

Master's Thesis

A canonical PDF Text Extraction Algorithm

Reconstruction of Electronic Signatures from Printouts

submitted by

Gregor Kofler, December 2009

Advisor: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhard Posch

Supervisor: Dipl.-Ing. Dr. techn. Thomas Rössler

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTÄTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am

.....
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

.....
date

.....
(signature)

Danksagung

Zu Beginn möchte ich mich bei all denjenigen bedanken, die mich während meines Studiums und in der Ausarbeitung der vorliegenden Masterarbeit unterstützt haben.

Besonderer Dank gilt:

Meinen Eltern, die mir dieses Studium ermöglicht haben und mir immer mit Rat und Tat zur Seite standen.

Meiner Freundin, Stefanie die diese schwierige Zeit mit mir durchgestanden hat und immer wieder ein offenes Ohr für meine Ideen hatte.

Meiner Oma und "Opa" Fritz, die durch Ihre Unterstützung mein Studium wesentlich erleichtert haben.

Meinen Betreuern Dr. tech. Dipl.-Ing. Thomas Rössler und Dipl.-Ing. Thomas Knall, die mich durch ihre konstruktive Kritik und hilfreichen Tipps sehr unterstützt haben. Vor allem Dr. tech Dipl.-Ing. Thomas Rössler der mir mit viel Engagement und Einsatz zur Seite stand.

Herrn Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch, der es mir ermöglicht hat auf dem IAIK diese Masterarbeit durchzuführen.

Abstract

This master's thesis deals with the reconstruction of Official Signatures from printouts. Official Signatures are advanced electronic signatures which play an important role in the field of eGovernment and bridge the gap between paper and electronic based documents. This thesis depicts and analyses the problems of Adobe's Portable Document Format (PDF) in view of text extraction as well as in view of reconstruction of Official Signatures from printouts.

The term eGovernment has become a synonymous with a modern, qualitative and efficient public administration by utilising new information and communication technology. In eGovernment, trust plays a central role in terms of authenticity and data integrity. This is enabled by electronic signatures. With the introduction of the well known "Citizen Card", every citizen has obtained a qualified electronic signature that is the legal equivalence to a personal handwritten signature in the digital world. Public authorities also have an electronic signature, called an Official Signature. Any document or notification from a public authority is affixed by an Official Signature which facilitates a secure communication to citizens by means of integrity and authenticity. In eGovernment, PDF has been well-established for interchanging document's because of its platform independence and preservation of the visual appearance of the original document. In order to support the reconstruction of electronic signatures based on printed PDF documents, only the document's text content is signed. Therefore a text extraction is needed. The issue is that PDF is well suited for illustrating documents but not for processing its text content. This results in the fact that PDF documents are not necessarily self-contained and self-descriptive. It utilises external objects such as system fonts which makes a text extraction on other systems infeasible.

This thesis proposes TEA a text extraction algorithm that is feasible, reliable and applicable in the field of eGovernment and fulfils the requirements of Official Signatures.

Kurzfassung

Die vorliegende Masterarbeit behandelt die Problematik der Rekonstruktion von elektronischen Signaturen, im Speziellen von Amtssignaturen, anhand papier-basierter Dokumente im Kontext von eGovernment. Amtssignaturen spielen eine wichtige Rolle in eGovernment und schlagen eine Brücke zwischen elektronischen und papier-basierten Dokumenten. Im Rahmen dieser Masterarbeit wird das weitverbreitete Portable Document Format (PDF) hinsichtlich des Einsatzes im eGovernment Bereich analysiert und die wesentlichen Schwächen im Bezug auf die Rekonstruktion von Amtssignaturen anhand des Dokumentenausdruckes aufgezeigt.

Der Begriff eGovernment steht für eine moderne, qualitative und effiziente behördliche Verwaltung unter Verwendung aktueller Informations- und Kommunikationstechnologien. Datensicherheit spielt dabei eine zentrale Rolle, vor allem in Hinsicht auf Integrität und Authentifizierung - Stichwort "elektronische Signaturen". Mit der Einführung der Bürgerkarte, hat jeder Bürger in Österreich die Möglichkeit Dokumente oder Formulare auch elektronisch zu unterschreiben, unter der gleichbedeutenden juristischen Gültigkeit einer handgeschriebener Unterschrift. Behörden besitzen, wie Bürger, auch eine elektronische Signatur, eine sogenannte Amtssignatur. Sie wird auf Bescheide und andere Erledigungen seitens einer Behörde aufgebracht, um damit kenntlich zu machen, dass es sich um ein amtliches Schriftstück handelt. Eine Amtssignatur gewährleistet einerseits, dass ein Dokument von einer Behörde stammt und andererseits, dass der Inhalt des Dokuments während der Übermittlung nicht verändert wurde. Im Bereich eGovernment, spielt im elektronischen Verkehr mit dem BürgerInnen das Portable Document Format (PDF) eine bedeutende Rolle. Das Format zeichnet sich vor allem durch seine Plattformunabhängigkeit und seine farb- und strukturtreue Darstellung aus. Bei sogenannten text-basierten Amtssignaturen wird für eine spätere Verifikation von behördlichen Dokumenten nur der Textinhalt der PDF Datei signiert, um die Prüfbarkeit der Signatur auch auf Basis des Ausdruckes gewährleisten zu können. Die Herausforderung dabei besteht in der Text-Extraktion, da PDF für die Betrachtung und nicht für die weitere Textverarbeitung entwickelt wurde.

Im Rahmen dieser Arbeit werden die Probleme der Text-Extraktion von PDF Dokumenten aufgezeigt, sowie ein zuverlässiger und praktikabler Ansatz zur Text-Extraktion für den Einsatz im Bereich eGovernment präsentiert.

Contents

1	Introduction	3
1.1	Motivation	3
1.1.1	Legal Requirements	4
1.1.2	Visual Representation of Electronic Signatures in Austria	6
1.1.3	Reconstruction of Electronic Signatures from Printouts	7
1.2	Objectives of this Thesis	10
1.3	Structure of this Thesis	10
2	Portable Document Format	11
2.1	History and Motivation	11
2.2	Components	13
2.2.1	File Structure	13
2.2.2	Objects and Indirect Objects	17
2.2.3	Document Structure	22
2.3	Graphic Model	25
2.3.1	Content Streams	26
2.3.2	Marked Content Streams	27
2.3.3	Graphic Objects	27
2.3.4	Coordinate Spaces	29
2.4	Text	30
2.4.1	Characters and Glyphs	31
2.4.2	Fonts	32
2.4.3	Text Objects and Operators	37
2.4.4	Text Positioning	37
3	Specification	42
3.1	The problem to be dealt with	43
3.2	PDF/A-1 (ISO 19005-1)	44
3.3	Tagged PDF	46

3.4	Algorithm Requirements	50
3.5	Algorithm Specification	51
3.5.1	Data Model	51
3.5.2	Hidden Text Recognition (HTR)	53
3.5.3	Text Extraction (TE)	60
4	Implementation	64
4.1	System Overview of PDF-AS	64
4.2	Architecture overview	66
4.2.1	Pipes and Filters	66
4.2.2	System Architecture	69
4.3	Technical Overview	70
4.3.1	Model-Package	71
4.3.2	Pipes-Package	72
4.3.3	Filters-Package	74
4.4	Testing	76
4.4.1	Testing Procedure	76
4.4.2	Test results	83
5	Discussion	85
5.1	Future work	87
A	Appendix	88
A.1	Acronyms	88
A.2	Glossary	89

Chapter 1

Introduction

1.1 Motivation

New technologies enrich and make our life easier day for day. The progress of information and communication technologies (ICT) have especially changed our society and modern life is unthinkable without these technologies. This changeover does not only affect the private and business sector, but also the public sector. The appliance of new information and communication technologies in the public domain can be summarized under the term eGovernment. The term eGovernment, also called E-Gov, digital Government or on-line Government has become a synonym for a modern, qualitative and efficient public administration. Applications of eGovernment are diversified and geared to improve the quality and transparency of information, communication and transaction between public authorities and citizens or businesses.

In the European Union, Austria is one of the leading countries in the field of eGovernment. The major topics of eGovernment are authentication and integrity by means of communication over untrusted channels such as the internet. Especially documents from public authorities require particular properties for showing citizens that a document originates from an official authority. Moreover, in eGovernment the most application forms require the signature of the person filling it out, which until now had to be signed on the paper itself. With the technological change, authenticating by signature has to be carried out electronically.

Public authorities have an electronic signature too, a so called the Official Signature, which is used to sign digital documents. The Official Signature is an advanced "electronic signature" attached by an authority to an administrative notice or document. Thereby, not only the authenticity and data integrity of the document can be verified by means

of the Official Signature but the printout of the signed document is also treated as being equivalent to the official document by the authorities. In other words, Official Signatures build a bridge between electronic documents and printouts in eGovernment.

At the beginning of the work on this thesis there was the problem of reconstructing electronic signatures and especially of Official Signatures from printouts. In eGovernment, many legal act needs an electronic signature to come into force. This means most of the documents or notifications issued by a public authority are affixed by an Official Signature. The resulting signature depends on the document's text content (i.e. the content to be signed) and a secret key only known by the issuer. The receiver of the document (e.g. a citizen) is able to verify the signature by using the public key of the issuer and applying mathematical operations on the document's text content. Only if the verification is successful the document's text content can be considered being authentic and unmodified.

In eGovernment, the Portable Document Format (PDF) is well-established because of the platform independence, the free access of PDF readers for nearly every operating system and because of the preservation of the visual appearance of the original document from which the PDF document is created. Furthermore almost every application supports the export into PDF. Thus, PDF has become the first choice for interchanging documents not only in eBusiness but also in eGovernment.

The legal situation in Austria requires, that citizens or public authorities should be able to reconstruct an Official Signature from printout. Beside the cryptographic verification of the signature, this implies also the reconstruction of the document's text content either by typing or scanning the printout's text. Of course, the extracted text should be equal with the one, that has been signed. This thesis points out in a later chapter, that text extraction of PDF documents is a great challenge, especially considering a later reconstruction. In the course of the investigations on this topic, a canonical text extraction algorithm for a subset of PDF documents is proposed which provides a standardized and unique result regarding the documents reading direction.

1.1.1 Legal Requirements

Austria was one of the first European states that introduced a series of laws to facilitate eGovernment services. The major laws are the E-Government Act (E-GovG), Signature Act (SigG) and Signature Regulation (SigV). They define the terms electronic signature, identity and authenticity. Furthermore, the legal regulation lays down the basic requirements and conditions for the technical eGovernment infrastructure in Austria, such as for

the well known Austrian Citizen Card as well as for Official Signatures. Official Signatures are one major motivation of this master's thesis. Therefore it is worth mentioning the definition of Official Signatures by the Austrian E-Government Act which is as follows:

Official Signature §19 E-GovG¹

[...]

(1) An official signature, being the electronic signature of a public authority, is an advanced electronic signature within the meaning of the Signaturgesetz (Signature Act), the peculiarity of which is indicated by an appropriate attribute in the signature certificate.

(2) An official signature serves to facilitate recognition of the fact that a document originates from a controller in the public sector. It may therefore only be used by this controller in accordance with the detailed conditions laid down in sub paragraph 3, when signing electronically or drawing up the documents issued by them.

(3) The official signature in views of electronic documents shall be displayed by means of an image which the public sector controller has published on the Internet in secure form as its own and a reference within the document confirming that it has been officially signed. The information needed for the validation of the electronic signature has to be provided by the controller of the public sector.

[...]

As recently as the definition of Official Signatures is the definition of the probative value of printouts, which supports the verification of the printout of the signed document.

Probative Value of Printouts §20 E-GovG¹

[...]

An electronic document of an authority printed out on to paper is assumed to be authentic (Paragraph 292 ZPO, RGBl. No. 113/1895) if signed with an official signature. The official signature has to allow verification by reconvertng the printout of the document into its electronic form or the document must be verifiable by other means provided by the authority. The document shall

¹"Federal Act on Provisions Facilitating Electronic Communications with Public Bodies", Austrian Federal Law Gazette BGBl. I Nr. 10/2004, amended by BGBl I Nr. 7/2008

include a reference to the source on the Internet, containing the procedure for reconvertng the printout into the electronic form and the applicable verification mechanisms, or a reference to another verification process.

[...]

The E-GovG differs between the reconstruction and verification of Official Signatures. However, from the technical point of view the two terms are equivalent and will be used synonymously in this thesis.

1.1.2 Visual Representation of Electronic Signatures in Austria

As already mentioned, Official Signatures base on advanced electronic signatures within the meaning of the Signature Act and replace the seal and stamp of public authorities. An Official Signature provides an standardized visual representation to show various signature information. Figure 1.1 illustrates the design of an Official Signature exemplarily. The main element of an electronic signature is the so called signature value. It depends on various cryptographic attributes and on the document's text content by so called text-based signatures. However, beside the signature value, the Official Signature provides


Signature Value	2lqZqgbg3VQQ7Cwdw5ViG+CKte4xK7QzrtndrCVTnLXohY7G1gsjl28t0iwuiMWu	
	Signatory	serialNumber=886647532789,givenName=Gregor,SN=Kofler,CN=Gregor Kofler,C=AT
	Date/Time-UTC	2009-10-29T22:27:17Z
	Issuer-Certificate	CN=a-sign-Premium-Sig-02,OU=a-sign-Premium-Sig-02,O=A-Trust Ges. f. Sicherheitssysteme im elektr. Datenverkehr GmbH,C=AT
	Serial-No.	243588
	Method	urn:pdfsigfilter:bka.gv.at:text:v1.1.0
	Parameter	etsi-moc-1.1@1c84e76a
Verification	Information about the verification of the electronic signature and of the printout can be found at: https://www.signaturpruefung.gv.at	

Figure 1.1: Visual representation of an Official Signature.

various so called signature attributes. The most important attributes are:

Signature value (Signaturwert) represents the value of the electronic signature calculated by a cryptographic function (see Figure 1.2).

Signatory (Unterzeichner) describes the name of the signatory.

Date/Time-UTC (Datum/Zeit-UTC) represents the date and time of signing.

Issuer certificate (Aussteller-Zertifikat) attests the identity of the certificate service provider that has issued the signatory signing certificate.

Serial-nr. (Serien-Nr.) represents the signatory's certificate's number.

Method (Methode) describes the used signature method or algorithm.

Parameter (Parameter) provides additional attributes of the signing method applied.

Note (Prüfhinweis) gives or points to information for verifying the signature.

Image/Blip designates that the signature is an Official Signature.

1.1.3 Reconstruction of Electronic Signatures from Printouts

The process of reconstruction of electronic signatures corresponds to the manual verification of signatures. Therefore, we have to take a closer look how a signature is created and verified in general. Figure 1.2 shows the signing and verification process of a notification, represented by a PDF document. The steps for signing and verifying are summarized briefly as follows (following the PDF-AS technology):

Signing

1. The text content of the underlying notification (PDF document) is extracted by a well-defined text extraction algorithm. The extracted text represents the content to be signed.
2. A hash value of the extracted text is calculated by a defined hash function (e.g. SHA-1).
3. The calculated hash value is signed with the private key of the signatory.
4. The signature value and the signatory's certificate, which affirms the authenticity of the sender and keeps the public key for verification, are attached on the PDF document. However, an Official Signature keeps a signatory certificate's number instead of the signatory's certificate and in addition a issuer certificate of the certificate service provider that has issued the signatory signing certificate. So the signatory's certificate's have to be retrieved by the certificate service provider.

Any step at signing is carried out in background from the PDF-Amtssignatur (PDF-AS) application, which will be discussed in a later chapter in this thesis. The verification of the signature is performed analogue in a slight modified order.

Verification

1. The signature information is separated from the document.
2. The content of the underlying notification (PDF document) is extracted by a well-defined text extraction algorithm. If the notification is only available as printout, the text content of the document is typed out by hand from a citizen.
3. The hash value of the extracted text from step 2 is calculated by a defined hash function (e.g. SHA-1).
4. The public key from the signatory certificate is used to decrypt the hash value calculated by the sender.
5. The calculated hash value is compared with the decrypted hash value, if they are equal, the notification is authentic and unmodified.

For document verification, there are two different methods. On the one hand, if the document has been received electronically as PDF file then the document is verified automatically by the PDF-AS application. On the other hand, if the document is only available as printout, a citizen can verify the document by entering the printout's text by hand. Thus, at the time of signing the method of text extraction from PDF documents is crucial for a later verification of the document by means of reconstructing the signature from printout. Only if the "automatic" text extraction applied by PDF-AS during signing produces the same result as the manual one, that bases on the entered text by hand, the verification of the Official Signature, based on printout, can succeed. Due to the nature of PDF, automatic text extraction regarding a later reconstruction from printout is a challenging task. Moreover, some features of PDF do not satisfy the legal requirements of Official Signatures. So one challenge is to identify and to eliminate those features of PDF, that prevent a proper extraction of the signed text, in order to develop a reliable text extraction algorithm.

In the course of this thesis, we will point out the problems of generic PDF documents regarding text extraction by means of the later reconstruction and how to tackle these problems with the new PDF/A-1a format. Moreover, this thesis introduces TEA an text extraction algorithm for PDF/A documents which aims to satisfy the legal requirements of Official Signatures.

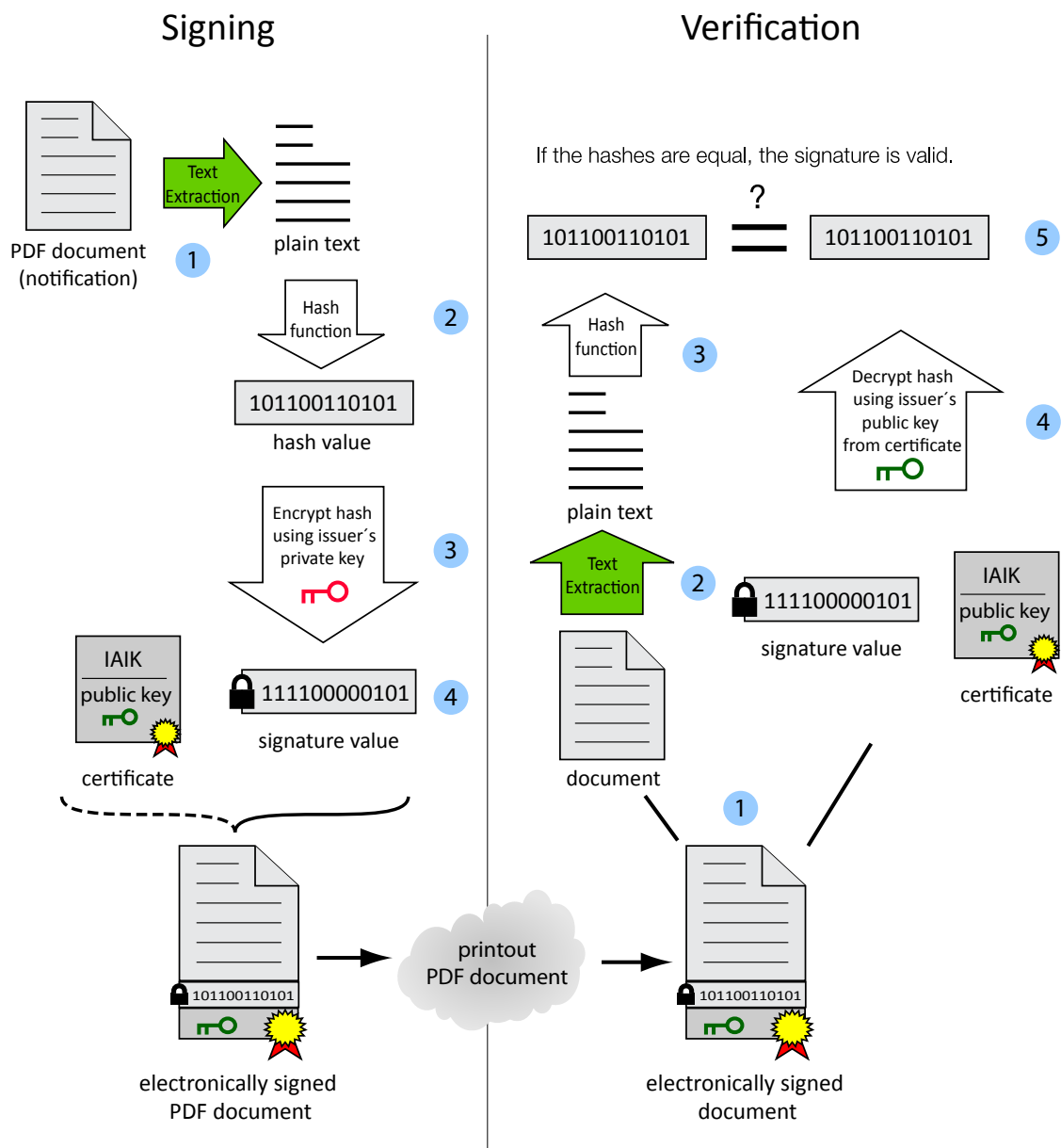


Figure 1.2: Work-flow of signing and verifying documents in eGovernment.

1.2 Objectives of this Thesis

In order to clarify the boundaries and content of this thesis, the objectives of this work are defined as follows:

- The primary focus is on the specification of an canonical text extraction algorithm which follows the reading direction with following key aspects:
 - technically neutral
 - correct handling of footer and header
 - based on PDF/A specification
- The secondary objective focuses on the implementation of the specified algorithm to be used within PDF-AS library, a core library used to create electronic signatures in respect to the needs of Official Signatures as given by the E-GovG.

1.3 Structure of this Thesis

The results initiated by the motivation stated above are summarized in this thesis, which is organized as follows:

Chapter 2 offers a survey regarding the sophisticated Portable Document Format (PDF) version 1.4. It continues with the examination of PDF syntax, structure and how it manages text in general. This chapter is mainly based on the *"PDF Reference: Adobe Portable Document Format Version 1.4"* from Adobe Systems [PPS01] and the German book *"Die Post Script- und PDF-Bibel"* from Thomas Merz [MD01].

Chapter 3 is divided into two parts. The first part deals with the problems concerning the rich-feature nature of PDF in terms of electronic signatures and shows the problems of PDF 1.4 in processing the document's text content. The second part proposes a canonical Text Extraction Algorithm (TEA) for PDF/A-1a documents. It specifies how to extract text from PDF/A-1 documents for signing and verification in eGovernment.

Chapter 4 provides a detailed overview of the implementation of TEA which involves the introduction of PDF-AS, the system architecture and basic design of TEA.

