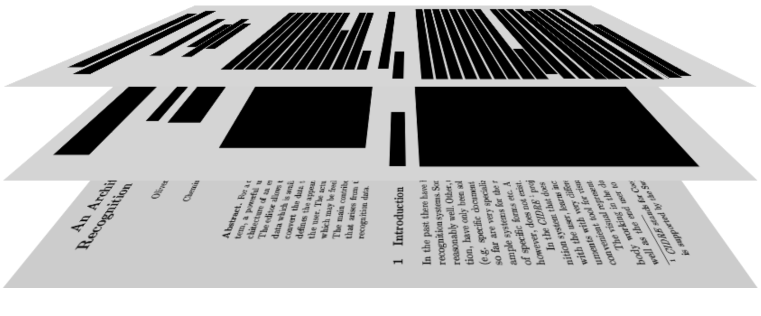


Figure 6.3: A document decomposed into three layers: document image, blocks, and text lines.

The xmillum prototype allows to take advantage of this characteristic by offering the layers functionality. The data to present is organized in layers that can be activated and deactivated individually. This enables the user to mask out unwanted information in order to concentrate on pertinent information only.

Listing 6.10 shows two layers in xmillum. The first layer is called *Document Image* and the second one *Text Blocks*.

The content of the <*xmi:layer*> element cannot be described any further at the moment. It depends upon the data we want to present. We will see how this content looks like and how it is related to the other components of the internal data structure in the next couple of sections.

```
<xmi:layer name="Document Image">
  ...
</xmi:layer>

<xmi:layer name="Text Blocks">
  ...
</xmi:layer>
```

Listing 6.10: Declaration of the two layers *Document Image* and *Text Blocks*.

6.2.2 Objects

Objects are the basic building blocks of xmillum's data visualization. The goal of the object declaration section is to define what plugin is used to draw an item.

This is done by associating an object name to a Java class, as illustrated in listing 6.11. After this declaration, every time an element named *block* appears in one of xmillum's layers, the Java class *package.Block* will be called to draw the graphical object.

```
<xmi:object name="block" class="package.Block">
  ··· Initialization Parameters ···
</xmi:object>
```

Listing 6.11: Declaration of an object class called *block*.

Usage of Declared Objects

Once an object is declared, it can be used to draw a graphical item in a layer. This is illustrated in listing 6.12.

```
<xmi:layer name="Text Blocks">
  <block x="100" y="28" w="324" h="74"/>
</xmi:layer>
```

Listing 6.12: An object *block* is drawn at (100,28).

The employed strategy is very simple: The xmillum prototype iterates over all elements inside the layers and tries to match the element name with a declared object. When a match is found, the corresponding plugin is called and the element is passed to it.

Per-Instance Parameters

xmillum itself does not care about any attributes or children elements which may be present in the element. However, since the element is passed to the plugin, attributes and children elements may be used to pass parameters to the plugin.

In the case of rectangular regions, for example, these parameters might consist of the exact coordinates where the graphical object will be rendered. This is also sketched in listing 6.12.

Initialization Parameters

Listing 6.11 mentions initialization parameters. In fact, it is important to note that xmillum does not define the content of the *<xmi:object>* element. xmillum only "sees" as far as to the *<xmi:object>* element and defines that there need to be *name* and *class* attributes. Everything else is up to the plugin to define.

Listing 6.13 shows how such initialization parameters might typically look: A *style* parameter is given in form of a key–value pair. This parameter will be valid everywhere an object called *block* is used. In this precise example, all *block* objects will be drawn using the style *yellow-style*.

6.2.3 Styles

Using styles, xmillum provides a uniform way of dealing with the description of visual appearance. xmillum's styles are similar to the styles concept in well-

```
<xmi:object name="block" class="package.Block">
  <param name="style" value="yellow-style"/>
</xmi:object>
```

Listing 6.13: A plugin may require initialization parameters.

known applications such as word processors. A style is a set of visual attributes and is identified by a unique name through which the individual components of xmillum have access to them. The xmillum prototype does not provide hierarchical styles as in word processors, where styles can inherit from other each other, but this would be a quite straightforward extension.

A sample style is declared in listing 6.14. It is composed of a name identifying it and an arbitrary number of parameters in the form of key–value pairs. Table 6.1 lists all keys supported by the prototype and their meaning.

```
<xmi:style name="yellow-style">
  <param name="background" value="yellow"/>
  <param name="fill" value="true"/>
  <param name="transparency" value="0.4"/>
</xmi:style>
```

Listing 6.14: Declaration of a style called *yellow-style*.

Emphasizing Objects

Styles are not only used for telling xmillum how a particular type of object should be rendered, but also for telling it how to emphasize objects tagged with a specific tag. For instance, imagine an application where the user is presented all textual blocks of a newspaper document image and he is asked to select the ones that represent article titles. In order to reflect the user's selection, the selected textual blocks need to be drawn with a style different from all the unselected blocks. In this case, xmillum applies *both* styles, the original one as well as the style associated to the tag. The second style modifies the desired attributes of the original style.

An attribute especially implemented for this use is the *hilight* parameter: It highlights the color of the original style and thus produces a style that very well contrasts the original style.

Name	**Description**
foreground, background:	Foreground and background color.
xor:	Use the exclusive–OR drawing mode for drawing pixels.
fill:	Specify the drawing of filled objects rather than hollow ones.
hilight:	Lights the current color in order to result a high-lighted version of it.
transparency:	Draws transparently instead of solid.
fontfamily, -weight, -slope, -size:	Specifies the font for textual objects.
textdirection:	Allows left–to–right and right–to–left text.
resolution:	Resolution in dots–per–inch. Used for calculating the pixel size of textual objects.
stroke-width:	Specifies the width of strokes.

Table 6.1: All possible parameters of styles.

6.2.4 Flag Sets

Many document recognition operations consist in assigning labels to objects. During the logical structure recognition, for example, objects are associated to labels describing the role of these objects in the document. And a character recognition process boils down to assigning the correct character-labels to objects.

The xmillum prototype provides native support for labels through the *flag set* feature. A set of labels that can be assigned to objects is called a *flag set*.

To every label can be associated a style. If an object is flagged, it will be redrawn immediately with the new style.

Listing 6.15 shows how a flag set is declared. A flag set called *title-type* for labeling titles is defined. It contains the three labels *chapter*, *section* and *subsection*. Chapter titles are drawn using the *yellow-style* style, sections using *blue-style* and subsection using *red-style*.

Flags and Plugins

Flags are mainly useful for the implementors of plugins.

Whenever an object is labeled, an event is triggered. By subscribing to these events, a plugin can react accordingly. A plugin that connect xmillum to an

```
<xmi:flag name="title-type">
  <value name="chapter" style="yellow-style"/>
  <value name="section" style="blue-style"/>
  <value name="subsection" style="red-style"/>
</xmi:flag>
```

Listing 6.15: Definition of a flag set called *title-type*.

external classification engine, for instance, might use these events to feed the objects selected by the user to the external engine for learning or recognition purposes.

Instead of a real-time notification when a label changes, a plugin can also get access to all objects labeled with a given label. This allows for interaction scenarios such as "Select all objects you would like to correct and click *Next* to go on."

Selection, a Predefined Flag Set

There is a special, builtin flag set, the selection. It contains only one label and is used by the xmillum prototype to keep track of the objects that are selected. Therefore this flag set only contains one label, *selected*. All objects that are not labeled with this label are unselected.

Associated to this label is a style that highlights the original style using the *hilight* parameter from table 6.1.

6.2.5 Handlers

Up to now, we have dealt with visualization, as well as with some xmillum-internal labeling. Since this thesis is about *interactive* document recognition, there need to be means for modifying the data that is visualized. This is where event handlers enter.

Event handlers have been created in the style of events in HTML. Whereas in HTML events are handled by scripts (typically using the JavaScript language), in xmillum an event handler is a plugin that gets called whenever an event is triggered.

The <*xmi:handler*> element declares and configures a plugin so it is used as an event handler. In listing 6.16, an event handler called *popup* is declared whose code is in the plugin *package.PopupMenu*.

As with the other plugins in xmillum, the initialization parameters are defined by every event handler. xmillum only defines the <*xmi:handler*> element as well as the *name* and *class* attributes.

```
<xmi:handler name="popup" class="package.PopupMenu">
  <param name="flag" value="type"/>
  <param name="allow-clear" value="true"/>
</xmi:handler>
```

Listing 6.16: Declaration of an event handler *popup*.

At the source of the events is the xmillum environment. All events are routed through the xmillum environment to the objects in question. The object then invokes a handler if it is configured to do so.

