Publishing Interactive Paper Documents with OpenOffice

Master Project

Patrick Ponti
<pontip@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Nadir Weibel

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

1.0
12th August 2007

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Copyright © 2007 Global Information Systems Group.

Abstract

Earlier predictions of the paperless office no longer seem realistic. Paper as a medium has many advantages over digital media in terms of how people can work with it, both individually and in groups. It is portable, cheap and robust. It is much more convenient to scan through a book than to browse a digital document. Paper supports forms of collaboration and interaction that are difficult to mimic in current digital world. Instead of replacing paper, we have focused on linking the paper with the digital world, enabling users to freely navigate within information spaces that span over printed and digital resources.
The main focus of this project is to further investigate and implement general approaches enabling a dynamic mapping from physical (x,y) positions on paper to the original digital content as defined at authoring time. The goal is to define a general infrastructure capable of retrieving digital elements within the original authoring tool based on their physical position on paper. The proposed solution has been applied to the particular case of the OpenOffice authoring suite, but it has kept general enough to be used within other authoring tools (e.g. MS Word). Finally, to prove the correctness of the proposed solution, an application based on the document review facilities of OpenOffice has been deployed.

Contents

1 Introduction 1

2 Motivation and Related Work 5

3 Existing infrastructure 9
- 3.1 iPublish 10
- 3.2 iDoc 10
- 3.3 iServer 13
- 3.4 iPaper 14
- 3.5 iGesture 14
- 3.6 Intelligent Character Recognition 14

4 Document Life-Cycle 17

5 Technologies 21
- 5.1 OpenOffice and OpenOffice SDK 21
- 5.2 Page Description Languages 22
  - 5.2.1 PostScript 22
  - 5.2.2 Portable Document Format 23
  - 5.2.3 XML Paper Specification 24
- 5.3 Anoto 27

6 Design and Implementation 31

7 Exemplar Application: PaperProof 45
- 7.1 Operations 45
- 7.2 Implementation 47

8 Results and Discussion 51
- 8.1 PaperProof in use 51
- 8.2 Infrastructure 55

9 Future Work 57
- 9.1 iPublish plug-in 57
- 9.2 iDoc extension 58
- 9.3 PaperProof 59

List of Figures

3.1 Existing Infrastrucure . . . 9
3.2 Mixed Digital and Physical Model . . . 11
3.3 Element Model . . . 11
3.4 The iDoc Framework . . . 12
3.5 iPaper Plug-In . . . 13

4.1 The Document Workflow . . . 17
4.2 The Publishing Process . . . 18

5.1 Glyph . . . 24
5.2 The XML Representation of a `Glyphs` node . . . 25
5.3 The XML Representation of a `Path` Node . . . 26
5.4 Anoto Pattern . . . 28
5.5 Anoto-Enabled Pen . . . 29

6.1 Highlighter Hierarchy . . . 32
6.2 A Highlighted OpenOffice Document . . . 33
6.3 a) `Section` and b) `Paragraph` Nodes . . . 34
6.4 Elements Hierarchy . . . 35
6.5 The Sub Elements . . . 36
6.6 The XML Representation of a `Glyphs` node . . . 37
6.7 Deobfuscation Process . . . 38
6.8 Finder Classes . . . 40
6.9 Publisher Classes . . . 40
6.10 Automatic Granularity Processing . . . 42
6.11 AdapterXPSOO . . . 43
6.12 Retrieval Process . . . 43

7.1 Operations and Related Gestures . . . 46
7.2 TrackChangers Hierarchy . . . 47
7.3 The `Corrections` Constructor . . . 49
7.4 The `execute()` Method . . . 50

8.1 The Authoring Phase . . . 51
8.2 Free Form Annotations on a Paper Document . . . 52
8.3 PaperProof Corrections within OpenOffice . . . 52
8.4 Delete Operation . . . 53
8.5 Replace Operation . . . 53

8.6 Insert Operation 53
8.7 Move Operation 54
8.8 Two Point Annotation Operation 54
8.9 Side Annotation Operation 55

1 Introduction

The concept of the paperless office made its first appearance a few decades ago. The advent of new technologies, such as the personal computer, and their increasing interactivity drove people to imagine a world in which it was possible to completely abandon paper. The continuous development of the existing technologies, further developed the idea of an empty office, where all information is stored and ordered within personal computers.

However, many years after the early predictions of the paperless office, this idea has become a myth. Paper still exists in all the offices of the world and its use even increased [1], probably due to the parallel development and the massive distribution of printers and copy machines that made the creation of paper documents easy and fast.
Moreover, although at first this might strike as a paradox, the birth and rise of e-mailing contributed to increase the amount of paper in the offices. This may be easily explained by considering the number of e-mail messages printed every day in the world.
Even with today's screen quality and resolution, it is still more confortable to read a document printed on paper, rather than directly on the screen. In the same way, it is much simpler to proof-read or annotate a document on paper, especially considering mobility: paper allows to access information in any place, like for instance during a train journey.
Another important aspect of paper, beyond the lightness and cheapness, is its use as a mean of interaction between different persons. How many times did we try to explain something to somebody and in order to better clarify the concepts we used a piece of paper?
Its mobility, the natural usage and the fact that it does not need any kind of power, make paper nowadays still one of the best way to distribute and access information.

To sum it up, nowadays we are not yet ready to abandon paper. On the other side, it would be a waste to give up all the new and important features that current developments in technology provide us with. In fact, it is much simpler to distribute or update a document or to create many documents from the same template if they are in digital form.

From these considerations it is easy to see how much one could gain from joining the paper and digital worlds, creating thus some form of communication and interaction between them and, at the same time, reducing the gap that still exists between these two realities.
A feasible approach might try to make paper a physical interface to the original digital document. This would mean making the printed document a way to transpose some actions from paper to digital documents. The user would interact directly with the printed instance and the result of its action would automatically be visible in the digital document.

The integration of such a model requires two basic steps. First, one needs to define a link between the user intentions on paper and the corresponding digital actions. The information available on paper should therefore be captured and translated into a digital language. Moreover, an interpretation of such information should be provided, in order to locate the digital document or service the user wants to access through paper. The interpretation should also be advanced enough to allow the interaction with the digital counterpart in a smart way, for example recognising the elements within the digital document that the user actually selected on paper.

Beside paper, we could consider other media to the digital document. It should be possible to link together elements of any type such as web pages, videos, audio tracks, and so on, creating thus a platform capable of providing cross-media information management. The Global Information Systems Group [2] at the ETH Zurich developed such a general Integration Server called *iServer* [3].
The iServer framework enables cross-media linking based on a set of link management information concepts. It not only supports links between different kinds of digital media, but also allows for the integration of physical and digital content. Its architecture is designed as a platform that can be extended based on a plug-in mechanism. By implementing media-specific instances of resource components, any new type of media can be integrated. A plug-in called *iPaper* [4] has been implemented for the integration of paper and digital content. Further plug-ins, like the one for XHTML documents, movie clips and sound files have also been developed. Thanks to the implementation of iPaper, the iServer framework may therefore be used to allow the interaction between the physical paper world and the digital one.

At this point the next step would be the creation of such a link based not only on the physical position of the elements within the document, but also on semantic aspects of the document's contents. In order to achieve this objective, it is first required to be able to link not only the two instances of the document, but also all single elements composing them. Acquiring semantic information about the single elements of the printed document is therefore an essential step.

A step forward in this direction has been made by the Global Information Systems Group, who developed a model capable of combining the metadata coming from both digital documents (semantic information) and from printed documents (physical information). This model is the base of the *iDoc* framework [5], which is used for the publishing of interactive documents. When a document is printed, the semantic information about the structure and the content of the document is stored within a database, so that it can be retrieved later. This

information would otherwise be lost during the printing process. However, extracting the semantic information of every existing element of a document would be a massive operation. The solution of tracking all single elements within a document (e.g. up to characters), calculating their positions and making them persistent within a database is not a feasible solution, since it requires an extremely large amount of space and a huge computation time. A more appropriate approach is to make the operation more dynamic, grouping the small elements into higher granularity sets (e.g. characters into words or words into paragraphs) and tracking the sub elements only when they are really needed. This approach will be extensively described in chapter 3.2 when describing the iDoc framework.

The first phase of this project involves the identification of a generic approach to dynamically map a physical position on paper to the digital elements within the original authoring tool, including all possible semantic information.
The second phase consists in the implementation of a general infrastructure capable of managing the physical representation on paper of the digital documents. We particularly focus here on text documents.
The last part includes the deployment of an interactive paper application based on OpenOffice and its document tracking facilities that exploits the features of the defined infrastructure. The outlined example has also the purpose of illustrating a real-world use of such an infrastructure. It proposes a solution for the very common problem of the automatic insertion of corrections and annotations that a user did on a printed document into the original digital document. How many times did we write a document with an authoring tool, print it on paper, read it, correct it and then copy the corrections within the original authoring tool? This essentially involves the same work twice.

Our interactive tool and the related infrastructure prevent such a redundant work by building a link between the printed and the digital document and translating the concepts expressed by the user on paper directly into the corresponding digital document in a transparent way.

In chapter 2, we analyse two existing applications and highlight theirs weaknesses, in order to understand the needs that our project tries to satisfy.
Chapter 3 gives an overview of the existing infrastructure, giving a brief description of all the existing components.
Chapter 4 outlines our vision of the document life-cycle, explaining in details what we are proposing with this project.
In chapter 5 we describe all the software and hardware tools and the technologies that were used during this work, providing a brief explanation of the benefits we gained from their use.
Chapter 6 introduces first the design of our system and then provides the reader with some implementation details.
In chapter 7, we discuss our approach, focussing in particular on the application provided for its validation: PaperProof.
Chapter 8 highlights the results obtained from the use of PaperProof giving at the same time some considerations about the presented infrastructure.
Finally, in chapter 9, we present the current weaknesses of the implemented system, along with new ideas for future improvements and developments.

2 Motivation and Related Work

Nowadays there exist several technologies that allow linking printed documents to digital entities. Probably the most advanced is the Anoto technology [6], which along with a dedicated printing approach provides a special pen equipped with a camera able to track the position of the pen with respect to the paper. Moreover, it provides a streaming-mode which makes it possible to maintain a continuous flow of information between the two worlds. More on the Anoto technology will follow in chapter 5.3.

The iServer framework [3], which will also be extensively discussed in chapter 3.3, introduces with its iPaper plug-in an approach based on the Anoto technology and the concept of “active areas” on paper. This approach allows the definition of regions on a paper document taking the role of link source on the physical instance of the document. Once the digital pen is positioned on one of these areas, a request is sent to a server which is responsible for the activation of the digital target. A typical example of such an approach is given by the hyperlinks: while reading a document on paper, the user may point on a link on paper and automatically the link will be activated and the corresponding digital element displayed on a computer screen.

Until now, however, this approach tends to be one-way, meaning that it is mainly used to generate interactive paper documents from their digital instance, rather than bearing bidirectional and continuous transactions between the digital and paper instances of the same document.
In response of this need, a model was developed, capable of combining the metadata coming from the paper and digital instances of the same document. The iDoc framework, created for the authoring and publishing of interactive paper documents, is substantially based on this model.

Taking into account the authoring process, we realised that in this phase a large amount of semantic information about the content of the document is available within the authoring tool

itself. This information is however lost at the moment that the document is sent to a printer device. Our main purpose is to maintain this semantic information and to store it somewhere where it is active and accessible even when we are dealing only with the printed instance of the document. Using this additional information, we are then able to select a single logical element of the printed document (such as a word) and to retrieve the corresponding digital counterpart in the source document at a semantic level.

But what does this mean? If, for example, we add an annotation on paper beside a word, this is linked to the corresponding digital word not thanks to the physical position it occupies on paper, but by tracking the position that the word has with respect to the paragraph or section that contains it. Moreover, we are able to access the characters composing the word as well as the paragraph or the section the word is contained in, allowing a freely granularity-dependent transition between the different logical elements composing the document.

This feature is one of the biggest peculiarities which diversify our solution from other existing projects based on similar technologies.

In the remainder of this chapter we present a short overview of two systems, which are able to add annotations to a document using a digital pen or a tablet PC.

- **ProofRite**
  ProofRite [7, 8] is a word processor, developed by Conroy, Levin and Guimbretière, which supports digital and physical document annotation. It is able to merge the annotations that different users made on the printed instance of a document with the digital source. As soon as the users are back to the computer, they may continue the writing process on the digital document. ProofRite is able to reflow the markings accordingly to the changes made by the users. Furthermore, ProofRite uses a repositioning algorithm that permits the document's content to change, allowing users to modify content and structure while preserving the meaning of annotations.

  The main purpose of ProofRite is to automatically transport the user annotations made on paper into the corresponding digital documents. However, ProofRite does not actually execute any task based on these annotations. If the user corrects part of the text on paper, ProofRite transfers the notion that the text needs to be corrected also in the digital instance of the document, but does not actually correct it nor selects it within the authoring tool. Indeed, ProofRite does not provide an "auto-correct" or an "apply annotations" tool which is able to programmatically execute the intentions of the annotations.

  Another weakness of ProofRite is that it tracks the exact position of annotations on paper and anchor them to the elements (i.e. words) rendered at the same position, but the created link is based only on the physical position of the annotations and the anchored elements and not on the semantic meaning that the link is actually carrying. This means that, if the digital elements change format after printing (for example the font size gets bigger), the physical positions (x,y) of elements may differ from the ones recorded at printing time and ProofRite is not able to link the right elements anymore.

- **XLibris**
  XLibris [9, 10] is a tool presented by Golovchinsky and Denoue which works mostly on tablet PCs. Here the situation differs with respect to the one highlighted before, since the user interacts with the system without using a piece of paper but by means of a tablet PC, however the basic concepts are still similar. XLibris proposes an annotation system similar to the one of ProofRite, but here the emphasis is put on the concept of "free-annotations", which do not have to conform to any structural constraint. The algorithm behind XLibris is able to preserve the position of the annotations even when the document is viewed with different font or font sizes, with different aspect ratios or on different devices. This is possible, thanks to the fact that each annotation is attached to a location which is independent of the document pagination. Changes on the document's content are instead not allowed.

  Also this approach, as the one proposed by Conroy et al., bases the insertion of annotations only on the physical information concerning the position of the annotation on the "paper-like" display and does not consider any semantic components of the anchored elements which could provide additional useful information.

Both approaches presented here do not take into account any of the semantic information provided by the user at authoring time. Once a digital document is printed on paper, it is no longer possible to directly access the semantic "meaning" of the augmented physical element in its digital counterpart, nor it is possible to programmatically establish a link between the printed elements and the original objects within the source document. This lack of semantic information is a relevant weakness of the two highlighted systems. Trying to overcome this problem by storing also the semantic information available during the authoring phase is therefore an important aspect that has to be considered in the development of new applications.

3 Existing infrastructure

The existing infrastructure for interactive paper applications (Figure 3.1) is composed of a set of four main components, which will be explained one by one in details later in this chapter.

