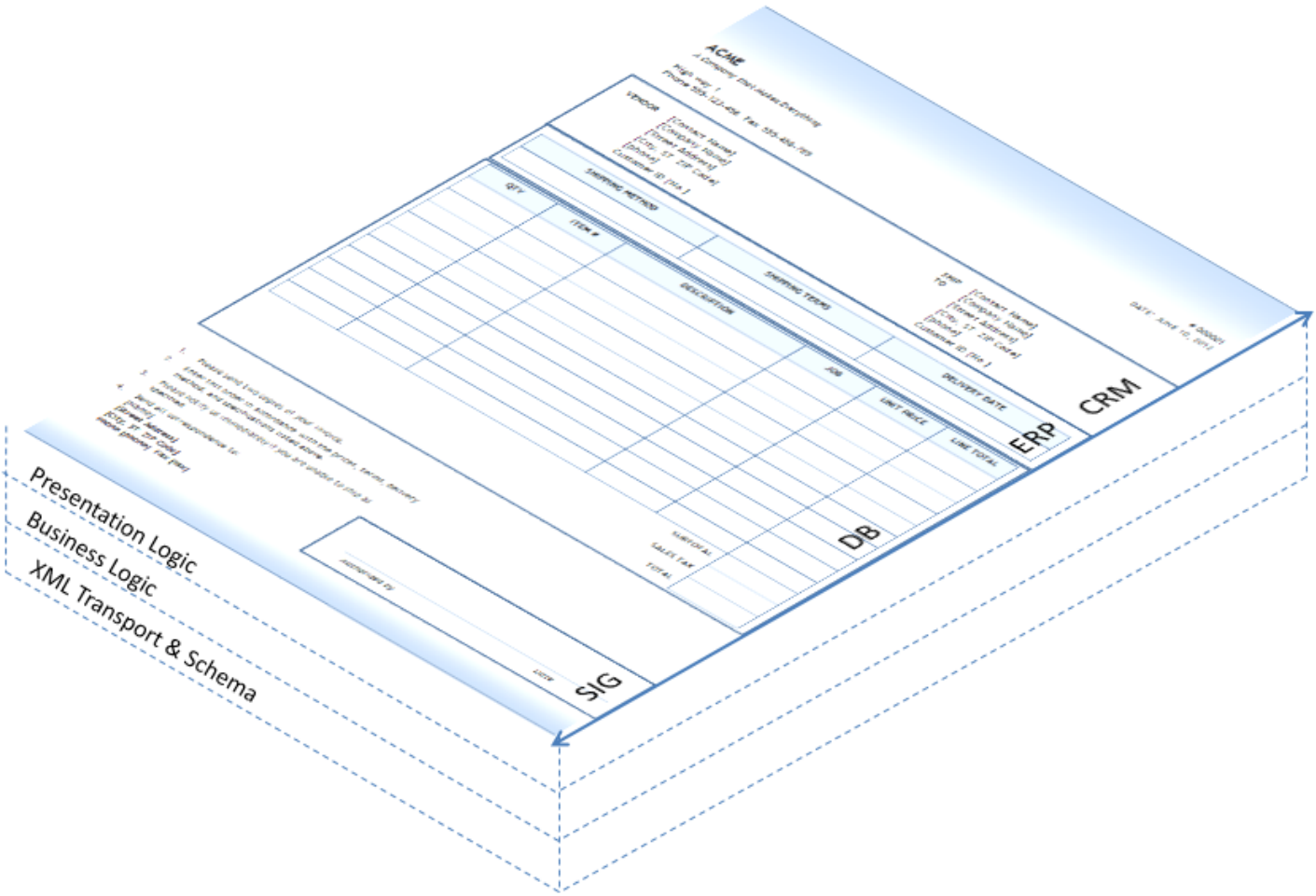
ZUGFeRD

The Future of Invoicing



by **Bruno Lowagie**

ZUGFeRD: the future of invoicing

iText Software

This book is for sale at http://leanpub.com/zugferdthefutureofinvoicing

This version was published on 2017-06-21

Contents

Introduction

Whenever I receive an invoice as a PDF document in my Inbox, I take a look at the *Document Properties* of the file to find out which tool was used to create that invoice. I'm always happy when I see that iText was used.

Next I check if the PDF is future-proof. That is: if it complies with the PDF/A standard, and if it can easily be interpreted by a machine. Usually that's not the case. That makes me less happy.

Only when I've performed these two simple checks, I look at the actual content of the invoice. Whether or not *that* makes me happy, depends on the amount I have to pay.

In this tutorial,

- I'll explain why conforming to the PDF/A standard is important,
- I'll show you how you can assure that a machine can read and process the invoices you create, and
- I'll introduce the Central User Guide for Electronic Invoicing in Germany (ZUGFeRD), a standard that was developed to meet these requirements.

Using some simple examples, I'll demonstrate how you can easily create ZUGFeRD-compliant invoices by applying some small changes to your iText-driven invoicing process.

1. ZUGFeRD: the core concepts

ZUGFeRD wasn't created out of the blue. It builds on top of existing standards that describe the Portable Document Format (PDF) and Electronic Data Interchange (EDI). Let's take a look at these standards before we dive into the specifics of the ZUGFeRD standard.

The Portable Document Format

In 2008, the PDF specification was published as an ISO standard: **ISO 32000-1**. This wasn't the first ISO standard for PDF. Figure 1.1 shows that there's an umbrella of PDF standards, each having its own specific purpose, often in the context of a specific sector or industry. ISO 32000 was written as the core standard that is used as the basis for all the sub-standards under this umbrella.

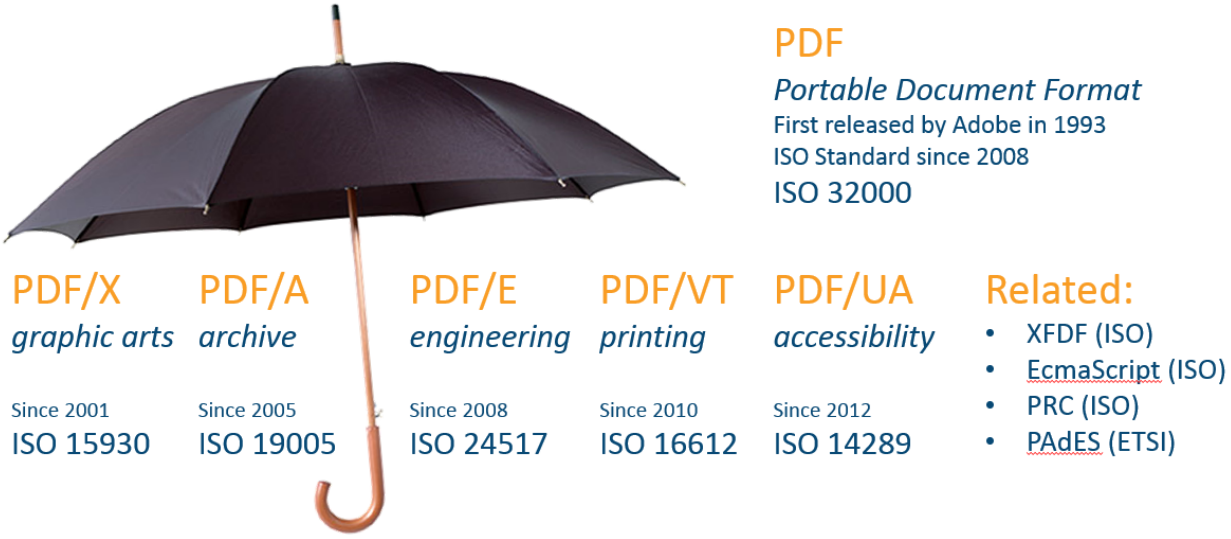


Figure 1.1: PDF, an umbrella of standards

The Portable Document Format was originally designed by Adobe. The first version of the specification was released in 1993. New versions were published on a regular basis, adding more and more functionality. ISO 32000-1 was based on version 1.7 (2006) of Adobe's PDF specification.