Finally, in **Chapter 5**, the strengths and weaknesses of TEA are outlined and an outlook for future work are given.

Chapter 2

Portable Document Format

When talking about text extraction from PDF documents it is worth taking a closer look at this frequently used document format. In this chapter we will examine Adobe's PDF. However a complete treatment of PDF is behind the scope of this thesis. Hence, we will focus on the components of a PDF File (Section 2.2), graphic model of PDF (Section 2.3) and how text is managed in PDF (Section 2.4). This chapter is mainly based on the "*PDF Reference: Adobe Portable Document Format Version 1.4*" from Adobe Systems [PPS01] and the book "*Die Post Script- und PDF-Bibel*" from Merz Thomas [MD01]. At the end of this chapter we will show the problems of PDF 1.4 by means of processing text and how these problems have been tackled by the PDF/A-1 specification in Section 3.2.

2.1 History and Motivation

In 1993, the Californian company Adobe Systems developed and published a platform independent document exchange format called PDF. Adobe's Portable Document Format emerged from the demand for an software, hardware and operating system independent document exchange format, with the aim of preserving the document's fidelity and appearance.

The Portable Document Format is closely related to Adobe's Post Script. Both formats have been designed as a device independent page description language for creating graphical output. While PostScript focuses on the standardization of document print-outs, PDF is optimized for document exchange by preserving the uniform presentation on screen, regardless of the environment in which documents were created.

Nowadays, there are more than 250 million PDF documents on the World Wide Web and over 500 million copies of the Acrobat Reader around the world [PPS01]. Currently,

the Portable Document Format is used in all areas of the private sector and industry as well in the public domain. The reason for this success is the fact that Adobe’s Reader for reading and printing PDF documents has become freely available since PDF version 1.1. On the other hand the Internet has grown explosively and has become more attractive for document exchange. As a consequence, PDF could distribute rapidly over the Internet and has become the *de facto* standard as interchange format in the World Wide Web.

Over the years, the PostScript and Portable Document Format have been improved several times since they were first published. New versions of the Portable Document Format are published in parallel with the releases of Adobe’s software suite, see Table 2.1.

Year	Event
1985	PostScript Level 1
1991	PostScript Level 2
1993	Portable Document Format 1.0 / Acrobat 1.0
1994	Portable Document Format 1.1 / Acrobat 2.0
1996	Portable Document Format 1.2 / Acrobat 3.0
1997	PostScript Level 3
1999	Portable Document Format 1.3 / Acrobat 4.0
2001	Portable Document Format 1.4 / Acrobat 5.0
2003	Portable Document Format 1.5 / Acrobat 6.0
2005	Portable Document Format 1.6 / Acrobat 7.0
2006	Portable Document Format 1.7 / Acrobat 8.0
2008	Portable Document Format 1.7 Ext. Level 3 / Acrobat 9.0

Table 2.1: Development history of PostScript and PDF.

On July 2008, the Portable Document Format was published as open standard ISO 32000-1:2008 with the title "Document management – Portable document format – Part 1: PDF 1.7". In addition, subsets of PDF have been, or are being, standardized under the International Organization of Standardization (ISO):

PDF/A-1 for long-term preservation of electronic documents in corporate, government and academic environments published as ISO 19005-1:2005; based on PDF 1.4

PDF/X for the exchange and printing of graphics, published as ISO 15930.

PDF/E for the exchange of complex technical documents for engineering and manufacturing as well as for architecture and construction.

PDF/UA for universally accessible PDF file.

The listing above can not be claimed to be complete and only shows a fragment of the available standards. Every standard covers a special domain. The PDF/A-1 standard for long-term preservation of electronic documents. The PDF/A-1 standard does not describe a new document format but it describes a profile of Adobe's PDF 1.4 which makes it more suitable for archiving electronic document over long periods of time. The PDF/A-1 standard is well suited for text extraction. So it will be discussed in Section 3.2, "*PDF/A-1 (ISO 19005-1) and Tagged PDF*".

2.2 Components

As mentioned before, the Portable Document Format has been designed as a device independent page description language. Like every language, it has its own alphabet, syntax and structure. This section outlines the different components of a PDF file containing the *file structure* and *document structure* as well as the provided *objects and indirect objects* of the PDF specification. The *Backus-Naur-Form (BNF)* is used in the following section in order to describe the PDF Syntax.

2.2.1 File Structure

A PDF document is optimized for efficient random access and incremental updates. Therefore, PDF 1.4 [PPS01] specifies four elements which are mandatory for a PDF document:

- A *header* representing the first line of a PDF file and specifying the applied PDF version of the document.
- A *body* containing a sequence of non-ordered *indirect objects* which will be discussed in Section Objects and Indirect Objects. An indirect object contains one or more objects and supports randomized access. An object represents a simple data type, such as a Number, a String, a Name. Furthermore, PDF provides a huge number of predefined complex objects such as pages, fonts and images.
- A *cross-reference table* points to the most important *indirect objects* in the file.
- A *trailer* describes where to find the cross-reference table in the file.

Beside these elements, there are some lexical conventions in a PDF file. A PDF character is either a *regular*, *delimiter* or *white-space character*. The ASCII characters NULL (hex-value:00), TAB (hex-value:09), LINE FEED (hex-value:0A), CARRIAGE RETURN (hex-value:0D) and SPACE (hex-value:20) represent a white-space character. Every line is terminated by a carriage return, line feed or both of them. The delimiter characters (,),

<, >, [,], {, }, / delimit and group characters to object tokens. The special character % introduces a comment statement. An entire syntactical specification of each character type is shown in Listing 2.1 in BNF.

```

<whitespace-char> ::=
    NULL|TAB|LINE_FEED|CARRIAGE_RETURN|SPACE
<delimiter-char> ::=
    '('|')'|'<'>'|'>'|'['|']'|'{'|'}'|'/'
<eol> ::=
    CARRIAGE_RETURN|LINE_FEED|CARRIAGE_RETURN LINE_FEED
<digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<number> ::= <digit>|<digit><number>

```

Listing 2.1: Definition of the most important character types in BNF

The following section describes the most important elements of the file structure such as the cross-reference table and the file trailer. Moreover, PDF's smart technique for updating PDF documents is proposed.

Cross-Reference Table

An essential part of a PDF file is the *cross-reference table*. It enables a randomized access to any indirect object in the body without need to read the entire file. Each cross-reference table is introduced by the keyword **xref** and can be divided into multiple sections. Each section consists at least of a header and an entry. A detailed syntactical specification of the cross-reference table and its elements is defined in Listing 2.2 in BNF.

```

<byte-offset> ::= <digit><digit><digit><digit>
                <digit><digit><digit><digit><digit><digit>
<gen-number> ::= <digit><digit><digit><digit><digit>
<free-or-in-use> ::= '<'>'<n'>
<section-header> ::= <number> <number><eol>
<section-entry> ::=
    <byte-offset> <gen-number> <free-or-in-use><eol>
<section> ::= <section-header><section-entry>
<sections> ::= <section>|<section><sections>
<cross-reference-table> ::= 'xref'<eol><sections>

```

Listing 2.2: PDF Cross-Reference Table specification in BNF

The section-header consists of an indirect object number and the number of indirect objects contained in this section. A section contains at least one section-entry. A section-entry consists of a 10-digit byte offset number, a 5-digit generation number and a marker

indicating that the affected entry is free "f" or in-use "n". If an entry is marked as in-use, the 10-digit number represents the number of bytes from the beginning of the file to the beginning of the indirect object. Otherwise, if an entry is marked as free, the indirect object number is not in-use and the 10-digit number represents the number of the next free object in the entire cross-reference table. All free entries build a circular linked list starting with **obj 0**, which is the first object (entry) in the cross-reference table. The last free entry links back to the first table entry. Beside the offset-number, each section-entry has a generation number which describes how often an object was updated, the number frees and in-uses. As a convention the first table entry (**obj 0**) is marked as free and has the generation number 65535.

To enable a better understanding, Listing 2.3 illustrates a cross reference table with two sections and three section-entries, whereas **obj 1**, which is the second section-entry, is marked as in-use and begins at the byte 123 from the beginning of the PDF file. The **obj 2**, which is the third section-entry, is marked as free, which means the object number is not in-use anymore (e.g. the object was deleted).

```
xref
% section has one entry and first object is obj 0
0 1
% obj 0, first free entry of first section
0000000000 65535 f
% section has two entries and first object is obj 1
1 2
% obj 1, first in-use entry of second section
0000000123 00000 n
% obj 2, second free entry of second section
% it was deleted and points to the next free object
0000000000 00001 f
```

Listing 2.3: Example of a Cross-Reference Table in PDF

File Trailer

The *file trailer* of a PDF File assigns the position of a cross-reference table. More precisely, it holds the number of bytes from the beginning of the PDF file to the beginning of the cross-reference table. It has the advantage that a PDF viewer application does not have to search the complete document for the cross-reference table because the file trailer is always at the end of the PDF File. Listing 2.4 illustrates an example of a trailer in a PDF file.

```

trailer
% begin of trailer dictionary
<<
    % size of the cross-reference table
    \Size 10
    % catalog dictionary
    \Root 2 0 R
% end of trailer dictionary
>>
startxref
% bytes from the beginning of the file to the
% beginning of the cross-reference table
1234
% end of file marker
%%EOF

```

Listing 2.4: Example of a Trailer Entry in PDF

The file trailer is introduced by the keyword **trailer**, a trailer dictionary, the keyword **startxref** and the number of offset bytes from the beginning of the PDF file. A dictionary is a simple data type. An overview of all the data types available in PDF gives the next Section, Objects and Indirect Objects. The trailer dictionary contains additional information about the trailer such as the document *catalog dictionary* or the size of the cross-reference table. The benefit of the catalog dictionary will be explained in detail in the course of this section. Listing 2.5 shows the specification of the file trailer in detail.

```

<trailer> ::= 'trailer'<eol><dictionary><eol>
            'startxref'<eol><number><eol>'%%EOF'

```

Listing 2.5: PDF file trailer specification in BNF

Incremental Updates

As mentioned before, a section entry is either marked as "in-use" or as "free". If a section entry is in-use, it represents an indirect object of the PDF document; otherwise, a free section entry means an indirect object has been deleted. Consequently the indirect object number can be allocated for a new indirect object.

This mechanism is used intensely by a PDF creator application such as Adobe Acrobat. This mechanism also allows an incremental update of PDF documents which provides quick saving of small changes on large documents. If an incremental update is performed,

a new trailer is created and each new or changed object is appended to a new cross-reference section. The new trailer contains all the entries of the old trailer. The new cross-reference section not only contains the indirect objects that have been changed, replaced or deleted but the entry of obj 0 as well.

2.2.2 Objects and Indirect Objects

The PDF specification defines eight basic types of objects which are similar (but not equivalent) to data types and structures of other high-level programming languages such as Java (see Table 2.2). In the following section, the term *object* is equivalent to the terms *data type* and *data structure* and vice versa. In this section the BNF notation is used for the syntactical description of the PDF data types. The PDF specification is extended step-by-step in BNF.

Portable Document Format 1.4	Java 5
Boolean value	boolean
Numeric value	integer, float, double
String	String, char[]
Name	-
Array	Object[]
Dictionary	Map<Object,Object>
Stream	String
Null	null

Table 2.2: Comparison of data types between PDF 1.4 and Java 5 (source:[PPS01]).

Beside the following basic object types, there exist more complex data structures (such as *dates*, *name trees*, and *number trees*) as well *functions* and file specifications. These complex elements are not further discussed here. A detailed description of these elements can be found in Section 3.8 of the PDF Reference [PPS01].

Boolean values

The keywords **true** and **false** introduce a *boolean object*. It can be used as value in arrays and dictionaries or as operand in functions and conditional statements.

```
<boolean-value> ::= 'true' | 'false'
```

Listing 2.6: BNF of boolean values in PDF

Numeric values

The PDF Reference allows integers as well as floating point numbers with the restriction of the exponential format.

```
<integer> ::= ('-')?<number>
<numeric-value> ::= <integer>.<number>
```

Listing 2.7: BNF of numeric values in PDF

Strings

A *string object* contains text and can be defined in two different ways: either as a literal string by a sequence of literal characters enclosed in parentheses or as a byte-sequence in hexadecimal representation enclosed in angle brackets. Both representations are illustrated in Listing 2.8. A literal string can represent characters outside of the ASCII character set by specifying a three digit octal number with a leading \ character. A detailed specification of PDF strings is shown in Listing 2.9.

```
% string representation by a sequence of literal characters
(This is a literal string.)
% hexadecimal representation
% interpreted as ASCII code 41(A), 42(B) and 43(C) -builds the
string ABC
<414243>
```

Listing 2.8: Example of the different string representations in PDF

```
<special-char> ::= <delimiter-char>|
                  '*'|'!'|'&'|'^'|'.'|'@'|'%'
<octal-digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'
<escape-char> ::=
                  '\<octal-digit><octal-digit><octal-digit>
<literal> ::= 'a'|'b'|'c'...|'x'|'y'|'z'|
               'A'|'B'|'C'...|'X'|'Y'|'Z'|
               <special-char>
<string> ::=
            '('(<escape-char>|<literal>)*')'|'<'(<number>)*>'
```

Listing 2.9: BNF of Strings in PDF

Names

A *name* is introduced by the slash character / and a valid sequence of characters. A valid sequence must not contain a delimiter or white-space character. The character "#" is only allowed in conjunction with a hexadecimal value. For instance, the valid literal name /@Master's#20Thesis_ results in @Master's Thesis_. Name objects are unique and used to identify parameters or functions. Unique means that two name objects with the same character sequence identify the same object.

```
<all-literal>      ::= <literal>|<special-char>
<hex-value-names> ::= '#'<number>
<name>            ::= '/'(<all-literal>|<hex-value-name>)+
```

Listing 2.10: BNF of Name Objects in PDF

Arrays

A PDF *array* is similar to an object array (Object[]) in Java. In PDF, an array is a group of elements enclosed in square brackets [,] and can have different object types. Listing 2.11 shows an example of a PDF array containing a number, a string and a name object. Listing 2.12 specifies the array object in BNF.

```
[42 (This is a string) /Name]
```

Listing 2.11: Example of PDF Array storing different object types

```
<primary-datatype> ::= <boolean-value>|<numeric-value>|
<name>|<string>|<dictionary>
<array>            ::=
'['(<primary-datatype>|<indirect-obj-reference>)+']'
```

Listing 2.12: BNF of Array type in PDF

Dictionaries

A *dictionary* is a data structure containing key/value pairs as elements. A key must be a name object and a value can be any type of object. The double assignment of a key results into an undefined value. A dictionary structure is bounded by two angle brackets « and ».

```
<<
  /Type      Listing
  /SubType   SubListing
  /FirstItem 1          %key: FirstItem, value: 1
```

```

/SecondItem (a string) %key: SecondItem,value: a string
/SubDict <</FirstItem 1.1 >>
>>

```

Listing 2.13: Example of a Dictionary Object

By convention, the entries **Type** and **SubType** identify the type of dictionary. In PDF there is a huge set of predefined dictionaries with clear specified key/value pairs. These dictionaries are also called complex objects. For instance, a Type1 font object has the key values pairs **Type/Font** and **SubType/Type1**.

```

<dictionary-entry> ::=
    <name> (<primary-datatype>|<indirect-obj-reference>)
<dictionary> ::= '<<'(<dictionary-entry>+)'>>'

```

Listing 2.14: BNF of Dictionary Type in PDF

Stream Objects

In contrast to the aforementioned data types, a *stream object* is always defined as an *indirect object*. A stream object consists of a *stream dictionary* and a *stream sequence*. A stream dictionary contains additional information about the stream sequence such as the sequence length and the applied *filters*. The stream sequence represents the data byte-stream. It is enclosed by the keywords **stream** and **endstream**. Listing 2.15 shows a minimal example of a stream object defined as indirect object.

```

2 0 obj          % indirect object
<</Length 42>>
stream
... sequence of 42-bytes...
endstream
endobj

```

Listing 2.15: Minimal example of a Stream-Object

```

<stream-sequence> ::= 'stream' BYTE-SEQUENCE 'endstream'
<stream-object> ::=
    <indirect-obj-def-header>
    <dictionary><stream-sequence>
    <indirect-obj-def-tail>

```

Listing 2.16: BNF of Stream Objects in PDF

Filters

Filters are optional parameters of stream objects. Filters represent no data type but a decompression or decoding algorithm embedded in a PDF viewer application. More precisely, filters describe how to interpret the byte-sequence of a stream object. Thus filters are classified into two groups:

- **Decoding filters** describe how to decode arbitrary 8-bit binary data. The PDF Reference version 1.4 specifies two filters which allow the decoding of encoded data into ASCII hexadecimal representation.
- **Decompression filters** describe how to decompress the compressed data. The compression of large data volumes is valuable reducing storage requirements and transmission time.

In addition, the PDF specification allows cascading of two or more filters. Therefore each filter is registered in an array and is executed in a registered order.

```
% Stream directory
<<
  /Length ...
  /Filter [/ASCIIHexDecode /FlateDecode]
>>
```

Listing 2.17: Filter example of a Stream Object

The example in Listing 2.17 shows a stream directory which contains an array with two specified filters: ASCIIHexDecode and FlateDecode. The data of the stream will be decoded at first with the ASCIIHexDecode algorithm and then the result will be decompressed with the zlib/deflate algorithm. The encoding of the data must be realized in reverse order (zlib/deflate and ASCIIHexEncode). Table 2.3 summarizes the most important standard filters.

Filter	Description
ASCIIHexDecode	Decodes encoded ASCII data in hexadecimal representation
ASCII85Decode	Decodes encoded ACSII base-85 data
LZWDecode	Decompresses data compressed using the LZW (Lempel-Ziv-Welch) adaptive compression method.
FlateDecode	Decompresses data compressed using the zlib/deflate compression method.
RunLengthDecode	Decompresses data compressed using a byte oriented run-length compression algorithm.

Table 2.3: Extraction of available standard filters.(source:[PPS01])

Indirect Objects

Indirect Objects describe no objects in the sense of data types but indirectly reference objects by using unique object numbers. They are broadly similar to object references known from Java. An indirect object is enclosed by the keywords **obj** and **endobj**.

```
<indirect-obj-def-header> ::=
    'obj' <number> <number><eol>
<indirect-obj-def-tail>   ::= 'endobj'
<indirect-obj-definition> ::=
    <indirect-obj-def-header>
    (<primary-datatype>|<dictionary><stream-sequence>)<eol>
    <indirect-obj-def-tail>
<indirect-obj-reference> ::=
    <unique-obj-Nr> <gen-obj-number> 'R'
```

Listing 2.18: Indirect Object specification in BNF

Only indirect objects can be referenced by other objects. An indirect object is referenced by its unique number and the keyword **R**. The definition of an indirect object is introduced by the keyword **obj**, a unique object number and a generation number. The generation number determines how often an object has been deleted and reused over the incremental update mechanism.

2.2.3 Document Structure

A PDF document can be seen as a composite of indirect objects contained in the body section of the PDF file. To guarantee fast access to large documents, and to provide additional information about the appearance of a PDF document on the screen or printout, this composition of objects is managed hierarchically in a tree structure (see Figure 2.1).

In computer science, tree structures are frequently used because of simplicity and to aid better understanding. A recursive definition of a tree structure is: a tree consists of a root, a left sub-tree and a right sub-tree. A node without children is called *leaf*. A node with a parent and children is called an *inner tree node*. In a tree, there is only one node without a parent - it is the root node.

At the root of the *document's structure* is the document's catalog, which is a directory. The document's catalog references to other dictionaries containing references to *simple* or *complex objects*. A complex object is a predefined dictionary which can be accessed through defined key/value pairs. The PDF Reference specifies a large number of different

complex objects, and predefined dictionaries, such as page objects or fonts. The PDF specification describes several tree structures for maintaining different logical structure in a document such as *outlines* and *article threads*. An overview of the complete PDF Object Hierarchy is illustrated in Figure 2.1. It is important to understand how the content of a

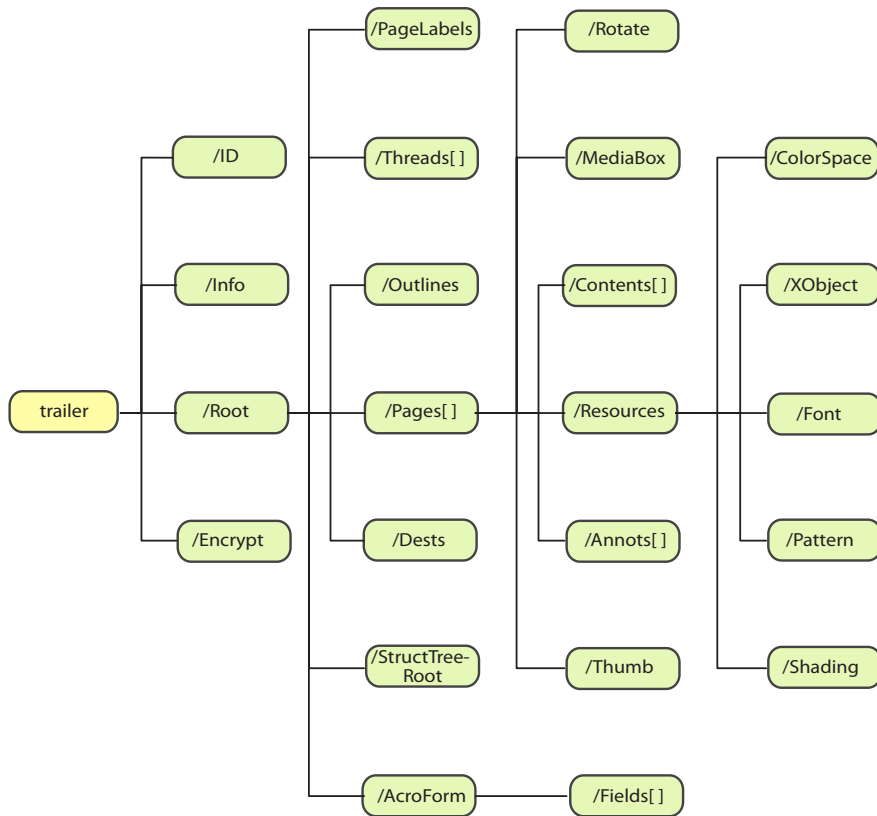


Figure 2.1: PDF Object Hierarchy (source:[MD01]).