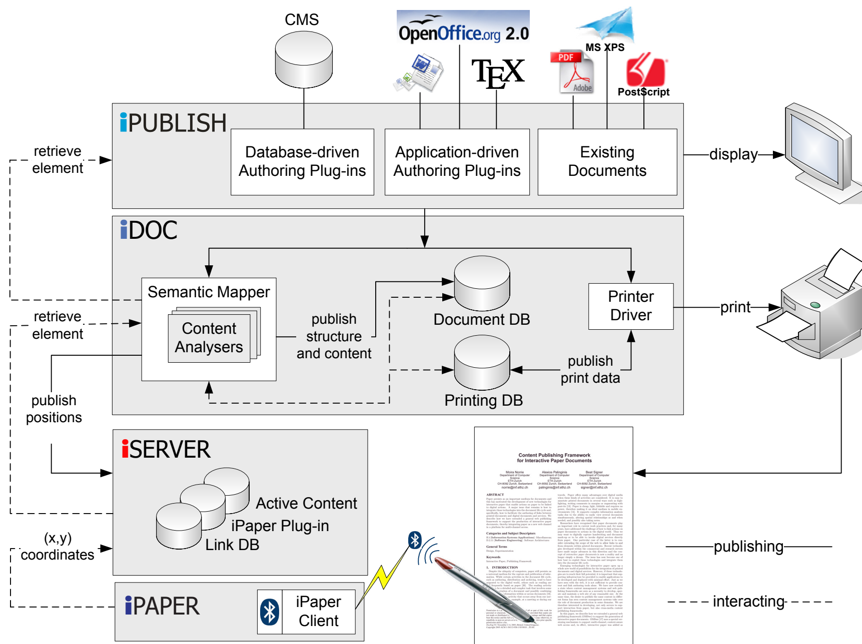


Figure 3.1: Existing Infrastrucure

The publishing component, called iDoc, enables the mapping between paper and digital instances of a document. This component is tightly bound on one side to the iPublish layer, which enables a bridge between the authoring of different kinds of documents and their publishing on paper, and on the other side to the iServer/iPaper framework, which supports interaction with paper documents. Apart from these four components, there exists two other

important ones: the iGesture framework, for gesture recognition, and the MyScript software, which provides an Intelligent Character Recognition (ICR) system. They may play an important role, especially during the development of applications, as we will see at the end of this report, when the exemplar application will be explained. Since iGesture and the handwriting recognition are in relation with interactive paper applications it makes sense to introduce them in this chapter too, in order to give a complete overview of the used infrastructure.

3.1 iPublish

The iPublish component consists of a series of plug-ins defined for different kinds of documents. There are three main categories of documents:

- documents automatically generated using data from a database or a content management system (CMS)
- existing paginated documents without access to the original source document (e.g. PDF, PS, etc.)
- documents generated with a classical authoring tool (e.g. OpenOffice, MS Word, etc.)

Depending on which of the three categories we are interested in, iPublish allows us to define a specific plug-in that tracks structured elements from within the authoring tool, or analyses existing paginated documents. The basic purpose of the iPublish layer is to identify physical and semantic information about the document's contents. Two interesting applications taking into consideration the first category are the EdFest system [11, 12], an interactive paper application able to support interaction for users on the move, and a similar application based on the text book produced by the BBC (British Broadcasting Corporation) for their series on ocean life, called Blue Planet [13]. For the second category, the Print-n-Link [14] system is a good example. It uses interactive paper technologies in order to enhance the reading process of existing printed documents. Users are allowed to access digital information and to search for cited documents, using only a digital pen for interaction. An example application for the third category of documents will be proposed in chapter 7, when the PaperProof application will be explained in details.

3.2 iDoc

The iDoc framework is based on a mixed digital and physical model, which is able to store metadata about both the digital and paper instances of a document. The model is shown in Figure 3.2. This mixed model is based on two distinct parts representing metadata coming from i) logical structures, referred to as digital documents and represented in the upper-right part of the model, and ii) paginated formats, referred to as physical documents and represented in the lower-left part of it.
Through the `HasPhysicalDescription` association the two parts are connected together, allowing in this way a full mapping between the digital and physical instances of the same document.

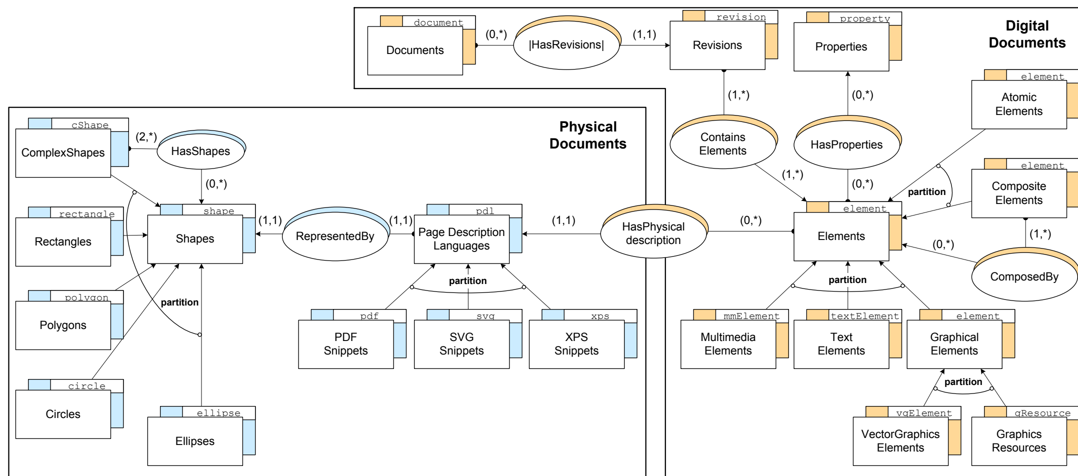


Figure 3.2: Mixed Digital and Physical Model

If we analyse the model in more detail, we see that we can split it in four basic parts: the first one dealing with documents, the second one embodying the concept of element, the third one representing the physical description of an element and the last one handling the geometric concept of shapes. For each digital document, a `Document` object is created in the `Documents` collection. Since it is possible to update several times the same document, revisions have also been introduced, in order to allow different contents for the same document. A new revision is defined each time a new version of the document is printed.

Each revision contains one or more `Element` objects as defined by the `ContainsElements` association. Three types of `Element` have been defined, depending on the nature of the `Element` itself: `Multimedia`, `Text` and `Graphical` elements.

The idea behind the iDoc framework is to have the possibility to refer to the logical representation of the objects at every level of granularity (e.g. from section up to character level). This implies that the storage processing of revision's elements has to be executed in a smart way, otherwise the amount of data to be transferred and stored into the database would be huge and this is unacceptable.

For this reason the concept of element has been developed according to a composite design pattern as shown in Figure 3.3.

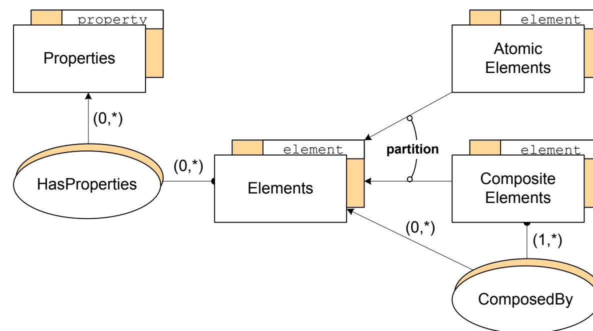


Figure 3.3: Element Model

An Element may belong to the AtomicElements collection if it does not contain any lower-level objects. Otherwise, it is inserted in the CompositeElements collection and a ComposedBy relation is created for each of its lower-level elements.
Using this approach the elements' storage is first executed only at high-level granularity, so that the number of objects that have to be initially stored into the database decreases. For the storage of elements at lower-level granularity a dynamic approach is taken. Since the composite pattern creates a hierarchy of Elements (mantaining this way also an important semantic information), we start from the highest granularity level and we compute the sub elements (up to the desired granularity) only when they are really needed in a recursive manner. The sub elements of the current Element object are computed and inserted into the AtomicElements collection, while the current element is moved into the CompositeElements collection. At the same time ComposedBy associations are created for each sub element. This process might be further repeated until the desired granularity level is reached.

Each Element can be described by a Page Description Language (PDL) snippet. A PDL is a language that describes the appearance of a printed page, introducing some additional information such as position coordinates, font size, font name, and others. The PDLs supported by the model are currently three: PDF [15], XPS [16] and SVG [17], but the model can be easily extended at any time. A detailed description of three PDLs examples follows in chapter 5.2.
Once the physical description of the elements has been defined, the HasPhysicalDescription association has the function of connecting the digital Element extracted from the document with its physical information stored in the corresponding PDL's snippet. From the PDL snippet a shape is finally built which contains the exact physical appearance that the digital element will have on paper. Several types of shape are defined and a composite pattern is used here in order to model the concept of complex shapes (Figure 3.2, left side).

The iDoc framework operates during the printing process of the document in strict collaboration with the corresponding iPublish plug-in. We want now to analyse more in details the different components used in the iDoc framework, giving a detailed description of the main parts composing its core (Figure 3.4).

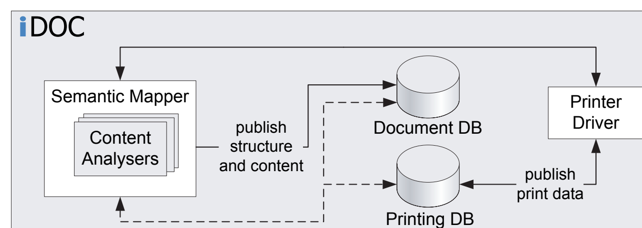


Figure 3.4: The iDoc Framework

The main part of the iDoc core is the *document database*. This database implements the functionalities needed to support the model explained before. All the logical and physical metadata needed are stored there, in order to allow a later mapping back from the physical

instance of the document to its structure defined at the logical level.
Another important part of iDoc is the *printing database*. It is used to store information about the printing technology used in the publishing process (e.g. Anoto). Depending on the printing technology, the printer driver generates the augmented version of the printed document.
The last component of the framework is the *Semantic Mapper*. It is responsible for three main operations. The first one is the insertion of the logical elements into the document database: this is achieved by the analysis of the document structure identified during the iPublish phase and the definition of the logical elements. The second operation is the computation of the position of the different elements and their shapes, based on the information stored in the PDL snippets. The last operation is the publishing of the elements' shapes into the iServer/iPaper framework and the definition of links with the corresponding logical elements stored in the document database.

3.3 iServer

As already mentioned in chapter 1, iServer enables cross-media linking based on a set of link management information concepts. Links within the iServer framework are defined as directed and bidirectional entities, therefore allowing them to have at least one target and one or more source entities. The simplest kind of entity is a resource, which represents an entire information unit. In order to be able to control the granularity of link sources and targets, the concept of selector has been introduced. It allows to address parts of a resource. The links can be defined also between parts of resources and in order to achieve this a specific implementation (plug-in) for the resource and selector has to be provided. An example of such an implementation is the iPaper plug-in, which is broadly explained in the next section. Figure 3.5 shows a simplified version of the iServer link model and a plug-in example: iPaper. For a detailed explanation of iServer and the related architecture please refer to [18].

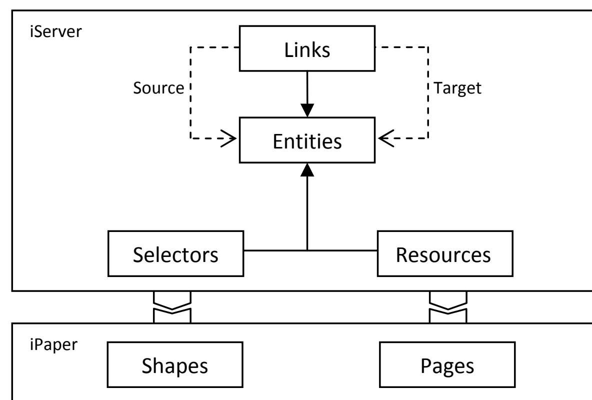


Figure 3.5: iPaper Plug-In

Recently, iServer introduced the concept of *active component*. An active component, which is basically a piece of code, can be integrated into iServer as source or target of a link. For

example, when a link is visited and selected, an active component may be executed and the program code in it is loaded, allowing the active component to become a handler for the information coming, for example, from a digital pen like the one based on Anoto technology. A very interesting feature of the active components is their reusability. They can be reused across different applications and each new implementation of an active component makes the existing pool bigger. The designer of new interactive applications can access this pool, allowing the creation phase to get much faster and reducing at the same time the efforts of the application's developer team.

3.4 iPaper

The base unit for linking paper documents is a page. According to Figure 3.5, a resource is represented by a single page and a selector is an active area defined by a shape within that page. iServer links can be defined from an active area of the page to any other iServer resource. Each time a user points to a position (x,y) within an active area, the iPaper plug-in resolves the selected shape and its associated links will be activated. In order to resolve the right shape, the iPaper plug-in needs three inputs: the document identifier, a page number and the (x,y) position where the user is pointing to. This information is general enough to avoid that iPaper depends on a specific pen technology.
iServer does not store a digital version of the document within its framework, it only stores some metadata about the printed document, such as dimension and number of pages, in the form of a digital document model. This information is used by iPaper to exactly understand which shape the user is interested in and consequently which link has to be activated by iServer.

3.5 iGesture

iGesture [19] is a Java-based gesture recognition framework which is implemented following the principles of extendibility and cross-application reusability. The framework is designed to be extendable for the integration of new functionalities and for the fulfilling of any needs or new requirements. The iGesture framework is used in order to interpret the user's intentions. Thanks to its functionalities, it is possible to understand from the gestures that users do with the digital pen, which are the purposes they want to achieve. In collaboration with the Intelligent Character Recognition explained in the next section, iGesture is the entry point for users' inputs.

3.6 Intelligent Character Recognition

The MyScript Intelligent Character Recognition (ICR) algorithm [20] is currently part of the existing infrastructure and is designed to recognise writing captured by a digital pen, a writing capture device or a graphics tablet. The main difference between the ICR method and the most known OCR (Optical Character Recognition) method, is that the first one bases the text recognition on how the text was written (e.g. speed, sequence of strokes, etc.), while the second works with bitmap images from scanned documents. Therefore, in addition to

iGesture, which interprets the actions that users want to perform, the ICR algorithm allows the existing infrastructure to be able to deal also with textual inputs.

4 Document Life-Cycle

In the previous chapters we introduced the iDoc framework as a tool to build a continuous and bidirectional mapping between the physical and digital instance of a document.
Analysing the life-cycle of a document within the iDoc framework we can identify five essential concepts that form a logical and cyclic workflow that starts and ends with the authoring tool (Figure 4.1).