Due to the breadth of the PDF specification, developers of PDF software have a lot of freedom when creating a PDF document. This results in many different flavors of PDF. For instance: to reduce file size, it's possible to make a document depend on external fonts. However, this can result in an illegible or odd-looking page if those fonts are missing on the end user's device. A PDF document can contain JavaScript which can update PDF form fields in specific circumstances. Clearly these are some PDF "features" to be avoided in the context of finalized invoices.

PDF/A: long-term preservation of documents

ISO 19005, or PDF/A, was originally developed to meet long-term archival needs. Part 1 was released in 2005. It was defined as a subset of version 1.4 of Adobe's PDF specification (which, at that time, wasn't an ISO standard yet). It introduced a series of obligations and restrictions:

- *The document needs to be self-contained:* all fonts need to be embedded; external movie, sound or other binary files are not allowed.
- *The document needs to contain metadata in the eXtensible Metadata Platform (XMP) format:* ISO 16684 (XMP) describes how to embed XML metadata into a binary file, so that software that doesn't know how to interpret the binary data format can still extract the file's metadata.
- *Functionality that isn't future-proof isn't allowed:* the PDF can't contain any JavaScript and may not be encrypted.

From the start, it was determined that approved parts of ISO 19005 could never become invalid. New, subsequent parts would only define new, useful features.

ISO 19005-1:2005 (PDF/A-1) defined two conformance levels:

- *Level B ("basic"):* ensures that the visual appearance of a document will be preserved for the long term.
- *Level A ("accessible"):* ensures that the visual appearance of a document will be preserved for the long term, but also introduces structural and semantic properties. The PDF needs to be a *Tagged PDF*.

ISO 19005-2:2011 (PDF/A-2) was introduced to have a PDF/A standard that was based on the ISO standard (ISO 32000-1) instead of on Adobe's PDF specification. PDF/A-2 also adds a handful of features that were introduced in PDF 1.5, 1.6 and 1.7:

- *Useful additions include:* support for JPEG2000, Collections, object-level XMP, and optional content.
- *Useful improvements include:* better support for transparency, comment types and annotations, and digital signatures.

PDF/A-2 also defines an extra level besides Level A and Level B:

- *Level U ("Unicode"):* ensures that the visual appearance of a document will be preserved for the long term, and that all text is stored in UNICODE.

ISO 19005-3:2012 (PDF/A-3) was an almost identical copy of PDF/A-2 (even the typos were copied). There was only one difference with PDF/A-2: in PDF/A-3, attachments don't need to be PDF/A. You can attach any file to a PDF/A-3 document, for instance: an XLS file containing calculations of which the results are used in the document, the original Word document that was used to create the PDF document, and so on. The document itself needs to conform to all the obligations and restrictions of the PDF/A specification, but these obligations and restrictions do not apply to its attachments.

PDF as a format for invoices

All the qualities of the PDF/A standard are also highly desirable for invoices. Alas, PDF/A alone doesn't solve the problem of processing invoices (yet). Few PDF invoices today are structured; they lack the information that is required for automatic extraction of key data such as the amount due, addresses, taxes, wiring information, and so on.

If the PDF is Tagged, you already get some information about the semantics of the different content elements in the document. You could apply some artificial intelligence to detect the subtotal, the taxes and the grand total. But this will only work to a certain extend. Tagged PDF wasn't designed for this purpose. The process will never be flawless, especially if Optical Character Recognition (OCR) is involved. Numbers can easily be misinterpreted: a zero can be read as the letter O, a 5 can be scanned as a 6; It's an error-prone process in an area where there's little tolerance for errors.

Ideally, we'd create our invoice as a PDF/A-3 invoice and attach a document that contains all the necessary data in a format that allows a machine to interpret the invoice without any human intervention. To find a format that meets this requirement, let's take a look at how large corporations exchange data.

Electronic Data Interchange

For decades, large corporations have used EDI to exchange information with each other in the form of structured data that is transmitted without (or with minimum) manual intervention. This required bilateral arrangements between the companies, defining which data is to be exchanged and which format is to be used. Implementing EDI isn't trivial. In the case of small and medium businesses, the low volume of transactions doesn't justify the cost of putting in place an EDI system. Several standardization organizations have tried to reduce this cost by introducing industry standards.

Electronic Business eXtensible Markup Language

In 1999, the United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) and the organization for the Advancement of Structured Information Standards (OASIS) started an initiative that resulted in a suite of standards that were approved by the International Organization for Standardization (ISO) in 2004. They were released under the general title **ISO 15000: Electronic Business eXtensible Markup Language (ebXML)**.

This standard enables enterprises in any industry, of any size, anywhere in the world to conduct business over the internet. Originally, ISO 15000 consisted of these four parts:

- **ISO 15000-1:2004:** ebXML Collaborative Partner Profile Agreement
- **ISO 15000-2:2004:** ebXML Messaging Service Specification
- **ISO 15000-3:2004:** ebXML Registry Information Model
- **ISO 15000-4:2004:** ebXML Registry Services Specification

The goal for these standards was to make EDI less expensive and less difficult to implement, by providing companies with a standard method to exchange business messages, conduct trading relationships, communicate data in common terms and define and register business processes.

OASIS and UN/CEFACT also developed a common set of semantic building blocks that represent general types of business data. Existing business vocabularies were restructured and new business vocabularies were created. These were published in a fifth part of the ISO standard **ISO 15000-5:2005, the ebXML Core Components Technical Specification (CCTS)**.

Uniform Business Language and the Core Components Library

OASIS then went on to produce a data format in full conformance with the CCTS: the **Universal Business Language (UBL)**. UBL became the foundation of a number of successful international frameworks such as ePrior, PEPPOL, and many other specifications. It's an XML only specification of which the data model is not normative.

UN/CEFACT released several versions of a Core Components Library (CCL) based on ISO 15000-5. A CCL is a repository of easily reused generic business data components. It provides templates discribing postal adresses, tax information, payment information, and so on. UBL uses Core Components, but it's an XML only specification. The Core Components of the CCL are syntax-independent. They can be used to create syntax solutions other than XML.

In 2014, ISO released an update of part 5 of ISO 15000: **ISO 15000-5:2014 ebXML Core Components Specification (CCS)**. Unlike UBL, this specification is normative and syntax neutral.

The Cross Industry Invoice standard

ISO 15000-5:2014 and the CCL were used by UN/CEFACT as the basis for specific business document models, such as the Cross Industry Order (CIO), the Cross Industry Order Response (CIOR), the Cross Industry Invoice (CII), and so on.

The goal of these uniform, standardized models is to permit the exchange of data electronically in a syntax-independent, interoperable way, without any human intervention. Using the CII standard, companies can easily process any number of invoices in an automated way, based on mutual agreements on which data should be shared, and in which form, for instance using XML.

At the European level, the European Committee for Standardization (CEN) derived different Message User Guides (MUG) from these standards, such as the Core Invoice Data Model MUG which is a subset, derived from the CII standard. CEN Workshop Agreement (CWA) documents CWA 16356-1, -2 and -3 describe the setup, content and data structures of a minimum scope in the context of sending invoice data. The Core Invoice Data Model defines about 100 field types related to an invoice.

Importance of the evolution of EDI standards

Electronic Data Interchange has been standardized and simplified in such a way that it is no longer impossible for small and medium businesses to implement. This has been recognized by the German Forum for Electronic Invoicing (FeRD) who used the CII and CEN's Message User Guides as the basis for the ZUGFeRD Model.