An event handler, when it is called, has full access to all the internal data structures of the xmillum prototype. It has therefore all the possibilities to apply all kinds of modifications.

Supported Types of Events

Since all events are routed through the xmillum environment to the objects, it is the environment that defines the types of events that are supported. The xmillum prototype supports only the following couple of events:

click1...3 Triggered when mouse button 1, 2 or 3 is clicked (i.e. pressed and released within a given time *t*) inside the rectangular bounding box of an object.

press1...3 Triggered when mouse button 1, 2 or 3 is pressed (i.e. not released for a given time *t*) inside the rectangular bounding box of an object.

over Triggered when the mouse moves over the rectangular bounding box of an object.

Events Triggered by Objects

Upon reception of an event, the objects are free to hand them on to handlers. Since the xmillum environment only checks if the event has taken place inside the rectangular bounding box of an object, an object which is not rectangular might want to further analyze the exact location to decide if the event is really relevant or not.

The handler that is then invoked by an object is typically defined at object declaration time. Listing 6.17 shows an object that calls a handler declared under the name *popup* when the user presses the left mouse button on the object in question.

```
<xmi:object name="block" class="package.Block">
  <param name="style" value="yellow-style"/>
  <param name="press1" value="popup"/>
</xmi:object>
```

Listing 6.17: This object calls the *popup* handler when the left mouse button is pressed on it.

Passing a Further Parameter to the Handler

As we have already seen in the correction application in section 6.1.2, there are handlers that provide multiple actions.

In order to tell the handler what operation to invoke, a further parameter needs to be sent to it. This is illustrated in listing 6.18. All mouse buttons invoke the same event handler, but a different option parameter can be given along with the event in order to decide what operation to invoke.

```
<xmi:object name="block" class="package.Block">
  <param name="style" value="yellow-style"/>
  <param name="press1" value="popup" opt="perform-operation-1"/>
  <param name="press2" value="popup" opt="perform-operation-2"/>
  <param name="press3" value="popup" opt="perform-operation-3"/>
</xmi:object>
```

Listing 6.18: All mouse buttons call the same event handler, but a different operation can be chosen using the *opt* attribute.

Finding the Object that Triggered the Event

When a handler is called, it also receives the element that has triggered the event. Since we are working on a transformed version of our XML data, this information is not of much use. In order to be able to identify the element in the original, untransformed input data, this information needs to be made available somehow.

In section 5.2.4 we have explained how this is done in our prototype: When the original XML data is read, a unique identifier is added to every element. This is done in the form of the attribute *tmp:refvalue*. Listing 6.19 shows the attributes that are added to input data when it is read in. This value is then used as a link between the transformed version of our data and the original XML data (explained in detail in section 5.2.4).

```
<document image="1-4-a-1.tif" tmp:refvalue="1">
  <thread x="7" y="3" w="3915" h="3" tmp:refvalue="2"/>
  ...
</document>
```

Listing 6.19: When an XML document is read, *tmp:refvalue* attributes with unique values are added in order to identify every element.

6.2.6 Tools

The most flexible way of extending the xmillum prototype are tools. These plugins can serve many different purposes, be it for the implementation of custom views whenever the default WYSIWYG-style view is too restricting, for interfacing xmillum to external programs (e.g. a recognition engine) or simply for guiding a user through a well-defined interaction step.

The declaration of a tool that guides the user through an interactive object labeling process is shown in listing 6.20. The plugin *package.Wizard* is used for this purpose.

As already seen in the declaration of objects, only the *<xmi:tool>* element is imposed by xmillum. Everything contained in this element is specific to the tool and is for initializing it.

Similar to event handlers, tool are also granted full access to the internal data structures of the xmillum prototype and can do therefore everything they like to our XML data.

However, since tools are not invoked by xmillum events, they need implement their own event handling. Typically, a tool creates an additional window and reacts on events concerning this window.

6.3 Plugins Implemented in the Prototype

This section presents the different plugins that have been implemented in the prototype: (Graphical) Objects, Event Handlers and last but not least Tools.

```
<xmi:tool class="package.Wizard">
  <step flag="type" value="title">
    <prompt>Select title zones.</prompt>
  </step>
  <step flag="type" value="author">
    <prompt>Select author zones.</prompt>
  </step>
  <step flag="type" value="abstract">
    <prompt>Select abstract zones.</prompt>
  </step>
</xmi:tool>
```

Listing 6.20: Declaration of a tool for guiding a user through a manual labeling process.

6.3.1 Graphical Objects

Rectangular Blocks

For the presentation of document recognition results, the ability to visualize simple rectangular blocks is without doubt one of the primary requirements. Many basic document recognition algorithms result in such rectangular blocks: bounding boxes of connected components, character, word and line segmentation, etc.

The plugin *iiuf.xmillum.displayable.Block* can be used to draw rectangular blocks with many visual attributes. Blocks are declared using the excerpt shown in listing 6.21. The *Block* plugin takes several configuration parameters in form of key–value pairs, the most important being the *style* parameter. It defines the exact visual appearance of blocks drawn with particular instance of this plugin. All other parameters define handlers that are called when specific events occur. The list of all possible key–value pairs is presented in table 6.2.

```
<xmi:object name="character" class="iiuf.xmillum.displayable.Block">
  <param name="style" value="character-style"/>
</xmi:object>
```

Listing 6.21: Declaration of the *Block* plugin.

A declared *Block* plugin can now be used as sketched in listing 6.22. The top left coordinate as well as the width and height are given as x, y, w and h

Name	Description
style	Specifies the style to be used for rendering this block. The style must be declared before.
click1...3	Specifies the handler that is called when mouse button 1 to 3 is clicked (i.e. pressed and immediately released) on this rectangle.
press1...3	Specifies the handler that is called when mouse button 1 to 3 is pressed on this rectangle.
over	Specifies the handler that is called when the mouse button is over this rectangle.

Table 6.2: The key–value pairs of the *Block* object.

attributes.

```
<xmi:layer name="My-Layer">
  ...
  <character x="..." y="..." w="..." h="..."/>
  ...
</xmi:layer>
```

Listing 6.22: A declared block plugin takes four parameters describing the location and size of the rectangle to visualize. The link between <*character*> and the *Block* plugin is the object name *character* from listing 6.21.

Bitmap Image

Using the graphical object *iiuf.xmillum.displayable.Image*, bitmap images can be visualized in xmillum. *Image* is a requirement for creating WYSIWYG views.

Using a standard install of the Java Virtual Machine, only GIF and JPEG images (PNG and JPEG since Java 1.4) can be read. In the document recognition domain, other image formats are more common (e.g. TIFF for scanned documents), *Image* optionally uses JAI, the Java Advanced Imaging Toolkit[3], a

[3]JAI is available at `http://java.sun.com/products/java-media/jai/`

Java add-on from Sun. JAI enlarges Java by a significant quantity of additional image formats as well as a huge library of image processing operations. On the Sparc and Intel i386 platforms, JAI even uses native implementations of some operations, what speeds them up significantly compared to their pure Java counterpart. The use of natively accelerated operations is completely transparent, i.e. if JAI is properly installed and on a platform where native processing is supported, image are automatically processed using the native versions of the image processing operations.

If JAI is installed, xmillum uses it for reading images and for scaling them, if it is not, only the stock Java classes are used.

The *Image* class is declared using the excerpt shown in listing 6.23. It takes an optional parameter *visible*, which defaults to *true*. If *visible* is set to *false*, the image will be invisible. In some situations, this can be a useful feature.

```
<xmi:object name="image" class="iiuf.xmillum.displayable.Image">
  <param name="visible" value="true"/>
</xmi:object>
```

Listing 6.23: The *Image* object can be declared with the optional *visible* parameter.

Once declared, the *Image* object is used as presented in listing 6.24. The *src* attribute is a URL pointing to the image to visualize. This URL is relative to the location of the original XML file (i.e. the XML file that is viewed). If instead of the attribute *src*, the attribute *asksrc* is present, a dialog pops up and asks the user to choose an image to visualize.

```
<xmi:layer name="My-Layer">
  ...
  <image src="any-image.tif"/>
  ...
</xmi:layer>
```

Listing 6.24: The *src* attribute contains the URL of the image to show.

By default, images are drawn at the top left corner. Using the x and y attributes, it is possible to change the location of the image.