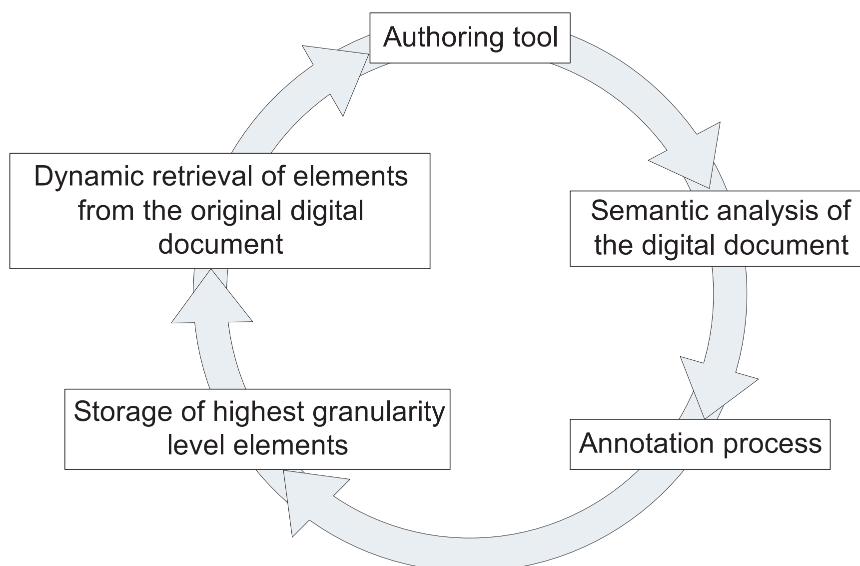


Figure 4.1: The Document Workflow

The workflow starts with the creation of a digital document within a standard authoring tool (e.g. OpenOffice, Microsoft Word, ...). When the user is satisfied by the work and the document is sent to the printer, the printing job is captured by the iPublish plug-in.
The iPublish plug-in performs a semantic analysis on the document contents based on

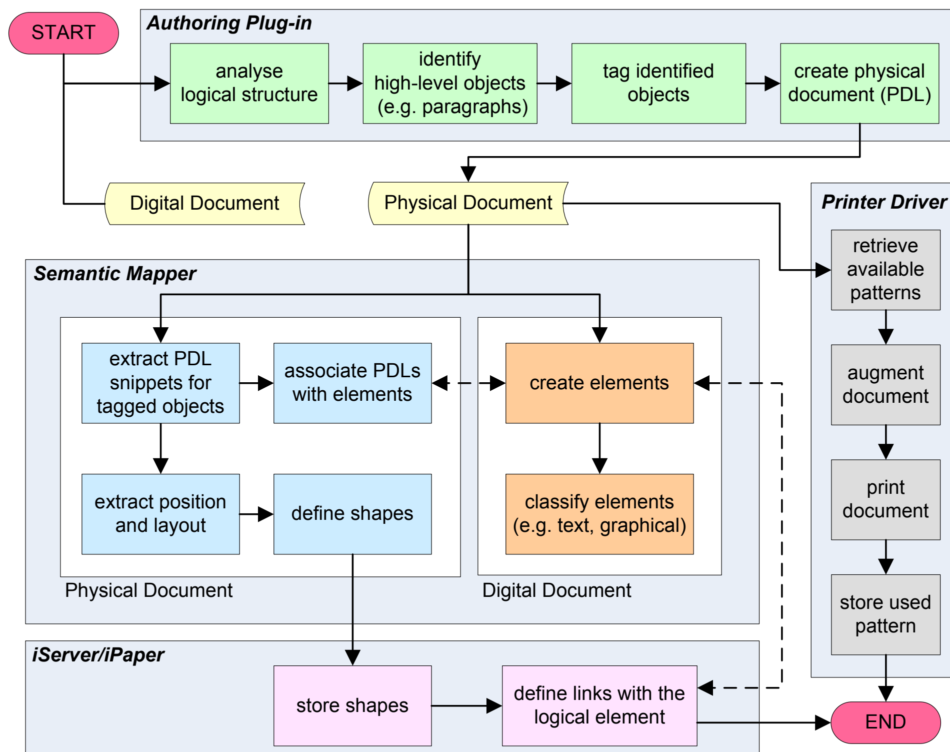


Figure 4.2: The Publishing Process

the document tree stored within the authoring tool and extracts the high-level elements (i.e. sections or paragraphs). Afterwards the digital document is transformed into a physical document with the help of a Page Description Language (e.g. PDF, XPS, or SVG), enriched with the semantic information extracted in the previous phase. This process is defined as the PDL's annotation process and is depicted in the upper part of Figure 4.2.

The annotated PDL is then forwarded to the iDoc component which splits the remaining process in two tasks. In one task, a copy of the PDL is sent to the printer driver which retrieves information about the augmenting technology to be used from the printing database and prints the augmented version of the document. In the other, a second copy of the PDL is sent to the Semantic Mapper, which parses the PDL file in order to identify the newly defined elements. The PDL snippets of the new elements are stored into the document database, and for each of the snippets a logical element is created and inserted into the database as well. At the end a link between the elements and the corresponding snippets is provided, by the `HasPhysicalDescription` association, as described in the model of Figure 3.2. The Semantic Mapper and its tasks are shown in the mid-part of Figure 4.2.

After that, the Semantic Mapper computes the positions and shapes of the logical elements using the physical information stored in the PDL snippets and publishes this information to the iServer/iPaper framework. For each element, a link between its physical position on paper and its digital counterpart is created and added to the iServer database (Figure 4.2, lower part).

Within text-based documents, the lowest granularity level is the character. Since the storage of every character of a document is obviously not an optimal strategy due to the large storage requirements and the time needed for such a computation, our system computes the character shapes and positions only when they are actually needed: thus the system offers a dynamic storage of elements, that refines the granularity of the stored semantic information on a per-request basis, by splitting coarse elements into sub elements and performing a more in-depth analysis only when this is required. The concept of sub elements is modeled as explained in chapter 3.2, according to a composite design pattern.

After publishing a document we have to retrieve the digital elements from the original digital file by means of the physical description of the document. This is the fifth stage of the iDoc workflow. Let us consider a practical example. If a user highlights a word on paper with a digital pen, the iServer/iPaper framework forwards the information about the document, the page and the position of the highlighted object to the Semantic Mapper, which looks for the word within the document database. If the word has not yet been stored within the database, the Semantic Mapper retrieves the corresponding paragraph element (the element with higher granularity which contains the position highlighted by the user) and splits it into its sub elements, according to the content of the paragraph's PDL snippet. For each sub element, the position and shapes are computed and inserted into the system. At this point the search of the desired word within the digital document may start and the authoring tool's plug-in is called with the physical word position as a parameter. The plug-in identifies the word within the document and highlights it, thus closing the circle.

The framework and the concepts highlighted in this chapter enable the construction of a reliable bridge between the two worlds, physical and digital, in which a document can exists, giving to the users the freedom to perform the same type of operations on a document no matter whether they work on a printout of it or directly on the screen.
In the remainder of this report we will look at how this may be practically achieved explaining first some of the technologies used, in chapter 5, then the design and implementation of our system in chapter 6, validating finally our approach with an exemplar application, PaperProof, in chapter 7.

5 Technologies

In this chapter we introduce the software and hardware tools and some technologies that were used for this work.

5.1 OpenOffice and OpenOffice SDK

OpenOffice [21] is one of the best open-source office suite available today. It is developed by Sun Microsystems [22] and is available for many different operating systems, although the primary development platforms for OpenOffice are Linux, Microsoft Windows and Solaris. OpenOffice has been implemented according to the OpenDocument standard [23] for data interchange and is available under the GNU Lesser General Public License (LGPL) [24].
The OpenDocument standard (ODF) is a XML-based document file format for electronic office documents. It is used for documents containing text, spreadsheets, chart and graphical elements. Since it is an open format, new applications can be freely implemented to read and write this kind of documents.

OpenOffice is a collection of several applications that provide all the features expected from a modern office suite (as, for instance, Microsoft Office [25]):

- **Writer**: a word processor similar to Microsoft Word [26] that offers a comparable set of functions and tools.
- **Calc**: a spreadsheet similar in concepts and features to Microsoft Excel [27].
- **Impress**: a presentation program similar to Microsoft PowerPoint [28]. It makes it possible to export the presentation's slides to Adobe Flash [29].
- **Base**: a database program similar to Microsoft Access [30].
- **Draw**: a vector graphics editor comparable in features to Corel Draw [31] or, in lesser measure, to Microsoft Publisher [32].

- **Math**: A tool for creating and editing mathematical formulae, in analogy to the Microsoft Equation Editor.
- **QuickStarter**: A small executable provided for Windows and Linux that is launched when the computer starts and loads the OpenOffice core files and libraries to allow for a faster start of the suite applications.
- **The macro recorder**: An useful application able to record user actions and replay them to automatise repetitive tasks.

Most of the OpenOffice applications also include the ability to export Portable Document Format (PDF) files (see below) without the need of additional tools. Another interesting aspect is that the OpenOffice applications are able to read the files created by the corresponding programs of the Microsoft Office suite.

Along with the office suite, Sun Microsystem provides the user with a Software Development Kit (SDK) [33], that covers all the applications of the suite. The SDK supports two programming languages: Java and C++. In this context, the choice of the OpenOffice.org's SDK was motivated by the fact that most of the existing infrastructure for interactive paper applications was indeed implemented in Java.
Besides providing programmatic access to every feature a user can execute within the OpenOffice user interface, the SDK allows the implementation of custom plug-ins which can then be imported as part of the suite itself.

5.2 Page Description Languages

A "print-ready" format is a description language that describes the physical appearance of a document as precisely as possible. The first and most intuitive usage of such a language was to drive the printing process, but nowadays with the design and development of new description languages, some other features have been added to the concept of print ready formats.

We will now look at two Page Description Languages (PDL) which made the history of PDLs, the oldest one, the PostScript language (PS) [34] and the today's most used Portable Document Format (PDF) [15]. We will then have an outlook inside a new very promising PDL from Microsoft: the XML Paper Specification (XPS) format [16].

5.2.1 PostScript

PostScript is a PDL, but also a programming language designed to do just one thing: describe in an accurate way the final appearance of a (printed) page.
Every programming language needs a processor to run or execute its code. In the case of PostScript, this processor is a combination of software and hardware which typically resides in a printer. Such a processor is called a Raster Image Processor (RIP). A RIP takes PostScript code as input parameter and renders it as rasterised image.
The PS programming language is an interpreted, stack-based language with strong dynamic typing, data structures and, in the newest versions, garbage collector.

PostScript syntax

The syntax of the PS language uses *reverse Polish notation* [35], as a consequence of the fact that it is a stack-based language.
As an example of this notation, let us look at the following arithmetical expression:

7 5 sub 2 9 add mul

This has the same meaning as:

*(7 - 5) * (2 + 9).*

In order to produce graphics, PS uses a similar syntax and a Cartesian coordinate system with origin at lower-left of the page. The following example positions the output tool at the point with coordinates (200, 200) and then draws a line from that point to the point with coordinates (350, 425):

200 200 moveto 350 425 lineto stroke.

A deeper discussion of the technical features of the PS is beyond the scope of this chapter, however it is interesting to know that the PostScript language along with its RIP interpreter is still the most used language in the area of professional high-end digital imaging, but will probably be supplanted in the future by one of its own descendants: the PDF, that we will discuss next.

5.2.2 Portable Document Format

PDF is an open standard and platform-independent file format. Additional than being one of the most used file format for storing, sharing and distributing digital documents, PDF is also a page description language. Built largely on the PostScript language, PDF has taken the PS-approach a step further. Besides describing the layout of a page, the PDF format can also store images, interactive hyperlinks, keywords for searching and indexing, fonts, and so on.

A PDF file is essentially a PostScript file that has already been interpreted by a RIP processor. It basically combines three technologies:

- A sub-set of the PS page description programming language, for generating the layout and graphics (some flow control commands such as *if* and *loop* have been removed, while graphics commands such as *lineto* have been maintained).
- A font embedding system to allow fonts to be kept together with the document itself.
- A storage system, which is able to structure all the document's elements into a single file, with data compression where possible.

The content of a PDF file is described by means of content streams. A PDF content stream contains a sequence of instructions used to describe the appearance of text and graphical entities.

Graphical objects in the PDF format fall in one of four classes:

- **Path object**: represents an arbitrary shape.
- **External object** (XObject): represents a graphical object defined outside the content stream.
- **Inline image object**: represents a small raw image directly stored within the content stream, by means of a special syntax.
- **Shading object**: represents a shape whose color varies according to a shading function.

For text, a unique object is provided. It is composed by one or more characters referencing a sequence of *glyphs*. There is a clear separation between the concepts of character and glyph. The first one is an abstract symbol, while the second is a specific graphical rendering of the first. Glyphs are stored into fonts, which can be considered vocabularies defining glyphs for a particular character set: as an example, the *Times* font defines glyphs for a set of standard Latin characters.

Figure 5.1 shows a glyph for the character 'f'. The grey box is the smallest rectangle that can enclose the glyph shape. The rendering position of a glyph depends on the position of the previous rendered glyph. In fact, the origin of the first glyph of a line sets the position (0, 0) where the glyph will be drawn. The origin of the following glyphs is the origin of the previous glyph shifted by the *Advance width* of the same glyph (see Figure 5.1).

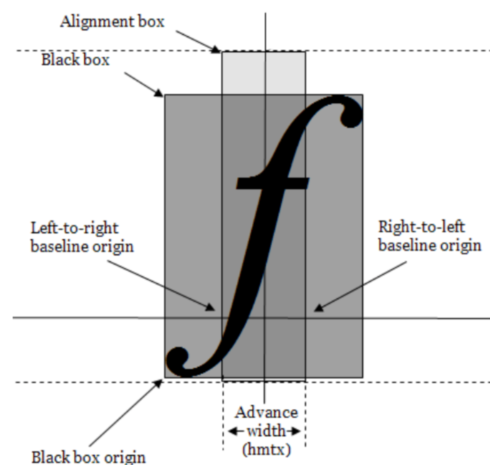


Figure 5.1: Glyph

5.2.3 XML Paper Specification

The Xml Paper Specification (XPS) is the newest page description language format proposed by Microsoft [16]. This format represents a set of pages with a fixed layout, which are parts of one or more documents. A file that implements this format is a simple zip archive with extension *.xps*. It includes everything (i.e. fonts and images, collectively referred to as

```
<Glyphs
Fill="#ffff0000" FontRenderingEmSize="16.0006"
FontUri="/Documents/1/Resources/Fonts/493B8DF4-19BD-4088-A175-03717D006BB3.odttf"
Indices="36,73;79,27;69;72,45;85;87;3;40;76;81;86;87;72;76;81" OriginX="123.04" OriginY="164.16"
StyleSimulations="None" UnicodeString="Albert Einstein" />
```

Output: Albert Einstein

Figure 5.2: The XML Representation of a `Glyphs` node

resources) necessary for rendering documents on a display device or on a physical medium (e.g. paper).