Zentraler User Guide des Forums elektronische Rechnung Deutschland

The "Forum elektronische Rechnung Deutschland" (FeRD) is a German platform of associations, ministries and companies promoting electronic invoicing. In 2014, FeRD published the Central User Guide for Electronic Invoicing (ZUGFeRD) as a new standard for invoicing. This standard consists of the *ZUGFeRD data model* describing which contents make up an invoice (the semantics) and the *ZUGFeRD format* describing how these contents will be transferred.

The ZUGFeRD Data Model

For the data model, ZUGFeRD uses the CCI standard and CEN's Message User Guides to which the UN/CEFACT Naming and Design Rules (NDR) are applied, resulting in the ZUGFeRD XML schema. Every invoice needs to contain an XML file that validates against this schema.

Figure 1.2 shows the three different profiles that are supported.

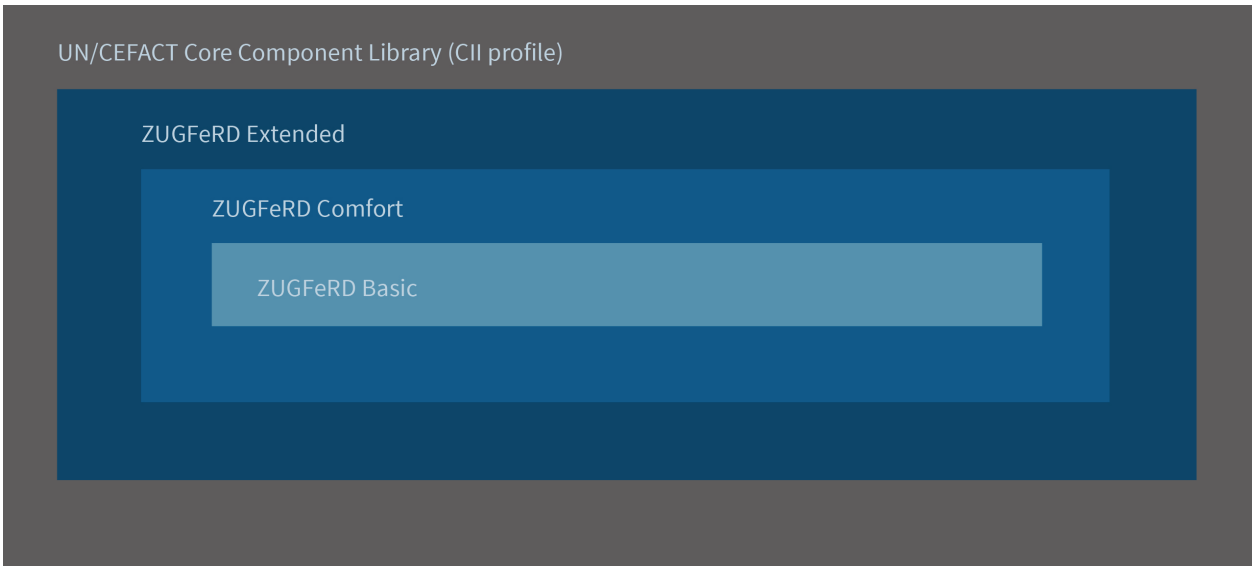


Figure 1.2: Semantic profile of the ZUGFeRD standard

- **The Comfort profile** supports processing posting, payment and checking of invoices. The information required to do so is present either in structured form or as qualified text. Other data can be included as free text. Free text imposes no requirements in terms of coding of the information itself, but qualified text needs to be accompanied by a code qualifying the content.
- **The Basic profile** is a subset of the Comfort profile. It reduces the requirements for structured data to the absolute minimum, such as posting and payment information. Other information can be added as free text.
- **The Extended profile** is a superset of the Comfort profile. It covers the cross-industry requirements as fully as possible. All relevant data is available either in structured form, or as a qualified text field. Other data, such as a note on an advertising campaign, can be included as free text.

ZUGFeRD Basic only supports commercial invoices, notifications and credit notes (code 380). ZUGFeRD Comfort also supports debit and credit notes related to financial adjustments (code 84). ZUGFeRD Extended also supports self-billed invoices and self-billed credit notes (code 389).

We'll go into much more detail, explaining which data is required and what these codes are about, when we discuss how to use iText to create invoices that conform to the Basic and the Comfort profile.

The ZUGFeRD Format

PDF/A-3 was chosen for the ZUGFeRD format and as a rule each PDF/A-3 file contains one and only one invoice. This PDF file will contain all data used for automated processing as an embedded XML file (either conforming with the requirements of the Basic, Comfort, or the Extended profile). Due to the nature of PDF/A-3, invoicees who want to process the PDF manually aren't hindered by this additional file; they can open the document in one of the many PDF viewers which come preinstalled on virtually all PCs, smart phones and other devices. The fact that ZUGFeRD uses PDF/A also means that the visual representation of the document is preserved for the long term.

We will take a closer look at how the XML file is embedded into the PDF file in the examples that demonstrate how to create ZUGFeRD invoices with iText.

Bridging the Gap between SMBs and EDI

The aim of ZUGFeRD is to close the gap between the simple exchange of invoices as printed pages or image files, and EDI processes, which consist purely of structured data. Figure 1.3, taken from the ZUGFeRD specification, shows the spectrum. The more your process is EDI-friendly, the more automation increases, but without a PDF page as part of the invoice, the less easy it becomes for humans to consume the document.

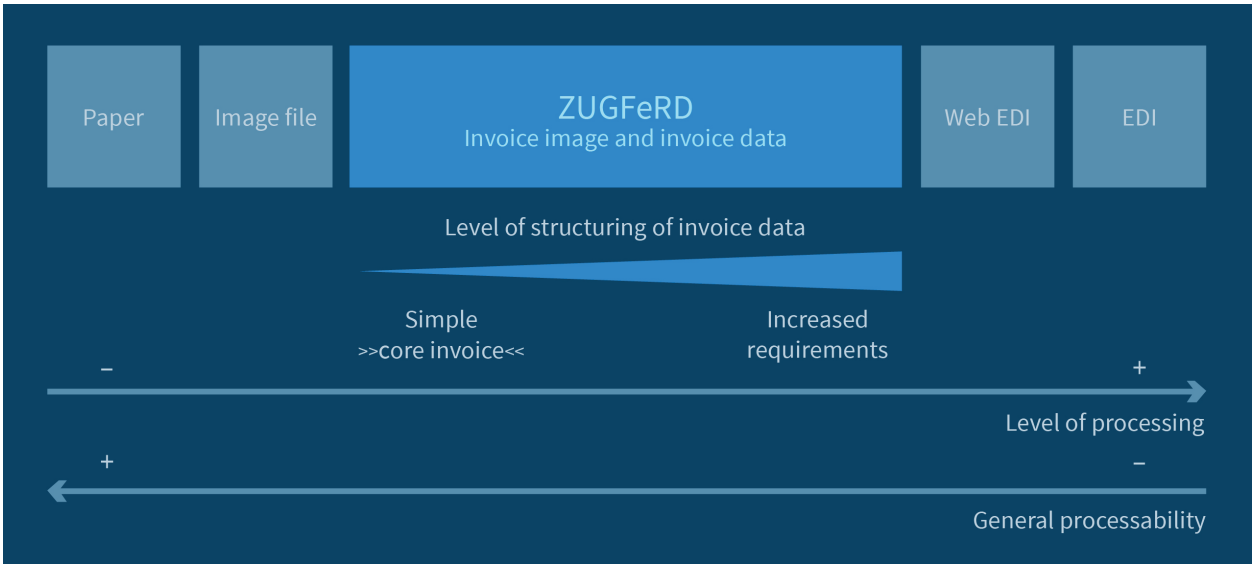


Figure 1.3: Closing the gap between paper and EDI

FeRD created ZUGFeRD to lower the threshold for small and medium businesses to implement EDI-like invoicing processes, even if they only issue and receive a small number of invoices. Thanks to ZUGFeRD, they will be able to exchange invoices with companies of any size without any prior consultation or agreement.

iText wants to contribute to this evolution. Every small and medium business is already able to create invoices in the PDF format. With a tool such as iText, it's not difficult to comply with the PDF/A standard and to attach an XML attachment. Thanks to iText, even small and medium businesses can afford to create ZUGFeRD compliant invoices.