Polygon

As the name suggests, the plugin *iiuf.xmillum.displayable.Polygon* allows to draw arbitrary polygons. A sample declaration is shown in listing 6.25. The polygon plugin takes exactly the same parameters as the *Block* plugin for drawing rectangles (see table 6.2).

```
<xmi:object name="polygon" class="iiuf.xmillum.displayable.Polygon">
  <param name="style" value="any-style"/>
</xmi:object>
```

Listing 6.25: The *Polygon* plugin and its most important parameter.

Since an actual polygon is defined by an arbitrary number of vertices, the coordinates therefore need to be given differently than with the plugins seen so far. Listing 6.26 shows how the coordinates of the vertices are passed to the *Polygon* plugin.

```
<xmi:layer name="My-Layer">
  ...
  <polygon>
    <point x="..." y="..."/>
    <point x="..." y="..."/>
    <point x="..." y="..."/>
    ...
  </polygon>
  ...
</xmi:layer>
```

Listing 6.26: Every *Polygon* requires an arbitrary number of points.

Text Block

The *iiuf.xmillum.displayable.TextArea* allows to visualize text blocks. It writes text into a rectangular area.

The *TextArea* plugin is declared as illustrated in listing 6.27. It takes the same parameters as the *Block* plugin (see table 6.2), the most important being the *style* parameter. In the style, the exact font characteristics are described (refer to table 6.1).

```
<xmi:object name="text" class="iiuf.xmillum.displayable.TextArea">
  <param name="style" value="title-text"/>
</xmi:object>
```

Listing 6.27: Declaration of the *TextArea* plugin.

In listing 6.28, an invocation of the *TextArea* plugin is shown. The four attributes x, y, w and h describe the rectangle containing the text, while the *text* attribute contains the text to write.

```
<xmi:layer name="My-Layer">
  ...
  <text text="Write this text." x="..." y="..." w="..." h="..."/>
  ...
</xmi:layer>
```

Listing 6.28: The text to render by the *TextArea* plugin is passed in the *text* attribute.

6.3.2 Event Handlers

Select

As introduced in section 6.2.4, xmillum supports the concept of a selection. This means that objects can be selected and deselected. Whenever the selection changes, all plugins that are interested in this kind of change are notified. This can be used to have multiple views of the same data among which the selection is synchronized. Selecting an object in one view will automatically select it in the other view and vice-versa.

The handler *iiuf.xmillum.handlers.Select* changes the selection state of an object. It's use is straightforward, as illustrated in listing 6.29. This listing makes all objects of type *block* selectable.

PopupFlagger

The *iiuf.xmillum.handlers.PopupFlagger* event handler allows a user to label objects using a simple popup menu such as shown in figure 6.4. The possible labels are given in a flag set as introduced in section 6.2.4.

```
<xmi:handler name="select" class="iiuf.xmillum.handlers.Select"/>

<xmi:object name="block" class="iiuf.xmillum.displayable.Block">
  <param name="style" value="my-style"/>
  <param name="click1" value="select"/>
</xmi:object>
```

Listing 6.29: Usage of the *Select* handler.

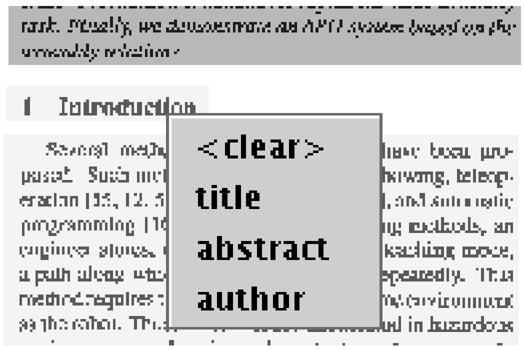


Figure 6.4: The *PopupFlagger* allows to label objects with a popup menu.

Listing 6.30 shows a sample usage of the plugin. Lines 1 to 5 declare a flag set called *type*, that contains three different flags with their corresponding styles. The declaration of the plugin is given in lines 7 to 10, where the beforementioned flag set is referenced using the *flag* parameter. The *PopupFlagger* plugin constructs its menu according to the flag values of this flag set. The *allow-clear* parameter tells the plugin to add a menu entry that removes all flags from the object in question. Finally, line 14 tells xmillum to call the *PopupFlagger* whenever mouse button 3 is pressed on an object.

Once set, the labels can be collected by other xmillum plugins. For instance, it would be possible to use this information for training recognition algorithms.

Info

The plugin *iiuf.xmillum.handlers.Info* works in conjuction with another plugin, the *iiuf.xmillum.tool.InfoWindow* tool.

The *InfoWindow* tool and the *Info* are used to show a simple informational message in a window. The window is opened by the *InfoWindow* tool. The

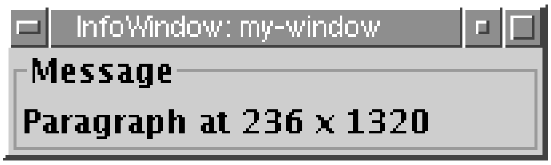
```
<xmi:flag name="type">
  <value name="title" style="title-style"/>
  <value name="author" style="author-style"/>
  <value name="abstract" style="abstract-style"/>
</xmi:flag>

<xmi:handler name="popup" class="iiuf.xmillum.handlers.PopupFlagger">
  <param name="flag" value="type"/>
  <param name="allow-clear" value="true"/>
</xmi:handler>

<xmi:object name="block" class="iiuf.xmillum.displayable.Block">
  <param name="style" value="default-style"/>
  <param name="press3" value="popup"/>
</xmi:object>
```

Listing 6.30: Usage of the *PopupFlagger* plugin.

Info handler is then used to send messages to that window. Figure 6.5 shows an example of such a message.



Paragraph at 236 x 1320

Figure 6.5: A message shown using the *Info* handler.

Listing 6.31 shows how the *Info* handler is used. Line 1 defines a window called *my-window* used to display messages sent from the handler. Lines 3 to 5 define the handler that sends the messages. In line 9, this handler is referenced. It will be called whenever the mouse pointer moves *over* a block object. Lines 14 and 15, eventually, show an object that triggers the handler. The text that is shown is given in the *info* attribute of the element. Whenever the mouse pointer moves over such an object, a message such as shown in figure 6.5 appears.

Split

The *iiuf.xmillum.handlers.Split* handler is tailored to a very specific application, unlike the general-purpose handlers presented so far. Suppose that you

```
<xmi:tool class="iiuf.xmillum.tool.InfoWindow" name="my-window"/>

<xmi:handler name="info" class="iiuf.xmillum.handlers.Info">
  <param name="name" value="my-window"/>
</xmi:handler>

<xmi:object name="block" class="iiuf.xmillum.displayable.Block">
  <param name="style" value="default-style"/>
  <param name="over" value="info"/>
</xmi:object>

<xmi:layer name="Zones">
  <xsl:for-each select="paragraph">
    <block x="{@x}" y="{@y}" w="{@width}" h="{@height}"
           info="Paragraph at {@x}, {@y}"/>
  </xsl:for-each>
</xmi:layer>
```

Listing 6.31: Declaration and use of the *Info* handler and *InfoWindow* tool.

have a document that is segmented into rectangular blocks representing paragraphs, text lines, words etc. Some of these blocks may be incorrectly segmented and you might want to correct such errors directly on the document. The *Split* handler allows you to do exactly this: You may choose visually what blocks you want to split and where exactly you want to split them.

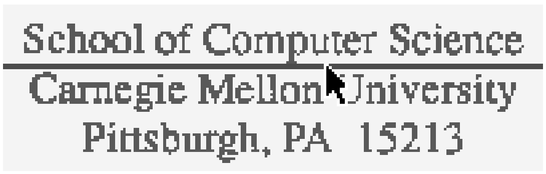


Figure 6.6: The *Split* handler allows to split blocks along a separation line.

Figure 6.6 shows the handler in action. When the mouse pointer moves over the block in question, a line appears showing where the block will be split in two. A click on the left mouse button then splits the block in two.

As illustrated in listing 6.32, this handler does not require any initialization parameters when it is declared. However, referencing the handler is somewhat special compared to the handlers seen so far. We want this handler to become active when the mouse pointer is over the block in question (for drawing the separation line), but also when the mouse button is clicked (for splitting the block). This means that the same handler is called for different purposes. In order to tell the handler what it needs to do, we have added the *opt* attribute to the <*param*> element. It is a simple parameter that is passed to the handler. In lines 5 to 7, the handler is referenced with the three parameters *split*, *turn* and *show*. The *split* parameter tells it to split the current block. The *turn* switches between vertical and horizontal splitting. This switch is done when the right mouse button is clicked. Finally, using the *show* parameter, the separation line is drawn whenever the mouse pointer is over a block.

```
1  <xmi:handler name="split" class="iiuf.xmillum.handlers.Split"/>
2
3  <xmi:object name="block" class="iiuf.xmillum.displayable.Block">
4    <param name="style" value="default-style"/>
5    <param name="click1" value="split" opt="split"/>
6    <param name="click3" value="split" opt="turn"/>
7    <param name="over" value="split" opt="show"/>
8  </xmi:object>
9
10 <xmi:layer name="Zones">
11   <xsl:for-each select="line">
12     <block x="{@x}" y="{@y}" w="{@width}" h="{@height}"
13            ref="{@tmp:refvalue}"/>
14   </xsl:for-each>
15 </xmi:layer>
```

Listing 6.32: Using the *Split* handler.