PDF document is managed and structured in a PDF file. Hence, we take a closer look to the page tree and page objects of a PDF file.

Page Tree

All the pages of a PDF document are accessed through a *page tree*. A page tree is a complex object which defines the order of all the pages in the document. The page tree root is referenced by the document's catalog. Inner tree nodes are represented by page tree nodes and leaves are represented by *page objects*. A *page object* is a directory which represents a page in a PDF document. It disposes different attributes such as the last modified date or its two dimensional clipping area. The page tree structure ensure an efficient and quick access of each page in a document. A simple example of a page tree is shown in Listing 2.19. It illustrates a page tree of a PDF document with two pages:

the document's catalog on the first and the page tree root node on the second. The page tree (1 0 obj) references to two page objects, 5 0 R and 7 0 R. The page tree structure has nothing in common with the logical structure of the document. This means an inner tree node does not necessarily represent a chapter or section in the document but may represent fragments of a page only.

```

%-----Catalog-----%
3 0 obj
  <<
    /Type Catalog          % directory type must be Catalog
    /Pages 1 0 R          % reference to page tree
  >>
endobj
%-----Page Tree-----%
1 0 obj
  <<
    /Type Pages           % directory type must be Pages
    /Kids [5 0 R 7 0 R]  % references to children
    /Count 2              % number of children
  >>
endobj
%-----Page Objects-----%
5 0 obj                  % first child node
  <<
    /Type Page           % directory type must be Page
    /Parent 1 0 R        % parent node in tree
    /MediaBox [0 0 600 700] % defines a rectangle space
    /Resources <</Font   % reference to Font F2
                  <</F2 11 0 R >>
    >>
    /Contents [21 0 R 28 0 R]
  >>
endobj
7 0 obj                  % second child node
  <<
    /Type Pages ...
  >>
endobj

```

Listing 2.19: Basic example of a PDF document with two pages using page tree

Page Objects

As mentioned before, *page objects* represent leaves in the page tree. They contain two dimensional information about a page in a document and provide various inheritable attributes. In the example provided in Listing 2.19, the attributes **MediaBox** and **Resources** given in `5 0 obj` are mandatory and inheritable, where inheritable means that all child nodes and their children nodes inherit their attribute from their parent node. A child node can override inherited attributes by simply overwriting them.

The PDF specification defines various inheritable attributes. Inheritance is a mighty instrument and is used by many object oriented programming languages such as in Java for instance. Table 2.4 summarizes the core entries of a page tree node and page object. One the most important entries of a page object is the Resources entry. It points to external resources of a page such as a font directory or a stream containing an image.

Key	Type	Description
Type	<i>name</i>	<i>(Required)</i> It must be Pages for an inner node rather root node (page tree node) or Page for a leaf node (page object).
Parent	<i>dictionary</i>	<i>(Required; must be an indirect reference)</i> Defines the predecessor node of this node. The root element has an empty entry.
Kids	<i>array</i>	<i>(Required for page tree nodes)</i> An array of indirect references to the immediate children of this node.
Count	<i>integer</i>	<i>(Required for page tree nodes)</i> Defines the number of leaf nodes within the page tree.
Resources	<i>dictionary</i>	<i>(Required for page object; inheritable)</i> It might be an empty dictionary containing any resources required by the page.
MediaBox	<i>rectangle</i>	<i>(Required for page object; inheritable)</i> A rectangle area which the page is intends to display.
Contents	<i>stream or array</i>	<i>(Optional)</i> A single stream or array of <i>content streams</i> (see Section 2.3.1) which describe the content and appearance of this page.

Table 2.4: Core entries of page tree nodes and page objects.(source: [PPS01])

2.3 Graphic Model

After having outlined the PDF components we will have a closer look at the graphic model of PDF. More precisely, we will discuss *content streams*, *graphic objects*, and the various *coordinate systems* in PDF. This section summarizes the main features of Chapter 4 "Graphics", of the PDF Reference [PPS01].

2.3.1 Content Streams

The "P" of PDF is for portable, which means device independent. Adobe has developed a graphic model for PDF which enables a device independent page description using two dimensional graphic structures such as *paths*, *text* and *external objects (XObject)* like images. The graphic model provides several *graphic operations* for painting graphical shapes on a page.

Every page of a PDF document is represented by a page object which specifies different page attributes. Each page object includes a **Contents** entry (see Table 2.4), which contains one or more *content streams*. Content streams are streams containing a sequence of graphic operations to be applied on a page. An operation is introduced by a graphic operator, a predefined keyword. An operator may require some operands which can be any PDF data type except a stream or an indirect object. Listing 2.20 shows an example of a content stream in PDF without an encoding (see Section 2.2.2, Filters). The content stream will paint the string "Uncompressed streams can be read easily" at the upper left corner of a page. The example shows very good the use of some graphic operators. PDF defines a huge number of different operators for various painting operations. The most important operators and their functions are described briefly in the following sections.

```
...
21 0 obj
<</Length 78 >>
stream
% save the current graphic state
% and set the color to black
q 0 0 0 rg
BT
% paint a string with font F1 in size 12
% at position (56.8,774.1) in user space
56.8 774.1 Td /F1 12 Tf
[(Uncompressed streams can be read easily)] TJ
ET
% restore graphic state
Q
endstream
endobj
...
```

Listing 2.20: Example of an uncompressed Content Stream in PDF.

2.3.2 Marked Content Streams

A *marked content stream* is a content stream with so called marked content operators. Marked content operators designate interesting elements of a content stream to a particular application. Furthermore, marked content operators facilitate associating portions of a content stream with additional properties or externally specified objects. Table 2.5 summarizes the available marked content operators.

operands	operator	Description
<i>tag</i>	MP	Designates the next content operator. <i>tag</i> is a name object indicating the role of the operator.
<i>tag properties</i>	DP	Designates the next content operator. <i>tag</i> is a name object indicating the role of the operator. <i>properties</i> is a name or dictionary object; if it is a name it is associated with a <code>Properties</code> dictionary of the current page resource dictionary.
<i>tag</i>	BMC	Designates a a sequence of content operators. <i>tag</i> is a name object indicating the role of the operator. <i>Properties</i> is a name or dictionary object; if it is a name it is associated with a <code>Properties</code> dictionary of the current page resource dictionary.
<i>tag properties</i>	BDC	Designates a a sequence of content operators. <i>tag</i> is a name object indicating the role of the operator.
<i>none</i>	EMC	Terminates a marked-content sequence.

Table 2.5: Overview of marked content operators (source: [PPS01]).

The DP and BDC marked content operators associate a property list with a marked-content element within a content stream. This is a dictionary containing information meaningful to the current context.

2.3.3 Graphic Objects

Graphic objects are not objects in the sense of a data structure or indirect object, as mentioned in the previous sections; a graphic object rather introduces a specified graphical action and defines an environment for a set of graphic operators in a content stream. PDF specifies six graphic objects:

- A *path object* represents a simple shape consisting of line segments, rectangles and cubic Bezier curves. A path can be stroked, filled and clipped.
- A *text object* represents a sequence of characters being painted on a page. Special text operators specify place and style of character glyphs.

- An *external object* - *XObject* represents an external object outside of the content stream.
- An *in-line image object* represents a self-containing image such as raster and vector graphics.
- A *shading object* represents a geometric shape whose colour is a function of the position within a shape.

Adobe's PDF Reference specifies precisely which graphic object can be combined with another object object. Not every graphic object is nestable. For instance, a text object cannot be defined in another text object but a path object can be defined in a path object. An overview of the most important graphic operators is given by Table 2.6.

OPERANDS	OPERATOR	DESCRIPTION
-	q	Push current graphic state on the <i>graphic stack</i> . The graphic stack is a general stack which works after the LastIn-FirstOut principle.
-	Q	Pop graphic state from the graphic stack and override the current state.
<i>a b c d e f</i>	cm	Change the current transformation matrix by multiplying the defined a,b,c,d,e,f matrix.
$CTM = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix} * CTM$		
<i>lineWidth</i>	w	Set the line width to <i>lineWidth</i> which is number.
<i>lineCap</i>	j	Set the line cap to <i>lineCap</i> which is an integer. Only the values 0 for butt cap, 1 for round cap) and 2 for projecting square cap are allowed.
<i>gray</i>	G	Set the gray level for stroking operations to <i>gray</i> which is a number between 0.0. and 1.0.
<i>gray</i>	g	Same as G for non-stroking operations.
<i>r g b</i>	RG	Set a red-blue-green color for stroking operations to <i>r g b</i> which are a numbers between 0.0. and 1.0.
<i>r g b</i>	rg	Same as RG for non-stroking operations.
-	BT	Introduce a text object and initializes the text matrix T_m , text line matrix T_{lm} and text rendering matrix T_{rm} . For more details see at Section 2.4 "Text".
-	ET	End the current text object and discharge T_m .

Table 2.6: Overview of several graphic operators (source:[PPS01])

2.3.4 Coordinate Spaces

Every graphic object which is painted on a page is positioned by coordinates in a Cartesian coordinate system. In PDF, a coordinate system has at least one origin, a definition about the orientation of the x and y- axis and the length of the units along each axis. In order to provide a device independent appearance of PDF documents, Adobe has introduced different coordinate spaces. Figure 2.2 shows the relationship between the coordinate spaces, where each arrow represents a transformation from one space into another. In the following sections, each coordinate space is outlined briefly.

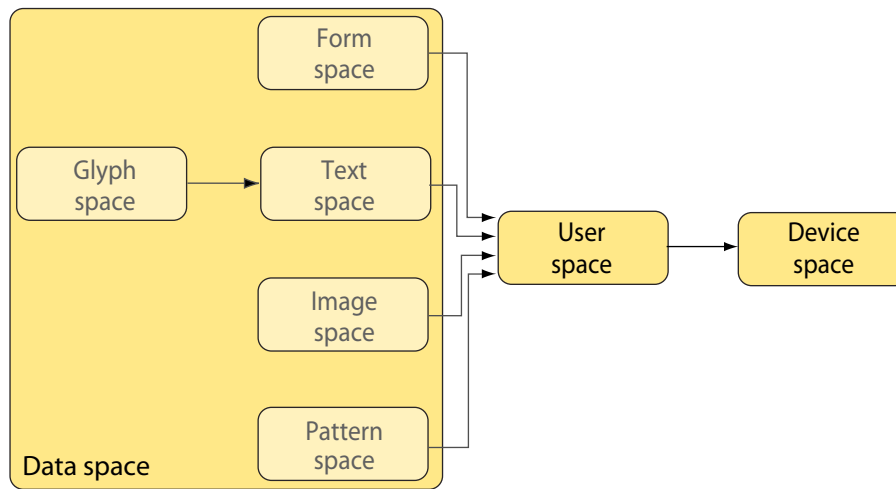


Figure 2.2: Overview of various Coordinate Systems in PDF and their relationships (source:[PPS01]).

Data Space

The *data space* involves the *form*, *glyph*, *text*, *image* and *pattern space*. The *form space* defines the coordinate system of external objects, called XObjects. They are treated as self-contained graphical elements. The transformation from form space to user space is specified by a form matrix contained in the XObject.

The glyph space is the space for glyphs. A glyph describes the graphical shape of a character. Glyphs are managed and specified in fonts. The transformation from glyph space to text space is performed by a predefined font matrix which maps 1000 units of glyph space to one unit of text space. The text space defines the coordinates of text objects. The text matrix describes the transformation from text space to user space. Glyphs and text objects are discussed in the next Section "*Text*".

All raster images are defined in the image space. The transformation from image space to user space is static, which means it is not adjustable. Each image is one unit high and one unit wide in the user space.

Finally, there is the pattern space, which represents repeating figures or smoothly varying color gradients. Patterns are always combined with painting operations such as stroke, fill, and show text. They are defined in pattern space.

User Space

The *user space* coordinate system builds an interface between the different data spaces and the device space. The user space is defined by the `CropBox` entry in the page object of the according page in a PDF document. By default, the `CropBox` is initialized with the size of the `MediaBox` of a page. The origin of the user space is at the lower left corner; the positive x axis is defined from left to right and the positive y axis is defined from bottom to top; 1/72 inches represents one unit on both axis. Coordinates from the user space are transformed into device space by the Current Transformation Matrix (CTM). The CTM ensures the same visual representation on every output device, because the resolution can vary from output device to output device (e.g. screen 72dpi, printer 600dpi). Hence, a PDF viewer application can adjust the CTM to the currently demanded device resolution.

Device Space

Finally, a PDF document is rendered on an output device. Each device has its own coordinate system the so called *device space*. Each device space may differ in the place of origin, orientation of the axis and length of units on the axis (resolution). It would be infeasible to ensure the same visual appearance of a document on different devices if the coordinate system were defined in device space instead in user space.

2.4 Text

After outlining PDF's graphic model we will discuss how PDF deals with the text. More precisely, this section summarizes Chapter 5 of the Adobe PDF Reference [PPS01] with the focus on the following questions:

- How is the relationship between *characters* and *glyphs*?
- How are glyphs organized in *fonts*?
- Which *text objects* and *text operators* are available in PDF?

2.4.1 Characters and Glyphs

In order to answer the first question, we will have to define the terms “character” and “glyph”. A common definition of the term character is: a character is a mark or (abstract) symbol of a written language ([MD01] page 178). Examples for characters are the letters of the Latin alphabet. A glyph is the graphical representation of a character. It defines the graphical shape of a character. For instance, the character "C" can be rendered as C, **C** and *C*, whereas the three representations illustrate three different glyphs. It is important to distinct between these two terms to avoid misunderstandings.

Before we take a closer look at the different font formats, we will explain some terms about glyph positioning and metric. The term metric summarize various kinds of information about a glyph such as the glyph’s *width*, *baseline*, *coordinate system* and the glyph’s *origin*. A glyph has its own coordinate system the so called glyph space. All the coordinates and metrics are interpreted in glyph space. The font matrix specifies the transformation to text space and is given by the font data structure. Initially, the glyph’s origin represents the point (0.0) in the glyph coordinate system (see Figure 2.3). The origin of the glyph space is positioned to the origin of the text space at the beginning of a Text Object. A text object may contain various text operators such as a text showing operator. A text showing operation may adjust the text matrix to perform a translation or rotation of a text.

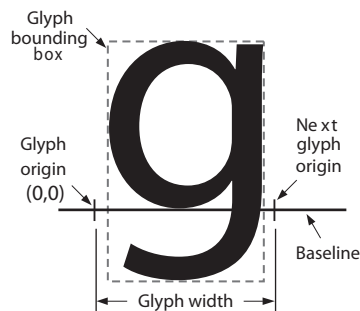


Figure 2.3: *Glyph metrics* (source:[PPS01]).

The *width* of a glyph is also called (*horizontal*) *displacement*. It represents the distance from the origin of the given glyph to the origin of the next glyph. In addition to the horizontal displacement there is a vertical displacement which is used generally in some Asian languages. In order to use the vertical displacement, PDF distinguishes between a horizontal and vertical writing mode which aligns text accordingly.

The dashed rectangle in Figure 2.3 illustrates the smallest box that envelopes the entire glyph. It is the so called *bounding box* of a glyph which is specified by the corner coordinates of the glyph.

Glyphs are defined and managed by Fonts. In the following sections, some important features of Fonts will be discussed briefly.

2.4.2 Fonts

A font represents a data structure that organizes and provides information for the proper illustration and positioning of glyphs. Generally, a font defines glyphs for a set of characters. The ultimate description of a glyph is generated by a *font program* written in a special font language. The font language is interpreted by an embedded font interpreter in the PDF viewer application. A font is represented in PDF as a dictionary specifying the type of font, its encoding, its metric and additional information.

Font Formats

Fonts are a central element of many applications. Therefore, over the years, various producers such as Adobe, Microsoft and Apple have invested a lot in developing digital fonts, mostly for their own interests. In consequence different font types and font formats have been created.

Two different techniques have been developed for defining the appearance of glyphs in fonts. On the one hand, a glyph can be described by means of discrete points, which are to be filled similarly to bitmap. Fonts using this technology are called bitmap fonts. On the other hand, a glyph can be described by defining its outlines (contours). Fonts of this kind are called outline-fonts. giving name outline-fonts. Each technique has its own advantages and disadvantages. For example, bitmap-fonts have a better performance but they are not scalable. This leads to artefact viewing in higher resolution. In contrast, outline-fonts look good but lose their aesthetic appearance through rasterisation (transformation of outlines into point-grid).

Different font formats have been developed over the years. The most important formats are:

- Type 1 fonts are outline-fonts which are fast in processing and improve the output quality with so called hints. Hints are additional information which improve the quality considerably, especially in lower resolutions. The Type-1 format has been

designed with the focus of the output on PostScript compatible printers. Adobe has forced its PostScript format with high quality fonts.

- TrueType is the standard format of Windows and Mac operating systems. The TrueType format has been designed for use with screen output.
- OpenType is Adobe's answer to TrueType. It unions PostScript and TrueType and operates on any operating system. OpenType is based on the data structure of TrueType. It supports hinting and different illustration techniques.

PDF supports a huge range of different font formats and covers the entire spectrum of available fonts, see Table 2.7.

Type	SubType	Description
Type 1	Type1	Type 1 font defines glyphs shape and how to encode them.
	MMType1	A multiple master font that extends the Type 1 font, see Section 5.5.1, Type 1 Fonts in the PDF Reference [PPS01]
Type 3	Type3	Type 3 fonts represent user defined fonts that are always embedded as stream in the PDF file.
TrueType	TrueType	A TrueType font.
Type 0	Type0	A composite font is one whose glyphs are obtained from other fonts or from font like objects called CIDFonts.
CIDFont	CIDFontType0	A CIDFont describes glyph shape based on the Type 1 technique, see Section 5.6.3, CID Fonts in the PDF Reference [PPS01]
	CIDFontType2	A CIDFont describes glyph shape based on the Type 1 technique, see Section 5.5.1, Type 1 Fonts in the PDF Reference [PPS01]

Table 2.7: Font types available in PDF (source:[PPS01]).

PDF provides the embedding of fonts in a document. Therefore, PDF provides a specified *font object*. Table 2.8 describes the most important dictionary entries of the font object.

Key	Type	Description
Type	<i>name</i>	(Required) Defines the font type which must be a type from Table 2.7.
Subtype	<i>name</i>	(Required) Defines the font sub type which must be a sub-type from Table 2.7.
BaseFont	<i>name</i>	(Required) The PostScript or TrueType name of the font.
FirstChar	<i>integer</i>	(Required) The first character code defined in the font's width array.

LastChar	<i>integer</i>	(Required) the last character code defined in the font's width array.
Width	<i>array</i>	(Required) Describes the glyph width of each defined glyph in the font which applies to the characters in the range of FirstChar - LastChar. For characters outside the range of FirstChar to LastChar the value MissingWidth is assigned. The glyph width is measured in glyph space.
FontDescriptor	<i>dictionary</i>	(Required; indirect object) General description of the font.
Encoding	name or dictionary	(Optional) - Defines the encoding used which is either a name (MacRomanEncoding, MacExpertEncoding or WinAnsiEncoding) or an encoding dictionary that describes the differences from the font's built in encoding or specified encoding.
ToUnicode	<i>stream</i>	(Optional) A stream containing a CMap file that maps character code to Unicode values.

Table 2.8: Overview of common font dictionary entries (source: [PPS01]).

Character encoding

The character encoding describes the mapping between character codes which are obtained from text strings and glyphs.

In general, font programs have a built-in encoding. For certain reasons, it might be necessary to overwrite the build-in encoding. This can be done by including an Encoding entry in the PDF font dictionary. The value of the Encoding entry is either a name of a base-encoding (the name of one of the predefined MacRomanEncoding, MacExpertEncoding, or WinAnsiEncoding) or an encoding dictionary. The encoding dictionary describes the differences to the encoding specified by a base-encoding.

Listing 2.21 illustrates an example of embedding the *Helvetica* font. It shows a definition for a font dictionary and defines an appropriate encoding. The encoding used is specified in `9 0 obj` which describes differences between the encoding to be applied and the font's built-in base-encoding. The specified array contains integers or names. An integer represents a character code and is the first index (begin) in a sequence of characters which should be changed. The character code is paired with the subsequent character name. Consecutive code indices are paired with subsequent names until a next character code appears. For example, in Listing 2.21: the character code 39 is alternated to the glyph `"'` (quotesingle) and code 40 is alternated to `"(" (leftparenthesis).`


```

7 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /FirstChar 0
  /LastChar 255
  /Widths [
    500 500 500 500 ..... 500 500 500 500
    500 500 500 500 ..... 500 500 500 500
    235 235 332 507 ..... 219 353 219 381
    507 507 507 507 ..... 594 604 594 402
    819 677 606 667 ..... 551 924 731 739
    . . . . .
    . . . . .
    455 546 546 478 ..... 561 445 326 386
  ]
  /Encoding 9 0 R
  /BaseFont /Helvetica
  /FontDescriptor 10 0 R
>>
endobj

9 0 obj
<<
  /Type /Encoding
  /Differences
  [ 39 /quotesingle /leftparenthesis
    96 /grave
    128 /Adieresis /Aring /Ccedilla /Eacute /Ntilde
      /Odieresis /Udieresis /aacute /agrave
    174 /AE /Oslash
    177 /plusminus
    180 /yen /mu
      .....
  ]
>>
endobj

```

Listing 2.21: Example of a font dictionary that embeds the WarnockPro-Italic (source: [MD01]).

Unicode CMaps

Beside the mapping of character codes to glyphs, it is sometimes necessary to know the information content of glyphs. This means the presented glyphs of a document must be interpreted in sense of text such as for text searching, indexing and extracting. The Unicode standard defines a system for numbering all of the common characters used in a large number of languages. It is a suitable scheme for representing the information content of text, but not its appearance, since Unicode values identify characters, not glyphs [PPS01].

Under certain circumstances, the information content of a character can not be determined because of insufficient information. Glyphs can still be shown, but the characters cannot be identified and consequently, they cannot be converted to Unicode values without additional information. To tackle this problem, PDF provides an optional ToUnicode entry in the font dictionary, whose value is a stream object containing a special kind of CMap file which maps character codes to Unicode values.