The content of a single page is expressed as a XML document tree. The XML schema behind an XPS page is based on an element, `FixedPage`, defining the width and height of the page and a list of children:

1) *Glyphs*: they are used to represent a sequence of uniformly-formatted text from a single font. Figure 5.2 is an example of a `Glyphs` node, as it appears in the XPS file.
There are different attributes describing a `Glyphs`:

- **Fill**: Describes the brush used to fill the shape of the rendered glyphs. In practice it represents the color of the text.
- **FontRenderingEmSize**: Specifies the font size in drawing surface units, expressed as a float in units of the effective coordinate space[1]. A value of 0 results in no visible text.
- **FontUri**: The URI of the physical font from which all glyphs in the run are drawn. The URI must reference a font contained in the package and thus it is a local reference.
- **Indices**: Specifies a series of up to 4 glyph indices, used for rendering the glyphs sequence. The first value is the character index and is used in order to retrieve it whithin a font file. The second defines where the next character has to be drawn with respect to the first. The third and fourth values are used to adjust the character position along x- and y-axis.
- **OriginX**: Specifies the x coordinate of the first glyph in the run, in units of the effective coordinate space[1].
- **OriginY**: Specifies the y coordinate of the first glyph in the run, in units of the effective coordinate space[1].
- **StyleSimulations**: Specifies a style simulation. Valid values are *None*, *ItalicSimulation*, *BoldSimulation*, and *BoldItalicSimulation*.
- **UnicodeString**: Contains the string of text rendered by the `Glyphs` element.

[1]By default, elements are rendered in a coordinate space with units of 1/96 inches. The effective coordinate space for a particular element is created by sequentially applying each parent and ancestor element's affine matrix transformation, specified with the Transform or RenderTransform properties, from outermost to innermost, including the element's own affine matrix transformation.

```
<Path
Data="F0 M 118.72,108.64 L 209.6,108.64   209.6,224.16   118.72,224.16 118.72,108.64 z"
Fill="#ff99ccff" />

<Path
Data="F1 M 164.16,224.16 L 118.72,224.16 118.72,108.64 209.6,108.64 209.6,224.16 164.16,224.16"
Stroke="#ff000000"
StrokeThickness="0.32"
StrokeLineJoin="Round"
StrokeStartLineCap="Round"
StrokeEndLineCap="Round" />
```

Output:

Figure 5.3: The XML Representation of a `Path` Node

2) *Path*: they are the sole means of adding vector graphics and raw images to an XPS page. Figure 5.3 is an example of a Path node, as it appears in an XPS file.
The meanings of the attributes of the Path node as they are defined in the XPS official specification are:

- **Data**: Describes the geometry of the path.
- **Fill**: Describes the brush used to paint the geometry specified by the Data property of the path.
- **Stroke**: Specifies the brush used to draw the stroke.
- **StrokeThickness**: Specifies the thickness of a stroke, in units of the effective coordinate space[1].
- **StrokeLineJoin**: Specifies how a stroke is drawn at a corner of a path. Valid values are *Miter*, *Bevel*, and *Round*.
- **StrokeStartLineCap**: Defines the shape of the beginning of the first dash in a stroke. Valid values are *Flat*, *Square*, *Round*, and *Triangle*.
- **StrokeEndLineCap**: Defines the shape of the end of the last dash in a stroke. Valid values are *Flat*, *Square*, *Round*, and *Triangle*.

The highlighted attributes do not cover all the ones defined for Glyphs and Path elements, but they constitute the most used. For further explanations, please see [16].

3) *Canvas*: they are used to group together a set of `Glyphs` or `Path` nodes. Such a grouping operation can be used to identify the grouped elements as a unit (for example in order to identify a hyperlink destination) or to apply a composed property value to each child of the `Canvas` node. The original idea of canvas was thus to group elements based on format similarity rather than semantically.
In addition to page description information, XPS may provide a set of metadata which enriches the XPS file with semantic information about the contents of the file itself. This

feature is absolutely unique to the page description languages we analysed so far. Currently, XPS files can either be created from within the Microsoft Office suite by means of the dedicated *XPS Plug-in creator*, or from any application through the XPS virtual printer. Still, while the XPS Plug-in creator stores all the semantic information associated to the document in the XPS file, this information is completely absent in files generated with the virtual printer.

XPS was chosen as the PDL to be used within our project for several reasons:

- The images needed by the XPS rendering process are stored directly into the XPS archive in their original format without the need of extra processing as is the case for a PDF file. The fonts are also stored inside the XPS archive, however the first time they are accessed they need to be de-obfuscated in order to be read. We will see in the next chapter how this process actually takes place.
- The XPS zip package can be extended with custom files or updated with the replacement of existing files. We took advantage of this feature for the insertion of semantic information into the XPS file: a detailed explanation of this point will follow in chapter 6.
- The XML files describing the content of the XPS file are human readable and can be easily edited with a simple text editor.

5.3 Anoto

Anoto [6] is one of the pioneering leaders of digital pen and paper technology. In collaboration with well-known enterprises like Sony-Ericsson [36], Nokia [37], Logitech [38], Hewlett-Packard [39] and others, Anoto delivered easy-to-use, digital communication based on advanced digital techniques that bridge the gap between writing and computing. In the last year they also began to produce their own Anoto-branded hardware.
There are two important components in the Anoto technology: a digital pen, produced by the Anoto's partners mentioned above, and a particular paper technology. This technology consists of a sheet of ordinary paper on which a proprietary pattern of dots has been printed. Since the spacing between the dots is of only 0.3 mm, the human eye perceives the pattern as a slightly off-white surface. An example of the pattern in its original size and magnified 19 times may be seen in Figure 5.4

The digital pens are equipped with a micro camera which captures snapshots of the dotted pattern. Light is emitted by a built-in infrared (IR) LED, reflected by the paper surface and captured by the camera. Since the Anoto dot pattern is printed with IR-absorbing black toner, the dots absorb the infrared light and the pattern will appear as white dots on the captured camera images. If the amount of black toner particles used in the printed document exceeds a certain limit, the printing will interfere with the pattern and the camera is no longer able to detect the dots on paper. This means that no black colour should be used in areas that are to be interactive.

Thanks to this technology, a record of the movements of the pen can be used to recreate in the digital world what the user has written on paper. Figure 5.5 shows the features

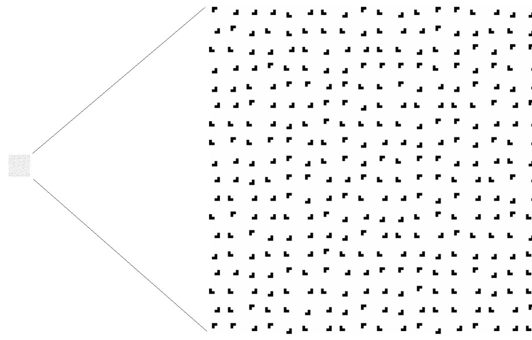


Figure 5.4: Anoto Pattern

of an Anoto enabled-pen. Beside the infrared camera cited above, the pen provides also a processor, able to elaborate the inputs coming from the camera, a memory slot where information are stored in the case the pen is used in batch mode and a rechargeable battery which enables to use the pen wherever you want.
Thanks to the Bluetooth transceiver (which is not mandatory), some digital pens have the potential to be used for direct interaction: indeed, they are able to transmit the position information continuously (in streaming-mode) and not only in batch mode.

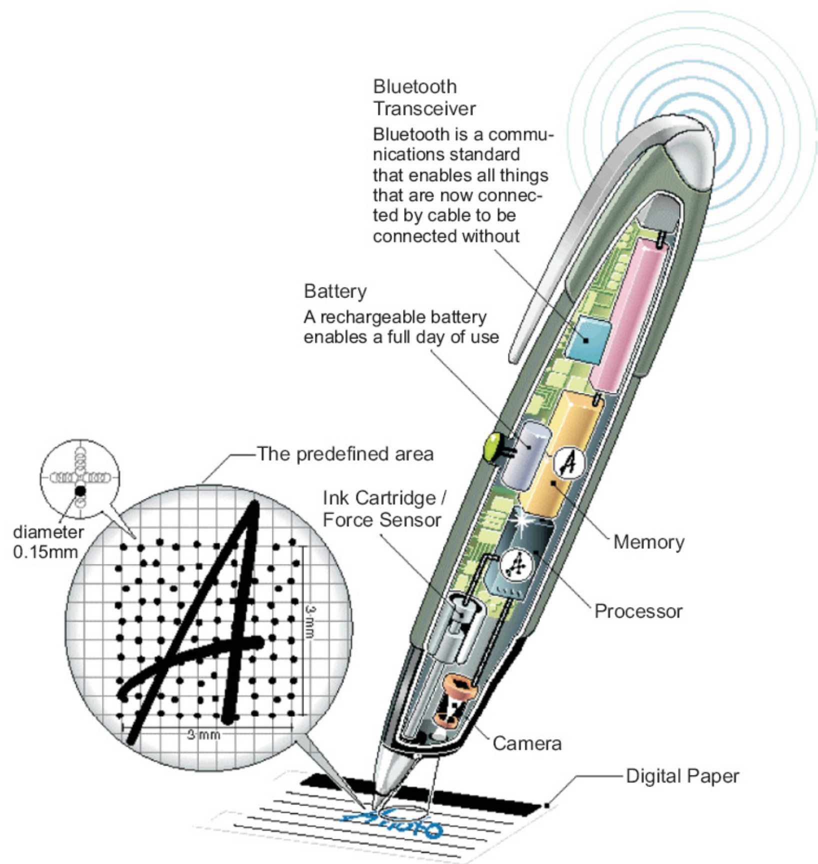


Figure 5.5: Anoto-Enabled Pen

6
Design and Implementation

As thoroughly discussed in the previous chapters, this project aims at delivering tools for bridging the gap between the physical and digital instances of a document. In terms of applications we particularly focus in the retrieval of elements marked on a printed instance within the digital document. Many of the existing tools already allow for the interaction between paper and digital elements, but they do not currently support access to the semantic information associated to the document, limiting their interest to the physical information that they retrieve from paper.

The goal of this work is to extend the capabilities of the existing tools, making use also of the semantic information associated to the document. Given a position on paper and depending on the level of granularity, there are several elements that can be retrieved within the digital document (e.g. words, characters). The semantic information augments the physical information by allowing the navigation from a single element to all the elements semantically connected to it, without making use of any additional physical information coming from paper.
According to the existing infrastructure, the project has been divided into three main sections: (i) the implementation of an iPublish plug-in for OpenOffice, (ii) the design and implementation of an iDoc extension that uses XPS files to retrieve elements in the original digital document, and (iii) the creation of an exemplar application that exploits the benefits of the implemented code.

iPublish plug-in

Thanks to the OpenOffice SDK [33] it is possible to get programmatic access to all the features provided by OpenOffice. The choice of OpenOffice as authoring tool prevented us to use the Microsoft Office plug-in for the creation of the XPS files. As discussed in the previous chapter, this means that we completely lacked the semantic information about the document's contents. In order to store such information, we introduced the concept of an

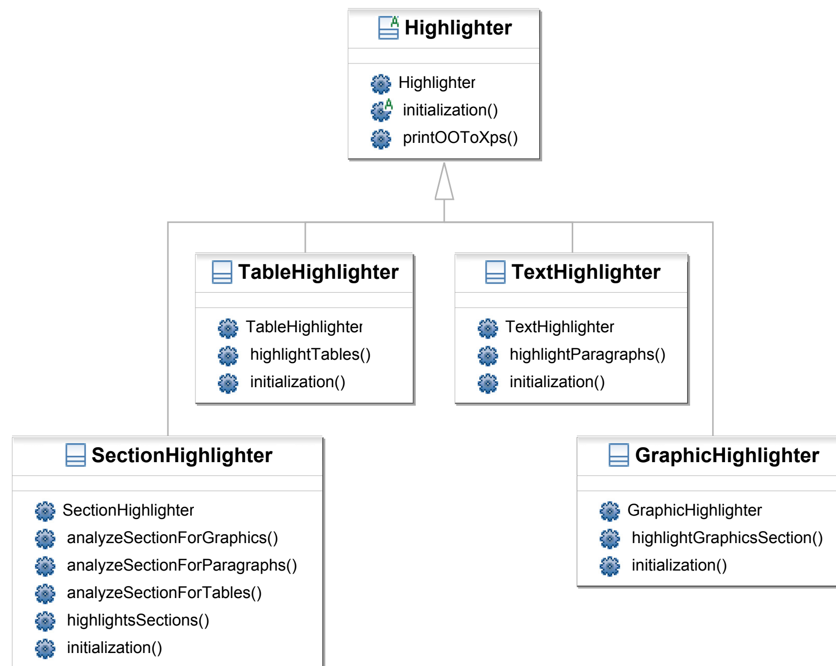


Figure 6.1: Highlighter Hierarchy

Highlighter. The idea behind the Highlighter is to use shapes as semantic containers and draw them within the OpenOffice document along with an ID string which labels them. There are several types of Highlighter, depending on the semantic element they have to enclose (e.g. Section, Table, Graphic and Text). In practice once the structural elements (sections, paragraphs, etc.) have been retrieved from the document tree, their representation is "highlighted". Once the highlighting process is completed the document is printed to an XPS file. This allows the extraction of physical information about the shapes themselves, meaning that it is therefore possible to define a physical shape for the semantic elements of the digital file.

The Highlighting process was implemented by means of an abstract class `Highlighter` that is responsible first for the activation of the connection with the OpenOffice instance and later for the printing of the document to an XPS file. To highlight the different elements, different highlighters have been developed: `SectionHighlighter`, `TableHighlighter`, `GraphicHighlighter` and `TextHighlighter`. These specific Highlighters provide an implementation of the abstract method `initialization()` declared within the `Highlighter` base class and additional specific methods for the execution of the highlighting process. Figure 6.1 outlines the UML schema of the highlighters.
In order to better understand the concept of the highlighters, Figure 6.2 shows the result of the execution of the `highlightParagraphs()` method within the `TextHighlighter` class. Each paragraph of a section has been enclosed by a polygon responsible of "bundling together" all the words that are contained within the same paragraph. For completeness of

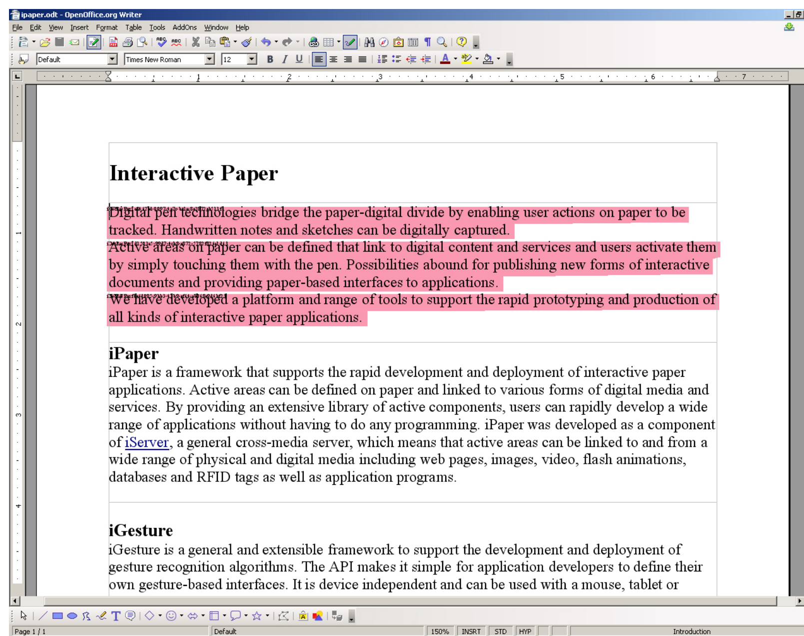


Figure 6.2: A Highlighted OpenOffice Document

information, we mention that the drawing operation was wrapped by means of the `Drawing` class, which allows for the insertion of rectangular and polygonal shapes. In the next section we will look at how this enclosing shape is used for extracting semantic information.
At the end of the semantic enrichment ("highlighting") the OpenOffice document is finally printed by means of the Microsoft XPS virtual printer and a XPS file is generated.

iDoc extension

There are two main components in the iDoc extension: the *SnippetExtractor* and the *Semantic Mapper*. Given an highlighted XPS file, the SnippetExtractor is responsible for tracking all the highlighting shapes, extract physical information about their position and annotate the XPS content with semantic information. The SnippetExtractor works on the following semantic levels: Section, Paragraph, Table and Graphic. The process workflow is summarised next:

1. Create an empty *references.xml* file inside the XPS archive.

2. For each highlighted shape, create a corresponding XML node (`Section`, `Paragraph`, `Table` or `Graphic` node) into the *references.xml* file and add a child element *Shape* containing an XML description of the surface occupied by the shape.