World-wide implementation of the ZUGFeRD standard could yield financial, technical and operational benefits across the entire economy, regardless of organization size or nationality.

2. Creating PDF/A files with iText

Before we start creating invoices, let's find out how to create a PDF document using iText, more specifically: how to create a PDF document in the PDF/A-3 format.

Creating a regular PDF file

Creating a PDF file with iText 7 is very easy. It requires four simple steps:

1. Create a `PdfDocument` that writes PDF to a `PdfWriter` (low-level functionality),
2. Create a `Document` to which you can add simple building blocks (high-level functionality),
3. Add content to the `Document` in the form of building blocks[1],
4. Close the `Document`.

We'll implement these four steps to create a PDF file that looks like the document shown in Figure 2.1:

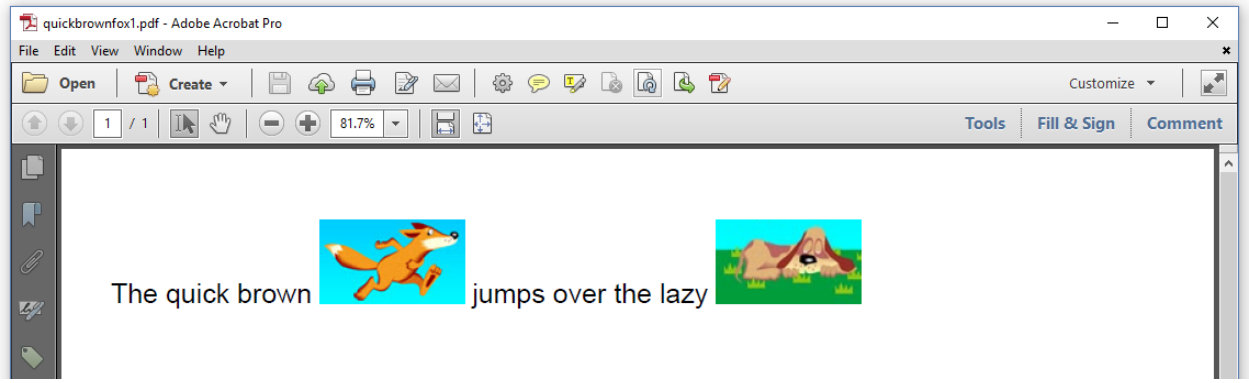


Figure 2.1: Quick brown fox jumps over the lazy dog example: regular PDF

This PDF was created using the SimplePdf.java[2] example:

[1] http://developers.itextpdf.com/content/itext-7-building-blocks

[2] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pdfa/SimplePdf.java

```
1  public void createPdf(String dest) throws IOException {
2      // step 1
3      PdfDocument pdfDocument = new PdfDocument(new PdfWriter(dest));
4      pdfDocument.setDefaultPageSize(PageSize.A4.rotate());
5      // step 2
6          Document document = new Document(pdfDocument);
7      // step 3
8      document.add(
9          new Paragraph()
10             .setFontSize(20)
11             .add(new Text("The quick brown "))
12             .add(new Image(ImageDataFactory.create(FOX)))
13             .add(new Text(" jumps over the lazy "))
14             .add(new Image(ImageDataFactory.create(DOG))));
15     // step 4
16     document.close();
17 }
```

We can easily discover the four steps in iText's PDF creation process in this code snippet:

1. In line 3-4, we create a `PdfDocument` object for a PDF document with page size A4 in landscape format. By default, the A4 page is in portrait; the `rotate()` method changes it into landscape.
2. In line 6, we create an instance of the `Document` class. `PdfDocument` is a low-level class that can be used for low-level operations; `Document` is a high-level class to which we can add high-level objects.
3. In lines 8 to 14, we compose content using high-level objects such as `Paragraph`, `Text` and `Image`. We add this content to the document.
4. In line 16, we close the document.

This creates a regular PDF file. In Figure 2.2, we take a look at the *Document Properties* of the file. At the bottom of the *Description* tab, we see **Tagged PDF: No**.

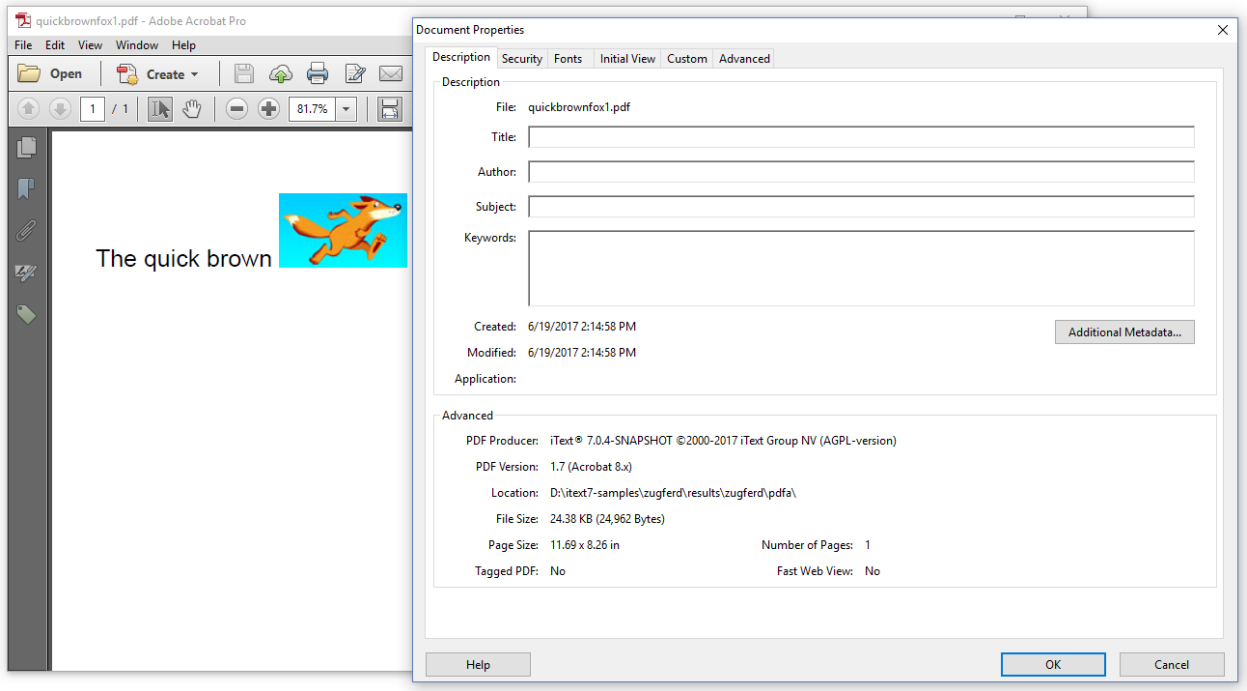


Figure 2.2: Document properties regular PDF

When we look at this document, we see a sentence in which the words "fox" and "dog" are replaced by images of a fox and a dog. Being human, we'll read *"The quick brown fox jumps over the lazy dog"*. A machine however, will only read *"The quick brown jumps over the lazy"* and won't know that the image of the fox and the dog are meant to be part of the sentence. This is a problem when the document is accessed by people who are visually impaired. They depend on assistive technology (AT) such as the "Read out loud" functionality in Adobe Reader. Only when the PDF is a *properly* Tagged PDF, will AT be able to read the full sentence.

We'll fix this problem *partly* by making some small changes to our code in the next couple of examples.