A CMap file must follow a clearly defined syntax, as described in Section 5.6.4, *CMaps* in the PDF Reference [PPS01]. A CMap file must contain **begincodespacerange** and **endcodespacerange** operators, which define the source character code range. In Listing 2.22, the CMap file comprises a Unicode mapping for the characters 0 - 255 in decimal representation. In addition, a CMap file must use **beginbfrange**, **endbfrange**, **beginbfchar**, **endbfchar**, **beginbfrange** and **endbfrange** operators to define a mapping from character codes to Unicode values. In Listing 2.22, the characters code from hex-value 0x00 to hex-value 0x5E are mapped to the Unicode values U+0020 to U+007E, where Unicode values are written as U+ followed by four hexadecimal digits. Furthermore, the character codes in hexadecimal 0x5F, 0x60 and 0x61 are mapped to the Unicode strings U+0066U+0066, U+0066U+0069 and U+0066U+0066U+006C.

```
...
1 begincodespacerange
  <00> <FF>      % source character code range
endcodespacerange
2 beginbfrange
  % mapping of characters 0x00-0x5E to 0x20-0x7E
  <00> <5E> <20>
  % Unicode representation of characters 0x5F, 0x60 and 0x61,
  % they are mapped to U+0066U+0066 "ff", U+0066U+0069 "fi" and
  % U+0066U+0066U+006C "ffI"
  %
```

```

<5F> <61> [<00660066> <00660069> <00660066006C>]
endbfrange
...

```

Listing 2.22: Fragment of a CMap stream (source:[PPS01]).

In view of text extraction, the mapping of character codes into Unicode values is an important point. Another important point, how text is managed in a PDF document. The following sections will address this question.

2.4.3 Text Objects and Operators

Beside graphic objects, PDF introduces so called *text objects* for covering all the painting mechanisms and operations that deal with text. Similar to general a graphic object, a text object builds an environment that initializes the *text state* for a set of certain *text operators*. A text operator can show text strings, moves text or modifies the text state. Text objects are not nestable which means a text object contains no other text object. In addition there are some volatile parameters which only persist at the current text object:

- T_m represents the text matrix
- T_{lm} represents the line matrix
- T_{rm} represents the text rendering matrix

A text object is enclosed by the operators **BT**, which begins a text object, initializes the text matrix, and text line matrix to the identity matrix. The operator **ET** which ends the text object and discharges the text matrix. In general, a text object may contain one or more text operators. A detailed description of text operators is given in Table 2.9.

2.4.4 Text Positioning

In general, text is shown in text space. For the illustration of text in user space a transformation of text space coordinates is needed. The text space to user space transformation can be represented by the following equation [PPS01]:

$$T_{userspace} = \begin{bmatrix} T_{fs} * T_h & 0 & 0 \\ 0 & T_{fs} & 0 \\ 0 & T_{rise} & 1 \end{bmatrix} * T_m \quad (2.1)$$

$T_{userspace}$ is computed before a glyph is painted by a text-showing operation indicated by the operators **TJ** or **Tj**. After a glyph is painted, the T_m matrix is updated for positioning

the next glyph, which can be performed as follows [PPS01]:

$$T_m = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} * T_m \quad (2.2)$$

$$t_x = \begin{cases} (\frac{font_w - t_j}{1000} * T_{fs} + T_c + T_w) * T_h \\ 0 \text{ if vertical writing mode} \end{cases}$$

$$t_y = \begin{cases} \frac{font_h - t_j}{1000} * T_{fs} + T_c + T_w \\ 0 \text{ if horizontal writing mode} \end{cases}$$

where

$font_w$ is the glyph's horizontal displacement.

$font_h$ is the glyph's vertical displacement.

t_j is a position adjustment specified by a number in a TJ array, default value is 0.

T_{fs} and T_h are the current text font size and horizontal scaling parameters in the graphics state.

T_c and T_w are the current character and word spacing parameters in the graphics state, if applicable.

To note is, beside positioning, the formula can be used for calculating the bounding box of a glyph. This will be required for the box model of the presented text extraction algorithm in the next chapter.

Table 2.9 summarizes the most important Text operators available in PDF. PDF provides various operation for adjusting the position of glyphs on a page to produce different text effects.

OPERANDS	OPERATOR	DESCRIPTION
<i>charSpace</i>	Tc	<p>Set the character space to <i>charSpace</i> which is a number; initial value:0.</p> <p>The character spacing parameter adds its value to glyph displacement vector, whereas the positive values increase and the negative values decrease the total character spacing. For instance, a T_c value of 0.25 looks as follows:</p> <div style="text-align: center;"> <p>$T_c = 0$ (default) Character</p> <p>$T_c = 0.25$ C h a r a c t e r</p> </div>
<i>wordSpace</i>	Tw	<p>Set the word spacing to <i>wordSpace</i> which is a number; initial value:0.</p> <p>In addition to the character space it is possible to adjust the space between words. The Tw operator is analogue to the Tc operator; for instance, a T_w value of 0.25 generates the following output:</p> <div style="text-align: center;"> <p>$T_w = 0$ (default) Word Space</p> <p>$T_w = 2.5$ Word Space</p> </div>
<i>scale</i>	Tz	<p>Set the horizontal scaling to $scale \div 100$; scale is a number; initial value:100.</p> <p>The scaling parameter T_z enables to stretch or shrink text in horizontal direction. A T_z value under 100 shrinks text, see at following example:</p> <div style="text-align: center;"> <p>$T_h = 100$ (default) Word</p> <p>$T_h = 50$ WordWord</p> </div>
<i>font size</i>	Tf	<p>Set the text font to <i>font</i> and font size to <i>size</i>; A <i>font</i> is referenced by the name of the font resource in the Font sub-dictionary; <i>size</i> is a number. There is no initial value of font and size, and must be set before text is shown.</p>

t_x t_y

TD

Go to the beginning of the next line and add the values of t_x , t_y as offset. Change the current line leading parameter T_l to $-t_y$.

The operator is equal to the following code:

$-t_y$ **Tl** % adjust line leading

t_x t_y **Td**

a b c d e f

Tm

Replaces the current text and text line matrix as follows:








$$T_m = T_{lm} = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}$$

render

Tr

Set the rendering model to *render*, which is an integer; initial value:0.

PDF provides different text style manipulations for stroking, filling and clipping boundaries, as well as the combinations of all of them. The next table outlines all the available rendering modes:

MODE	EXAMPLE	DESCRIPTION
0		Fill text.
1		Stroke text.
2		Fill, then stroke text.
3		Neither fill nor stroke text (invisible).
4		Fill text and add to path for clipping (see above).
5		Stroke text and add to path for clipping.
6		Fill, then stroke text and add to path for clipping.
7		Add text to path for clipping.

$t_x t_y$	Td	Go to the beginning of the next line and add the values of t_x, t_y as an offset. The Td performs following matrix multiplication: $T_m = T_{lm} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{pmatrix} * T_{lm}$
-	T*	Performs a matrix multiplication and sets a new text matrix T_m and text line matrix T_{lm} with the following code: <code>0 T1 Td</code>
<i>string</i>	Tj	Display a text string.
<i>array</i>	TJ	Display one or more text strings and allows individual positioning of glyphs. An element of array is either a string or a number, where a number is expressed in thousands of units and translates the text matrix, T_m by that amount.
<i>array</i>	Tc	Array can contain text strings and/or numbers. If an array entry is a text string it is shown. If an array entry is a number the amount is subtracted from the current coordinate, whereas positive values translates glyphs to left or down, depends on the current writing mode. This operator enables a character-wise positioning of text.

Table 2.9: Overview of available text operators in PDF version 1.4 (source:[PPS01]).

Finally, the page description language PDF provides a huge number of features to support the representation of also eccentric documents. This chapter has given a small impression of some features of PDF only. A detailed description of PDF is beyond the scope of this thesis. The next section will focus on the text extraction of PDF documents. Therefore, firstly, the problems of PDF regarding text extraction will be outlined and secondly, a text extraction algorithm will be introduced.

Chapter 3

Specification

PDF provides a rich set of features for illustrating documents. This ranges from illustrating of forms with dynamic elements to illustrating of complex figures. However, the nature of PDF creates difficulties in enabling the reconstruction of Official Signatures, as required by the law. PDF does not support an unambiguous text extraction because PDF documents are not necessary self-contained and self-describing, which is important to enable a further processing of the document's text content.

The "new " document format PDF/A-1a solves these problems by restricting PDF. PDF/A-1a describes a "profile" for PDF documents with the focus on self-containing and self-describing. However, PDF/A-1a defines a ruleset for PDF 1.4 that forbids and enforces special features to support a text processing. PDF/A-1a documents become more suitable for text extraction and, consequently, for Official Signatures. However, PDF/A-1a does not solve all problems with respect to Official Signatures, which will be discussed in the following sections.

This chapter is divided in two parts: The first part addresses the problems of text extraction from PDF and PDF/A-1a documents. Therefore, an overview of the most important issues is given and the PDF/A standard is described. The second part specifies the canonical Text Extraction Algorithm (TEA). TEA represents a proof-of-concept with the aim to enable a text extraction from PDF/A-1a documents with respect to the requirements of Official Signatures. TEA will replace the current text extraction algorithm of the underlying PDF-AS system 2.0 [EG08]. Furthermore, the requirements of TEA are defined in Section 3.4 and the major concepts of TEA are specified in Section 3.5.

3.1 The problem to be dealt with

PDF is well established as an interchange format because of its platform and application independence. In order to also use the document format in the field of eGovernment, it is mandatory to affix an Official Signature on it. As discussed at the beginning of this thesis, this is because of legal regulations by the E-GovG. The §4 of SigV¹ extends the requirements of the interchange format PDF, it instructs that:

[...] die signierten Daten sowohl bei der Signaturerstellung als auch bei der Signaturprüfung zweifelsfrei und mit gleichem Ergebnis darstellbar sind [...]

[...] the signed data must be absolutely certain (unambiguous) and shown equally at the time of signature creation and verification [...] (loose translation)

In order to satisfy this requirements the use of *dynamic* and *external objects* in PDF documents are forbidden in eGovernment. Moreover, the statute ensures that a signed document comprises no *hidden elements* in particular, *hidden text*. Hence, dynamic, external and hidden objects must be identified and handled suitably. The following chapter classifies problematic features of PDF in terms of electronic signatures, more precisely in terms of text extraction and how to tackle them with PDF/A-1. The following classification is partly based on Study of secure signatures with PDF/A [Neu07].

Dynamic Objects

PDF has allowed the embedding of JavaScript since version 1.4. JavaScript is well known in web-development as script language. It provides an object-oriented access to html files. It is heavily used for developing dynamic websites which change their content on demand. In PDF, JavaScript is used for dynamic maintaining of formula fields. With regard to signatures, JavaScript can be dangerous. For example, a script can be used to change the amount of an account at the time of signing and verification. The document content can be modified without breaking the signature of document. Hence, the property of unambiguousness of signed data is not given any more.

External Objects

The best example of external objects are fonts. In most cases, fonts are not embedded in the associated PDF file. Thus fonts will be substituted by other fonts if they are not present on the system when viewing the document; and this, in turn, results in that

¹"Verordnung des Bundeskanzlers über elektronische Signaturen (Signaturverordnung - SigV)", BGBl. II Nr. 30/2000, BGBl. II Nr. 527/2004

the font encoding, the mapping of character codes to glyphs, can not be recognized any more. For instance, the 'ö' glyph may become a different glyph. This is a high potential threat because the signature would not be broken in this case although the appearance and printout of the PDF document is modified. This state depends on the font encoding, how a glyph is selected by a character code. Another problem in terms of encoding is the character encoding, which is the mapping of character codes to characters. In this case the glyphs can still be shown, but the characters cannot be converted to Unicode values without additional information. A reliable text extraction becomes infeasible.

Text-In-Picture

Text-in-picture means the representation of text as a picture. For instance, somebody creates a picture that contains text. This picture is added to a document which is exported as a PDF file. Thus, on viewing the PDF document it seems that the document shows text but it does not; it shows a picture. The reconstruction or verification of the printout's signature becomes infeasible because at printout it is not possible to differentiate between a text and a picture that contains text. Hence, the signature that is generated on the text content of the PDF document is not reproducible/verifiable by having only the printout. The same applies for Paths (geom. shapes) in PDF.

Hidden Elements

Hidden elements are elements that are not visible but present in the PDF document. Even hidden text can become a problem at printout. While the electronic document can be verified successfully, the printout is not verifiable because the hidden text get lost during printing.

The PDF problems lead to the conclusion that dynamic and hidden objects as well as pictures and shapes should be removed and external objects should be integrated into the PDF file. Only then PDF supports the reconstruction of Official Signatures from printouts.

3.2 PDF/A-1 (ISO 19005-1)

The problems of archiving PDF documents are similar to those discussed above. Also here a document format is needed to ensure that documents remain readable, capable of being rendered and accessible under preserving its integrity and visual appearance for the long-term. This subsection points out the intention of PDF/A-1 (ISO 19005-1) [Int05], the benefit of Tagged PDF and the relations between them.

The feature-rich nature of PDF creates difficulties in preserving information over long-term. PDF documents are not necessarily self-contained, drawing on system fonts and dependencies on external objects; on the other hand as time goes by technology changes, external connections will be broken and so information gets lost.

There are various methods for archiving electronic documents such as storage then as picture (TIFF) or as microfilm but with the disadvantage of losing the possibility of having access to its text based content (e.g. text search). In order to use the advantages of PDF and to meet the requirements of long-term archiving of electronic documents, the ISO 19005-1 (PDF/A-1) international file format standard has been established. The PDF/A-1 standard was published on October 1, 2005 with the full title: "ISO 19005-1: Document management – Electronic document file format for long-term preservation – Part 1: Use of PDF 1.4 (PDF/A-1)". PDF/A-1 represents the first part in a new family of ISO standards which defines a file format for long-term preservation of electronic documents based on PDF 1.4. The scope of this standard is on preserving the visual appearance and the maintenance of information in electronic documents, independently of tools and systems which are used for creating, storing or rendering over archival time spans. A second part, PDF/A-2, will be published in 2009 based on a newer PDF version.

In fact, the PDF/A-1 standard focuses only on how to use PDF 1.4 for long-term archiving. PDF/A-1 does not specify an archiving strategy and does not cover for electronic signatures. The standard specifies only a minimum set of PDF 1.4 features in order to ensure preserving the visual appearance of electronic documents. In other words, PDF/A-1 represents only a "profile" of PDF 1.4 .

Currently, PDF/A-1 specifies two different conformance levels:

- PDF/A-1a (ISO 19005-1a): Level “A” conformance: A PDF/A-1a document ensures the preservation of structural and semantic properties of the document. Level 1a uses “Tagged PDF” and Unicode character maps to preserve the document’s logical structure and content text stream in natural reading order. The Unicode standard defines a system for numbering characters used in a large number of languages in order to ensure a unique identification of characters.
- PDF/A-1b (ISO 19005-1b): Level “B” conformance: A PDF/A-1b document ensures the exact visual reproduction of the author’s original only and does not contain any information about the logical structure of the document and the character encoding.

A PDF/A-1 document should be self-contained and self-descriptive which means they contain all resources (fonts, structure information) necessary for rendering, printing and for processing the document. The following list outlines the most important restrictions and instructions of PDF/A-1 documents compared to the PDF 1.4 specification. More information about the technical requirements of PDF/A-1 is in the PDF/A-1 specification [Int05].

- The encryption of a PDF/A-1 document (as described in Section 3.5 [PPS01]) is not permitted.
- Transparency is not allowed in PDF documents.
- All colours shall be specified in a device-independent manner.
- Each font used must be embedded, which involves the font program, glyph metrics and character encoding so that the font is embeddable for unlimited, universal rendering. Each associated font directory includes a `ToUnicode` entry, whose value is a CMap file which maps every character code to a 2-byte Unicode value (see Section, 2.4.2 `ToUnicode` CMaps).
- A stream directory shall not contain the `F`, `FFilter`, or `FDecodeParams` keys and the `LZWDecode` filter is not permitted due to license rights.
- Audio and video content are forbidden.
- JavaScript and launching of executable files are prohibited.
- Interactive form fields and external links (e.g. hyper links) are forbidden.
- A Level A (PDF/A-1a) conform document have to meet the requirements of Tagged PDF (see Section 9.7, Tagged PDF in [PPS01]).

Tagged PDF is important for text extraction from PDF/A-1a documents. Only a "tagged" PDF document delivers enough information to extract its text content. The basic idea of Tagged PDF is discussed in the next paragraph.

3.3 Tagged PDF

PDF is well suited for the illustration and printing of documents containing text, images and other information. On the other hand, PDF is not well suited for maintaining a document's content. For instance, the selection or copying of text from a PDF document does not always work properly. Which means the copied text does not represent the illustrated

one. The problem is the character encoding; it defines the mapping of character codes to characters themselves. In general, characters can be identified according to a particular standard character set, for instance Unicode. Unicode represents a unified character set that allows a unique character identification by means of code points (character code). Without knowing about the character encoding, characters can not be interpreted (Unicode value) without additional information but glyphs can still be shown in PDF.

To resolve the issue of character identification, Tagged PDF defines a set of rules for representing text in the page content so that characters, words, and text order can be determined reliably. In addition, Tagged PDF defines conventions for explicitly declaring and describing the logical structural with respect to the document's content. Tagged PDF is based on two mechanisms of PDF 1.4, which are:

- marked-content operators to designate fragments of a content stream and
- logical structure facilities to define structural information.

Marked-content operators have already been discussed in Section 2.3.2, Marked-content streams.

In order to describe the document's logical and visual structure, Adobe has introduced special logical structure facilities. The logical structure of a document is described by a structure tree. The root of the tree is called the *structure tree root* and other nodes are called structure elements. Every structure element has a type and one or more content items. A content item can be a marked-content sequence embedded within the content stream, a complete PDF object such as an annotation or another structure element. The type of structure element is determined by the **S** entry which characterizes the role of the content item within the document. PDF specifies a set of standard structure types which are mostly similar to HTML tags. Users can also define its own structure types. To interchange user types with other applications, a role map is needed which maps user types to equivalent standard types. In addition to structure types, a structure element can be described by structure attributes. A structure attribute is similar to an HTML attribute which assigns specific properties to an element. Furthermore, PDF allows the grouping of elements together by a *class map*. This is needed to build a class of structure elements with similar attributes.

The major focus of Tagged PDF is to provide a set of standard structure types to stylize PDF content with the intent to provide additional structural information to Tagged PDF consumers. This includes the possibility for consumers to make their own layout decisions while preserving the author's document layout. Tagged PDF has introduced four different

categories of structure element types which are been divided into:

- Grouping elements group elements into hierarchies. Examples of grouping elements are:
Document, Part, Art, Sect Div, BlockQuote, Caption
- Block level structure elements layout text or other content such as the heading, list item or footnote. Examples of Block-level structure elements are:
P, H1-H6, Table, TR, TH, TD, L, LI.
- Inline level structure elements specify the styling of text or other content. Examples of Block level structure elements are:
Span, Quote, Note, Reference.
- Illustration elements are any structure elements whose structure type is one of following:
Figure, Formula, Form.

In addition to the logical structural facilities, Tagged PDF encompasses a set of conventions [PPS01]:

- A Tagged PDF document involves a structure tree.
- A Tagged PDF document contains information about the content order regarding the natural reading direction.
- Each character of a Tagged PDF document can be mapped to a conform Unicode representation.
- A Tagged PDF document provides information to distinguish between real content and artifacts of the layout process. The real content of a document covers all the PDF objects in a file which are important with respect to the document's content. All other objects, which are not related to the original document's content, are called artifacts. Artifacts are created in course of layout or technical processes during the PDF generation.
- Representing text in a form from which an Unicode representation and information about font characteristics can be unambiguously derived.
- Representing word breaks unambiguously.

We assume that Tagged PDF is well suited for providing information about the document's content and its logical structure. Tagged PDF facilitates PDF to embed structure information which can be used for text extraction.

However, a PDF creation application, for instance Adobe PDF Driver, creates a PDF/A-1a file from an original document. The logical structure, which is represented by the structure tree, is derived from the original document. The issue is that there don't exist clear defined guidelines (policies) for creating the logical structure tree, such as: If the y-position of an element A (e.g. an image) is greater than the y-position of an element B (e.g. some text), then element A is inserted before element B in the logical structure tree. In consequence, the structure tree of a PDF/A-1a document can vary from PDF creation application to application. The uniqueness of the logical structure tree is not given any more. This gives leeway in interpreting the logical order of a document and may produce different results in text extraction. So the reconstruction of an Official Signature from printout, that bases on the logical structure tree, becomes infeasible. This is a knock-out criterion to use the logical structure tree of a PDF/A-1a document for text extraction regarding Official Signatures.

3.4 Algorithm Requirements

This section describes the algorithm requirements [AR] and preconditions of TEA in detail.

- **[AR1] TEA requires a PDF/A-1a conforming document as input.**
Text extraction from traditional PDF documents could become infeasible. Thus, TEA is performed on a PDF/A-1a conforming document that is self-contained and provides a minimum of information as follows:
 - information to map any character to its Unicode representation.
 - information to identify words by defining clear word breaks.
- **[AR2] TEA must deliver a unique result.**
In this context, unique means that two different PDF/A-1a documents deliver two different results. Two documents are different if, and only if, its visible text content is different. This condition includes the fact that the text extraction is done in a standardized manner. The text content of a PDF/A-1a document comprises every character that is shown by the "text-showing" operators Tj and TJ . In addition, the result of TEA should only contain the visible text content of the document's printout. Hidden text must be excluded from the result. Text is called hidden if, and only if, it meets any of the following points:
 - It is not present at printout.
 - It is crossed out at printout.
- **[AR3] The result of TEA must be a sequence of characters according to the document's visible text content on printout.**
With regard to the reconstruction of a text-based signature it is important to only sign visible text content of the document's printout. In other words, only the visible text content of a PDF/A-1a document should be signed.
PDF artifacts must be recognized and treated in view of the reproduction of the document's printout. Artifacts are graphic objects which are not part of the original document. They result because of visual effects or mechanical processes during PDF generation. For example, some PDF documents emulate a bold character by two overlapped characters with marginal, horizontal displacement. In general, a font provides separate rendering information for illustrating bold text with single characters.
- **[AR4] The extracted sequence of characters must follow the natural reading direction of the document from top left to bottom right.**

3.5 Algorithm Specification

This section focuses on the specification of TEA. Hence, we will describe the principal concepts and give generic information about TEA according to the defined requirements. Figure 3.1 shows the two phase concept of TEA. Any phase is performed successively on the input PDF/A-1a document. The result of any phase is the input of the next phase. Finally, TEA returns the text content of the given PDF/A-1a document.