3. For each `Glyphs` and `Path` node of the XPS file, extract the physical information and check if they reside within a highlighted shape. In the case of a `Canvas` node, extract its children and do not consider the enclosing canvas anymore.

4. For every `Glyphs` and `Path` node which reside within the highlighted shape, create an *XInclude* node as sub element of the node representing the highlighted shape.

The *XInclude* node follows the XInclude W3C, Version 1.0, recommendations [40]. Basically it is a link to an external XML element which is part of another XML document. In order to retrieve the element within the source document *XPath* expressions are used. Using such an approach, it is possible to reduce the amount of duplicated information within the XPS document, since the original XML nodes are not replicated in the *references.xml* file, but only linked. This approach needs the usage of particular XML parsers able to support this linking system. There exist several ones which are advanced enough to support this functionality. Among them, we chose the *Xerces* parser, version 2.9.0 [41].
If the OpenOffice document contains sections, the outlined process is run twice. Once for annotating the sections and the second time for annotating tables, graphics and paragraphs, enabling this way a double level of hierarchy, as shown in Figure 6.3.

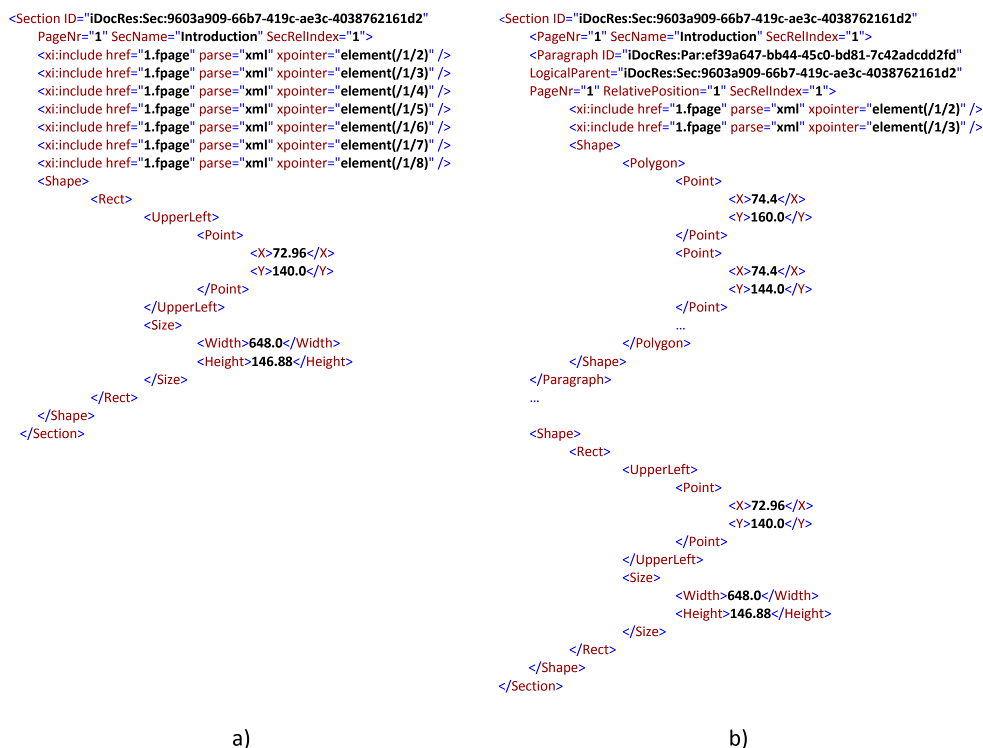


Figure 6.3: a) `Section` and b) `Paragraph` Nodes

The `ID` attribute is a unique identifier for all structural elements. The `PageNr` attribute represents the page where the element begins, `SecName` is a specific Section attribute and contains its name as specified within the authoring tool, while `SecRelIndex` is an index used in the case that a section is splitted on more than one page. In this case more highlighting shapes are drawn for the same section. They have all the same ID and the index is used in order to keep them in the right sequence once the addition of paragraph, table and graphic children is performed.
The `Paragraph` node shows two more attributes, the `LogicalParent`, which contains the ID of the Section enclosing it, and the `RelativePosition`, which defines the position of the Paragraph with respect to the parent.

The second main component of the iDoc extension is the Semantic Mapper. We defined all the relevant semantic elements as Java classes that extend a base class `Element`. This class contains all the basic fields and methods common to all the element types. Figure 6.4 shows the UML representation of the `Element` class and its subclasses.

Figure 6.4: Elements Hierarchy

Semantically, these elements build a hierarchical structure. The interaction is ensured by the method `getElements()` which computes the `Elements` composing the current element and stores them in a vector of sub elements, as shown in Figure 6.5.

The implementation of `getElements()` differs among the different classes. In the case of a section, `getElements()` first calls the iPublish plug-in in order to highlight the elements contained within the section itself. The document is printed and the XPS file is generated. Then the XPS file is parsed, the XML representations of the sub elements are retrieved and the corresponding Java objects are created.

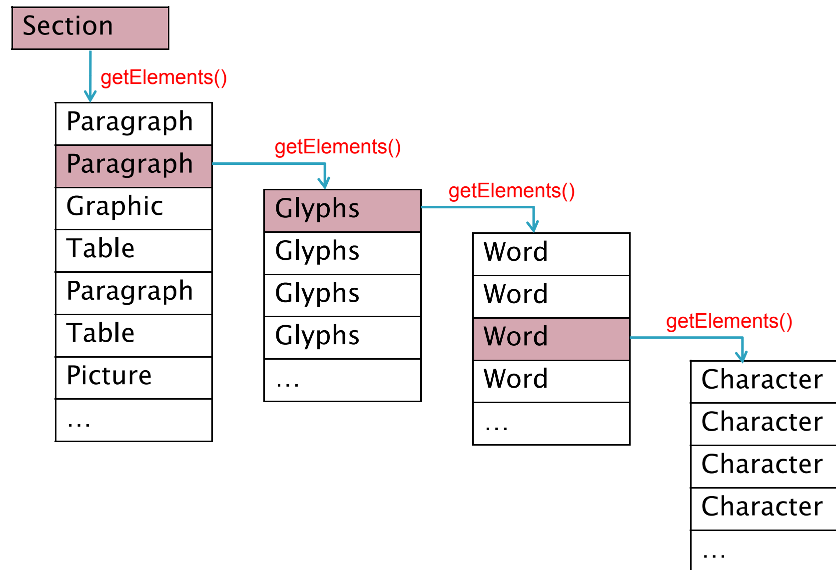


Figure 6.5: The Sub Elements

For graphical objects, the `getElements()` method does not make sense, since an image does not contain any further element other than the image itself. This practically means that its implementation is empty. For tables, the implementation is currently missing. This means that we currently get the object `Table` containing the `Path` and `Glyphs` elements from the section's `getElements()` method, but we are currently not able to analyse it further. For `Paragraphs` elements, `getElements()` returns a list of `Words`. To be precise, the method first creates `Glyphs` objects according to the `<Glyphs>` nodes contained in the XML representation of the paragraph within the XPS file, and then for each of these `Glyphs`, a further call to the corresponding `getElements()` method is executed. This two-step operation is necessary since the `Glyphs` objects do not have a semantic meaning, but they just represent a run of uniformly-formatted text from a single font, and nothing more. The sub elements of a `Paragraph` are hence the words composing it.
The UML diagram of Figure 6.4 shows methods available within the `Element` class. Most of them are just getter and setter methods, which parse the XML representation of the element itself and return or set the corresponding values. The implementing sub classes provide specific implementation of some of these methods in the case that they need a particular behaviour which differs from the general one. The other important method, beside `getElements()`, is `getShape()`. This method computes the shape of the current element and stores it as a `Shape` object, following the same convention as within the iServer framework.

```
<Glyphs
Fill="#ffff0000" FontRenderingEmSize="16.0006"
FontUri="/Documents/1/Resources/Fonts/493B8DF4-19BD-4088-A175-03717D006BB3.odttf"
Indices="36,73;79,27;69;72,45;85;87;3;40;76;81;86;87;72;76;81" OriginX="123.04" OriginY="164.16"
StyleSimulations="None" UnicodeString="Albert Einstein" />
```

Output: Albert Einstein

Figure 6.6: The XML Representation of a `Glyphs` node

In order to compute the position and the shape of the words of a paragraph, several operations have to be computed. Let us start by analysing how the information is stored within the `Glyphs` nodes composing the XML representation of a Paragraph object. For a better understanding the Glyphs structure already outlined in chapter 5 is proposed once again in Figure 6.6.
The node stores the physical coordinates of the beginning of the Glyphs, defined as `OriginX` and `OriginY`, the `URI` of the font used to render the text, the content of the Glyphs as `UnicodeString`, and a list of `Indices`. For each character composing the Glyphs, a list of up to four comma-separated parameters is stored in the `Indices` attribute. A semi-colon represents the separation of the single characters, indicating that the following attributes are now referring to the next character. The first value contains the index of the character, useful for retrieving it within a font file. The second index expresses the *advance width* of the character, which defines where the next character has to be drawn with respect to the current one. By using the advance width values and starting from the first character, it is therefore possible to compute the x position of each following character. The third and fourth indices allow the adjustment of the character position along the x- and y-axis respectively.

If the advance width is not specified in the Indices list, this information has to be extracted from the font file specified by the `FontUri` path. However, the XPS generation process encrypts the first 32 bytes of the font data in order to prevent misappropriation of the embedded fonts. This obfuscation mechanism prevents users from using standard ZIP utilities to extract fonts from the XPS files and install them on their system. In order to open and access font data, a decryption of the font has to be executed and a specific key is needed for every font. The key needed for this operation can be obtained directly from the font name. Indeed, a font is stored within an XPS document with a 128-bit number string. Considering each byte of the 128 bit as a single element represented as two hexadecimal digits, the name of the font can be viewed as a list of 16 single elements. The key is then simply the inverse of the list, starting from the last element and adding each time the previous one. A XOR operation between the key and the first 32 bytes of the obfuscated font leads to an un-obfuscated file, which can be freely accessed. Figure 6.7 shows a summary of the deobfuscation process.
Once the font has been deobfuscated, the advance width can be read and the position of the next character computed. Using the same information, it is obviously also possible to compute the position of each word, considering the blank spaces of the `UnicodeString` as stop points.

- First 32 bytes of the font are obfuscated...
- How to get the key for the deobfuscation?

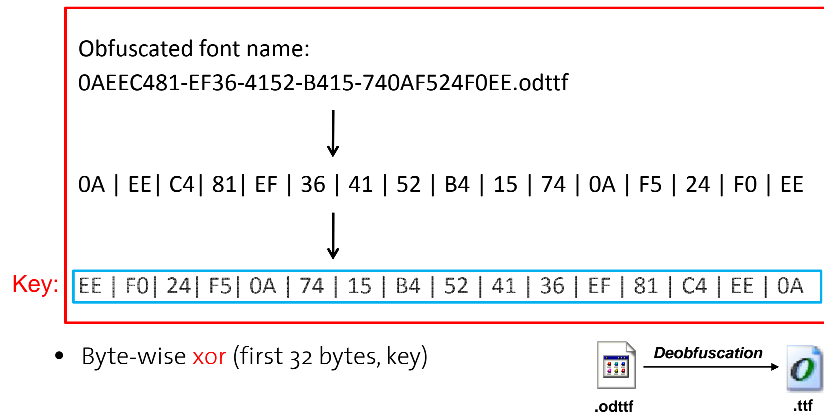


Figure 6.7: Deobfuscation Process

The computation of the origin `X` of a character is obtained with the formula shown in Equation 6.1:

X = Origin of the character
pC = The previous character
pX = The `OriginX` of the previous character
pAW = The `AdvanceWidth` of the previous character
$pFRES$ = The `FontRenderingEmSize` of the previous character

$$X = pX + \Delta(pAW) \tag{6.1}$$

where Δ is defined as follows if pAW is known:

$$\Delta(pAW) = 0.01 * pFRES * pAW \tag{6.2}$$

or as follows if pAW is not yet known:

$$\Delta(pC) = 0.01 * pFRES * font.getWidth(pC) * 0.1 \tag{6.3}$$

At this point, we need one more piece of information to fully characterise the shape containing characters and words: the height of the shape itself. This value depends on the height of the characters within the words, and again this has to be extracted from the font file. In the case of a word (that contains several characters), the maximal height among all characters is chosen.

Many technologies exist for the extraction of information from a font file. We decided to use iText [42], an open source Java library created to generate PDF on the fly. Even if we were not interested in the creation of PDF, this library provides a simple interface for accessing all the common information about single characters in a font file.

From paper back to digital

Two operations provide the link to the digital instance of the printed document enabling the retrieval of elements from the repository, and storing the newly analysed elements in the case that a further analysis is needed. These operations are currently implemented in two main classes, `XPSElementFinder` and `ReferencesPublisher`, that perform the retrieval and publishing of elements from and to an XPS repository. These classes implement the abstract `findElements()` and `publish()` methods declared in the base abstract classes `ElementFinder` and `Publisher`, respectively. These abstract classes have been introduced in order to enable the addition of *finding* and *publishing* operations in the case that repositories different from XPS will be used. In the near future, for example, the XPS repository will be substituted by the iDoc document database mentioned in chapter 3.2. In this case, after the highlighting process and the parsing of the XPS, the created Java objects will be directly inserted into the database, and any additional computations will rely on the data retrieved from it, allowing for a much faster process with respect to the parsing of the *references.xml* file. Once the iDoc database will be connected to the system, also retrieving an element selected on paper will be much faster. Linking the iDoc database to the existing implementation will require the implementation of the specific `DatabaseFinder` and `DatabasePublisher` classes, as outlined in Figure 6.8 and Figure 6.9.
Additionally, we provided also an implementation of the `ElementFinder` abstract class for normal, non-analysed XPS files. In this case, the only semantic categories available are *Word* and *Character*. This implementation allows us to deal also with normal XPS files not coming from OpenOffice. However, in this case we are only able to track words and characters and no semantic objects of higher granularity.