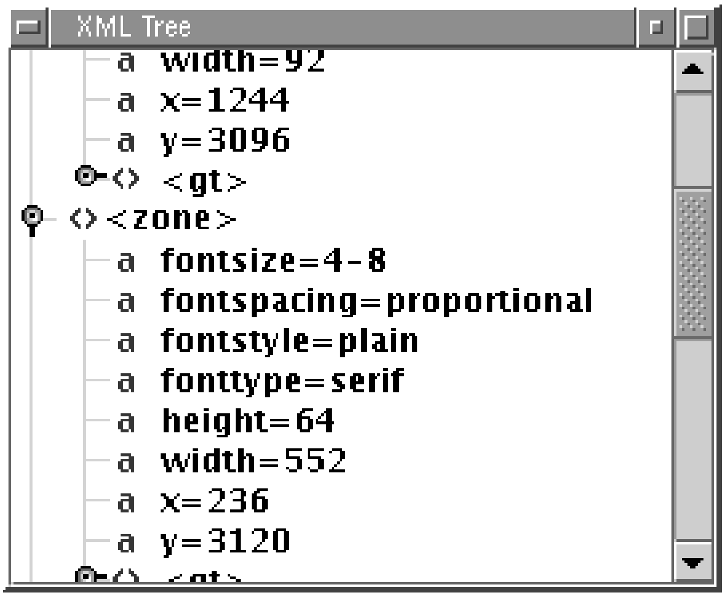
When the *Split* handler is called to split a block, it looks up the element referenced by the *ref* attribute. The size of the two new blocks is then calculated and the blocks are inserted into the XML structure. xmillum is then asked to rerun the XSLT transformation in order for the change also to be reflected in the transformed structure.

This handler is a proof of concept implementation. It shows that it is possible to use xmillum for manually correcting segmentation results. For a real application, it would need to be enhanced in several ways. For instance, it does not take into account the location of children elements of the element that is split. If a block representing a paragraph is split, all children of this block end up in one of the resulting partial blocks. If the children are text lines, they should end up in one block or the other depending on their location. In some circumstances, it might even be desirable to recursively split elements contained in the block to split. The exact specifications depend on the application in question.

6.3.3 Tools

XMLTree

The *iiuf.xmillum.tool.XMLTree* plugin is a very basic example of an additional view. It shows part of the original data structure in form of a tree, such as shown in figure 6.7. The *XMLTree* keeps the selections in the two views synchronized, selecting an element in the tree will automatically select it in the WYSIWYG view and vice-versa.



```
a width=92
a x=1244
a y=3096
<> <gt>
<> <zone>
a fontsize=4-8
a fontspacing=proportional
a fontstyle=plain
a fonttype=serif
a height=64
a width=552
a x=236
a y=3120
<> <gt>
```

Figure 6.7: The *XMLTree* gives access to the original data structure in form of a tree.

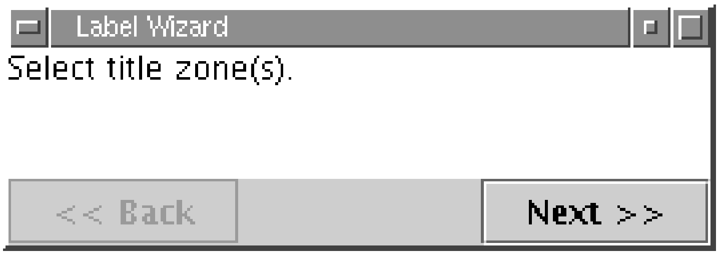
The *XMLTree* tool is used as shown in listing 6.33. It offers the following possibilities: The *start* attribute is a pointer to the root node of the tree. This allows for partial trees of the original XML structure to be represented. *showattributes* instructs *XMLTree* to also show attributes in the tree. If *showattributes* is 0 (zero), only elements are present in the tree. Finally, if attributes are shown, not necessarily all attributes are useful. The *filter* attribute is a comma-separated list of attributes you do not wish to include in the *XMLTree*.

```
<xmi:tool class="iiuf.xmillum.tool.XMLTree"
          start="{@tmp:refvalue}"
          showattributes="1"
          filter="id"/>
```

Listing 6.33: Using the *XMLTree* tool.

LabelWizard

The *iiuf.xmillum.tool.LabelWizard* tool has the same goal as the *PopupFlagger* handler seen in section 6.3.2, but it works differently. Instead of showing the user all choices in a popup menu, the labeling of objects is done step by step one class at a time using a Wizard-like user interface, hence the name. Figure 6.8 shows a simple dialog shown by this tool. Objects that are selected while this message is shown are tagged as title. A click on the *Next* button takes the user to the next message and then selected objects will be tagged differently.



Select title zone(s).

Figure 6.8: The *LabelWizard* asks the user to select objects of a given type.

The *LabelWizard* tool is used as shown in listing 6.34. Every step is modeled in a <*step*> element. For instance, lines 2 to 4 tell our tool that all selected objects should be tagged with the flag *title* of the flag set *type* while the user is shown a textual message asking him to select the title zones.

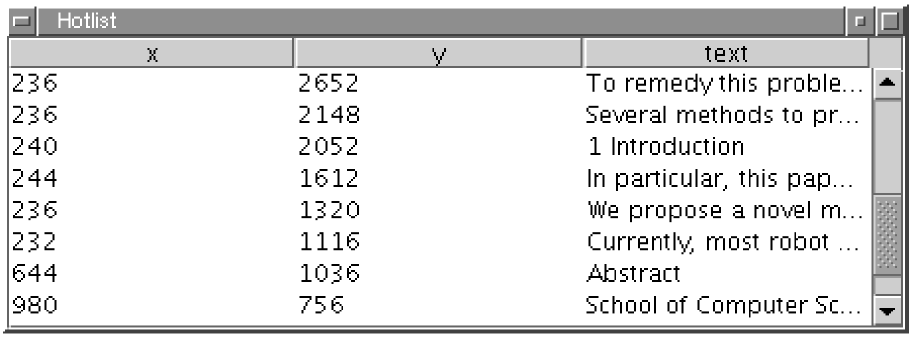
Of course, in order to work, the *LabelWizard* tool requires a correctly declared flag set. Also, it requires object selections to be set up. This can be done using the *Select* handler presented in section 6.3.2. The *LabelWizard* tool has access to the internals of xmillum where it intercepts changes in the selection state of objects and acts accordingly. No handler is required here.

```
<xmi:tool class="iiuf.xmillum.tool.LabelWizard">
  <step flag="type" value="title">
    <prompt>Select title zone(s).</prompt>
  </step>
  <step flag="type" value="author">
    <prompt>Select author zone(s).</prompt>
  </step>
  <step flag="type" value="abstract">
    <prompt>Select abstract zone(s).</prompt>
  </step>
</xmi:tool>
```

Listing 6.34: Initialization of the *LabelWizard* tool.

Hottable

As the name suggests, the *iiuf.xmillum.tool.Hottable* tool allows to show *hot* objects in tabular form. It provides the user with a shortcut to interesting objects. It could for example be used to show the user a list of all objects whose recognition confidence value is less than a given threshold. These objects are then to be inspected manually by the user. Figure 6.9 shows a sample table produced by the *Hottable* tool.



x	y	text
236	2652	To remedy this proble...
236	2148	Several methods to pr...
240	2052	1 Introduction
244	1612	In particular, this pap...
236	1320	We propose a novel m...
232	1116	Currently, most robot ...
644	1036	Abstract
980	756	School of Computer Sc...

Figure 6.9: The *Hottable* tool shows objects in tabular form.