Creating a Tagged PDF file

Let's take a look at the TaggedPdf.java[3] example. It is identical to SimplePdf.java[4], except for step 1:

```
1 // step 1
2 PdfDocument pdfDocument = new PdfDocument(new PdfWriter(dest));
3 pdfDocument.setDefaultPageSize(PageSize.A4.rotate());
4 pdfDocument.setTagged();
```

[3] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pdfa/TaggedPdf.java
[4] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pdfa/SimplePdf.java

We add a single line: `pdfDocument.setTagged()`. As we are using high level objects such as `Paragraph`, `Text`, and `Image`, iText will mark that content as structured elements. A `Paragraph` will be marked as `<P>`, a `Text` object as `<Span>` and an `Image` as `<Figure>`. This adds semantical information. The resulting document, shown in figure 2.3, looks identical to the one we had before, but we can now see the structure of the sentence in the *Tags* panel when opening the PDF in Acrobat Reader.

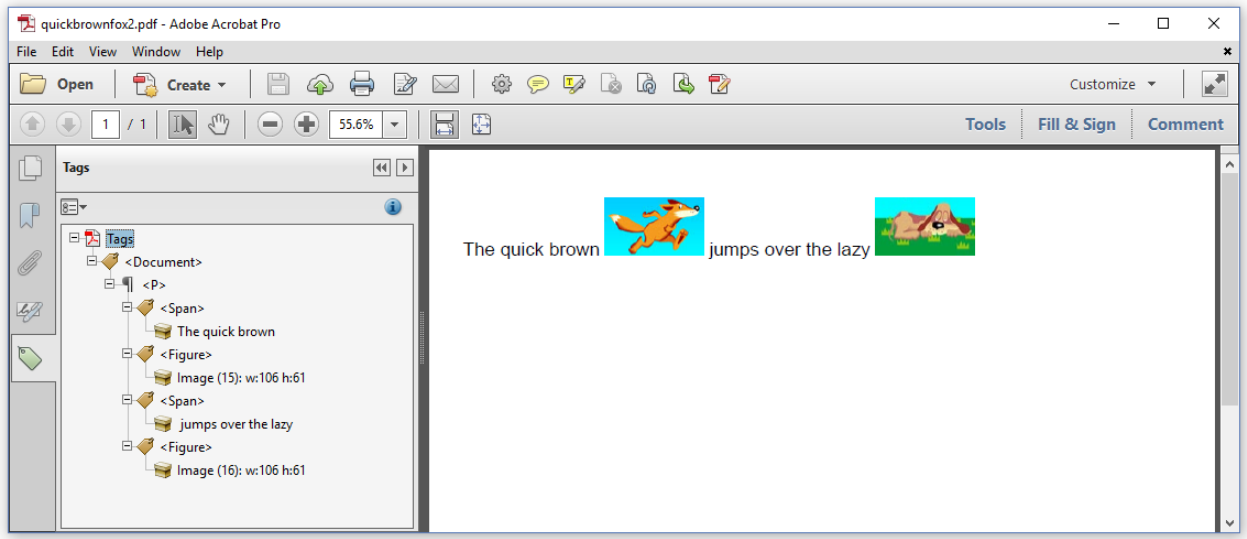


Figure 2.3: Tags in a Tagged PDF

This doesn't make the document accessible (yet). It's still not possible to replace the images by a meaningful and accurate word, but the structure of the paragraph can now be interpreted by a machine (which wasn't the case before), because a PDF parser can read a structure tree that indicates that the images are part of the paragraph.

We can make the content accessible by providing alternate text for the images. But before we do so, let's take a close look at the PDF/A format.

Creating a PDF/A-3 level B file

A PDF/A file needs to be self-contained. In the previous examples, we didn't specify a font. As a result, the default font Helvetica was used. Helvetica is one of the 14 *standard Type 1 fonts* that are assumed to be known by every PDF viewer. iText ships with 14 Adobe Font Metrics (AFM) files that contain the font metrics that are needed to calculate the width of words and sentences. iText doesn't ship with the full fonts, so whenever Helvetica is used, that font isn't embedded in the PDF.

- PDF/A requires that all fonts are embedded, so if we want to create a PDF/A-3 file, we'll have to provide a font file.
- PDF/A also requires an International Color Consortium (ICC) profile.

We'll introduce two constants with the paths to an ICC color file and a font file in the PdfA3b[5] example:

```
public static final String FONT = "resources/fonts/OpenSans-Regular.ttf";
public static final String ICC = "resources/color/sRGB_CS_profile.icm";
```

We'll need these files when we create the PDF.

- `FreeSans.ttf` is a OpenType font with TrueType outlines that looks very much like Helvetica.
- `sRGB_CS_profile.icm` is an ICC profile used to define an RGB color space.

Let's take a look at the `createPdf()` method:

```
1  public void createPdf(String dest) throws IOException {
2      // step 1
3      PdfADocument pdfDocument = new PdfADocument(
4          new PdfWriter(dest), PdfAConformanceLevel.PDF_A_3B,
5          new PdfOutputIntent("Custom", "", "http://www.color.org",
6              "sRGB IEC61966-2.1", new FileInputStream(ICC)));
7      pdfDocument.setDefaultPageSize(PageSize.A4.rotate());
8      // step 2
9      Document document = new Document(pdfDocument);
10     // step 3
11     PdfFont font = PdfFontFactory.createFont(FONT, true);
12     document.add(new Paragraph().setFont(font).setFontSize(20)
13         .add(new Text("The quick brown "))
14         .add(new Image(ImageDataFactory.create(FOX)))
15         .add(new Text(" jumps over the lazy "))
16         .add(new Image(ImageDataFactory.create(DOG))));
17     // step 4
18     document.close();
19 }
```

What's different in this code snippet when compared to our first example?

- *We use a different PDF document object.* Instead of creating an instance of `PdfDocument`, we now use a `PdfADocument` (line 3) and we define the conformance level `PDF_A_3B` (line 4).
- *An stream is created containing metadata in the eXtensible Metadata Platform (XMP).* You don't see this in the code, but iText will add this metadata automatically since XMP is a requirement for PDF/A documents.

[5] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pdfa/PdfA3b.java

- *We define output intents.* This is where we need the ICC profile (line 5-6).
- *We embed the font.* We use the `FontFactory` object to get a `PdfFont` object, making sure that we set the embedded flag to `true` (line 12). We use the `setFont()` object on the `Paragraph` instance so that the default font for all the `Chunks` added *after* the font is set, changes to this embedded font (line 25-26).

As a result, we have the same document as before, but now it verifies as a PDF/A-3B document. See figure 2.4.

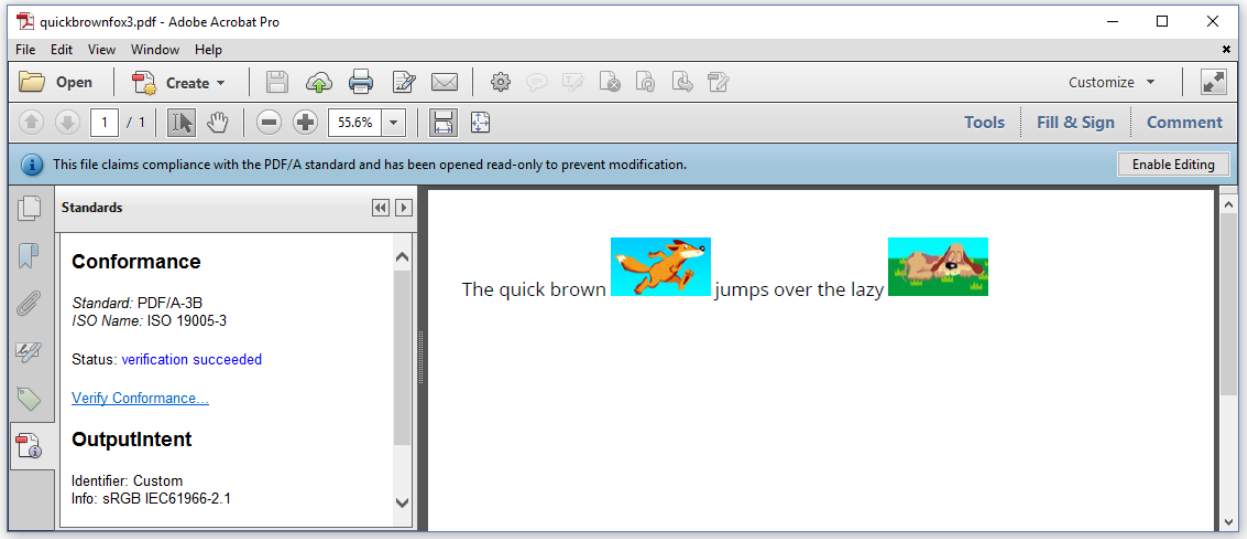


Figure 2.4: a PDF/A-3B example

Making this document accessible requires a handful of extra changes.