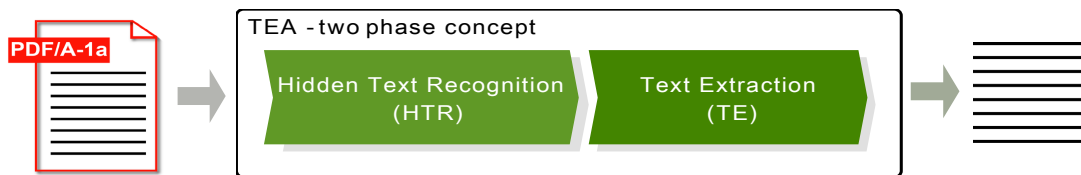


Figure 3.1: Basic concept of the text extraction algorithm TEA consisting of two phases: hidden text recognition (HTR) and text extraction (TE).

With respect to the algorithm requirements, only visible text content should be extracted. Therefore, TEA is divided into two parts. At first, a Hidden Text Recognition (HTR) algorithm determines hidden text fragments which should be removed from the document. Secondly, only the visible text content is extracted in natural reading order (logical order) from the top left to the bottom right corner. In the following sections, each phase will be described in detail.

3.5.1 Data Model

Before we specify each phase, it is worth looking at the data-model that is used in the following section. Therefore we will take a closer look at the composition of a PDF page. A PDF page can be imagined as a two dimensional plane on which various elements/objects are attached. The set of elements can be divided into three classes:

- **Image elements** are represented by external objects (`XObject`) or inline images, introduced by the PDF `Do` operator (see Section 4.8 in [PPS01]).
- **Path elements** are represented by PDF path operations which are used to draw lines and to define shapes of filled areas (see Section 4.4 in [PPS01]).
- **Text elements** are represented by single-characters of the document introduced by PDF text-showing `TJ` and `Tj` operators (see Section 2.4).

For further processing, each page element is abstracted by its bounding box. The bounding box of an element describes the smallest iso-oriented rectangle area that covers the complete element. Iso-oriented means that the left and right edge as well as the top and bottom edge of a rectangle are parallel to the y- and x-axes of the coordinate system (e.g. in user space). The bounding box of an element is defined by its position on document, height, width and the class of the associated page object. Figure 3.2 shows the different page elements with their bounding boxes. The bounding box of a text element can be

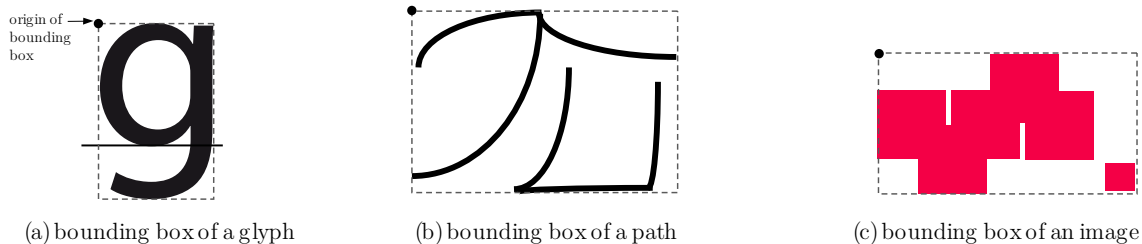


Figure 3.2: Bounding boxes of page elements.

determined by the bounding box of the associated glyph. This requires some calculations which are equal to the formulas in Section 2.4.4, Text Positioning, with slight adaptations, as follows:

$$glyph_{width} = \left(\frac{font_w}{1000} * T_{fs} + T_c + T_w \right) * T_h \quad (3.1)$$

$$glyph_{height} = \frac{font_h}{1000} * T_{fs} \quad (3.2)$$

where

$font_w$ is the glyph's horizontal displacement.

$font_h$ is value of the font entry Ascent in the associated font descriptor dictionary.

T_{fs} and T_h are the current text font size and horizontal scaling parameters in the graphics state.

T_c and T_w are the current character and word spacing parameters in the graphics state.

The bounding box of an image (see Figure 3.2) is calculated by applying the current transformation matrix on a unit square with lower left corner (0-0) and upper right corner (1-1). Furthermore, the bounding box can be determined by the transformed unit-square. In PDF, a path is defined by a sequence of points to describe a Bezier curve. The bounding box of a path object is calculated by its lowest left and highest right point. This can result into an overhead which is accepted.

3.5.2 Hidden Text Recognition (HTR)

At first, TEA performs a HTR algorithm to identify hidden text fragments in the document. There are three cases when text becomes hidden in a PDF/A-1a document. Each case must be taken into account during hidden text recognition:

Case 1 Text is rendered in text rendering mode three or seven which is introduced by the **Tr** text operator within a content stream (see PDF Reference [PPS01], Section 9.5.2 Marked Content and Clipping).

Case 2 Text is overlapped by other page elements such as images or paths (shapes).

Case 3 Text becomes hidden because of marginal colour differences (low contrast) between the text colour and background colour.

HTR Case 1 - rendering mode

This case is easily detected by scanning each content stream for the **Tr** operator and its value. PDF provides eight rendering modes which are used to support text effects. An overview of all rendering modes was given in Section 2.4. Text that is rendered in mode three or seven becomes hidden and is removed (Section 9.5.2 [PPS01]).

HTR Case 2 - overlapped text

The appearance and position of every page element is defined by the page's content streams (see Section 2.3.1, Content Streams). The position is defined by x- and y-coordinates in user space. Beside this, the imaging model of PDF is responsible for rendering. It separates the specification and rendering process. PDF 1.4 extends this model to facilitate transparency. PDF/A-1a does not allow transparency. This means the painting model of PDF/A-1a is a simple opaque imaging model in which each graphics object is painted independently of any previous object. The order in which an element is painted is equal to the order in which the element is specified within the content stream of a page. This means, objects which are specified at the end of a content stream overlaps objects, or only parts of it, which are specified earlier within the content stream. For instance, a text element can become overlapped by an image if the image is specified after the text element within the content stream, of course under the condition that their x and y position overlap.

This property of PDF/A-1a can be used to identify overlapped text or characters. The idea is to transform the overlapping text problem into a geometrical overlapping problem by exploiting the bounding-box abstraction. Therefore, overlapped text is identified by

performing a rectangle-intersection algorithm. Figure 3.3 shows the basic principle of the overlapped text recognition; red rectangles assign hidden characters regarding to the objects painting order. The intersected rectangle problem is a frequent problem in Computer Science. The algorithms of Edelsbrunner H. et al. [Ede83b], Wood D. et al. [HWD80] and Preparata F., Shamos M. [PS93] base on the famous plane-sweep technique. In the following section, Preparata's and Shamos's plane-sweep algorithm for the intersections of iso-oriented rectangles problem will be outlined briefly.

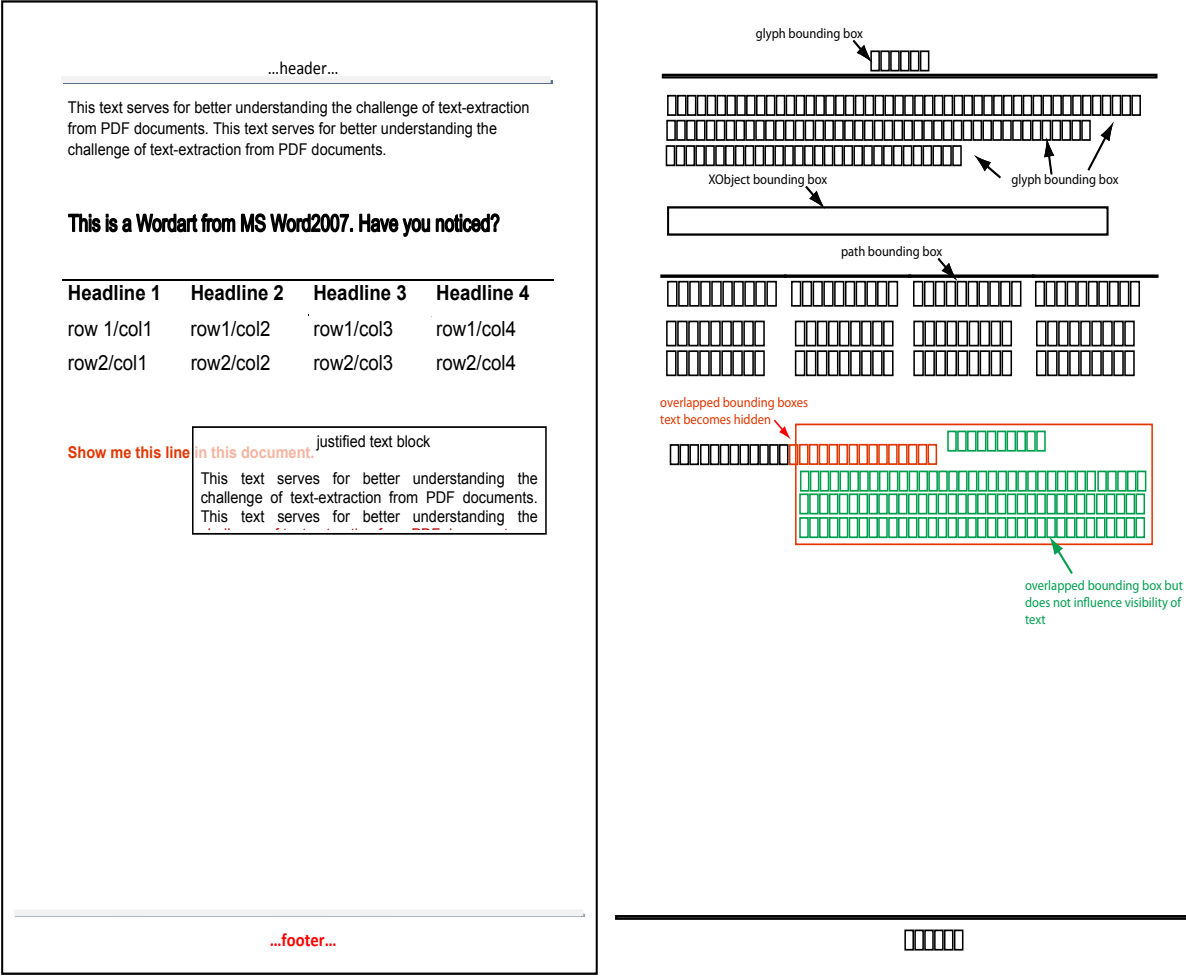


Figure 3.3: Page elements and their bounding boxes. Left: Original PDF Page; Right: Associated bounding boxes.

Plane-sweep algorithm for intersecting iso-oriented rectangles [PS93]

The traditional plane-sweep algorithm of Shamos und Hoey et al. [SH76] describes an algorithm for determining intersections of line segments. However, the core concepts can

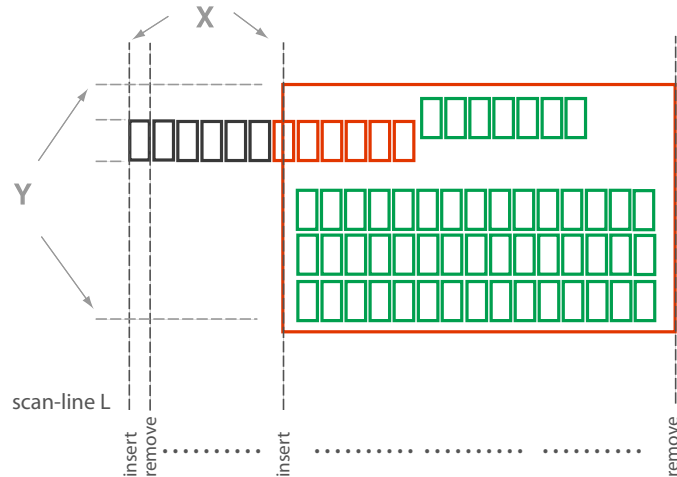


Figure 3.4: Plane-sweep algorithm for reporting intersected iso-oriented rectangles.[PS93]

be used for reporting intersected rectangles. The algorithm can be sketched up as follows: all rectangles are sorted in ascending x-order and a scan line \mathbf{L} sweeps in horizontal direction, from left to right, over the plane. If the sweep line encounters the begin of a rectangle then the rectangle is stored; otherwise if the end of a rectangle is encountered the rectangle is deleted and the corresponding overlapped rectangles are retrieved. As usual, a plane-sweep algorithm is event driven and supported by two data structures, an event schedule \mathbf{X} and a scan line status \mathbf{Y} . Event driven means that the algorithm or more precisely the scan line stops at defined events. Each event signals a change of the status of the algorithm (e.g. scan line reaches left edge of a rectangle). Following events are defined:

Event 1: left edge of rectangle.

Event 2: right edge of rectangle.

The event schedule \mathbf{X} stores all the x-values of left and right edges of rectangles, sorted in ascending x-order. The x-values must be stored in such a manner that is clear whether it refers the left or right edge of the correspond rectangle. In the course of the algorithm the scan line removes step by step the smallest x-value from the event schedule \mathbf{X} . In other words the event schedule represents all the events, or more precisely all the edges of rectangles which were not yet intersected by the scan line \mathbf{L} . If the removed value represents the left edge of a rectangle, the rectangle is inserted into the status structure \mathbf{Y} ; otherwise if the removed value represents the x-value of the right edge of a rectangle, the rectangle is removed from \mathbf{Y} . All the rectangles, which intersects the removed one, are reported.

Figure 3.4 show the principle of the plane-sweep algorithm. It shows the scan line \mathbf{L} , illustrated by vertical dashed lines, which sweeps over the plane and stops at each event in \mathbf{X} . The \mathbf{Y} list, illustrated by horizontal dashed lines, contains the y-intervals of rectangles in \mathbf{L} . In Figure 3.4, each red coloured rectangle represents a hidden text element. In fact, the plane-sweep algorithm recognizes more intersections as really necessary for hidden text recognition. So in addition, every intersected rectangle have to be checked if it represents a text object and if it is behind another element. To note, text becomes overlapped if it is rendered before another element is rendered at the same place. Therefore, the ordering of each page object must be taken into account for determining hidden text. For hidden text recognition, glyphs which are marginal overlapped only should not be recognized as hidden unless they are readable at printout. In order to recognize these false positive intersections of glyphs, a tolerance value for overlapping glyphs is crucial. This means, the plane-sweep algorithm is filled with shrinked bounding boxes, where the value for shrinking should be discovered empirically.

However, the status structure \mathbf{Y} stores the y-intervals (height) of rectangles only. That is because [PS93]: a rectangle intersects a given rectangle if the y-interval of both overlaps at the time of intersection of the scan line \mathbf{L} . However, the rectangle intersection problem is reduced to the problem of maintaining a set of intervals such that intervals can be inserted and deleted efficiently and a query asking for all intervals that intersect a given interval can be answered quickly. To achieve this, a special data structure an interval tree is required. An interval tree allows inserting and deleting in $O(\log n)$ [CLS01] time queries asking for all intervals that intersect a given interval is carried out in $O(\min(n, \log n + k))$ time [CLS01], where k is the number of intervals in the output list.

There are different ways for implementing an interval tree. Edelsbrunner H. et al. [Ede83a] proposes an 2-fold rectangle tree. Thomas H. et al. [CLS01] proposes an more intuitive way for implementing a interval tree by means of an augmented red-black tree.

Figure 3.5 shows an interval tree represented by an augmented red-black tree. The interval tree contains a set of intervals $int = [x_1, x_2]$. An interval has a low endpoint $low[int] = x_1$ and a high endpoint $high[int] = x_2$. Two intervals int_1 and int_2 overlap if $int_1 \cap int_2 \neq \emptyset$, that is, if $low[i] \leq high[int_2]$ and $low[int_2] \leq high[i]$. Each node n in the interval tree provides a interval $int[n] = [x_1, x_2]$, a key $key[n] = low[int[n]]$ represented by the low endpoint of the interval and a max value $max_{int}[n]$, which represents the maximum value of the high endpoint of all nodes rooted by the node n (see Figure 3.5). For instance, the node $n_x = [8, 9]$ has a $max_{int}[n] = 23$ which is the highest endpoint value of all children nodes. On the other side, a query of overlapped intervals for a given interval $[15, 24]$

returns the intervals $[15, 23]$, $[16, 21]$, $[17, 19]$ and $[19, 20]$.

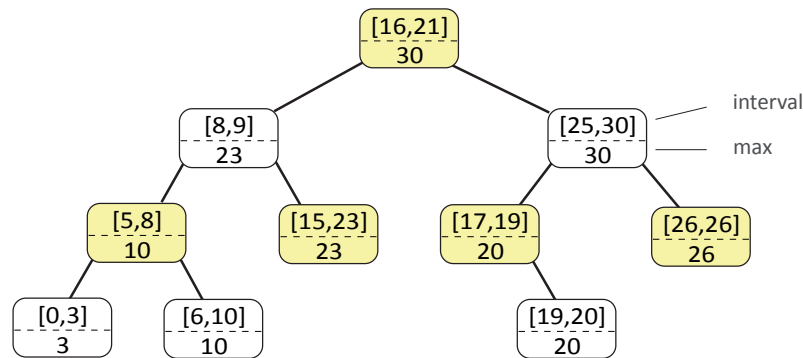


Figure 3.5: Interval tree with ten intervals sorted by the left endpoint.(source: [CLS01])

HTR Case 3 - marginal colour differences (low contrast)

The last case addresses the problem the instance that text can be hidden because of marginal colour differences between text colour and background colour.

Text can take arbitrary colours in a PDF document. In general, a colour value is given by one or more numbers depending on the colour space used. PDF supports different colour spaces and differentiates between device-dependent, device-independent (CIE-based) and special colour spaces, see Table 3.1.

device-dependent	device-independent (CIE-based)	special
DeviceGray	CalGray	Indexed
DeviceRGB	CalRGB	Pattern
DeviceCMYK	Lab	Separation
	ICCBased	DeviceN

Table 3.1: Supported colour spaces divided by same general characteristics. [PPS01]

Device colour spaces enable a page description to specify colour values that are directly related to their representation on an output device; while CIE-based colour spaces produce consistent results on different output devices, within the limitations of each device. Special colour spaces adds features or properties to an underlying colour space.

In general, CIE-based and Special colours are converted into the device's native colours by complex algorithms, typically DeviceGray, DeviceRGB, or DeviceCMYK. A PDF/A-1a document specifies the colour characteristics of the device on which it is intended to

be rendered by using a PDF/A-1a output intent, as described in Section 9.10.4 [PPS01]. The output intent describes the colour reproduction characteristics of a possible output device. The output intent dictionary includes a `DestOutputProfile` entry which defines an embedded ICC Profile stream that transforms PDF document's source colours (CIE-based and special) to output device colourants and vice versa. So PDF/A-1a does always allow a transformation into the CIE colour space, indirectly or directly. This is important because the most colour metrics for calculating the distance between two colours assume the CIE colour space. In the following, two metrics are proposed for calculating the difference between two colour values.

The International Commission on Illumination (CIE) suggest the ΔE_{ab}^* formula for calculating the difference between two colour values. More precisely, The ΔE_{ab}^* value represents the Euclidean distance between two colours in the Lab colour space. The ΔE_{ab}^* value is calculated as following [Nas98]:

$$\Delta E_{ab}^* = \sqrt{(L_2^* - L_1^*)^2 + (a_2^* - a_1^*)^2 + (b_2^* - b_1^*)^2}$$

ΔE_{ab}^*	Rating
0.0 ... 0.5	no or marginal difference
0.5 ... 1.0	difference for some persons
1.0 ... 2.0	visible difference
2.0 ... 4.0	perceived difference
above 4.0	different colour

The definition of ΔE_{ab}^* was extended in 1994 to address non-uniformities in recognition of colours by the introduction of application specific weights. The calculation is a little bit more complex but reduces the non-uniformity error significantly [Nas98]:

$$\Delta E_{94}^* = \sqrt{\left(\frac{L_2^* - L_1^*}{K_L}\right)^2 + \left(\frac{C_2^* - C_1^*}{1 + K_1 C_1^*}\right)^2 + \left(\frac{\Delta H}{1 + K_2 C_1^*}\right)^2}$$

$$C_1^* = \sqrt{a_1^2 + b_1^2}$$

$$C_2^* = \sqrt{a_2^2 + b_2^2}$$

$$\Delta H = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2 + (C_1^* - C_2^*)^2}$$

$$K_1 = \begin{cases} 0.045 & \text{graphic arts} \\ 0.048 & \text{textiles} \end{cases} \quad K_2 = \begin{cases} 0.015 & \text{graphic arts} \\ 0.014 & \text{textiles} \end{cases}$$

$$K_L = \begin{cases} 1 & \text{graphic arts} \\ 2 & \text{textiles} \end{cases}$$

Both calculations of ΔE_{ab}^* require a conversion from a colour space X into the Lab colour space. This conversion is done either indirectly through the conversion of X into the CIE-XYZ space and afterwards the conversion from CIE-XYZ space into Lab space; or directly by the associated ICC Profile defined by the output intent. The transformation of the CIE-XYZ space into the Lab space needs a reference white X_r, Y_r, Z_r in CIE-XYZ space. The reference white can be taken from the according colour space dictionary in the WhitePoint entry (see PDF Reference 4.5.4 "CIE-Based Colour Spaces" [PPS01]). The conversion is performed as follows:

$$p_r = \frac{P}{P_r} \quad \text{P is replaced by X, Y, Z}$$

$$\epsilon = \frac{216}{24389} \quad \kappa = \frac{24389}{27}$$

$$L^* = 116f_y - 16 \quad f_y = \begin{cases} \sqrt[3]{y_r} & x_r > \epsilon \\ \frac{\kappa y_r + 16}{116} & \text{else} \end{cases}$$

$$a^* = 500(f_x - f_y) \quad f_x = \begin{cases} \sqrt[3]{x_r} & x_r > \epsilon \\ \frac{\kappa x_r + 16}{116} & \text{else} \end{cases}$$

$$b^* = 200(f_y - f_z) \quad f_z = \begin{cases} \sqrt[3]{z_r} & x_r > \epsilon \\ \frac{\kappa z_r + 16}{116} & \text{else} \end{cases}$$

There many other formulas for calculating the distance between two colour values and it is beyond the scope of this thesis to evaluate each one. The defined metrics should only give a short overview of the complexity of this topic. Regardless which metric function is chosen. In order to satisfy AR2, hidden characters must be excluded from the text extraction.