The method `findElements()` implemented in the `XPSElementFinder` class returns one or more `Element` objects, depending on the elements selected on paper. The selection may rely on different gestures a user performs on paper to understand the a priori undefined intentions of the user (e.g. select all the elements above, all the intersecting elements, all the nearest elements, etc.). Moreover, the system does not know if the user wants to highlight an entire word or only some of its characters. We therefore implemented a simple algorithm for such a task. Starting from the highest granularity level (section), the system first check how many elements are involved with the user's gesture. In case that more than one element is found, it means that the selection is done at the current granularity level and therefore the system returns all of them. If only one element is found, it might be that the wanted granularity level is lower than the current one and the lower granularity elements (e.g paragraphs, tables and graphics) are investigated. This process goes on, recursively, until one of the following cases takes places:

1. no elements are found: the system goes back at the older granularity level and returns the parent element.

2. all elements are found: all the investigated sub elements are involved with the operation, the system goes back at the older granularity level and returns the parent element.

3. more than one element is retrieved: the system is as the right granularity level and it returns a collection containing these elements.

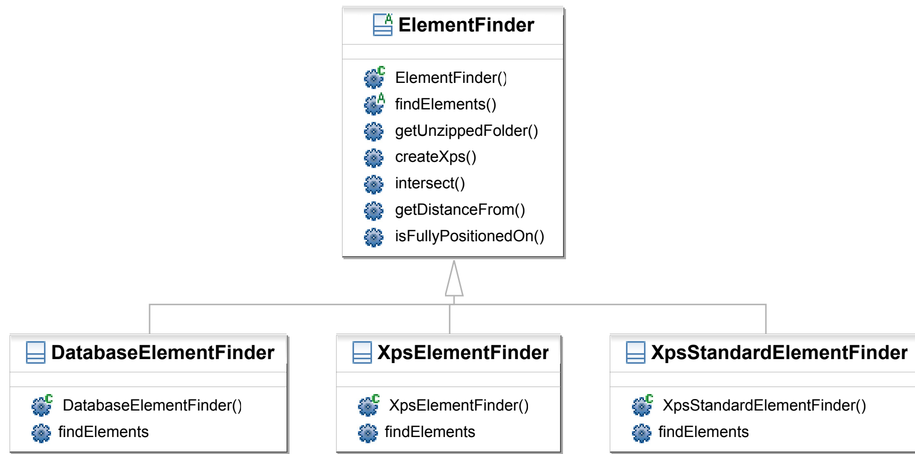


Figure 6.8: Finder Classes

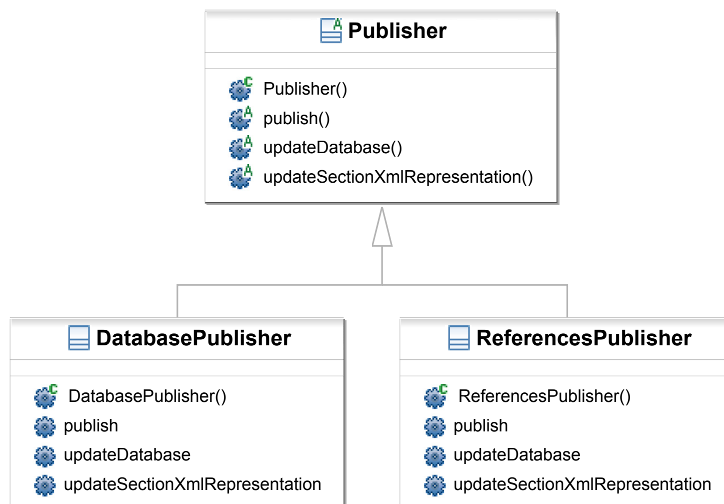


Figure 6.9: Publisher Classes

4. the smallest granularity (character) is reached and the system returns a collection containing one or more characters.

The flow diagram of Figure 6.10 illustrates the algorithm graphically.

The retrieval of the single elements from the digital document has to be general enough to be adapted to every authoring tool. For this reason, we decided to use a very simple method consisting in counting them: when the elements are published within the XPS database, a reference to their parent (`LogicalParent` attribute) and the relative position with respect to their parent (`RelativePosition` attribute) are inserted along with all the information about their content and physical position (see Figure 6.3). This additional information is then used in the retrieval phase to find the right elements.
In order to better explain the retrieval process, let us propose a practical example. With a digital pen we select a word on the printed document and we want to get access to the corresponding digital element within the authoring tool. Starting from the physical coordinates received from the pen, the `ElementFinder` retrieves from the database the elements which are involved with these coordinates. As explained above, these elements contain a reference to the parent element, in this case a paragraph, that contains it. With the same mechanism, the system retrieves the section that is the parent element of the paragraph.
The second phase runs in the opposite direction. The counting feature is implemented in the `AdapterXPSOO` class (see Figure 6.11): given one or more elements retrieved from the database, the `getOOEntity()` method retrieves the corresponding element(s) within the authoring tool. Using the example above, the section found in the first phase is retrieved within the authoring tool. Making use of the relative position of its children, the right paragraph is also retrieved. From the origin of the paragraph, the system counts until it reaches the position of the desired word and highlights it.
Figure 6.12 shows the retrieval process, outlining the different components of the Semantic Mapper.
If the document does not contain any sections, the process does not change with the difference that the element with the highest granularity would be of type `Paragraph` instead of `Section`.

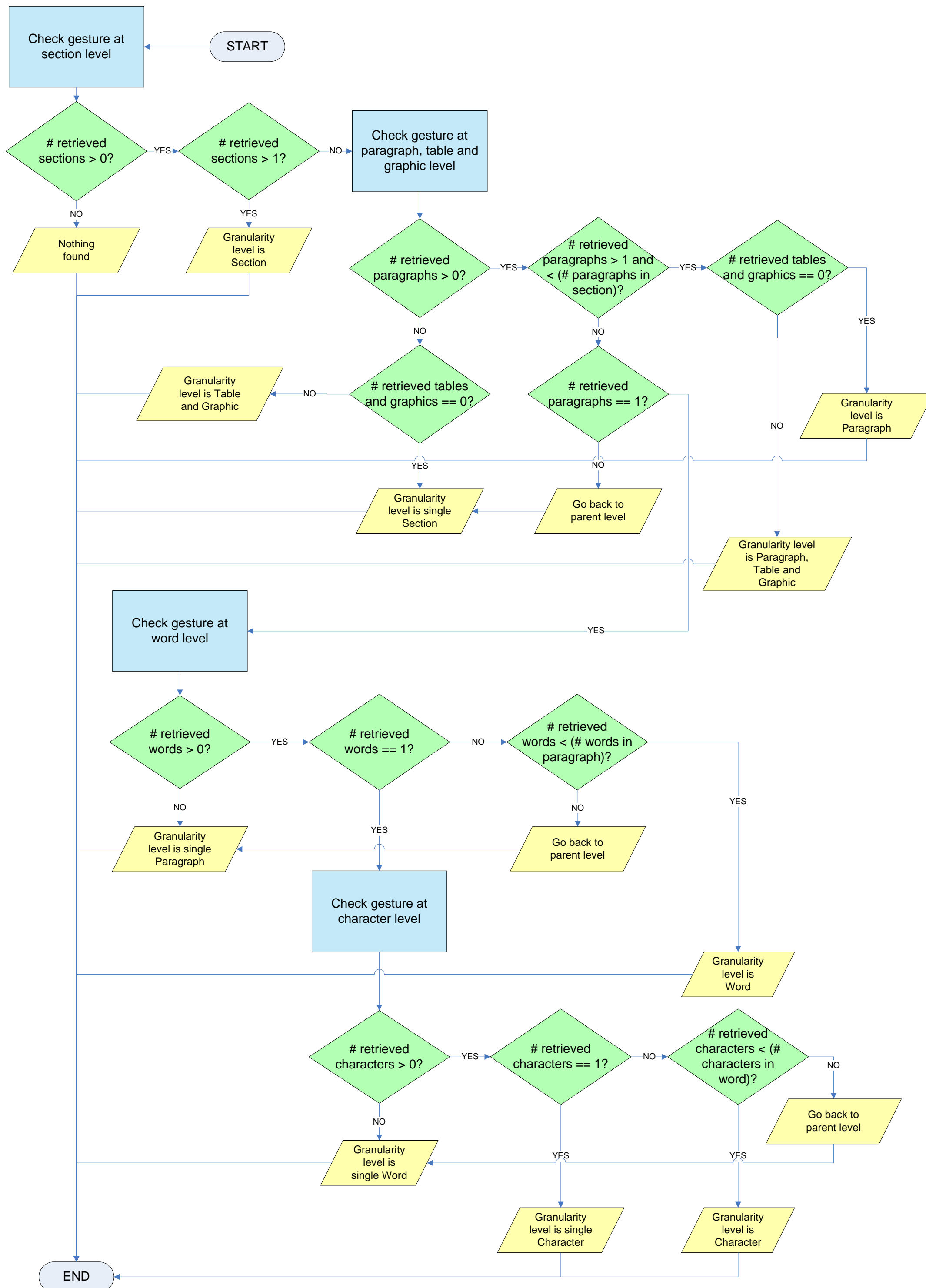


Figure 6.10: Automatic Granularity Processing

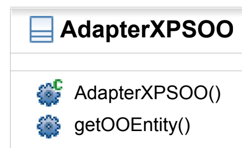


Figure 6.11: AdapterXPSOO

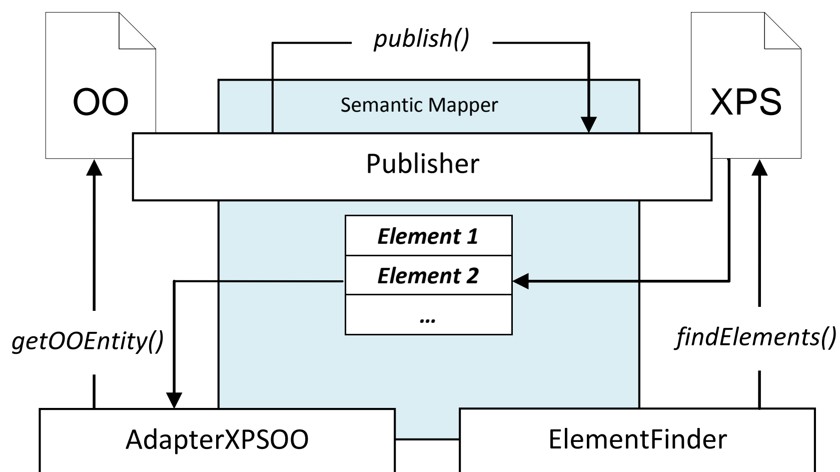


Figure 6.12: Retrieval Process

7 Exemplar Application: PaperProof

OpenOffice provides a way for tracking changes within a document. Whenever this functionality is enabled, any change to the existing text does not immediately replace the existing content, but changes are considered as tentative operations that can later be individually accepted or rejected, either by the same user or by others, in the case of collaborative drafting of the document.

This chapter presents an exemplar application that makes use of the OpenOffice tracking changes features along with the system implemented during this project and the other technologies introduced in chapter 5. We called this application *PaperProof*. *PaperProof* allows the translation of the actions a user performs on a printed document into the OpenOffice authoring tool. Since the tracking changes feature of OpenOffice is able to deal with different authors, our application can be easily extended to a multi-authors correction system. The document should be printed several times and each copy distributed to a different author. Digital pens have unique IDs that allow the system to assign the corrections to the different authors that performed them.

7.1 Operations

PaperProof accepts the following five operations:

- Insert
- Delete
- Replace
- Move
- Annotate

The iGesture framework (introduced in chapter 3.5), enables the system to recognise the intentions of the user according to the gestures registered by the digital pen during the editing process of the printed document. The ICR (Intelligent Character Recognition) software translates the information written by hand into a digital counterpart, which is used as a further input for some of the operations.
Figure 7.1 highlights the available operations, along with the sequence of needed input gestures.

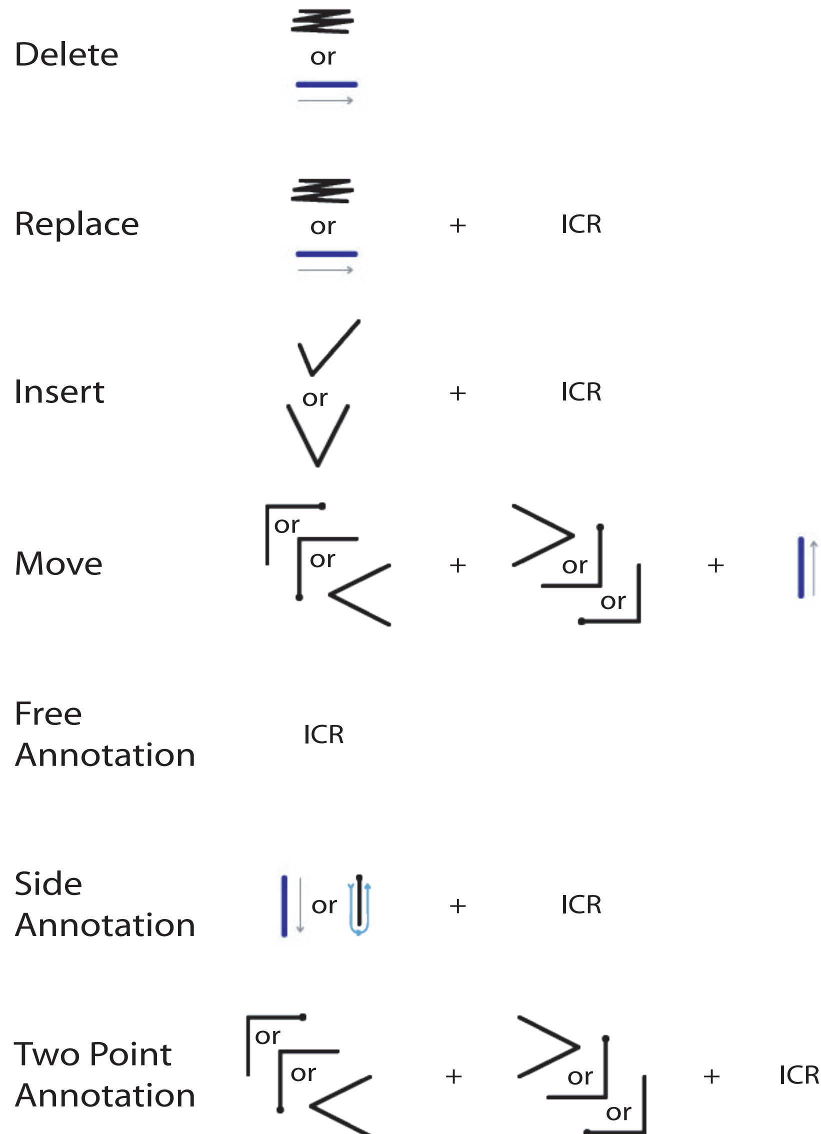


Figure 7.1: Operations and Related Gestures

Please note that there are three possible *Annotate* operations, depending on what the user wants to do.