Creating a PDF/A-3 level A file

We'll conclude this chapter with the PdfA3a[6] example:

```
1  public void createPdf(String dest) throws IOException {
2      // step 1
3      PdfADocument pdfDocument = new PdfADocument(
4          new PdfWriter(dest), PdfAConformanceLevel.PDF_A_3A,
5          new PdfOutputIntent("Custom", "", "http://www.color.org",
6              "sRGB IEC61966-2.1", new FileInputStream(ICC)));
7      pdfDocument.setDefaultPageSize(PageSize.A4.rotate());
8      pdfDocument.setTagged();
9      pdfDocument.getDocumentInfo().setTitle("The fox and the dog");
```

[6] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pdfa/PdfA3a.java

```
10      pdfDocument.getCatalog().setViewerPreferences(
11          new PdfViewerPreferences().setDisplayDocTitle(true));
12      pdfDocument.getCatalog().setLang(new PdfString("en-US"));
13      // step 2
14      Document document = new Document(pdfDocument);
15      // step 3
16      PdfFont font = PdfFontFactory.createFont(FONT, true);
17      Image fox = new Image(ImageDataFactory.create(FOX));
18      fox.getAccessibilityProperties().setAlternateDescription("fox");
19      Image dog = new Image(ImageDataFactory.create(DOG));
20      dog.getAccessibilityProperties().setAlternateDescription("dog");
21      document.add(
22          new Paragraph()
23              .setFont(font)
24              .setFontSize(20)
25              .add(new Text("The quick brown "))
26              .add(fox)
27              .add(new Text(" jumps over the lazy "))
28              .add(dog));
29      // step 4
30      document.close();
31  }
```

We've applied the following changes when compared to the previous example:

- *We change the conformance level to* `PDF_A_3A`. That's just a matter of changing one parameter in the `PdfADocument` constructor (line 4).
- *We create a Tagged PDF.* On line 8, we recognize the `setTagged()` method. This will create `<P>`, `<Span>` and `<Figure>` tags. (This is a PDF/A-3a and a PDF/UA requirement.)
- *We add a title to the metadata.* In line 9, we set the title to "The fox and the dog." (This is a PDF/UA requirement, not a PDF/A requirement.)
- *We make sure the document title is displayed.* We do this by setting the viewer preference `DisplayDocTitle` in line 10-11. (This is a PDF/UA requirement, not a PDF/A requirement.)
- *We define the language used in the document.* On line 12, we tell the document that its contents are in American English. (This is a PDF/UA requirement, not a PDF/A requirement.)
- *We provide Alternate text for images.* On lines 18 and 20, we tell the images that they represent a "fox" and a "dog" by defining *alternate text*. (This is a PDF/UA requirement.)

When we look at the document shown in figure 2.5, we see that the document has structure and the file knows that the image of a dog represents the dog. Just hover over the image and you'll see the alternate text appear as a tool tip.

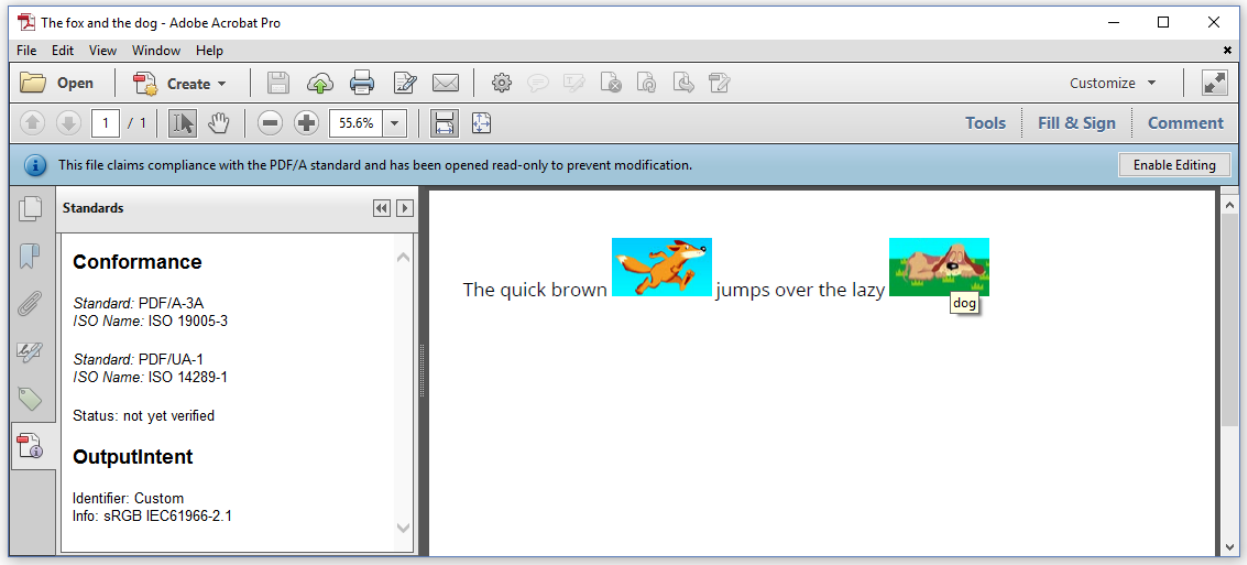


Figure 2.5: a PDF/A-3A example

The document is now accessible. Mind the title that is shown in the title bar, and the tooltip "dog" that is shown when you hover over the image of the dog. When the document is read using a AT, it will now read *"The quick brown fox jumps over the lazy dog."*

If you look at the left panel, you can see that the document is a PDF/A-3A document as well as a PDF/UA document, because we also introduced some features that are required by the PDF/UA standard. Adobe Acrobat can't verify the compliance with these standards. PDF/UA in general can't be verified programmatically because it takes a human to check whether a document is properly tagged. Incidentally, the PDF/A-3 files we've created are also compliant with the PDF/A-2 standard, because we didn't add any attachments yet. They are compliant with PDF/A-1 too, because we didn't use any of the new functionality introduced in PDF/A-2.

Once we create PDFs that represent invoices, we'll want to add an XML file that conforms to the ZUGFeRD model. Before we can do so, we need a database with invoice data.

3. A simple invoice database

Usually, all the data about your products, customers, sales, pricing, and so on, will be stored in a Customer Relationship Management (CRM) system. As this is a tutorial with examples that anyone should be able to download and execute without depending on a specific CRM, we've created a very basic invoice database that can be accessed using Hyper SQL Database (hsqldb).

Database diagram

Figure 3.1 shows the minimal set of tables and fields that we'll use for the examples in the next couple of chapters.