The result of HTR is a PDF/A-1a document which does not contain any hidden text fragments in terms of the algorithm requirements. Figure 3.6 shows a process diagram of the HTR algorithm. The diagram gives an overview of each step needed at the HTR

phase. The order of steps is not mandatory for the result; In order to improve performance the following order is recommended. At first, all the text elements whose rendering value is equal three or seven are removed. This reduces number of page elements. Furthermore, the rectangle-intersection algorithm is used to identify the background colour of text elements and to recognize overlapped text elements. The background colour is required to calculate the contrast of a text elements and in consequence the visibility of the text. Overlapped text need not become hidden, so various if-conditions are defined which check if text gets hidden or not (see Figure 3.6).

3.5.3 Text Extraction (TE)

This section describes the second phase of TEA the Text Extraction (TE). In general, a text extraction from PDF documents can be reduced to two independent tasks. On the one hand, every character code within a page stream must be mapped to an associated character value. This procedure is called character encoding. On the other hand, PDF stores text in an arbitrary order which does not need to be equal to document's logical structure. Therefore, the extracted text must be ordered to preserve the logical and semantic structure which is, in general, the natural reading direction from top left to bottom right in the western hemisphere.

The first problem is solved by requiring the PDF/A-1a "profile" for PDF documents, as claimed in AR1. As aforementioned in Section 3.2, a PDF/A-1a document ensures that each used font is embedded. This implies the font program, glyph metrics and the character encoding so that the font is embedded for unlimited, universal rendering. Regarding the character encoding, each font directory includes a **ToUnicode** entry, whose value is a CMap file which maps every character code into a 2-byte Unicode value (see Section 2.4.2 "ToUnicode CMaps"). So every character can be mapped to its Unicode representation.

The proper ordering of extracted text can be achieved in different ways. As aforementioned, PDF/A-1a provides information about the document's logical structure by introducing a structure tree. Every node of the tree references to an element on the page, where the logical order of an element is defined by the position of the associated node in the tree within a deep first search. However, as explained in Section 3.3, the structure tree can become ambiguous and is a security vulnerability in terms of electronic signatures. For instance, if a text extraction is carried out by using the structure tree, an attacker can manipulate the tree to adapt the text order of the PDF document without recognizing this at text extraction. Hence, the text of PDF document does not reflect the text order

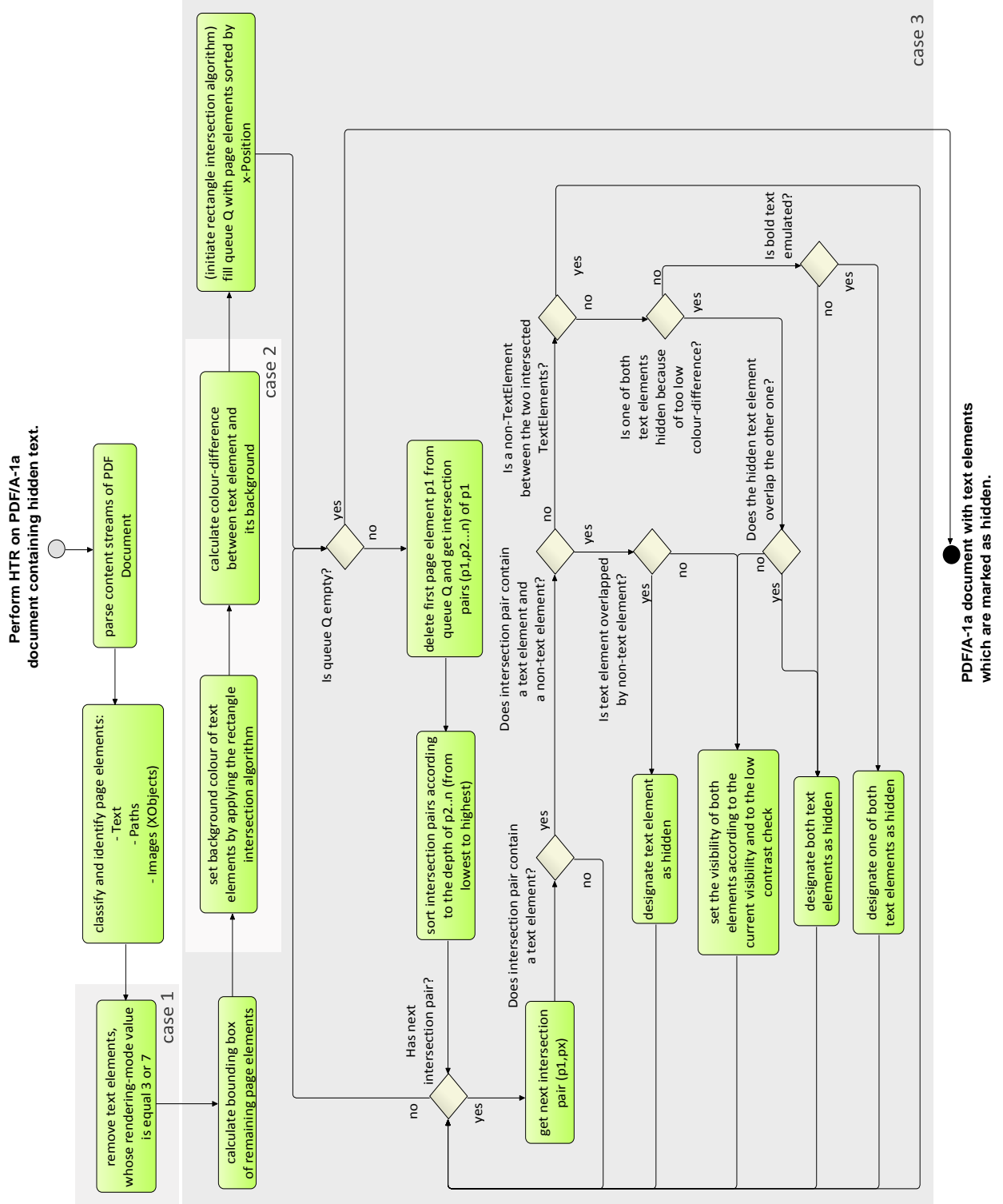


Figure 3.6: Process diagram of the first phase of TEA the hidden text recognition (HTR).

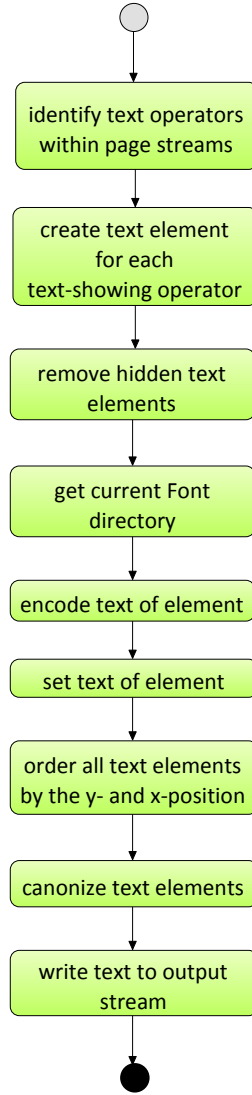
at printout any more, which is required by AR3. Due to the lack of technical protection mechanisms for PDF/A-1a, the structure tree can not be utilized for ordering.

Another approach to extract text in an consistent way is to use the graphical text position on a page. This method satisfies the AR1-AR4 algorithm requirements and it is resistant against attacks on PDF source-code, as modifying the position of an element results in changing the printout as well. The major disadvantage here is that all the semantic layout information such as from tables or multi-column documents are lost because they are not reproducible by means of position information. Nevertheless, this approach satisfies the requirements of Official Signatures and ordering by position is a straightforward approach. Every text element is ordered firstly by its y-position and secondly by its x-position. In addition, it is necessary to canonise the extracted text stream because text in PDF/A-1a does not contain any line breaking characters or space characters between paragraphs. In order to improve the readability of extracted text and to establish a standard formatting, line breaks and spaces are added or removed, depending on the following rules:

- A new line begins if a character is inserted with a new y-position. This results into a variation in representation between input document and canonized text but preserves the semantic meaning of it.
- A space character is inserted if the space between two characters is greater than the width of the average space character of the current font. If the font of the two characters is different, the average of space width is taken as threshold. Multiple space characters in a row are not allowed and reduced to one.

The canonisation improves readability of the text extraction. Furthermore, canonisation should make a later reconstruction easier. Figure 3.7 illustrates all steps of the text extraction phase in detail.

**PDF/A-1a document with marked
hidden text content after HTR**



**visible text content of
PDF/A-1a document.**

Figure 3.7: Process diagram of text extraction (TE).

Chapter 4

Implementation

This chapter deals with the implementation details of the Text Extraction Algorithm Module (TEA-M). Section 4.1 gives a schematic overview of the PDF-AS Application in which TEA-M is integrated. The Section 4.2 points out the internal architecture of TEA-M. Finally, Section 4.3 discusses the main components of TEA-M in detail and their relationship to each other.

4.1 System Overview of PDF-AS

The E-Government and Innovationszentrum (EGIZ) was established in 2005 as a joint initiative of the Federal Chancellery Austria and IAIK at Graz University of Technology. Since then EGIZ has supported the Chancellery in development and technical research in the field of eGovernment. EGIZ has conducted several projects in field of eGovernment, such as the Online Application Modules (MOA) architecture and the PDF-Amtssignatur (PDF-AS) application.

The PDF-AS application has been designed to attach an Official Signature on PDF documents. It affixes an electronic signature as well as an Official Signature on a PDF document and verifies an officially signed document. PDF-AS supports two modes of operation:

- binary based PDF signature
- text based PDF signature

In the first mode, PDF-AS signs the entire PDF file, or parts of it, at byte level. This comprises all the elements of the document, such as pictures, shapes and text. This method has the disadvantage that the Official Signature is not reproducible from the document's printout, which is a requirement as stated before. On the other hand, a text signature

enables the reproduction of the signature from the document’s printout.

At the time of working on this thesis, the PDF-AS application simply uses the open source library PDFBox from Apache [APA09] for text extraction. PDFBox provides an object based access to PDF documents for the Java programming language. PDF-AS is designed on a two-level architecture where the first level provides low-level access on PDF and the second level builds up complex objects. Furthermore, the library provides a basic structure for text extraction that avoids the development from the sketch. This makes PDFBox the first choice as the basis of TEA-M.

Returning to PDF-AS, Figure 4.1 shows a greatly simplified configuration of the entire PDF-AS system. PDF-AS uses a multitude of various open source libraries. The figure shows very good the provided interfaces of TEA-M to the other PDF-AS components.

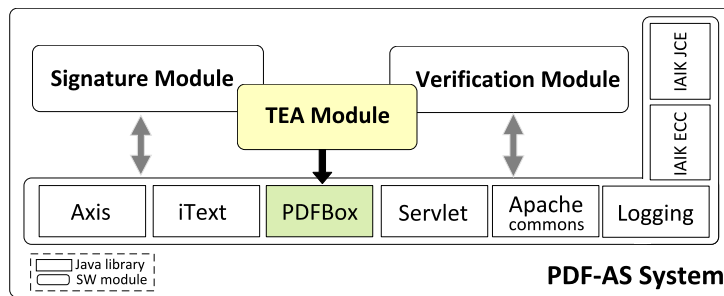


Figure 4.1: Schematic diagram of the PDF-AS application and its components.

TEA-M is a central part of the entire PDF-AS system and is primarily used by the signing and verification module. The major modules are:

TEA module is responsible for text extraction from PDF documents. Furthermore, the module provides an interface for other PDF-AS modules. The PDFBox library provides a low level access on PDF documents for TEA-M.

Signature module is responsible for creating an Official Signature.

Verification module is responsible for verifying an Official Signature.

Due to the PDF-AS architecture, the following implementation requirements [IR] have been defined in the scope of this thesis:

- [IR1] TEA-M shall be implemented as Java library.
- [IR2] TEA-M shall provide a standardized interface for communication with other PDF-AS components.

- [IR3] TEA-M shall provide a modular design to support the implementation and integration of arbitrary text extraction algorithms.

Furthermore, TEA-M requires Java 6.0 and the following Apache libraries: PDFBox 0.8 (core library), JempBox 0.8 (for working with XMP) and FontBox 0.8 (for accessing to PDF fonts) [APA09].

4.2 Architecture overview

The following sections outline the software architecture and summarize important design decisions.

4.2.1 Pipes and Filters

As introduced in Section 3.5, TEA follows a two phase concept for extracting text content of PDF/A-1a documents which is composed of a:

- Hidden Text Recognition (HTR) Phase and
- Text Extraction (TE) Phase

The core concept of TEA-M implementation is that the output of each phase is the input of the next one, which makes it well suited for the Pipes and Filters architecture pattern from [BMR⁺96]. The pattern will be described in the following paragraph.

"The Pipes and Filters architecture pattern provides a structure for systems that process a stream of data in sequential steps. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters in the Pipes and Filters pattern allows to build family of related systems." [BMR⁺96, S. 54ff.]

The core idea of the Pipes and Filters pattern is to divide large processing steps into smaller ones which are easier to reuse and to handle. This makes extensions and changes manageable in development. Furthermore, the pattern is notably, scaleable and adjustable. Thus, the pattern is well suited for parallel programming with only slight modifications to the pattern, as suggested by Ortega et al. [OA05] or Mattson et al. [MSM04]. Concurrency has a huge impact on the performance of large data sets and may reduce computation time dramatically. Mattson et al. describes two patterns which are very similar to Pipe and Filter pattern with the key difference of considering concurrency: the Pipeline and the Task Parallelism Pattern. Figure 4.2 illustrates the concept of the Pipeline pattern [MSM04] which can be phrased out as follows: a Pipeline is defined by

pipeline stages. At each stage a sequence of computations are performed concurrently by a sequence of filter components. The result of each filter is pushed to its neighbour filter. The pipeline starts again and each filter is applied on the new data set.

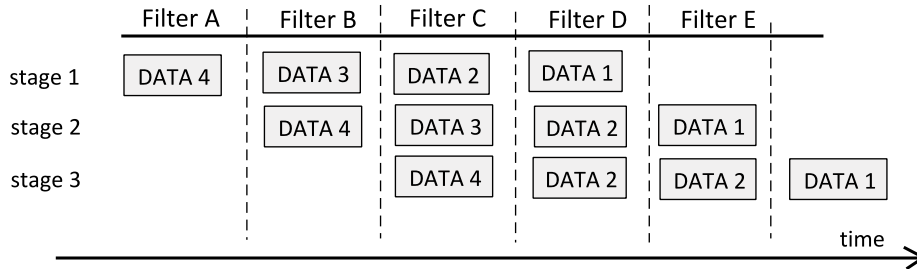


Figure 4.2: Pipeline Pattern of Mattson T. et al. [MSM04].

However, Figure 4.3 shows the conceptional TEA-M architecture by applying the Pipes and Filters pattern. Every pipe element performs a specific task on a `IPDFModel` object, an abstraction of a PDF page. The order of execution is formed by the pipeline. Each stage in the pipeline has privileged read and write access on a `IPDFModel` object for the time it is executed. The following paragraphs explain major components from Figure 4.3:

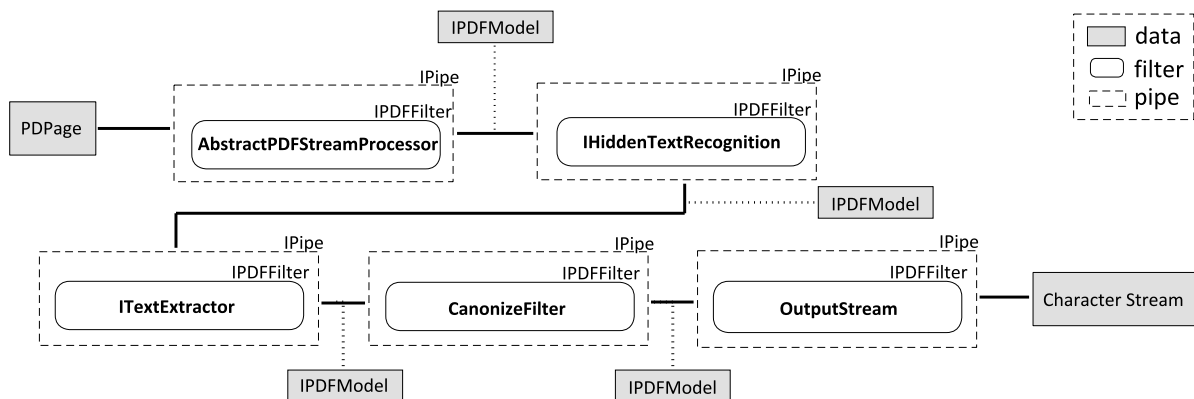


Figure 4.3: Simplified TEA-M architecture concept.

PDPPage

This is the in-memory representation of a PDF document page from PDFBox.

IPDFModel

The `IPDFModel` represents the in-memory data on which a filter is performed. It keeps information about a PDF page and a collection of `IPageElement` elements. TEA deals

with PDF page elements in order to determine hidden text. As introduced in the previous section, a PDF Page can be considered as a composition of Text, Path and Image elements. Each element of this kind is represented by the `IPageElement` interface in TEA-M. It keeps information about position, size and textual content of an associated page element. More precisely, an `IPageElement` object is given by its x- and y-coordinates of the upper left corner and its width and height value in user space. In addition, the coordinates of an `IPageElement` are invariant regarding page rotation. It has the advantage that an `IPageElement` is always defined by its upper left corner, unless page rotation.

IPDFFilter

An `IPDFFilter` component takes input data (e.g. `IPDFModel` object) from a pipe, performs a sequence of operations on the input data and sends output result data (e.g. `IPDFModel` object) to the pipe again. TEA-M defines a sequence of different `IPDFFilter` components, most of them implement algorithms and phases which were defined in section 3.5:

- **AbstractPDFStreamProcessor:** This filter will run through a PDF page content stream and executes certain operations to build a `IPDFModel` object.
- **IHiddenTextRecognition:** It provides the HTR algorithm which recognizes and designate hidden `IPageElement` objects from the input `IPDFModel`.
- **ITextExtractor:** This filter encapsulates the TE algorithm. It removes hidden and non text elements from the `IPDFModel`.
- **CanonizeFilter:** This filter encapsulates the canonization procedure. It formats the `IPDFModel` into human-readable style by adding or removing line-breaks and space-characters.
- **OutputStream:** This filter writes the resulting extracted character stream to a `java.util.OutputStream`.

IPipe

The `IPipe` component transfers data between filters and synchronises activities between neighbouring `IPDFFilter` objects. TEA-M provides three different implementation of `IPipe` objects namely the `FilterPipe`, the `LinearPipe` and the `ParallelPipe`. The `FilterPipe` servers as wrapper only which encapsulates a `IPDFFilter` object. However, the `LinearPipe` and the `ParallelPipe` enables the parallel execution of a set of `IPipe` objects. So, the `LinearPipe` object combines single `FilterPipe` objects to a line of pipes,

where each stage of the pipe is executed concurrently. The following sections describe the simultaneous execution of filters in detail.

4.2.2 System Architecture

Figure 4.4 presents the TEA-M architecture. The PDFBox and PDF-AS infrastructure is not shown for reasons of clarity. PDF-AS implements and handles the calling of the text extraction pipeline, as illustrated in Figure 4.3.

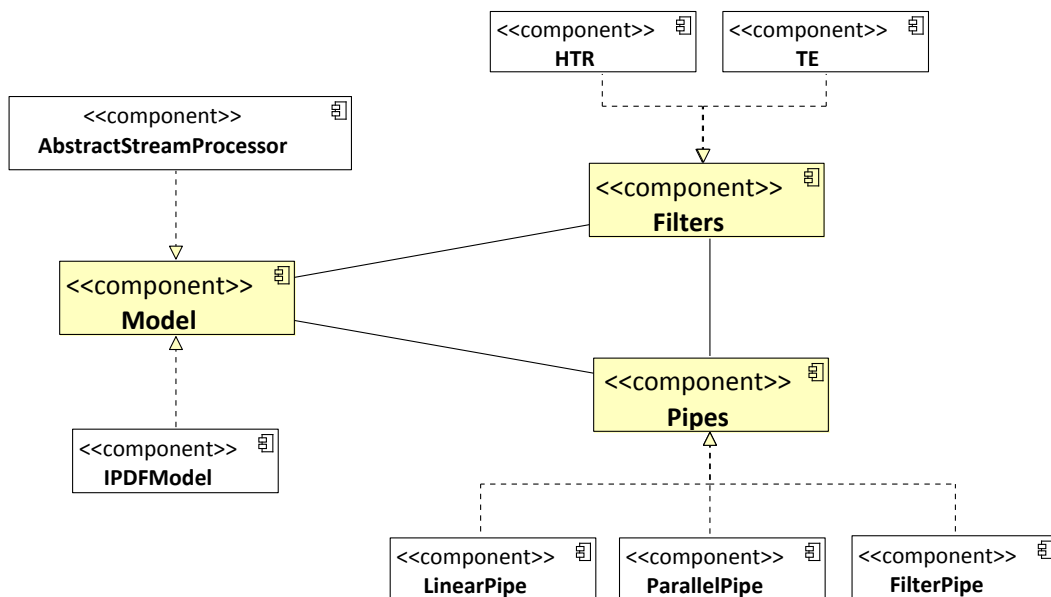


Figure 4.4: Block diagram illustrates system architecture of TEA-M.

The architecture, as shown in Figure 4.4, introduces some new parts of the pipeline mechanism. Pipes are used to encapsulate filters and to synchronize communication between other pipes. Pipes are connected to pipelines that are executed consecutively. The traditional Pipes and Filters pattern from [BMR⁺96] does not support multithreading from the sketch. TEA-M implements a slightly modified version of Pipe and Filters with multithreading support. The benefit of multithreading is higher performance due do simultaneous execution of all pipes in a pipeline. The parallel programming requires the partitioning of computation into smaller parts. This requires that computations are independent from each other. That means a filter can be carried out until completion without interference.[OA05]

TEA-M provides two methods for establishing a pipeline: either by connecting a pipe to another one by the IPipe interface or by using of a managed pipe such as LinearPipe

or `ParallelPipe`. The latter supports the simultaneous execution of a sequence of pipes. While the result of `LinearPipe` is formed by the last registered pipe, the result of `ParallelPipe` is formed by the union of each single pipe item.

The advantages of this architecture are the exploiting of concurrency, a easy maintenance and increased performance compared to traditional pipes and filters architecture [BMR⁺96]. The disadvantages of this architecture are the additional effort in development compared to a strategy pattern or to a traditional pipes and filters pattern due to the synchronisation of pipes. This implies the risk of deadlocks. Furthermore, objects for communication between pipes must be built up to a common denominator, which may limit a decoupling of objects.