The next section will analyse the implementation of the available operations.

7.2 Implementation

Every operation that a user executes on paper with the digital pen is classified as a *Correction*. Each correction corresponds to an instance of one of the Java classes that extend the basic `TrackChanger` class.
There exist four types of TrackChangers, as shown in Figure 7.2. The four classes, `TrackChangerGraphics`, `TrackChangerSections`, `TrackChangerTables` and `TrackChangerText`, define theirs operations depending on the type of OpenOffice element that they describe.

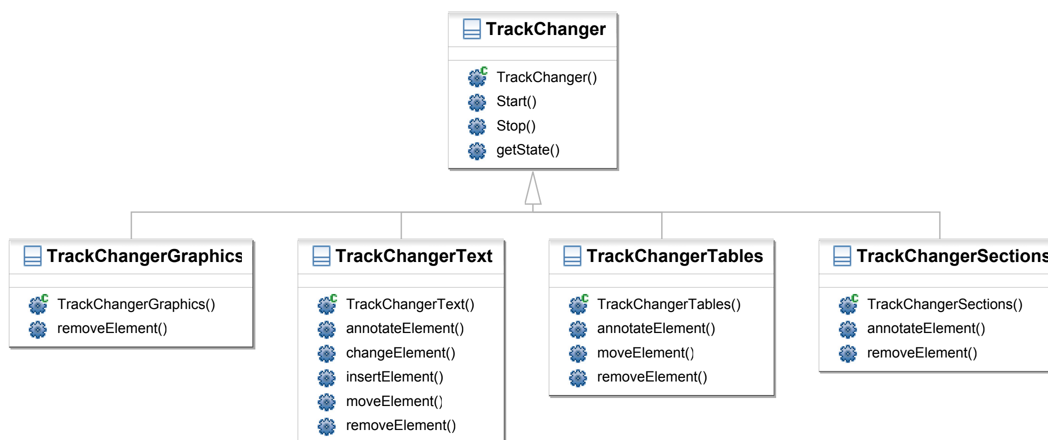


Figure 7.2: TrackChangers Hierarchy

As shown in the diagram, depending on the type of element that the system is dealing with, the operations have different behaviours:

- Sections
  - `annotateElement(String textNote)`: this section is annotated with the given textNote.
  - `removeElement()`: all the contents of this section are removed from the document.

- Graphics
  - `removeElement()`: this graphic element is removed from the document (the operation is not handled by the track changes functionality of OpenOffice).

- Tables
  - `annotateElement(String textNote)`: this table is annotated with the given textNote (the annotation is inserted within the first cell of the table).
  - `moveElement(Element destination)`: this table is moved after the given destination element.
  - `removeElement()`: this table is removed from the document; in order to satisfy

the requirements of the track changes feature, all the contents of the table are deleted, instead of the table itself.

- Text (paragraph, word or character)
  - `annotateElement(String textNote)`: this text element is annotated with the given textNote.
  - `changeElement(String newText)`: this text element is substituted by the given newText.
  - `insertElement(String newText)`: the newText is inserted *after* this text element.
  - `moveElement(Element destination)`: this text is moved after the given *destination* element.
  - `removeElement()`: this text element is removed from the document.

The correct operation is executed depending on the type of Element the users selected and the type of gesture they performed. As already explained in the previous chapter, the `AdapterXPSOO` class is provided for the definition of the granularity and the interpretation of the user intentions: once the `AdapterXPSOO` has established the granularity of the element(s) that the user has selected with the digital pen, the corresponding `TrackChanger` type is instantiated. Then the right operations are called depending on the gestures, and the corrections are automatically executed within OpenOffice.

Since users can make as many corrections as they want, it is important to ensure that an operation does not influence the following operations, resulting then in erroneous corrections. In order to prevent this situation, we moved the call to the method responsible of finding the right element(s) within OpenOffice into the constructor of each `TrackChanger` type. This way, instantiating a new `TrackChanger` automatically stores a reference to the right OpenOffice element. Moreover, before executing any corrections, we first instantiate all the TrackChangers needed for the execution of all the corrections. Also in the case of several authors, all their TrackChangers get instantiated at the same time. This means that from the very beginning, all the elements of the document that have to be corrected are properly referenced. In this way, even if they are moved by a previous operation, the corresponding TrackChangers are still able to retrieve them. We know that we can safely execute all the corrections without any undesired dependency.

The starting point of the whole application is the class `Corrections`. The constructor shown in Figure 7.3, indicates that instances of this class need four attributes: a vector of *operations*, containing all the corrections which have to be performed, the path and the version of the OpenOffice document which has to be corrected and the path of the XPS file that is used as repository.
The `Corrections` class is the connection between the existing infrastructure, presented in chapter 3, that handles the input sent by the Anoto pen during a proof-reading session, and the infrastructure implemented in this project: from the `ElementFinder` class, responsible of extracting information from the repository about the selected printed elements, through the `AdapterXPSOO`, which returns the corresponding OpenOffice digital elements, up to the `TrackChanger`, that eventually transforms the user intentions in real operations.
The whole computation is launched in the `execute()` method highlighted in Figure 7.4,

```
public Corrections(Vector<Operation> operations, String odtFilePath,
                          int versionNumber, String xpsFilePath) {

        this.operations = operations;
        this.versionNumber = versionNumber;

        this.xpsFilePath = xpsFilePath;
        this.odtFilePath = pdtFilePath;

        this.execute();

}
```

Figure 7.3: The `Corrections` Constructor

where only some lines of code are highlighted to give a better insight into the sequence of operations that are effectively performed. The green comments should clarify the meaning of the code.

```java
public synchronized void execute() throws OOException {

        for (Operation op : operations) {

                // Retrieve the elements from the XPS repository, where xps_x, xps_y,
                // xps_width and xps_height are positional parameters taken from
                // the digital pen's input.
                Vector<Element> results = finder.findElements(
                        op.getPageNr(), xps_x, xps_y, xps_width, xps_height);
                …

                // Depending on the granularity of the found elements, the right
                // TrackChanger is instantiated. Here, for example, a TrackChangerText
                // object is created. As arguments it takes a connection to the
                // OpenOffice document, the Vector of elements to be corrected and an
                // instance of the AdapterXPSOO which is used to get the OpenOffice
                // elements within the document.
                TrackChanger trackchanger = new TrackChangerText(connection, results, adapter);

                // The trackchanger is added to the vector of pending corrections, by means
                // of an instance of the class StoredCorrection. This class builds
                // <trackchanger, operation> pairs.
                allCorrections.add(new StoredCorrection(trackchanger, op));

                // At the end, all the operations are executed, depending on their type.
                for (int i = 0; i < allCorrections.size(); i++) {

                        // The current trackchanger and the current operation are taken from the
                        // vector of corrections.
                        TrackChanger trackchanger = allCorrections.get(i).getTrackchanger();
                        Operation operation = allCorrections.get(i).getOperation();

                        // Depending on the type of trackchanger and the type of operation, the
                        // right correction is eventually executed.
                        if (trackchanger instanceof TrackChangerText)
                        {
                                if (operation.getName().name().equals(OPERATION.REPLACE.name()))
                                {
                                        // Execute the correction (replace).
                                        ((TrackChangerText) trackchanger)
                                                .changeElement(((Replace) operation).getText());
                                }
                                …
                        }
                        …

                }
        }
}
```

Figure 7.4: The `execute()` Method

8 Results and Discussion

In this work, we presented a general infrastructure that aims at bridging the gap between the physical and digital instances of a document. The proposed approach allows users to move back and forth between these two realities. In chapter 7 we highlighted an exemplar application, called PaperProof, that is based on the proposed infrastructure. In order to discuss our approach, in the following we analyse a real-world PaperProof session.

8.1 PaperProof in use

The process starts within OpenOffice, where the digital document is authored (Figure 8.1). After that the document is printed on paper and it is possible to begin the proof-reading session.

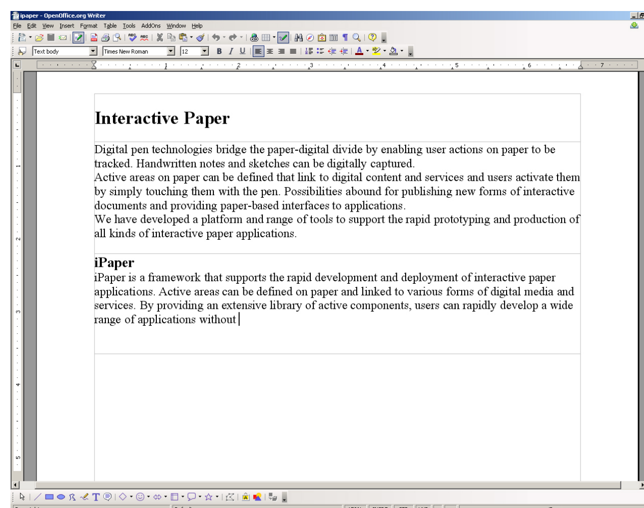


Figure 8.1: The Authoring Phase

Figure 8.2 shows an example of free form annotations and corrections performed by a user on the printed version of the OpenOffice document.

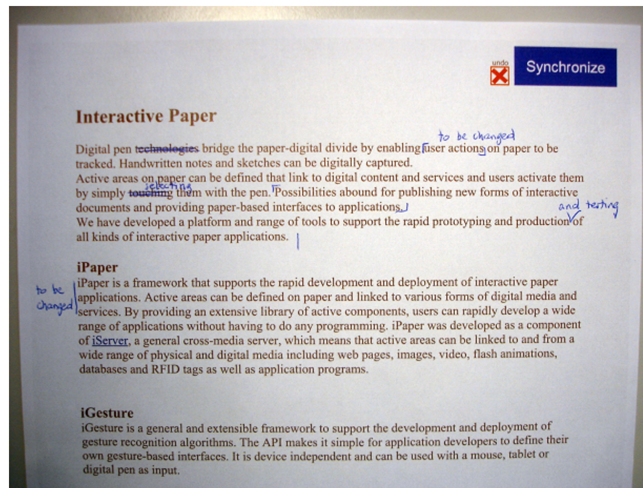


Figure 8.2: Free Form Annotations on a Paper Document

First the user performs the corrections and annotates the paper document. Every operation is inserted into a queue waiting to be transferred to OpenOffice. When the user is finally ready to transfer the corrections, he touches the “Synchronize” button in the top right part of the printed document with the digital pen. This causes the corrections to be transferred to the PaperProof application that transforms them into digital actions.

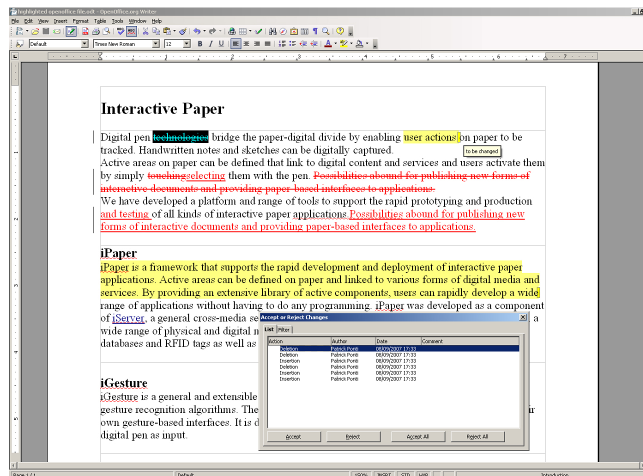
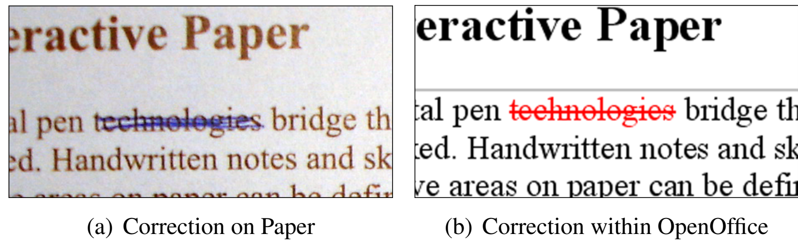


Figure 8.3: PaperProof Corrections within OpenOffice

Figure 8.3 shows the corresponding OpenOffice document after the execution of the corrections. Once the process has been finished, the ”Accept or Reject Changes” dialog is automatically opened, allowing the users to decide whether the corrections have to be applied or discarded. This dialog is useful also for multi-authors corrections, since it lists the name of the Author as well as the Date of execution.

We now discuss the operations executed on this document.

1. Delete: “technologies”.

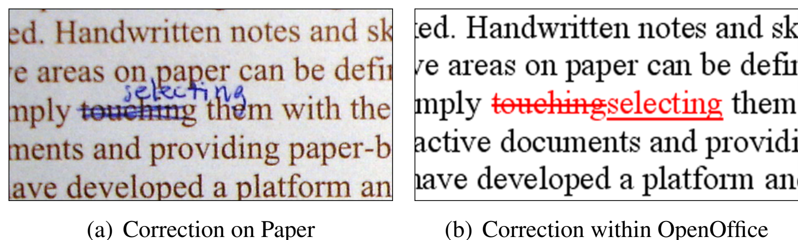


(a) Correction on Paper (b) Correction within OpenOffice

Figure 8.4: Delete Operation

The Delete operation is the simplest one, since it requires a single gesture (scratch-out, chapter 7 for details). At the end of the gesture, a timer starts. If a user begins to write something within a programmable timeout (currently 4 seconds), the Delete operation is transformed into a Replace operation; if instead the timer expires or another operation is executed, the Delete operation is inserted into the pool of pending operations.

2. Replace: “touching” with “selecting”.

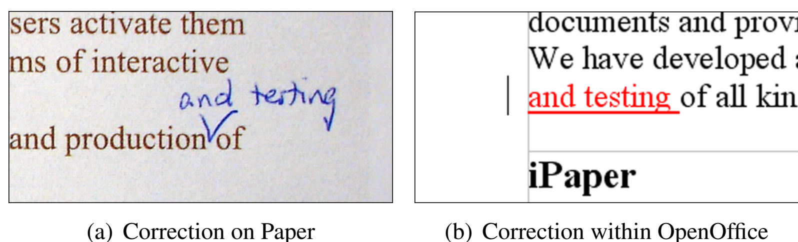


(a) Correction on Paper (b) Correction within OpenOffice

Figure 8.5: Replace Operation

In this case, a writing operation immediately followed a delete operation. In the “Accept or Reject Changes” dialog, the Replace operation appears as the deletion of “touching” and the insertion of “selecting”.