Listing 6.35 shows how the *Hottable* tool is used. The *columns* attribute in line 2 lists the columns the table will be composed of. Every row is then given with a *<row>* element and the chosen attributes. Using the *<xsl:for-each>* instruction in line 3, we iterate over all paragraphs whose confidence value is less than 80% [4]. Of these paragraphs, the x any y coordinates as well as the 32 first characters of the contained text will be shown in our table.

```
<xmi:tool class="iiuf.xmillum.tool.Hottable"
          columns="x,y,text">
  <xsl:for-each select="paragraph[@confidence &lt; 0.8]">
    <row x="{@x}" y="{@y}"
         text="{substring(text(), 1, 32)}"
         ref="{@tmp:refvalue}"/>
  </xsl:for-each>
</xmi:tool>
```

Listing 6.35: Initialization of the *Hottable* tool.

The *Hottable* tool also keeps track of object selections and synchronizes them with the main view of xmillum.

[4]The < character may not appear in XSLT expressions, which is why it needs to be written as &lt;.

Chapter 7

Conclusion and Perspective

The main result of this thesis is without doubt the xmillum prototype, which is a rather simple but nonetheless powerful tool. This chapter presents other projects using xmillum, it gives possible directions for further research and improvements and it summarized the accomplished work.

7.1 xmillum in other Projects

This thesis presents an attempt at creating a modular and reusable software framework for interactive document recognition. The result, xmillum, in itself is very useful as a visualization tool, but, only integrated into another applications, it can show its full potential.

The next couple of sections present projects that benefit from the use of xmillum.

7.1.1 *2(CREM)*

The *2(CREM)* project of Lyse Robadey [61] has already been briefly presented in section 2.2.8. It is a general classification technique that makes use of user input to incrementally and interactively build document models. The approach has not only proved to be capable of labeling physical document objects with logical tags, but also to merge consecutive text lines into blocks, a not so common use of a general classification technique.

In order to prove the feasibility of the classification technique, a flexible tool for visualizing the different aspects of document objects was required. This tool also needed to be capable of handling classification requests made by the user.

Since the *2(CREM)* approach builds the underlying document model in an incremental manner, every classification change made by the user may modify the document model in such a way that it affects the classification of other objects. In other words, by manually classifying an object, the classes of the other objects may change immediately because the old classifications no longer respect the new document model. This suggests that the *2(CREM)* user interface be online, that is, directly connected to the *2(CREM)* engine. This allows to send all classification actions made by the user to the *2(CREM)* engine and to update the visualization accordingly to reflect the changed situation.

All requirements can be fully satisfied by xmillum: The visualization of different types of document objects can be solved with the appropriate XSLT stylesheets. For interfacing the user interface with the *2(CREM)* engine, an xmillum tool has been developed. It presents two lists, one containing all the object elements to classify and the other containing the available classes. Selecting an element in one list automatically selects its class in the other list. At the same time, xmillum's main view scrolls to the element in question, showing the user where this element is located on the document image. The class can be changed by simply clicking on the new class. This change is then sent to the *2(CREM)* engine, which integrates it into its document model and reclassifies the other elements according to the new model.

The resulting application demonstrates the power of xmillum: Creating a user interface especially for *2(CREM)* from scratch would have been a very time-consuming task. With xmillum, many subtasks (e.g. the visualization) were already solved, it was only necessary to add a relatively small component which interfaces xmillum with the *2(CREM)* engine. This allows to concentrate on the main problem at hand, in the present example the creation of the *2(CREM)* engine. It can be said that the development of the *2(CREM)* prototype did heavily benefit from xmillum. Without doubt, the same is also true the opposite way around: xmillum would probably not have all the features it currently offers had it not been used for *2(CREM)*.

7.1.2 Edelweiss

The idea of Nicolas Roussel's Edelweiss [63] was to create an environment for making document recognition services available through the Web.

Edelweiss was one of the first attempts of solving the document recognition problem with Web Services. While at that time, the term "Web Services" did not yet resound throughout the land, the underlying concepts were already known at the time Edelweiss was created.

In Edelweiss, document processing chains are described in an XML structure called *eDocument*. An eDocument defines the source of the document to process, as well as the individual elementary document processing operations called *jobs*. eDocuments are stored in *repositories*, from where they are fetched by *schedulers* to execute part of the processing chain. In Edelweiss, all resources are addressed using URLs and data is transferred over HTTP. Thus, repositories are simply web servers serving eDocuments as well as source documents and schedulers are elementary document processing modules capable of fetching their input from web servers.

xmillum is integrated in Edelweiss as a document viewer for all types of data produced by the system. By using xmillum instead of an especially crafted viewer, more time could be invested on the other components of Edelweiss, while at the same time a very customizable viewer was provided.

7.1.3 DocMining

A very ambitious project using the xmillum framework is DocMining [13], a collaborative effort of the DocMining Consortium. The consortium consists of France Telecom R&D and four academic institutions in France and Switzerland.

The aim of DocMining is to create a general document interpretation framework for processing large numbers of heterogeneous documents involving not only textual components, but also graphical objects. Such systems tailored to very specific applications and domains exist, however, adapting them to other domains is a major effort because they rely largely on domain knowledge. DocMining tries to overcome this problem by offering a flexible plugin oriented architecture. Domain knowledge may be included in plugins, but it is not included in the platform itself.

The DocMining platform can best be characterized with the following key features:

Document-centered All communication goes through the document. All data that is produced during a processing stage is put into the document. This allows communication between processing stages without an outside communication channel.

Plugin architecture New elementary processing stages may be integrated into the system in the form of plugins.

Scenario-based Know-how is stored in the form of scenarios. These are simply programs that describe the sequence of processings that need to be applied to a given type of document in order to achieve a goal.

In the DocMining project, xmillum is the key user interface component. It is used for two tasks: For controlling the DocMining engine (also called the *controller*) and for the visualization of results. The controller is responsible for executing scenarios and for managing data and access to the data.

The xmillum framework has been chosing for this project mainly because of its extensibility. Many DocMining extensions to xmillum are currently being implemented.

7.1.4 PLANET

In his project PLANET [31, 32] (short for Physical Layout Analysis of Complex Structured Arabic Documents Using Artificial Neural Nets), Karim Hadjar tackles the layout analysis problem using neural networks.

In this project, xmillum is used for training the neural network by correcting over- and undersegmentation. Custom plugins have been developed for this purpose. The project benefits from the robust visualization possibilities of xmillum.

7.2 Future Work

7.2.1 XML Schemas and Plugins Exporting Their Interfaces

In xmillum, as it is presented in this thesis, it is very difficult to verify the validity of the internal XML format. The reason for this is that the internal format depends on the plugins that are used. Every plugin requires its data in its own specific format, and there is no way for xmillum to know these requirements.

It would be an interesting project to have all plugins export their requirements on the input data. This can be considered like part of a contract as advocated in Bertrand Meyer's *Design by Contract* software development methodology [53]: The plugin works if the input data meets given criteria. Doing so would allow the internal structure to be validated instead of having every plugin do the work.

The interesting part is how this requirements should be presented. Since there are techniques for validating XML structures, the idea to use these techniques is straightforward.

At the beginning of the work on this thesis, the only method available for describing and validating XML structures was the DTD (Document Type Definition), which lacks many important features. For instance, DTDs do not support the notion of types. Content is simply content, it is impossible to restrict it any further. From the point of view of xmillum, the most important deficiency of DTDs, however are the lack of Namespaces support.

Later on during the work on this thesis, another technique for representing XML grammars became known: XML Schemas [80, 81, 82]. XML Schemas solve many of the problems of DTDs. Using XML Schemas, it is possible to restrict the content of elements and attributes using elementary data types (integers, strings etc.). Based on these types, new types can be declared. XML Schemas also fully support XML Namespaces, making it possible to mix different grammars from different Namespaces, while retaining the possibility to validate the structure. But, the best of all is that XML Schemas are XML documents themselves and as such can be processed just like other XML documents.

Using XML Schemas, it should be possible to use a unique Namespace for every plugin and a corresponding XML Schema to validate content.

7.2.2 Generalizing xmillum

Considering how well xmillum works for presenting data from a very specific domain, it might be desirable to apply it also to other domains.

Clearly, given the early efforts around the Extensible Stylesheet Language (XSL), the idea of having one program or technology to visualize most or all kinds of XML data is hardly new. However, considering how long it took until half-decent implementations of XSL (especially the XSL-FO part) were showing up, the chosen approach of creating a huge vocabulary for specifying the formatting semantics has probably not been the most effective of approaches.

xmi relies on external plugins that offer a specific service. Given the right plugins, the xmillum approach should allow to visualize virtually all kinds of data. But, doing so will also put other decisions of xmillum in question. For instance, the layered structure of xmillum visualizations is very well suited for presenting document recognition data, but it may be troublesome for other domains.