4.3 Technical Overview

The technical design encompasses the package and class structure of TEA-M, as well as the interaction between the different components of TEA-M

The following list summarizes the package structure of TEA-M and briefly describes what each package does. The package structure relies basically on the system architecture discussed before.

- `at.gv.egiz.tea.model`: Abstract PDF page element data model; used by `IPDF-Filter` and `IPipe`
- `at.gv.egiz.tea.pipes`: Provides different pipe implementations of the Pipe and Filter framework.
- `at.gv.egiz.tea.filters`: Provides various filter implementations of the Pipe and filter framework.
- `at.gv.egiz.tea.filters.htr`: A HTR implementation of TEA .
- `at.gv.egiz.tea.filters.te`: A TE implementation of TEA.
- `at.gv.egiz.tea.util`: The util package compasses everything that is not tied to the core algorithms of TEA. Beside standard data manipulation functions this also includes logging functionality.

The following sections give an detailed overview of each package provided by TEA-M

4.3.1 Model-Package

Figure 4.5 depicts the key classes in parsing a PDF/A-1a document and constructing an `IPDFModel` object from it. Classes from `PDFBox` are omitted for clarity. The central classes or interfaces are the `IPDFModel`, `AbstractStreamProcessor` and `OperatorProcessor`.

An `IPDFModel` object keeps a collection of `IPageElements` - there are three different implementations: `TextElement`, `PathElement` and `ImageElement`. Each one reflects the bounding box of a PDF page element and its content. In fact, `IPageElement` is a `IRectangle2D` that is used from `IPlaneSweepAlgorithm` in the `Filter` package.

The `IPageElement` or `IPDFModel` object is constructed by an `AbstractStreamProcessor` instance. It is the interface that is used by other packages in the system, neatly implementing a bridge pattern of abstraction. The inner workings of the model package can be

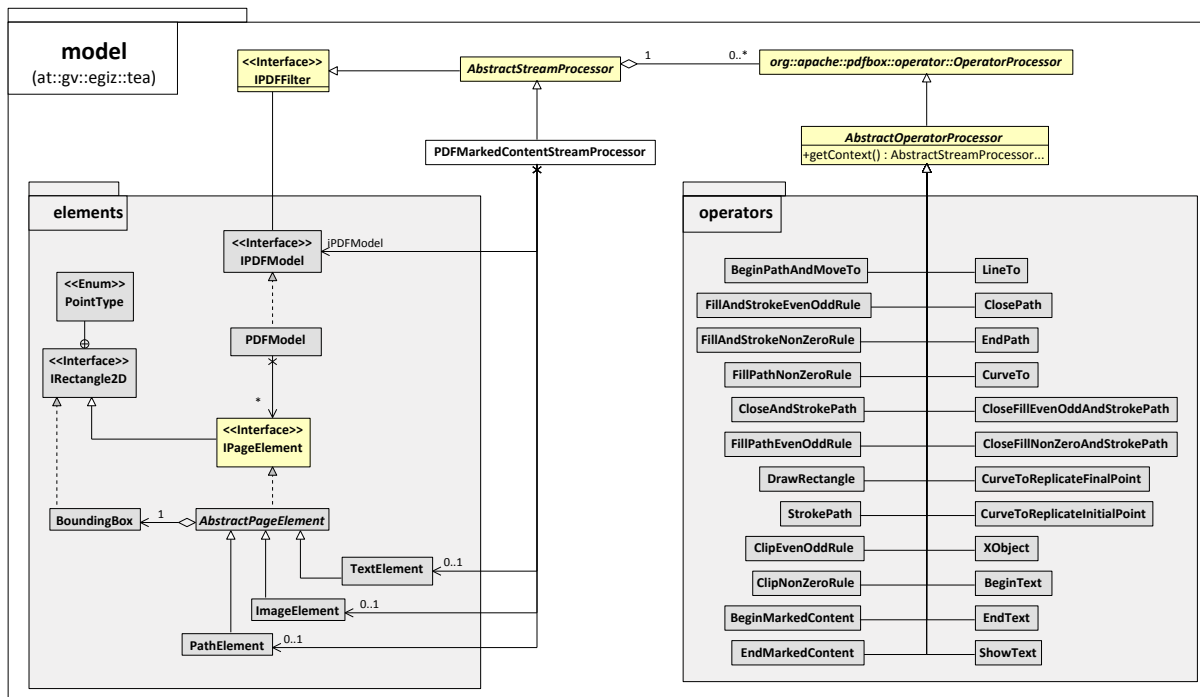


Figure 4.5: Simplified UML class diagram of the model package.

explained by looking at the specific case of building a `PDFModel`. The `AbstractStreamProcessor` in Figure 4.5 receives a request from the `IPDFFilter` interface that invokes a specific `org.apache.pdfbox.PDDocument` and `org.apache.pdfbox.PDPage` object. The processor parses the content stream of the `PDPage` and invokes an associated `OperatorProcessor` for each operator. TEA-M implements a sequence of `OperatorProcessors`

for specific content stream operations, as illustrated in Figure 4.5, in addition to those provided by PDFBox. When an `OperatorProcessor` is invoked it performs various operations on the caller `AbstractStreamProcessor` to build and to modify `IPageElements` applying a builder pattern.

The use of builder pattern enables easy programming and a clean abstraction of functionality. Elements can be modified through a shared context and must not be managed by each `OperatorProcessor`. Furthermore, it decouples the `IPDFModel` from `OperatorProcessors`.

4.3.2 Pipes-Package

Figure 4.8 shows how the Pipe and Filter framework is implemented and how it is associated with the rest of the system. The core elements in Figure 4.8 are `IPDFFilter` and `IPipe`, which provide an interface for other packages. The former will be discussed later.

The `IPage` interface inherits `java.util.concurrent.Callable`. It allows a concurrent execution of implemented classes by threads and returns a result if it is completed. As noted, TEA-M implements various pipes for different purpose. The differences between `FilterPipe`, `LinearPipe` and `ParallelPipe` are in the accomplishment of an associated `IPDFFilter`:

- `FilterPipe` represents the simplest pipe. It encapsulates an `IPDFFilter`, executes it on request of a neighbour pipe and if the associated filter is finished the pipe pushes the filter's result to the next pipe and invokes its execution.
- `LinearPipe` is itself a pipe and takes a series of pipes which forms a pipeline; In the case of execution, it carries out each pipe concurrently and pushes the result from first to second pipe, second to third pipe etc. and lastly to the `LinearPipe`'s output.
- `ParallelPipe` is similar to `LinearPipe`. It differs regarding the movement of execution as each pipe is executed concurrently but the result of each one is merged to one result and forwarded to the `ParallelPipe`'s output.

Figure 4.6 illustrates a linear pipe with three filters which is encapsulated in a `IPipe` object. All pipes are carried out parallel, where the result of `PipeA` is the input of `PipeB` and so on. The result of a linear pipe is the result of of the last pipe item (`PipeC`). In contrast, in figure 4.7, the result of all pipes are merged and forwarded to output which

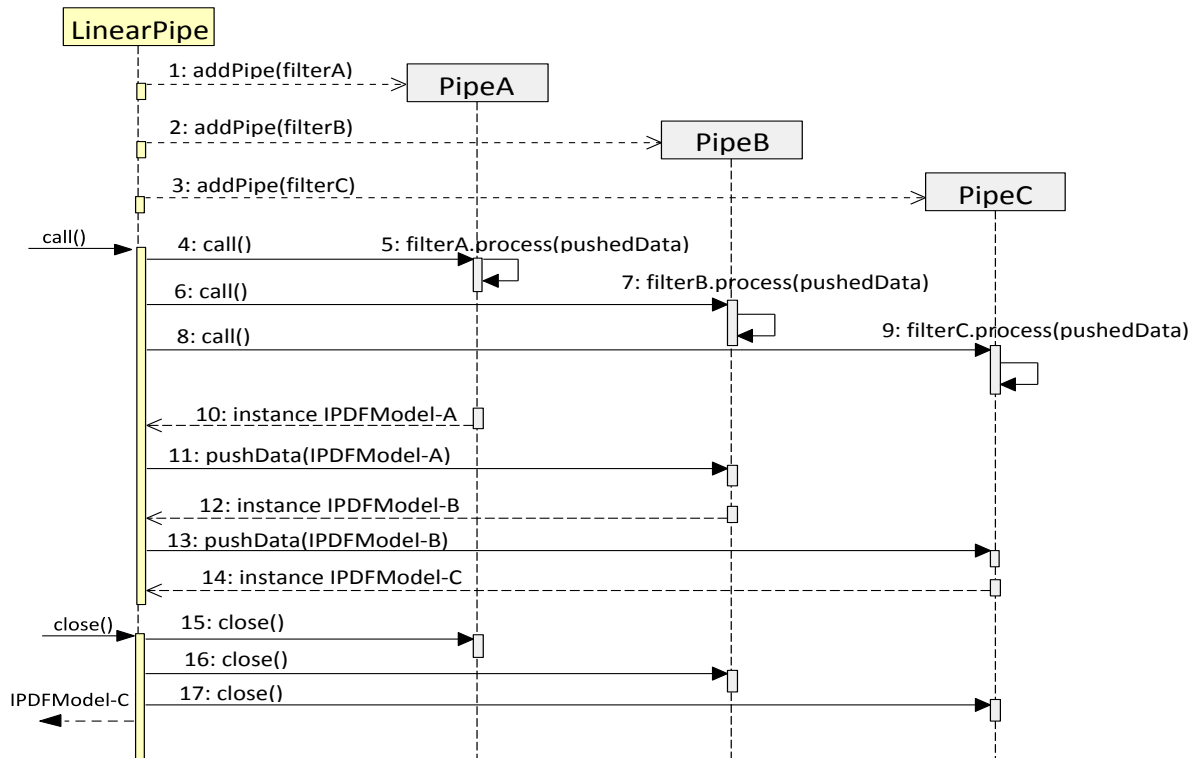


Figure 4.6: Simplified sequence diagram of *LinearPipe* in TEA-M.

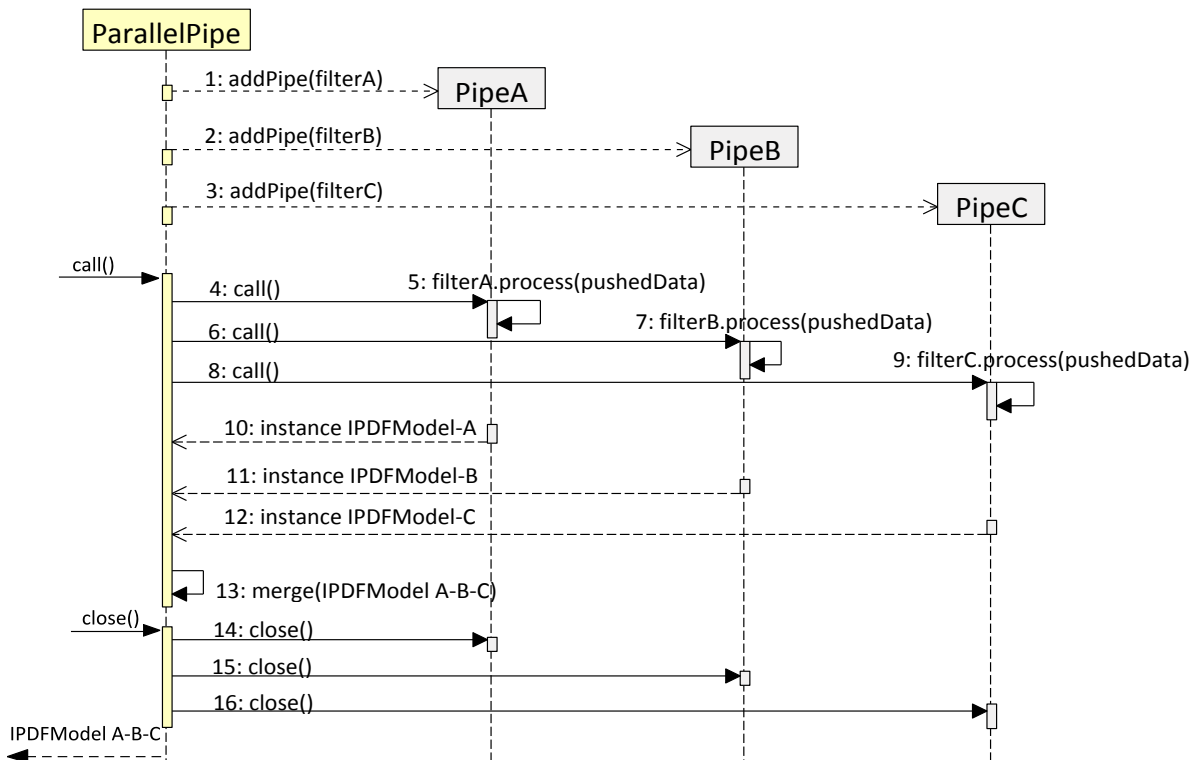


Figure 4.7: Simplified sequence diagram of *ParallelPipe* in TEA-M.

represents the behaviour of `ParallelPipe`. Finally, every pipe has to be closed to push the last input of the first pipe to the last pipe.

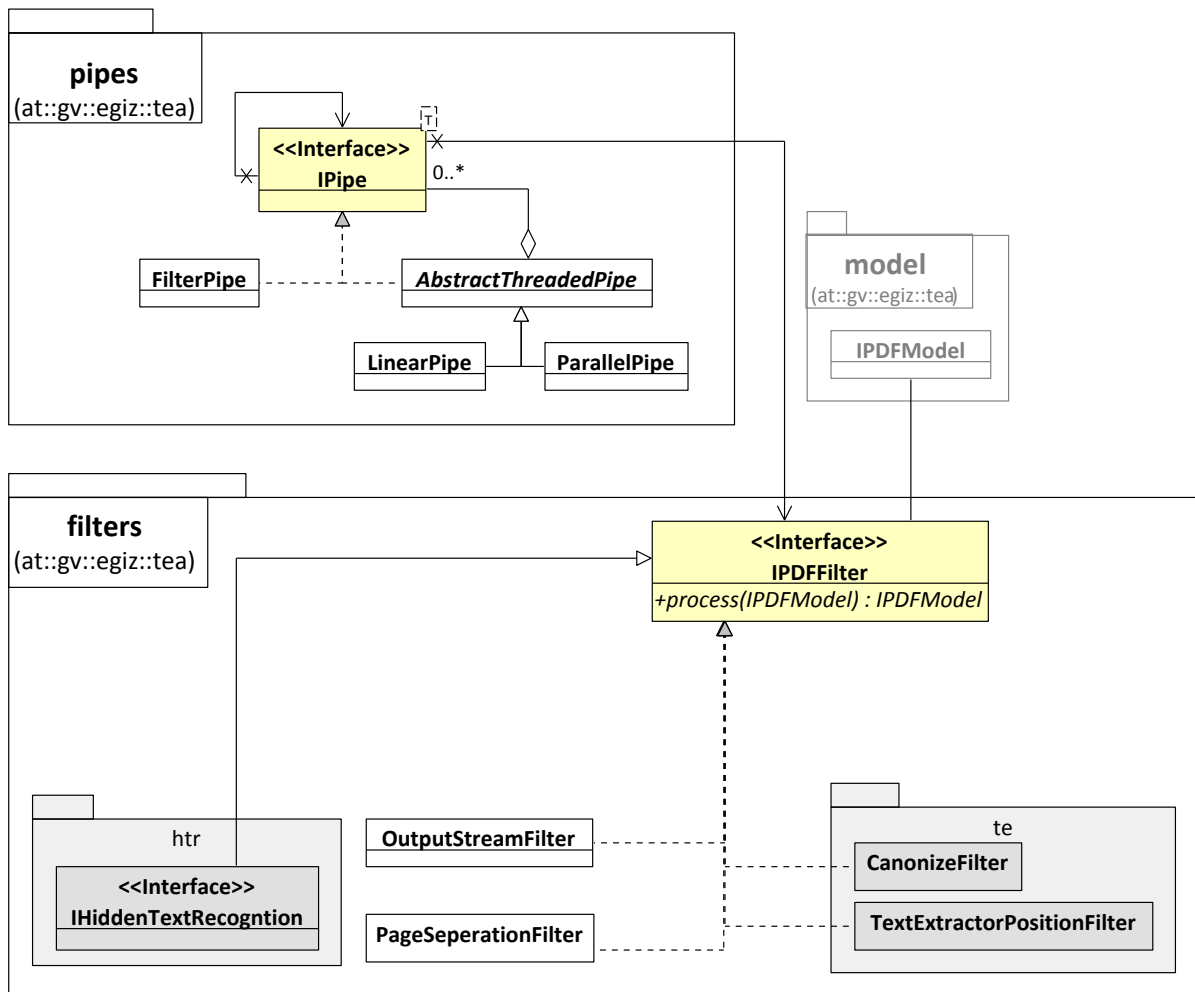


Figure 4.8: Simplified UML class diagram of the pipe and filter packages.

4.3.3 Filters-Package

The `IPDFFilter` interface must be implemented by any object that is used in a pipeline. In general, a filter encapsulates a specific algorithm which transforms a `IPDFModel` object. TEA-M applies a sequence of different filters for text extraction on PDF/A-1a documents. The most important will be discussed in the following sections.

Hidden Text Recognition (HTR)

The function of this filter is described in detail in chapter 3. In this section we only give some technical implementation notes. Figure 4.9 shows the class diagram of the

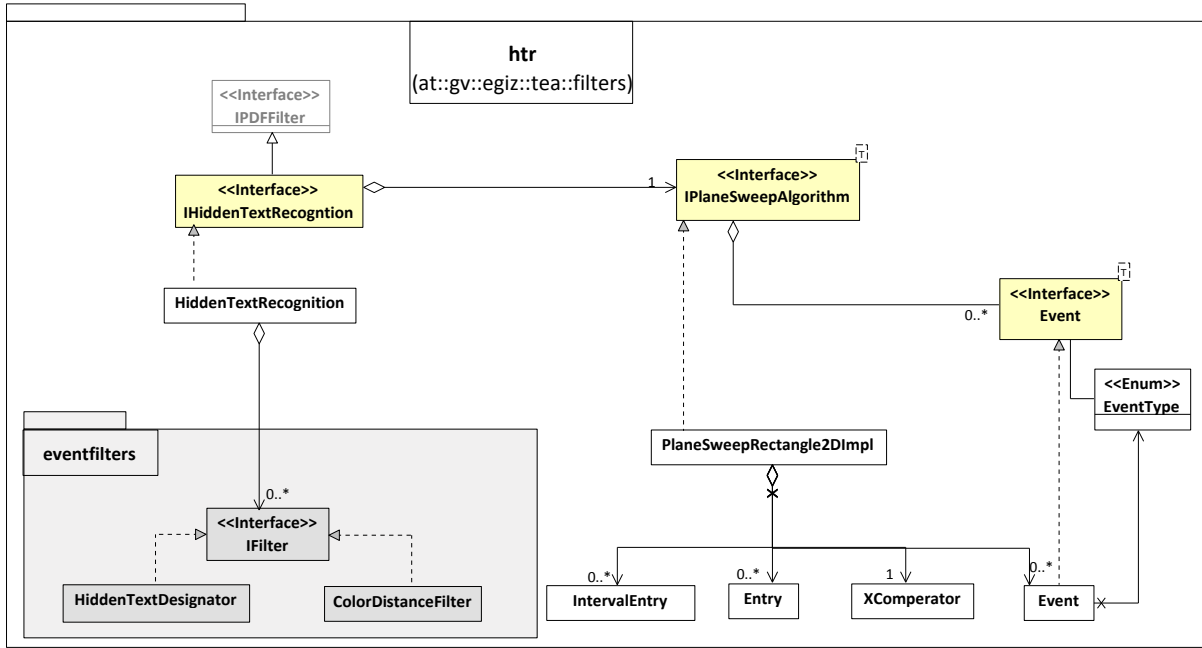


Figure 4.9: Simplified UML class diagram of the htr package.

htr-package. The focus is on the `IHiddenTextRecognition` and `IPlaneSweepAlgorithm` interfaces. The former provides an interface for other packages by applying the bridge pattern. The latter class provides the "business logic" of the `HiddenTextRecognition` which builds the core part of HTR. The `PlaneSweepRectangle2DImpl` class realises a rectangle intersection algorithm according to Shamos and Hoey [SH76] plane-sweep technique. The algorithm was described in detail in section 3.5.

Text Extraction (TE)

Figure 4.10 shows the class diagram of the TE implementation, which tightly follows the algorithm's specification in section 3.5.3. In general, an `IPDFModel` object keeps an un-ordered set of `IPageElements`, whereas each one represents a part of a PDF page. In the case of text-extraction, only `TextElements` are of interest. The challenge is the reconstruction of the reading order of the origin document. Therefore, TEA-M provides a TE implementation with the `TextExtractorPositionFilter` class. It orders all the text elements in an `IPDFModel` object by means of its y- and x- position on the page. The advantage of this approach is that it is always applicable, independent of the PDF document and it delivers a unique result. The disadvantage is that the reconstruction is not lossless; that is, in some cases the approach can not reconstruct the visual and logical structures of a document such as of tables. For example, a multi-line table without borders does not differ from a normal text block regarding the text position. On the other hand a table

can be read column by column and not from top left to bottom right, which is recognized from a reader easily but not from the algorithm. The `TextExtractorPositionFilter` is designed as an `IPDFFilter` to be use as a pipe element in a pipeline.

Furthermore, this package provides a canonization implementation, as defined in section 3.5.3, by the `CanonizeFilter` class. It is used to canonize an `IPDFModel` object, where only visible text objects are recognized. It implements the `IPDFFilter` interface to utilize it as a pipe item. The canonization process is implemented as follows: a map data structure represents a PDF page, where a key represents the y-position of a text-line and the associated value to key keeps the array of text elements at this line. If an element is inserted, the map is checked for a present key. If the key is not present a new line is introduced and the element is added. Otherwise only the element is added.

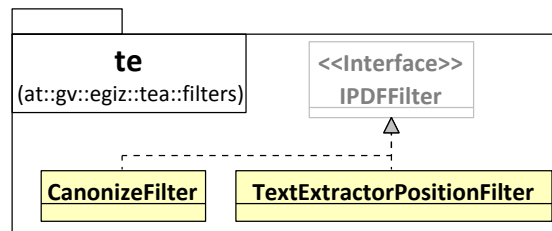


Figure 4.10: Simplified UML class diagram of the *te*-package.