3. Insert: “and testing” after “production”.



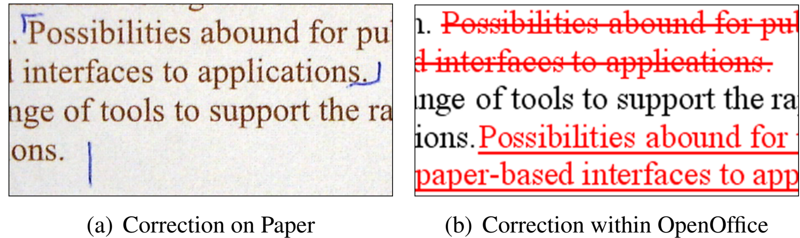
(a) Correction on Paper (b) Correction within OpenOffice

Figure 8.6: Insert Operation

The Insert operation can be executed at every text granularity level. If the gesture intersects an element, the Insert operation is executed immediately after it. In the case

that the gesture does not intersect any characters, words or paragraphs, the section is selected and the Insert operation adds text at the end of the last paragraph of the section.

4. Move: “Possibilities abound for publishing new forms of interactive documents and providing paper-based interfaces to applications.” to the end of section.

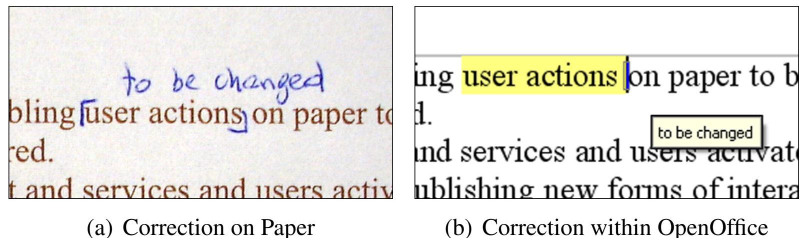


(a) Correction on Paper (b) Correction within OpenOffice

Figure 8.7: Move Operation

The Move operation is executed as a cut-and-paste operation within OpenOffice. This has the advantage of preserving the formatting of the cut text (e.g. **this formatting** would be preserved at the new position). In the same way as for Replace, the Move operation is represented in the dialog of operations as a deletion followed by an insertion.

5. Two Point Annotation: “user actions” with “to be changed”.



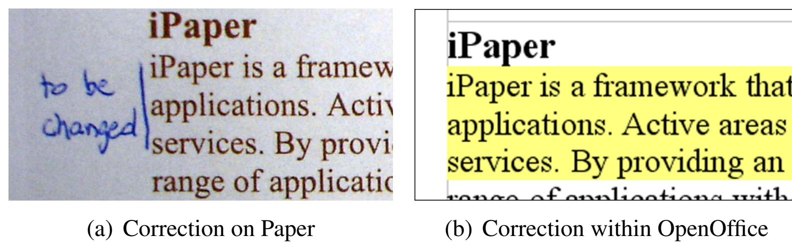
(a) Correction on Paper (b) Correction within OpenOffice

Figure 8.8: Two Point Annotation Operation

The Annotate operation is not listed within the dialog, since it does not change any texts. The annotated element’s background is made yellow and the note is added at the end of the highlighted region. Since there is not yet support for annotations within OpenOffice, in order to manage the annotations (i.e. to insert new annotations or delete the existing ones), we installed a very simple plug-in for OpenOffice developed by Christoph Jopp [43]. Thanks to this tool, the annotations inserted with PaperProof can then be managed in a simple way.

6. Side Annotation

   The Side Annotation is performed by drawing a vertical line on the left of the elements which have to be annotated. All what is located on the right of the line is annotated with the text recognised by the ICR software.



(a) Correction on Paper (b) Correction within OpenOffice

Figure 8.9: Side Annotation Operation

At the current state of development, PaperProof operates in a satisfactory way. If the users write or draw on paper in a clean and clear way when they correct and annotate the printed document, the operations are executed with a good reliability. Still, the lack of accuracy can easily lead to unexpected results. If for example, while scratching out a word the pen accidentally touches the next word, this also gets deleted, according to the automatic granularity explained in chapter 6. This is the consequence of the fact that users' intentions are not known a priori and therefore the system should be able to decide how to behave. What does it mean if users scratch all the characters of a word, but not the last one? Have they only been unprecise or they really wanted to remove only those characters? The algorithm for the automatic granularity detection tries to interpret the intentions of the users, resulting in some cases in a wrong interpretation.

The PaperProof gestures have been selected in a way that should feel natural to the end user. Still, only when the application has been used extensively by several users, it will be clear whether it provides a good interface. For this reason, PaperProof should be subjected to several user studies, in order to establish the effective user-friendliness of its interface. Indeed, absolutely no user studies have been performed on PaperProof so far.

8.2 Infrastructure

PaperProof successfully validated our infrastructure. The purpose of creating a bridge between the two instances of a document, not only at a physical level but also using the associated semantic features, has been achieved. The retrieval of digital elements starting from the corresponding elements on paper works well, even though it is still not optimal in terms of speed. This is probably due to the current implementation which uses the XPS file also as the elements repository. This is costly in terms of performance, since parsing an XML file and looking for an element within it is a rather slow operation. Moving to the database provided by the iDoc framework, which is planned for the near future, should make the retrieval of elements faster and consequently speed up all the applications based on this infrastructure, like for example PaperProof.
In conclusion, in this work we created an infrastructure that links the digital and physical instances of a document and showed that it works reliably. We implemented a framework on top of which applications like PaperProof may be implemented.

The retrieval of elements uses the relative position with respect to the parent elements containing them. This is a generic approach which can be used also in the development of

other plug-ins (like for example a PaperProof plug-in for Microsoft Word). If we indeed consider MS Word as a target for the development of a new PaperProof application, the future implementation might be based completely on the existing infrastructure. Moreover, the implementation of the iPublish plug-in, would be very simple, since the creation of an XPS file starting form a MS document through the MS Office 2007 plug-in for XPS already provides all the semantic information that our infrastructure needs. Therefore, the only parts that have to be significantly changed are the PaperProof application, to be compatible with the MS Word SDK, and the iDoc extension, with the implementation of a proper SnippetExtractor.

Of course there are many possibilities to improve the current system. In the next section, we will analyse some more points of the infrastructure that still can be improved and propose some possible developments for the near future.

9 Future Work

The existing implementation may be divided into three main components: (i) The iPublish plug-in, identifying the semantic information within the authoring tool; (ii) the iDoc extension, responsible for the storage of semantic and physical information and the retrieval of digital elements; (iii) the PaperProof application, transferring annotations and corrections on paper back in the original authoring tool. Future work in all of these three areas might range from simple improvements of the current implementation to the development of complete new features.

Here, we present a list of limitations of the current implementation ordered by the component to which they refer. This inventory can be taken as an incentive for the improvement of the existing infrastructure.

9.1 iPublish plug-in

- Since every shape drawn by the plug-in must provide an ID, which is also drawn within the shape, the highlighting shapes cannot be arbitrarily small. A minimum size is required to allow the shape to contain the ID which labels it.
- The section highlighting process is page-based, meaning that the width of the highlighting shape of a section is exactly the same as the width of the page. If the sections of a document span several columns, the `SectionsHighlighter` fails to recognise them. In the current implementation, in the case of multi-column documents, the semantic analysis must skip the section level and starts directly at the paragraph level.
- If the width of a table is not explicitly specified, meaning that the table takes the width of the element that contains it, the `TablesHighlighter` considers the width of the table equal to the width of the page. As in the previous point, this can cause problems if the document contains columns and the table is in one of them.

- The internal analysis of the tables has not been implemented. The existing code is able to highlight a table, but not to analyse its content. Thus, the elements contained within the table can not be resolved as single elements. Selecting any of them will make the PaperProof application select the entire table.
- The `TablesHighlighter` cannot highlight tables that span more than one page.
- Some kinds of bullets of OpenOffice are misinterpreted when the document is printed with the XPS virtual printer. The resulting XML code contains characters that are not accepted by standard XML editors.
- The MS XPS virtual printer fails also to print OpenOffice documents, if they contains several pictures. The resulting XPS pages contain only the pictures without the text.
- Within OpenOffice, several properties are defined for a paragraph. One of them is called "below-spacing" and represents the space that OpenOffice keeps before rendering the next paragraph. If the value of the "below-spacing" property is larger than 0, the `TextHighlighter` will not be able to understand the exact end position of the paragraph, resulting in an incorrect highlighting shape.

We should spend a few words for one last consideration on the iPublish plug-in. OpenOffice - as any other authoring tool - offers a very large palette of functionalities and therefore allows for a huge variability in the shape and content of digital documents that finally will have to be processed by the plug-in. The task of rendering the plug-in robust against such a large unpredictability is surely not trivial.

9.2 iDoc extension

- The existing code needs the OpenOffice document to contain sections as the root elements from which digital elements with finer granularity can be retrieved. Fixing this limitation will be straighforward, since the code has been designed to be able to start at finer granularity, and the limitation lies only in the implementation.
- When the original OpenOffice file is printed, the printing process is captured by the iPublish plug-in that starts the analysis for the definition of the semantic information. At the end of this process, the document is printed to an XPS file that is subsequently annotated by this iDoc extension, which enriches it with the semantic information of the elements with larger granularity (section elements). This operation has not yet been integrated into the whole infrastructure, meaning that the printing operation and the following XPS annotation have still to be launched manually.
- During the XPS annotation, the check for containment of a graphic element into a section uses the position of its upper left corner. It would be better to consider its centre instead, or even better, its centre of mass, since it can be composed by many graphics sub elements combined.
- The XPS annotation causes the XPS file to grow in size. When a section is analysed for its sub elements, a new temporary XPS file containing the annotation of the sub elements is created. This file contains the same fonts as the original one, but the XPS

creation process computes each time different names for the contained fonts. When the XPS snippets of the sub elements are embedded in the original file, the newly created font files are copied into the original file to avoid problems with the fontURI attributes. This leads to an increase of the file size. A possible solution would be to change the fontURIs of the new annotated elements according to the name of the font files of the original XPS file.

9.3 PaperProof

- Usability studies should be performed to improve the user interface of PaperProof. Feedback from the users might suggest that new gestures would be needed or that some of the existing ones should be corrected or replaced.

Acknowledgments

I would like to thank my supervisor Nadir Weibel for his unlimited support during the whole project. Furthermore, I want to thank Prof. Moira C. Norrie for giving me the opportunity to accomplish my Master Project in her group.

Bibliography

[1] A. J. Sellen and R. Harper. *The Mith of the Paperless Office*. MIT Press, November 2001.

[2] Global Information Systems Group. http://www.globis.ethz.ch/.

[3] Moira C. Norrie and Beat Signer. Information Server for Highly-Connected Cross-Media Publishing. *Information Systems*, 30:526–542, 2005.

[4] Moira C. Norrie, Beat Signer, and Nadir Weibel. General Framework for the Rapid Development of Interactive Paper Applications. In *Workshop on Collaborating over Paper and Digital Documents*, Banff, Canada, November 2006.

[5] Nadir Weibel, Moira C. Norrie, and Beat Signer. A Model for Mapping between Printed and Digital Document Instances. In *Proceedings of the 2007 ACM Symposium on Document Engineering*, 2007.

[6] Anoto AB. http://www.anoto.com/.

[7] ProofRite. http://www.cs.umd.edu/hcil/proofrite/.

[8] Kevin Conroy, Dave Levin, and Françqis Guimbretière. ProofRite: A Paper-Augmented Word Processor. In *Demo Session of UIST 2004, 17th Annual ACM Symposium on User Interface Software and Technology*, Santa Fe, USA, October 2004.

[9] XLibris. http://www.fxpal.com/?p=XLibris/.

[10] Gene Golovchinsky and Laurent Denoue. Moving Markup: Repositioning Freeform Annotations. In *Proceedings of UIST 2002, 15th Annual ACM Symposium on User Interface Software and Technology*, Paris, France, October 2002.

[11] EdFest. http://www.edfest.ethz.ch/.

[12] Moira C. Norrie. Paper on the Move. In *Workshop on Ubiquitous Mobile Information and Collaboration Systems (UMICS), CAiSE 2004, 16th International Conference on Advanced Information Systems Engineering*, Riga, Latvia, June 2004.

[13] BBC - The BluePlanet TV Series. http://www.bbc.co.uk/nature/programmes/tv/blueplanet/.

[14] Beat Signer, Moira C. Norrie, Nadir Weibel. Print-n-Link: Weaving the Paper Web. In *Proceedings of DocEng 2006, ACM Symposium on Document Engineering*, Amsterdam, The Netherlands, October 2006.

[15] Adobe Systems inc. *PDF Reference, Adobe Portable Document Format*. 2006.

[16] Microsoft XPS. http://www.microsoft.com/whdc/xps/default.mspx.

[17] Scalable Vector Graphics. http://www.w3.org/Graphics/SVG/.

[18] Beat Signer. *Fundamental Concepts for Interactive Paper and Cross-Media Information Spaces*. PhD thesis, ETH Zurich, Switzerland, 2006. Dissertation, ETH No. 16218.

[19] iGesture. http://www.igesture.org/.

[20] MyScript. http://www.visionobjects.com/.

[21] OpenOffice.org. http://www.openoffice.org/.

[22] Sun Microsystems. http://www.sun.com/.

[23] Oasis OpenDocument Format. http://www.oasis-open.org/.

[24] GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html.

[25] Microsoft Office. http://office.microsoft.com/.

[26] Microsoft Word. http://office.microsoft.com/word/.

[27] Microsoft Excel. http://office.microsoft.com/excel/.

[28] Microsoft PowerPoint. http://office.microsoft.com/powerpoint/.

[29] Adobe Flash. http://www.adobe.com/.

[30] Microsoft Access. http://office.microsoft.com/access/.

[31] Corel Draw. http://www.corel.com/.

[32] Microsoft Publisher. http://office.microsoft.com/publisher/.

[33] OpenOffice.org SDK. http://www.openoffice.org/dev_docs/source/sdk/.

[34] Adobe Systems inc. *PostScript Language Reference, Third Edition*.

[35] Reverse Polish Notation. http://mathworld.wolfram.com/ReversePolishNotation.html.

[36] Sony-Ericsson. http://www.sony-ericsson.com/.

[37] Nokia. http://www.nokia.com/.

[38] Logitech. http://www.logitech.com/.

[39] Hewlett-Packard. http://www.hp.com/.

[40] XInclude W3C Recommendation. http://www.w3.org/TR/xinclude/.

[41] Xerces XML Parser. http://xerces.apache.org/xerces2-j/.

[42] iText. http://www.lowagie.com/iText/.

[43] Christoph Jopp. OpenOffice Annotations Plug-In. http://en.ooo-info.org/documentation/annotation_tool.html.

[44] Michael Grossniklaus, Moira C. Norrie, Beat Signer, and Nadir Weibel. Producing Interactive Paper Documents based on Multi-Channel Content Publishing. In *Proc. of AXMEDIS 2007, 3rd International Conference on Automated Production of Cross Media Content for Multi-channel Distribution,*, Barcelona, Spain, November 2007.