No doubt using xmillum in other domains does not just imply new plugins, but also more far-reaching changes.

7.2.3 Hierarchical Nature of XSLT

The XML language is very well suited for hierarchical structures. More complex structures, such as graphs-like structures can be represented, but doing so quickly gets messy. The same holds for transformation rules formulated in XSLT.

For some documents, the hierarchy is not the most natural structure. This is true especially for intermediary document recognition data which is not yet completely structured, but laid out two-dimensionally.

Having another data format such as XML, but which is better suited for data not structured hierarchically, might help for working with such data.

7.2.4 Code, Code, Code

xmillum is a prototype that may be useful for very specific applications. However, in order to serve a larger public, it needs to be extended. To facilitate this, the whole xmillum code has been published at SourceForge, the world's largest Open Source software development website [1]. Everybody is invited to contribute!

7.3 Accomplished Work

We have discussed aspects of document recognition in general and aspects and motivations for interactive document recognition in particular.

We have shown how the family of XML technologies and a modern programming language such as Java can be combined to create a modular visualization and editing system for interactive document recognition. The resulting xmillum prototype is a *proof of concept* quality software package with many possibilities for improvements and extensions. It shows that our approach is feasible once XML document recognition data is available.

[1]The URL to the xmillum project is `http://xmillum.sourceforge.net`

Bibliography

[1] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An Appliance-Independent XML User Interface Language. In *Proceedings of the Eighth International World-Wide Web Conference*, 1999.

[2] Oronzo Altamura, Floriana Esposito, and Donato Malerba. Transforming Paper Documents into XML Format with WISDOM++. *International Journal on Document Analysis and Recognition*, 4(1):2–17, 2001.

[3] European Computer Machinery Association. *ECMAScript Language Specification*. European Computer Machinery Association, June 1997.

[4] Antoine Azokly. *Une approche générique pour la reconnaissance de la structure physique de documents composites*. PhD thesis no. 1076, University of Fribourg, 1995.

[5] H. S. Baird. Feature Identification for Hybrid Structural/Statistical Pattern Classification. In *Proceedings, CVPR '86 (IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Miami Beach, FL, June 22–26, 1986)*, IEEE Publ.86CH2290-5, pages 150–155. IEEE, 1986.

[6] Henry S. Baird. The Skew Angle of Printed Documents. In *Proceedings of the Conference of the Society of Photographic Scientists and Engineers*, Rochester, New York, May 1987.

[7] Frédéric Bapst. *Reconnaissance de documents assisté: architecture logicielle et intégration de savoir-faire*. PhD thesis, University of Fribourg, 1998.

[8] T. Berners-Lee, R. Fielding, U. C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI): Generic syntax. Technical report, RFC 2396, August 1998. `http://www.ietf.org/rfc/rfc2396.txt`.

[9] L. Bottou, P. Haffner, P. G. Howard, P. Simard, Y. Bengio, and Y. LeCun. High Quality Document Image Compression with DjVu. *Journal of Electronic Imaging*, 1998.

[10] David Brownell. *SAX2*. O'Reilly & Associates, Inc., Cambridge, MA, 2002.

[11] Rolf Brugger. *Eine statistische Methode zur Erkennung von Dokumentstrukturen*. PhD thesis, University of Fribourg, 1995.

[12] R. Cattoni, T. Coianiz, S. Messelodi, and C. M. Modena. Geometric Layout Analysis Techniques for Document Image Understanding: A Review. ITC-IRST 9703-09, 1998.

[13] Eric Clavier, Gerald Masini, Maurizio Rigamonti, Karl Tombre, and Joel Gardes. DocMining: A Cooperative Platform for Heterogeneous Document Interpretation According to User-Defined Scenarios. In *GREC'03: IAPR Workshop on Graphics Recognition*, pages 21–32, Barcelona, Spain, July 2003.

[14] T. M. Cover and P. E. Hart. Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

[15] Danny Coward and Yutaka Yoshida. *Java Servlet Specification Version 2.4*. Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, USA, November 2003.

[16] Chris Cracknell and Andy C. Downton. A Handwriting Understanding Environment (HUE) for Rapid Prototyping in Handwriting and Document Analysis Research. In *ICDAR'99: Fifth International Conference on Document Analysis and Recognition*, pages 362–365, Bangalore, India, September 1999.

[17] Andreas Dengel, Rainer Bleisinger, Rainer Hoch, Frank Hönes, Michael Malburg, and Frank Fein. Officemaid – a system for office mail analysis, interpretation and delivery. In A. Lawrence Spitz and Andreas Dengel, editors, *International Association for Pattern Recognition Workshop on Document Analysis Systems*, volume 14 of *Series in machine perception and artificial intelligence*, P. O. Box 128, Farrer Road, Singapore 9128, 1995. World Scientific Publishing Co. Pte. Ltd.

[18] Andreas Dengel, Rainer Hoch, Frank Hönes, Thorsten Jäger, Michael Malburg, and Achim Weigel. Techniques for improving ocr results. In H. Bunke and P. S. Wang, editors, *Handbook of Character Recognition and Document Image Analysis*, chapter 8, pages 227–258. World Scientific, 1997.

[19] Andreas R. Dengel and Bertin Klein. smartFIX: A Requirements-Driven System for Document Analysis and Understanding. In *DAS'2002: 5th International Workshop on Document Analysis Systems*, number 2423 in Lecture Notes in Computer Science, pages 433–444. Springer, August 2002.

[20] P. A. Devijver and J. Kittler. *Pattern Recognition: A Statistical Approach*. Prentice Hall, 1982. DEVIJVER82.

[21] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1, January 1999. Status: PROPOSED STANDARD.

[22] Dov Dori, David Doermann, Christian Shin, Robert Haralick, Ihsin Phillips, Mitchell Buchman, and David Ross. The Representation of Document Structure: A Generic Object-Process Analysis. In H. Bunke and P. S. Wang, editors, *Handbook of Character Recognition and Document Image Analysis*, chapter 16, pages 421–456. World Scientific, 1997.

[23] Philippe Dosch, Karl Tombre, Christian Ah-Soon, and Gérald Masini. A Complete System for the Analysis of Architectural Drawings. *International Journal on Document Analysis and Recognition*, 3(2):102–116, 2000.

[24] Richard Duda and Peter Hart. *Pattern Recognition and Scene Analysis*. John Wiley and Sons, 1973.

[25] International Organization for Standardization. *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland, August 1986.

[26] International Organization for Standardization. *ISO 8613:1994: Information processing - Text and office systems - Office Document Architecture (ODA) and interchange format*. International Organization for Standardization, Geneva, Switzerland, 1994.

[27] International Organization for Standardization. *ISO/IEC 10179:1996: Information technology — Processing languages — Document Style Semantics and Specification Language (DSSSL)*. International Organization for Standardization, Geneva, Switzerland, 1996.

[28] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol — version 3.0, March 1996. Internet draft draft-freier-ssl-version3-01.txt.

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[30] C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. CiteSeer: An Automatic Citation Indexing System. In *DL'98: Proceedings of the 3rd ACM International Conference on Digital Libraries*, pages 89–98, 1998.

[31] Karim Hadjar and Rolf Ingold. Arabic Newspaper Page Segmentation. In *ICDAR'03: Seventh International Conference on Document Analysis and Recognition*, Edinburgh, Scotland, August 2003.

[32] Karim Hadjar and Rolf Ingold. Physical Layout Analysis of Complex Structured Arabic Documents Using Artificial Neural Nets. In *DAS'2004: International Workshop on Document Analysis Systems*, pages 170–178, Florence, Italy, September 2004.

[33] P. V. C. Hough. Methods and Means to Recognize Complex Patterns. U.S. Patent 3,069,654, 1962.

[34] T. Hu and R. Ingold. A Mixed Approach toward an Efficient Logical Structure Recognition from Document Images. In Christoph Hüser, Wiebke Möhr, and Vincent Quint, editors, *Proceedings of the Fifth International Conference on Electronic Publishing, Document Manipulation and Typography, 13–15 April, 1994, Darmstadt, Germany*, volume 6(4) of *Electronic Publishing—Origination, Dissemination, and Design*, pages 457–468, New York, NY, USA; London, UK; Sydney, Australia, 1993. John Wiley and Sons.

[35] Tao Hu. *New Methods for Robust and Efficient Recognition of the Logical Structures in Documents*. PhD thesis, University of Fribourg, 1994.

[36] Matthew Hurst. Layout and Language: Challenges for Table Understanding on the Web. In Apostolos Antonacopoulos and Jianying Hu, editors, *Proceedings on the First International Workshop on Web Document Analysis*. Seattle, USA, 2001.