4.4 Testing

In this section we will evaluate TEA-M, if it satisfies the requirements as defined in section 3.5. Therefore, the test procedure and test data set are introduced. Furthermore, at the end of this section the test results are presented and briefly discussed.

4.4.1 Testing Procedure

Unit testing is a popular verification and validation method in software development for testing parts of a implementation as a whole. In order to validate TEA-M against the algorithm specification a sequence of unit tests have been defined and executed on TEA-M. The focus was put on testing major features of TEA, such as the correct text extraction according to the reading direction and the hidden text recognition.

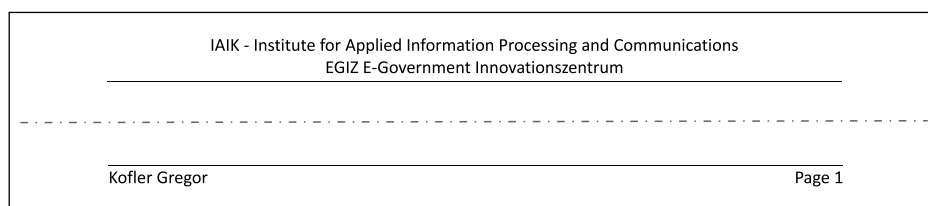
The input of every unit test was generated in OpenOffice Writer and exported as a PDF/A file. Furthermore, the text content of every PDF/A document was being extracted by TEA-M. If the extracted text matched with the expected one, the test was evaluated as

successful otherwise as failed. A detailed overview of all test results gives Table 4.22. In the following sections, every test is described briefly. This involves a list of test files in which the respective test is applied and the expected result of the test.

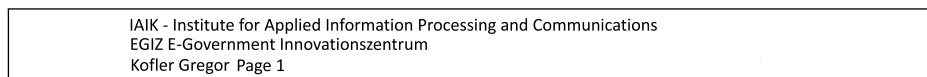
T1: Header and Footer Test

Figure 4.11(b) gives the test case for testing header and footer extraction. The simple PDFBox based extraction algorithm of PDF-AS had problems in recognizing headers and footers of multi page PDF documents correctly, where TEA solves these problems.

Test files: PDF/A 1, PDF/A 10, PDF/A 11



(a) header and footer test



(b) expected result

Figure 4.11: Figure shows the header and footer test and the expected result.

T2: Body Text Test

The body text test tests the text extraction from the body of a PDF document with respect to the reading direction. Figure 4.12 shows the test and the expected result.

Test files: PDF/A 2, PDF/A 10, PDF/A 11

T3: Text Block Test

Figure 4.13 illustrates a simple text-block provided by various word processing applications, such as Microsoft Word and OpenOffice Writer. This test tests the text extraction from a text-block.

Test files: PDF/A 3, PDF/A 10, PDF/A 11

This text serves for better understanding the challenge of text-extraction from PDF documents. This text serves for better understanding the challenge of text-extraction from PDF documents.

(a) header and footer test

This text serves for better understanding the challenge of text-extraction from PDF documents. This text serves for better understanding the challenge of text-extraction from PDF documents.

(b) expected result

Figure 4.12: Figure shows the body text test and the expected result.

----- justified text block -----
This text serves for better understanding the challenge of text-extraction from PDF documents.

(a) header and footer test

----- justified text block -----
This text servers for better understanding the challenge of text-extraction from PDF documents.

(b) expected result

Figure 4.13: Figure shows the text block test and the expected result.

T4: Table Test

Tables provide additional information concerning the document's reading direction. A reader easily knows the reading direction of a table which is not always from top left to bottom right. In particular, tables are often used for styling purposes. At printout, only the position of text is recoverable but not styling and structure information. Figure 4.14 shows a table with full borders and Figure 4.15 shows a table without borders. At printout, a distinction between regular styled text and text in tables is infeasible by means of the text position. For instance, the table in Figure 4.15 can be read column per column by a person instead of row by row such as TEA.

Test files: PDF/A 4, PDF/A 10, PDF/A 11

Headline 1	Headline 2	Headline 3	Headline 4
row1/col1	row1/col2	row1/col3	row1/col4
row2/col1	row2/col2	row2/col3	row2/col4

(a) table with full borders

```

Headline 1 Headline 2 Headline 3 Headline 4
row1/col1 row1/col2 row1/col3 row1/col4
row2/col1 row2/col2 row2/col3 row2/col4

```

(b) expected result

Figure 4.14: Figure shows the table test with borders and the expected result.

Headline 1	Headline 2
How tables are recognized at printout?	Tables provide additional information regarding the reading direction.
How tables are recognized at printout?	Tables provide additional information regarding the reading direction.

(a) table without borders

```

Headline 1 Headline 2
Tables provide additional information
How tables are recognized at
printout?
regarding the reading direction
Tables provide additional information
How tables are recognized at regarding the reading direction.
printout?

```

(b) expected result

Figure 4.15: Figure shows the table test without borders and the expected result.

T5: Listing Test

Listings and enumerations are used frequently in text document. As usual, text processing applications allow the adjustment of listing symbol before the text. In OpenOffice Writer, the OpenSymbol font is used for listing symbols. During work on this thesis some problems with the OpenSymbol font regarding the character encoding have been encountered. For instance, in Figure 4.16 the character "x" is encoded as "?", which makes a text extraction infeasible.

Test file: PDF/A 12

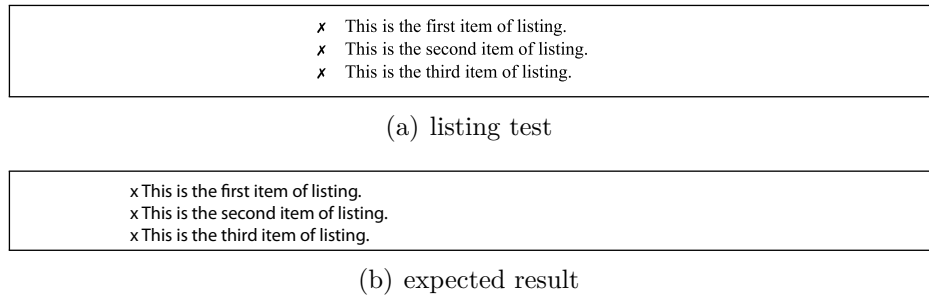


Figure 4.16: Figure shows the listing test and the expected result.

T6: Text-in-Picture Test

At the time of verification of a electronic signed printout, a major problem is to distinct between text and text which is embedded in a picture. This circumstance is tested with the following test case, given in Figure 4.17. TEA does not extract text from pictures, so no extracted text is expected.

Test file: PDF/A 13

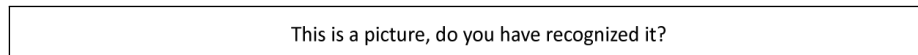


Figure 4.17: Figure shows the text-in-picture test only because the expected result is empty.

T7: Hidden Text - Rendering Mode Test

Text, that is rendered in mode three or seven, is hidden at printout and on screen. Hence, a visible representation of this test case cannot be generated.

Test files: PDF/A 5, PDF/A 10, PDF/A 11

T8: Hidden Text - Marginal Color Difference Test

The recognition of hidden text is a major requirement of TEA. Hidden text is divided into three types whenever the marginal color difference between text and its background is one of those types. In this example text becomes hidden because of low contrast and cannot be reconstructed at printout. Figure 4.18 illustrates the according test case with

the expected result.

Test files: PDF/A 9, PDF/A 10, PDF/A 11

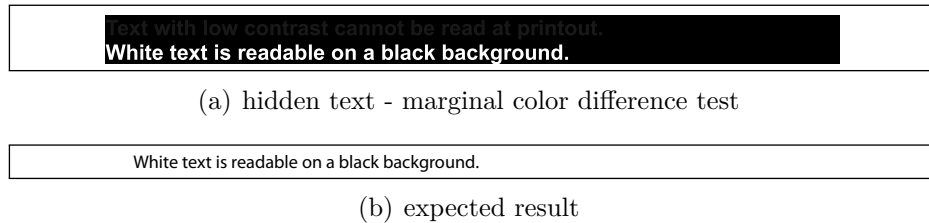


Figure 4.18: Figure shows the marginal color difference test and the expected result.

T9: Hidden Text - Overlapped Text Test

There are three different tests for testing the recognition of hidden text. Crossed out text is given in Figure 4.19. Crossed out text is removed from text extraction. Figure 4.20 and 4.21 test the recognition of hidden text which is overlapped by a text-block and a picture. Only the visible text should be extracted.

Test files: PDF/A 6, PDF/A 7, PDF/A 8, PDF/A 10, PDF/A 11

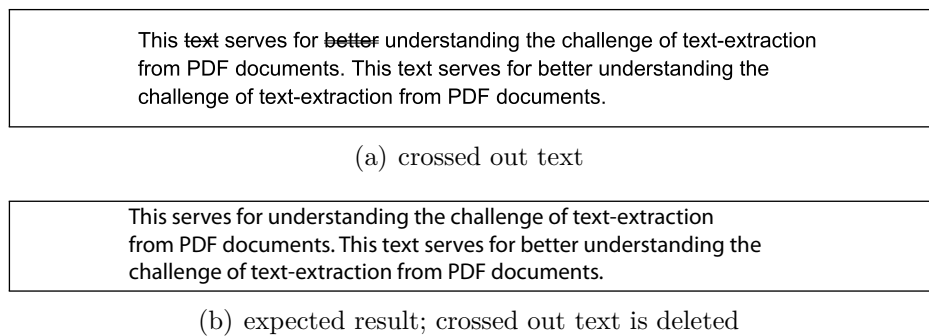
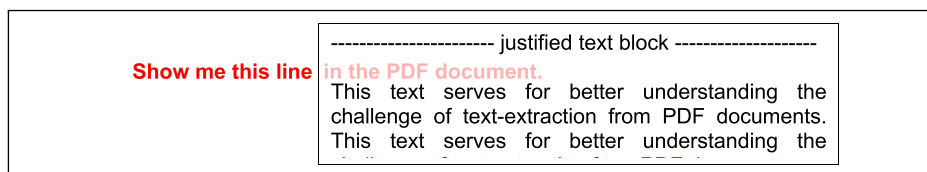
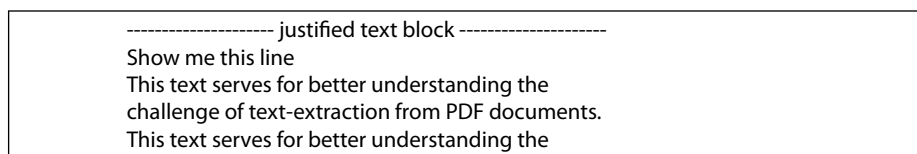


Figure 4.19: Figure shows the overlapped text test with crossed out text and the expected result.

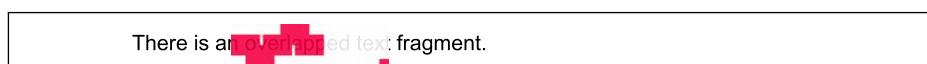


(a) overlapped text by a text-block



(b) expected result; overlapped text is deleted

Figure 4.20: Figure shows the overlapped text test, whereby text is overlapped by a text box, and the expected result.



(a) overlapped text by an image



(b) expected result; overlapped text is deleted

Figure 4.21: Figure shows the overlapped text test, whereby text is overlapped by an image, and the expected result.

T10: Combination Test

The tests header and footer, text, text-block, table and all overlapped tests are combined in this test. There is no image for this test. The expected result is the expected result of each single test.

Test file: PDF/A 10

T11: Combination Test with more pages

This test contains all test of the combination test and allocated over two pages. A figure was not generated for this test but the expected result is the expected result of each single test.

Test file: PDF/A 11

4.4.2 Test results

TEA-M was being tested by a sequence of PDF/A files, where each PDF/A file contains one or more unit tests. In Figure 4.22, a plus implies, the test was carried out successfully, a minus means the test was failed and a slash means the test was not performed with this file.

The table in Figure 4.22 shows that, beside document PDF/A 12 and PDF/A 13, all the tests are carried out successfully. The files PDF/A 12 and PDF/A 13 involves the tests for extracting a listing and the text from a picture. The extraction of the listing has failed because the listing symbol "x" is extracted as "?". Hence, the extracted character "?" does not match with the expected one "x". This is the result of a wrong character encoding by the OpenSymbol font which is used by OpenOffice for the listing symbol. The test in PDF/A 13 has failed, because TEA is not designed for recognizing text in pictures.

Summing up, TEA-M has accomplished all tests regarding the algorithm requirements. That is, TEA-M recognizes hidden text in a PDF/A document, which satisfies [AR2] and [AR3]. Furthermore TEA-M extracts only the visible text of the document following the reading direction from top left to bottom right, which satisfies [AR2],[AR3] and [AR4]. The algorithm requirement [AR1] is satisfied by taking PDF/A documents as input, which TEA-M fulfils.

Document-ID	visible text										hidden text		
	header & footer	body text	text-block	table	listing	text-in-picture	multi-pages	rendering mode	overlapped text	marginal color-diff			
PDF/A 1	+	/	/	/	/	/	/	/	/	/			
PDF/A 2	/	+	/	/	/	/	/	/	/	/			
PDF/A 3	/	/	+	/	/	/	/	/	/	/			
PDF/A 4	/	/	/	+	/	/	/	/	/	/			
PDF/A 5	/	/	/	/	/	/	/	+	/	/			
PDF/A 6	/	/	/	/	/	/	/	/	+	/			
PDF/A 7	/	/	/	/	/	/	/	/	+	/			
PDF/A 8	/	/	/	/	/	/	/	/	+	/			
PDF/A 9	/	/	/	/	/	/	/	/	/	+			
PDF/A 10	+	+	+	+	/	/	/	+	+	+			
PDF/A 11	+	+	+	+	/	/	+	+	+	+			
PDF/A 12	/	/	/	/	-	/	/	/	/	/			
PDF/A 13	/	/	/	/	/	-	/	/	/	/			

Figure 4.22: Overview of executed unit tests and its results; green plus = test was successful, red minus = test was failed, slash = not tested with this PDF file

Chapter 5

Discussion

The primary objectives of this thesis are the specification and implementation of a proof-of-concept text extraction algorithm for PDF/A-1a documents in order to enable a reconstruction of Official Signatures from printouts. Key aspects of the algorithm are that the extracted text is unambiguous, represents the visible text content of the underlying document and considers the reading direction from top left to bottom right. Moreover, the text extraction algorithm shall replace the underlying algorithm of the PDF-Amtssignatur (PDF-AS) application. PDF-AS was developed by the E-Government and Innovationszentrum (EGIZ). It facilitates the attachment of Official Signatures on PDF documents.

In the course of this work the difficulties of PDF regarding the text extraction and reconstruction of Official Signatures from printouts have been shown. To tackle the problems of PDF the new PDF/A-1a document format was introduced. The ISO standard PDF/A-1a focuses on the archiving of PDF documents over a long-term of time. This implies the preservation of the visual appearance and the ability of processing the document's text content (e.g. text extraction) with respect to the reading direction by providing a so called logical structure tree.

However, we have shown that the document format PDF/A-1a establishes a basis for text extraction but it does not satisfy all requirements of Official Signatures in terms of reconstruction from printouts. On the one hand this involves the unambiguous text extraction regarding the reading direction and on the other hand the consideration of hidden text. The provided logical structure tree of PDF/A-1a documents has turned out as "bad choice" for text extraction. This is because PDF/A-1a does not define any policies for creating the logical structure tree. Hence, the structure tree may change from PDF/1-a creation application to application. In consequence, the structure tree does not guarantee

an unique text extraction. Moreover, we have shown that the recognition of hidden text in PDF/A-1a documents is crucial for reconstructing Official Signatures from printouts.

To tackle these and further problems of PDF/A-1a, we have introduced the Text Extraction Algorithm (TEA) in Chapter 3. TEA provides a two phase concept for text extraction from PDF/A-1a documents. At the first phase a HTR algorithm designates hidden text fragments. At the second phase the visible text content is extracted by the text's vertical and horizontal position. Therefore a character is represented by a bounding box

In section 4, a Java implementation of TEA called TEA-M have been developed. Focus of the TEA-M architecture is the easy adaptation and extension of TEA. Therefore, a multi threaded version of the Pipes and Filter architecture [BMR⁺96] was introduced. Furthermore, in Section 4.4 TEA-M have been tested against a small set of unit tests. The test results in Section 4.4.2 shows that not all problems at the text extraction are solved by TEA. Although PDF/A-1a ensures a proper character encoding, there are problems in character encoding. This was shown by the test file PDF/A 12 in Section 4.4. Also the overlapped text recognition algorithm tends to false positive matchings. This results from the fact that not every overlapped character is unreadable at printout. However, TEA and TEA-M provides a reliable text extraction and builds a solid basis for the further development of PDF-AS.

5.1 Future work

A major target for future work is to evaluate TEA-M against a representative test data set in order to explore the weaknesses and strengths of the implementation. A weakness of TEA is that only PDF/A-1a documents are supported. This restricts the number of PDF documents for text extraction dramatically. Future development will concentrate on relaxing this limitation of TEA. One way could be the transformation of existing PDF documents into PDF/A-1a documents with the focus on adding Unicode information. Another approach could be the combination of TEA with OCR optical character recognition algorithms in order to extract text from non-conform PDF/A-1a documents. Another open question is how to tackle the text-in-picture problem regarding the reconstruction of Official Signatures from printouts. A rather generic solution of this problem would be the designation of pictures by watermarks at the time of signing in order to identify them at printout.

Appendix A

Appendix

A.1 Acronyms

BNF Backus-Naur-Form

CTM Current Transformation Matrix

ISO International Organization of Standardization

TEA Text Extraction Algorithm

TEA-M Text Extraction Algorithm Module

PDF Portable Document Format

PDF-AS PDF-Amtssignatur

HTR Hidden Text Recognition

OTR Overlapped Text Recognition

TE Text Extraction

IAIK Institute for Applied Information Processing and Communications

EGIZ E-Government and Innovationszentrum

ICT Information and communication technologies

MOA Online Application Modules

E-GovG E-Government Act

SigG Signature Act

SigV Signature Regulation

A.2 Glossary

Binary signature

A binary signature signs the entire document or parts of it at byte level.

Text-based signature or textual signature

A text-based signature signs the entire text content of a document within defined specifications of PDF-AS [EG08].

Official signature

An official signature is an electronic signature within the meaning of the Signaturgesetz (Signature Act). In other words an official signature is an advanced electronic signature which serves to facilitate recognition of the fact that a document originates from an authority.

E-Government

The term E-Government covers all electronic services, transactions and interactions between agencies and citizens.

Austrian E-Government Act

The Austrian E-Government Act provides a clear and solid legal basis for the electronic communication, procedure and proceeding within all layers of government. Furthermore it instructs to create technical prerequisites for establishing E-Government in Austria.

Signature Act

The Signature Act provides a legal framework for using and creating electronic signatures.

Portable Document Format (PDF)

The Portable Document Format (PDF) has been well established as an interchange format of electronic documents in private sector as well as for eGovernment.

ISO 19005-1 (PDF/A-1)

ISO 19005-1, Document management — Electronic document file format for long-term preservation — Part 1: Use of PDF 1.4 (PDF/A-1), is the first part in a new family of ISO standards which addresses the issue of maintaining information in electronic documents over

archival time spans. PDF/A-1 bases largely on the PDF 1.4 specification with exceptions of PDF Transparency, encryption, mulit-media and others. PDF/A-1 documents are self-contained and self-describing which means they contain all resources (fonts, structure information) necessary for rendering, printing and for processing its text content.

Signature attributes

An electronic signature is characterized uniquely by its attributes such as signature value, signature date and signature properties.

PDF-AS

The PDF-AS application is an project of the E-Government Innovation Center which attaches an official signature on a PDF document. Thereby it provides two different modes of operation for creating signatures: text-based and binary-based signature.

PDFBox

PDFBox is a open source java libraries that allows the creation of new PDF documents, manipulation of existing documents and the ability to extract content from documents.

TEA

TEA stands for Text Extraction Algorithm and is a canonical text extraction algorithm for PDF/A documents.

TEA-M

The TEA module represents an software implementation of TEA.

Bibliography

- [APA09] Apache pdfbox - java pdf library. Website, 2009. <http://incubator.apache.org/pdfbox> (accessed August 2009).
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Micahel Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.
- [CLS01] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, Januar 2001.
- [Ede83a] Herbert Edelsbrunner. A new approach to rectangle intersections part i. *International Journal of Computer Mathematics*, pages 209–219, 1983.
- [Ede83b] Herbert Edelsbrunner. A new approach to rectangle intersections part ii. *International Journal of Computer Mathematics*, pages 212–229, 1983.
- [EG08] E-GIZ. *PDF-Amtssignatur Spezifikation 2.0.0*. www.egiz.gv.at, 2008. <https://demo.egiz.gv.at/plain/content/download/527/3056/file/PDF-AS-Spezifikation-2.0.0.pdf>(accessed on July 2009).
- [HWD80] Six H.-W. and Wood D. The rectangle intersection problem revisited. *BIT Numerical Mathematics*, pages 426–433, 1980.
- [Int05] International Organisation Standard (ISO). *ISO 19005-1:Document management - Electronic document file format for long-term preservation, Part 1:Use of PDF 1.4 (PDF/A-1)*, Januar 2005.
- [MD01] Thomas Merz and Olaf Drümmer. *Die PostScript- und PDF-Bibel*. Dpunkt Verlag, second edition, March 2001.
- [MSM04] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Dpunkt Verlag, September 2004.

- [Nas98] Kurt Nassau. *Color for science, art and technology*. North-Holland Publishing Co, first edition, 1998.
- [Neu07] DI Edgar Neuherz. Dokumentation strategiekonformität - studie zu sichere signaturen mit pdf/a. Technical report, E-Government and Innovationcenter, 2007.
- [OA05] Jorge L. Ortega-Arjona. The parallel pipes and filters pattern:a functional parallelism architectural pattern for parallel programming. Technical report, Departamento de Matematicas, Facultad de Ciencias, UNAM, 2005.
- [PPS01] Chuck Geschke (Preface), John Warnock (Preface), and Adobe Systems. *PDF Reference: Adobe Portable Document Format Version 1.4*. Addison-Wesley, third edition, March 2001.
- [PS93] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer, 1993.
- [SH76] Michael I. Shamos and Dan Hoey. Geometric intersection problems. *Proc. 17th IEEE Symp. Foundations of Computer Science (FOCS '76)*, pages 208–215, 1976.