[37] Adobe Systems Inc. *PDF Reference*. Addison-Wesley, second edition, 2000.

[38] Adobe Systems Incorporated. *PostScript Language Reference*. Addison-Wesley, Reading, MA, USA, third edition, 1999.

[39] Adobe Systems Incorporated. *PDF Reference: Adobe Portable Document Format, version 1.4*. Addison-Wesley, Reading, MA, USA, third edition, 2001.

[40] Rolf Ingold. *Structures de documents et lecture optique: une nouvelle approche*. Presses Polytechniques Romandes, 1990.

[41] Min-Chul Jung, Yong-Chul Shin, and Sargur N. Srihari. Multifont Classification using Typographical Attributes. In *ICDAR'99: Fifth International*

Conference on Document Analysis and Recognition, pages 353–356, Bangalore, India, September 1999.

[42] Tapas Kanungo. *Document Degradation Models and a Methodology for Degradation Model Validation,*. PhD thesis, University of Washington, 1996.

[43] Tapas Kanungo, Chang H. Lee, Jeff Czorapinski, and Ivan Bella. TRUEVIZ: a Groundtruth/Metadata Editing and Visualizing Toolkit for OCR. In *Proceedings of SPIE*, San Jose, USA, January 2001.

[44] Stewart Kelland and Slawo Wesolkowski. A Comparison of Research and Production Architectures for Check Reading Systems. In *ICDAR'99: Fifth International Conference on Document Analysis and Recognition*, pages 99–102, Bangalore, India, September 1999.

[45] Donald Ervin Knuth. *The TEXbook*, volume A. Addison-Wesley, Reading, MA, USA, 1986.

[46] Gary E. Kopec and Philip A. Chou. Document Image Decoding Using Markov Source Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):602–617, June 1994.

[47] Louisa Lam, Yea-Shuan Huang, and Ching Y. Suen. Combination of multiple classifier decisions for optical character recognition. In H. Bunke and P. S. Wang, editors, *Handbook of Character Recognition and Document Image Analysis*, chapter 16, pages 79–101. World Scientific, 1997.

[48] Leslie Lamport. *LATEX: A Document Preparation System*. Addison-Wesley, Reading, MA, 2nd edition, 1994.

[49] D. X. Le, G. R. Thoma, and H. Wechsler. Classification of Binary Document Images into Textual or Nontextual Data Blocks Using Neural Network Models. *Machine Vision and Applications*, 8(5):289–304, 1995.

[50] Philippe Lefèvre, C. Felter, and P. Lobbrecht. Reconnaissance de documents: Passage du document papier à l'information électronique. *EPURE*, 58:15–25, April 1998.

[51] Philippe Lefèvre and François Reynaud. ODIL: An SGML Description Language of the Layout of Documents. In *ICDAR'95: Third International Conference on Document Analysis and Recognition*, pages 480–488, Montreal, Canada, August 1995.

[52] Heiko Maus. Workflow Context as a Means for Intelligent Information Support. In Varol Akman, Paolo Bouquet, Richmond Thomason,

and Roger A. Young, editors, *Modeling and Using Context*, pages 261–274. Springer-Verlag, Berlin, Germany, 2001.

[53] Betrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[54] S. Mori, C. Y. Suen, and K. Yamamoto. Historical Review of OCR Research and Development. *Proceedings of the IEEE*, 80:1029–1058, 1992.

[55] George Nagy. At the Frontiers of OCR. *Proceedings of the IEEE*, 80(7):1093–1100, July 1992.

[56] George Nagy, Sharad Seth, and Mahesh Viswanathan. A Prototype Document Image Analysis System for Technical Journals. *Computer*, 25(7):10–22, July 1992.

[57] George Nagy and Yihong Xu. Priming the Recognizer. In *DAS'96*, pages 263–281, Malvern, Pennsylvania, October 1996.

[58] Lawrence O'Gorman. The Document Spectrum for Page Layout Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1162–1173, 1993.

[59] W. Postl. Detection of linear oblique structures and skew scan in digitized documents. In *Proceedings, Eighth International Conference on Pattern Recognition (Paris, France, October 27–31, 1986)*, IEEE Publ. 86CH2342-4, pages 687–689, 1986.

[60] RAF Technology, Inc. *DAFS Library, Programmer's Guide and Reference*, August 1995.

[61] Lyse Robadey. *Une méthode de reconnaissance structurelle de documents complexes basée sur des patterns bidimensionnels*. PhD thesis, University of Fribourg, 2001.

[62] Lyse Robadey, Oliver Hitz, and Rolf Ingold. Segmentation de documents ideaux à structure complexe. In *CIFED'2000: Colloque International Francophone sur l'Ecrit et le Document*, pages 383–392, Lyon, France, jul 2000.

[63] Nicolas Roussel, Oliver Hitz, and Rolf Ingold. Web-based Cooperative Document Understanding. In *ICDAR'01: Sixth International Conference on Document Analysis and Recognition*, pages 368–373, Seattle, WA, September 2001.

[64] P. K. Sahoo, S. Soltani, A. K. C. Wong, and Y. C. Chen. A Survey of Thresholding Techniques. *Computer Vision, Graphics, and Image Processing*, 41(2):233–260, February 1988.

[65] Youssouf Saidali. *Modélisation et Acquisition de Connaissances: Application à une Plate-Forme de Traitement d'Images*. PhD thesis, Universitè de Rouen, France, 2002.

[66] R. Sivaramakrishnan, I. T. Philipps, J. Ha, S. Subramanium, and R. M. Haralick. Zone Classification in a Document Using the Method of Feature Vector Generation. In *ICDAR'95: Third International Conference on Document Analysis and Recognition*, pages 541–544, Montreal, Canada, August 1995.

[67] A. Lawrence Spitz. Style-directed document recognition. In *DLIA'99: Document Layout Interpretation and its Applications*, Bangalore, India, September 1999.

[68] Sargur N. Srihari. Recognition of Handwritten and Machine-Printed Text for Postal Address Interpretation. *Pattern Recognition Letters*, 14(4):291–302, 1993.

[69] Sargur N. Srihari and Venu Govindaraju. Analysis of Textual Images Using the Hough Transform. *Machine Vision and Applications*, 2(3):141–153, 1989.

[70] Sargur N. Srihari and Edward J. Kuebert. Integration of Hand-Written Address Interpretation Technology into the United States Postal Service Remote Computer Reader System. In *ICDAR'97: Fourth International Conference on Document Analysis and Recognition*, pages 892–896, Ulm, Germany, August 1997.

[71] World Wide Web Consortium (W3C). Cascading Style Sheets, level 2: CSS2 Specification. `http://www.w3.org/TR/REC-CSS2/`, 1998.

[72] World Wide Web Consortium (W3C). Document Object Model (DOM) Level 1 Specification. `http://www.w3.org/TR/REC-DOM-Level-1/`, 1998.

[73] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/REC-xml`, 1998.

[74] World Wide Web Consortium (W3C). HTML 4.01 Specification. `http://www.w3.org/TR/html4/`, 1999.

[75] World Wide Web Consortium (W3C). Namespaces in XML. `http://www.w3.org/TR/REC-xml-names/`, 1999.

[76] World Wide Web Consortium (W3C). XML Path Language (XPath) 1.0. `http://www.w3.org/TR/xpath`, 1999.

[77] World Wide Web Consortium (W3C). XSL Transformations (XSLT) 1.0. `http://www.w3.org/TR/xslt`, 1999.

[78] World Wide Web Consortium (W3C). Extensible Stylesheet Language (XSL) 1.0. `http://www.w3.org/TR/xsl`, 2001.

[79] World Wide Web Consortium (W3C). Scalable Vector Graphics (SVG) 1.0. `http://www.w3.org/TR/SVG`, 2001.

[80] World Wide Web Consortium (W3C). XSL Schema Part 0: Primer. `http://www.w3.org/TR/xmlschema-0`, 2001.

[81] World Wide Web Consortium (W3C). XSL Schema Part 1: Structures. `http://www.w3.org/TR/xmlschema-1`, 2001.

[82] World Wide Web Consortium (W3C). XSL Schema Part 2: Datatypes. `http://www.w3.org/TR/xmlschema-2`, 2001.

[83] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, 1999. Includes CD-ROM.

[84] K. Y. Wong, R. G. Casey, and F. M. Wahl. Document Analysis System. *IBM Journal of Research and Development*, 26(6):647–656, 1982.

[85] Abdelwahab Zramdini. *Study of Optical Font Recognition Based on Global Typographical Features.* PhD thesis, University of Fribourg, 1995.

Appendix A

Visualizing Segmentation Data

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

<xsl:stylesheet version="1.0"
 xmlns:xmi="http://xmillum.sourceforge.net"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="document">
  <xmi:document>

   <!-- Styles -->

   <xmi:style name="thread-style">
    <param name="foreground" value="yellow"/>
    <param name="transparency" value="0.4"/>
    <param name="fill" value="true"/>
   </xmi:style>

   <xmi:style name="frame-style">
    <param name="foreground" value="blue"/>
    <param name="transparency" value="0.4"/>
   </xmi:style>

   <xmi:style name="image-style">
    <param name="foreground" value="red"/>
    <param name="transparency" value="0.4"/>
```

```
     <param name="fill" value="true"/>
    </xmi:style>

    <xmi:style name="text-style">
     <param name="foreground" value="green"/>
     <param name="transparency" value="0.4"/>
     <param name="fill" value="true"/>
    </xmi:style>

    <xmi:style name="line-style">
     <param name="foreground" value="grey"/>
     <param name="transparency" value="0.4"/>
     <param name="fill" value="true"/>
    </xmi:style>

    <!-- Objects -->

    <xmi:object name="image" class="iiuf.xmillum.displayable.Image"/>

    <xmi:object name="thread-block" class="iiuf.xmillum.displayable.Block">
     <param name="style" value="thread-style"/>
    </xmi:object>

    <xmi:object name="frame-block" class="iiuf.xmillum.displayable.Block">
     <param name="style" value="frame-style"/>
    </xmi:object>

    <xmi:object name="image-block" class="iiuf.xmillum.displayable.Block">
     <param name="style" value="image-style"/>
    </xmi:object>

    <xmi:object name="text-block" class="iiuf.xmillum.displayable.Block">
     <param name="style" value="text-style"/>
    </xmi:object>

    <xmi:object name="line-block" class="iiuf.xmillum.displayable.Block">
     <param name="style" value="line-style"/>
    </xmi:object>

    <!-- Finally, the layers -->

    <xmi:layer name="Document Image">
     <image src="{@image}"/>
    </xmi:layer>

    <xmi:layer name="Threads">
     <xsl:for-each select=".//thread">
      <thread-block x="{@x}" y="{@y}" w="{@w}" h="{@h}"/>
     </xsl:for-each>
    </xmi:layer>

    <xmi:layer name="Frames">
     <xsl:for-each select=".//frame">
      <frame-block x="{@x}" y="{@y}" w="{@w}" h="{@h}"/>
     </xsl:for-each>
    </xmi:layer>

    <xmi:layer name="Images">
     <xsl:for-each select=".//image">
      <image-block x="{@x}" y="{@y}" w="{@w}" h="{@h}"/>
     </xsl:for-each>
    </xmi:layer>

    <xmi:layer name="Text Blocks">
     <xsl:for-each select=".//block">
      <text-block x="{@x}" y="{@y}" w="{@w}" h="{@h}"/>
     </xsl:for-each>
    </xmi:layer>

    <xmi:layer name="Text Lines">
     <xsl:for-each select=".//line">
      <line-block x="{@x}" y="{@y}" w="{@w}" h="{@h}"/>
     </xsl:for-each>
    </xmi:layer>

   </xmi:document>
  </xsl:template>
</xsl:stylesheet>
```

Appendix B

Correcting Undersegmentation

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

<xsl:stylesheet version="1.0"
 xmlns:tmp="tmp"
 xmlns:xmi="http://xmillum.sourceforge.net"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="document">
  <xmi:document>

   <!-- Styles -->

   <xmi:style name="thread-style">
    <param name="foreground" value="yellow"/>
    <param name="transparency" value="0.4"/>
    <param name="fill" value="true"/>
   </xmi:style>

   <xmi:style name="frame-style">
    <param name="foreground" value="blue"/>
    <param name="transparency" value="0.4"/>
   </xmi:style>

   <xmi:style name="image-style">
    <param name="foreground" value="red"/>
   <param name="transparency" value="0.4"/>
   <param name="fill" value="true"/>
  </xmi:style>

  <xmi:style name="text-style">
   <param name="foreground" value="green"/>
   <param name="transparency" value="0.4"/>
   <param name="fill" value="true"/>
  </xmi:style>

  <xmi:style name="line-style">
   <param name="foreground" value="grey"/>
   <param name="transparency" value="0.4"/>
   <param name="fill" value="true"/>
  </xmi:style>

  <!-- Handlers -->

  <xmi:handler name="split" class="iiuf.xmillum.handlers.Split"/>

  <!-- Objects -->

  <xmi:object name="image" class="iiuf.xmillum.displayable.Image"/>

  <xmi:object name="thread-block" class="iiuf.xmillum.displayable.Block">
   <param name="style" value="thread-style"/>

   <param name="click1" value="split" opt="split"/>
   <param name="click3" value="split" opt="turn"/>
   <param name="over" value="split" opt="show"/>
  </xmi:object>

  <xmi:object name="frame-block" class="iiuf.xmillum.displayable.Block">
   <param name="style" value="frame-style"/>

   <param name="click1" value="split" opt="split"/>
   <param name="click3" value="split" opt="turn"/>
   <param name="over" value="split" opt="show"/>
  </xmi:object>

  <xmi:object name="image-block" class="iiuf.xmillum.displayable.Block">
   <param name="style" value="image-style"/>

   <param name="click1" value="split" opt="split"/>
   <param name="click3" value="split" opt="turn"/>
  <param name="over" value="split" opt="show"/>
 </xmi:object>

 <xmi:object name="text-block" class="iiuf.xmillum.displayable.Block">
  <param name="style" value="text-style"/>

  <param name="click1" value="split" opt="split"/>
  <param name="click3" value="split" opt="turn"/>
  <param name="over" value="split" opt="show"/>
 </xmi:object>

 <xmi:object name="line-block" class="iiuf.xmillum.displayable.Block">
  <param name="style" value="line-style"/>

  <param name="click1" value="split" opt="split"/>
  <param name="click3" value="split" opt="turn"/>
  <param name="over" value="split" opt="show"/>
 </xmi:object>

 <!-- Finally, the layers -->

 <xmi:layer name="Document Image">
  <image src="{@image}"/>
 </xmi:layer>

 <xmi:layer name="Threads">
  <xsl:for-each select=".//thread">
   <thread-block x="{@x}" y="{@y}" w="{@w}" h="{@h}" ref="{@tmp:refvalue}"/>
  </xsl:for-each>
 </xmi:layer>

 <xmi:layer name="Frames">
  <xsl:for-each select=".//frame">
   <frame-block x="{@x}" y="{@y}" w="{@w}" h="{@h}" ref="{@tmp:refvalue}"/>
  </xsl:for-each>
 </xmi:layer>

 <xmi:layer name="Images">
  <xsl:for-each select=".//image">
   <image-block x="{@x}" y="{@y}" w="{@w}" h="{@h}" ref="{@tmp:refvalue}"/>
  </xsl:for-each>
 </xmi:layer>

 <xmi:layer name="Text Blocks">
  <xsl:for-each select=".//block">
```

```
      <text-block x="{@x}" y="{@y}" w="{@w}" h="{@h}" ref="{@tmp:refvalue}"/>
     </xsl:for-each>
    </xmi:layer>

    <xmi:layer name="Text Lines">
     <xsl:for-each select=".//line">
      <line-block x="{@x}" y="{@y}" w="{@w}" h="{@h}" ref="{@tmp:refvalue}"/>
     </xsl:for-each>
    </xmi:layer>

   </xmi:document>
  </xsl:template>
</xsl:stylesheet>
```

Curriculum Vitae

Personal Information

- **Full name:** Oliver HITZ
- **Date of birth:** August 12th, 1974
- **Nationality:** Swiss
- **Marital status:** Single
- **Languages:** Swiss german (native tongue)
  German (written and spoken)
  French (written and spoken)
  English (written and spoken)

Education

- **1998-2005** University of Fribourg, PhD student in computer science
- **1994-1998** University of Fribourg, studies of computer science and economics
- **1990-1994** Collège Ste. Croix, Fribourg, scientific maturity

Work Experience

- **2001-now** Founded the company net-track GmbH.
- **1998-2002** Teaching assistant at the University of Fribourg for the following classes: image processing and pattern recognition, theory of programming languages, computer architecture and C programming.
- **1997-now** Systems and network manager at senseLAN GmbH Internet Services Provider.
- **1997-1999** Systems manager at the University of Fribourg.