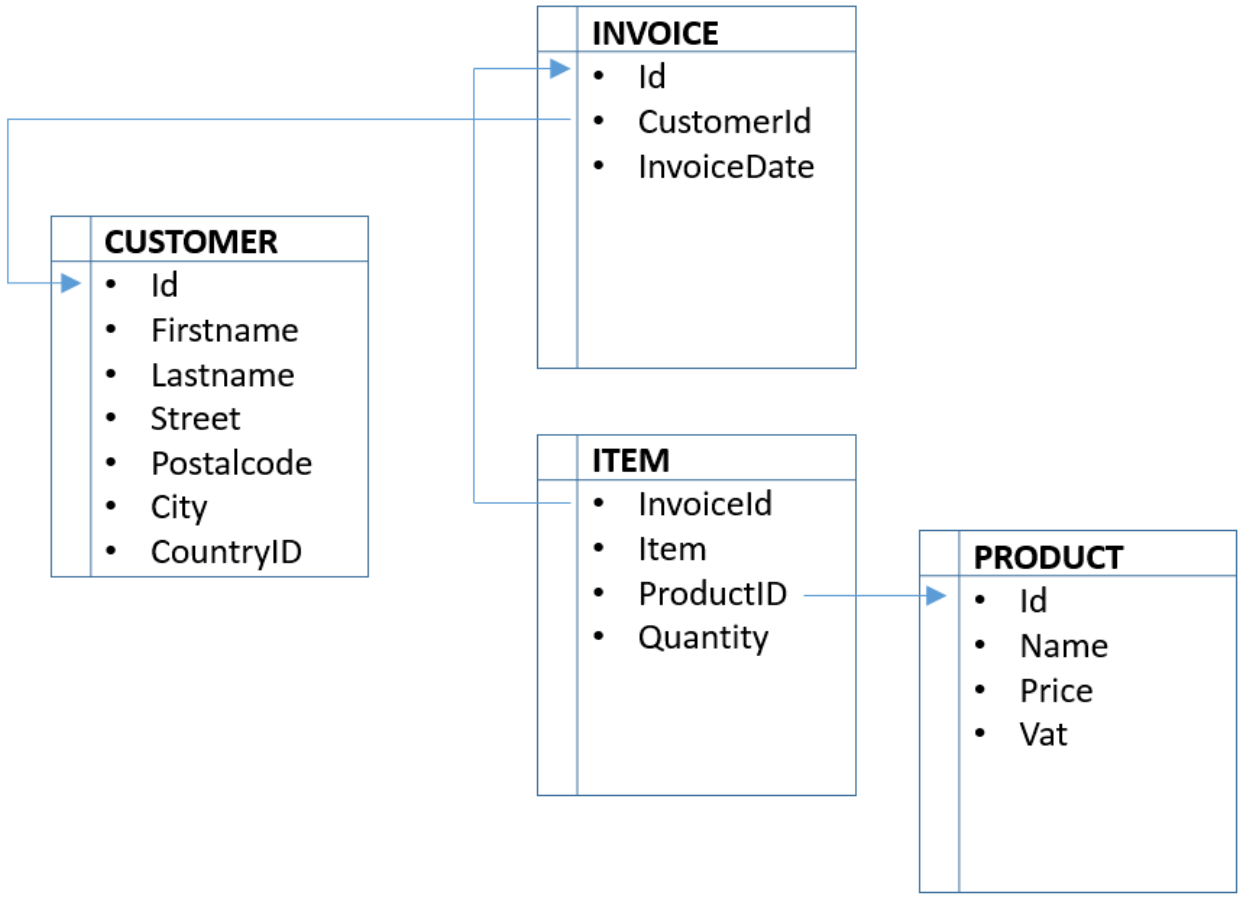


Figure 3.1: Simple invoice database schema

We'll work with only four tables:

- **INVOICE**: every record contains an id and an invoice date. The customer id refers to the **CUSTOMER** table.
- **CUSTOMER**: every record contains an id, a first name, last name, street, postal code, city and country id.
- **ITEM**: an invoice will involve one or more items. The invoice id refers to the **INVOICE** table; item is a sequential number for each separate invoice line. Product id refers to the **PRODUCT** table; the quantity field tells you how many products are being purchased.
- **PRODUCT**: every record contains an id, a name, a price and a VAT percentage.

In the real world, CRM databases contain much more information, but for the sake of this tutorial, we want to keep the complexity as low as possible.

- We won't create a country table,
- We'll assume that the customers are individuals without a company id,
- We won't store the address of the seller in our database,
- And so on.

This is not a tutorial on how to build a CRM system. We just need some data for our examples.

We will access our database using JDBC, and to make things simple, we'll work with a series of Plain Old Java Objects (POJO).

Creating database POJOs

We start by creating the classes `Invoice`[7], `Customer`[8], `Item`[9], and `Product`[10]. These are our POJOs: each class corresponds with a table in our database and consists of a series of member-variables (one for each field) and the corresponding getters and setters.

For instance: the body of the `Product`[11] class looks like this:

[7] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Invoice.java
[8] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Customer.java
[9] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Item.java
[10] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Product.java
[11] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Product.java

```
// member-variables
protected int id;
protected String name;
protected double price;
protected double vat;
// getters and setters
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public double getPrice() {
    return price;
}
public void setPrice(double price) {
    this.price = price;
}
public double getVat() {
    return vat;
}
public void setVat(double vat) {
    this.vat = vat;
}
```

We'll also add a `toString()` method that renders this content as a `String` value:

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append("\t(").append(id).append(")\t").append(name).append("\t")
        .append(price).append("\u20ac\tvat ").append(vat).append("%");
    return sb.toString();
}
```

Once we have created such a class for every table, we create a series of queries that will return

`Invoice`[12] objects. An `Invoice`[13] object contains a `Customer`[14] object and a list of `Item`[15] objects, each of which refers to a `Product`[16] object.

Creating a POJO factory

When working with hsqldb, all the data is stored in a `.script` file. In our case, the file is named `invoices.script`. We'll write a `PojoFactory`[17] class that connects to this database and that initializes a series of prepared statements:

```
1  protected Connection connection;
2  protected HashMap<Integer, Customer> customerCache
3      = new HashMap<Integer, Customer>();
4  protected HashMap<Integer, Product> productCache
5      = new HashMap<Integer, Product>();
6  protected PreparedStatement getCustomer;
7  protected PreparedStatement getProduct;
8  protected PreparedStatement getItems;
9  // constructor
10 private PojoFactory() throws ClassNotFoundException, SQLException {
11     Class.forName("org.hsqldb.jdbcDriver");
12     connection = DriverManager.getConnection(
13         "jdbc:hsqldb:resources/zugferd/db/invoices", "SA", "");
14     getCustomer = connection.prepareStatement(
15         "SELECT * FROM Customer WHERE id = ?");
16     getProduct = connection.prepareStatement(
17         "SELECT * FROM Product WHERE id = ?");
18     getItems = connection.prepareStatement(
19         "SELECT * FROM Item WHERE invoiceid = ?");
20 }
```

Let's take a closer look at the member-variables in the `PojoFactory`[18] class:

- **In line 1**, we have a `connection` object that is an object of type `java.sql.Connection`. We load the hsqldb database driver in line 11 and create the connection in lines 12 and 13. In our case, the `invoices.script` is stored in the folder `db`, which is a subdirectory of the folder `resources`

[12] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Invoice.java
[13] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Invoice.java
[14] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Customer.java
[15] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Item.java
[16] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Product.java
[17] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/PojoFactory.java
[18] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/PojoFactory.java

which is in turn a subdirectory of the working directory of our Java Virtual Machine (JVM). The username and password of that database are `"SA"` and `""`.

- **In lines 2-3 and 4-5**, we create a cache for `Customer` objects and a cache for `Product` objects by storing an integer (the id of a record) and the object containing the data of a record in `HashMap` objects. That way, we won't always have to execute a query on the database when a `Customer` or a `Product` is needed more than once; we can just fetch it from the `HashMap` that caches these objects. Obviously, this will only work if we never change any record in our database. Please keep in mind that this is just a demo database and some demo code. In the real world, your database and database access may be completely different.
- **In lines 6, 7, and 8**, we have three prepared statements. These are defined in lines 14-15, 16-17, and 18-19. As you can see, these are simple `SELECT` statements that return all the fields from the records in the tables `CUSTOMER`, `PRODUCT`, and `ITEM` based in an id.

You can get all the invoices at once, using the `getInvoices()` method. This method selects all the fields from all the records in the `INVOICE` table:

```
1  public List<Invoice> getInvoices() throws SQLException {
2      List<Invoice> invoices = new ArrayList<Invoice>();
3      Statement stm = connection.createStatement();
4      ResultSet rs = stm.executeQuery("SELECT * FROM Invoice");
5      while (rs.next()) {
6          invoices.add(getInvoice(rs));
7      }
8      stm.close();
9      return invoices;
10 }
```

The `getInvoice()` method will perform several sub-queries:

```
1  public Invoice getInvoice(ResultSet rs) throws SQLException {
2      Invoice invoice = new Invoice();
3      invoice.setId(rs.getInt("id"));
4      invoice.setCustomer(getCustomer(rs.getInt("customerid")));
5      List<Item> items = getItems(rs.getInt("id"));
6      invoice.setItems(items);
7      double total = 0;
8      for (Item item : items)
9          total += item.getCost();
10     invoice.setTotal(total);
11     invoice.setInvoiceDate(rs.getDate("invoicedate"));
12     return invoice;
13 }
```

You see that we even calculate the total sum of the invoice. This is a value that you'd usually store in your database redundantly, to avoid that the total invoice price of an old invoice changes when you introduce new prices for products mentioned on that old invoice. Once again: we have kept our database as minimal as possible.

The `Customer` object that corresponds with this invoice, is obtained with the `getCustomer()` method:

```
    public Customer getCustomer(int id) throws SQLException {
        if (customerCache.containsKey(id))
            return customerCache.get(id);
        getCustomer.setInt(1, id);
        ResultSet rs = getCustomer.executeQuery();
        if (rs.next()) {
            Customer customer = new Customer();
            customer.setId(id);
            customer.setFirstName(rs.getString("FirstName"));
            customer.setLastName(rs.getString("LastName"));
            customer.setStreet(rs.getString("Street"));
            customer.setPostalcode(rs.getString("Postalcode"));
            customer.setCity(rs.getString("City"));
            customer.setCountryId(rs.getString("CountryID"));
            customerCache.put(id, customer);
            return customer;
        }
        return null;
    }
```

If the `id` is found in the cache, we return the corresponding `Customer`[19] object. If not, we perform a query using one of our prepared statements, and we store the resulting `Customer`[20] instance in the cache before returning it.

When populating the `Invoice`[21] object, we get a `List` of `Item` objects using the `getItems()` method:

[19] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Customer.java
[20] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Customer.java
[21] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Invoice.java

```
public List<Item> getItems(int invoiceid) throws SQLException {
    List items = new ArrayList<Item>();
    getItems.setInt(1, invoiceid);
    ResultSet rs = getItems.executeQuery();
    while (rs.next()) {
        items.add(getItem(rs));
    }
    return items;
}
```

This method calls the `getItem()` method for every invoice line that belongs to a specific `Invoice`[22]:

```
public Item getItem(ResultSet rs) throws SQLException {
    Item item = new Item();
    item.setItem(rs.getInt("Item"));
    Product product = getProduct(rs.getInt("ProductId"));
    item.setProduct(product);
    item.setQuantity(rs.getInt("Quantity"));
    item.setCost(item.getQuantity() * product.getPrice());
    return item;
}
```

The `getItem()` method also fetches the corresponding `Product`[23]:

```
public Product getProduct(int id) throws SQLException {
    if (productCache.containsKey(id))
        return productCache.get(id);
    getProduct.setInt(1, id);
    ResultSet rs = getProduct.executeQuery();
    if (rs.next()) {
        Product product = new Product();
        product.setId(id);
        product.setName(rs.getString("Name"));
        product.setPrice(rs.getDouble("Price"));
        product.setVat(rs.getDouble("Vat"));
        productCache.put(id, product);
        return product;
    }
    return null;
}
```

[22] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Invoice.java
[23] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Product.java

Now we have everything we need to retrieve all the invoice data that is stored in our very simple invoice database.

Testing the database

Before we use this database to create ZUGFeRD XMLs and invoices, let's run the DatabaseTest[24] example to test the database:

```
1  public static void main(String[] args) throws SQLException {
2      PojoFactory factory = PojoFactory.getInstance();
3      List<Invoice> invoices = factory.getInvoices();
4      for (Invoice invoice : invoices)
5          System.out.println(invoice.toString());
6      factory.close();
7  }
```

We use the `PojoFactory`[25] to get a `List` of `Invoice`[26] objects and we write the content of such an object to the `System.out`. The result will be a sequence of text snippets (one for each invoice) that look like this:

```
Invoice id: 4 Date: 2015-04-01 Total cost: 1507.0€
Customer: 30
    First Name: Bill
    Last Name: Sommer
    Street: 362 - 20th Ave.
    City: BE 9000 Ghent
  #0  (28)   Running jersey      8.0€   vat 21.0%  Quantity: 9  Cost: 72.0€
  #1  (35)   Golf polo           8.0€   vat 21.0%  Quantity: 1  Cost: 8.0€
  #2  (41)   Threadmill          600.0€ vat 21.0%  Quantity: 2  Cost: 1200.0€
  #3  (23)   Pro steel dartboard 25.0€  vat 21.0%  Quantity: 2  Cost: 50.0€
  #4  (9)    My First Cookbook   17.0€  vat 6.0%   Quantity: 1  Cost: 17.0€
  #5  (37)   Golf kit            80.0€  vat 21.0%  Quantity: 2  Cost: 160.0€
```

This already looks more or less like an invoice, doesn't it? Now let's find out how all this data fits into the ZUGFeRD data model.

[24] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/DatabaseTest.java
[25] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/PojoFactory.java
[26] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Invoice.java

4. Creating XML Invoices with iText

In chapter 1, we discussed XML standards for invoicing and we explained that ZUGFeRD is based on the Cross Industry Invoice (CII) standard and the Message User Guides (MUG) from CEN. In this chapter, we'll create a series of such invoices using iText.

Interfaces for the Basic and Comfort profile

iText ships with two interfaces that can be found in the `com.itextpdf.text.zugferd.profiles` package (shipped with the pdfInvoice add-on[27]):

- `IBasicProfile`: contains 56 `get()` methods for you to implement.
- `IComfortProfile` extends the `IBasicProfile` interface and adds 89 more methods for you to implement (144 in total).

These methods look like this:

```
1  public String getSellerName();
2  public String getSellerPostcode();
3  public String getSellerLineOne();
4  public String getSellerLineTwo();
5  public String getSellerCityName();
6  public String getSellerCountryID();
7  public String[] getSellerTaxRegistrationID();
8  public String[] getSellerTaxRegistrationSchemeID();
```

As we're dealing with XML and as XML consists of data stored as text, most of the methods expect that you return `String` values, or arrays of `String` values, even when the data consists of numbers. In cases where dates or Boolean values are involved, the method has a `Date`, a `Date[]`, a `boolean`, a `Boolean[]`, or a `Boolean[][]` as return type.

iText ships with an implementation of these interfaces: `BasicProfileImp` implements the `IBasicProfile` interface; `ComfortProfileImp` extends `BasicProfileImp` and implements the `IComfortProfile` interface. These classes store all the data in `boolean`, `Date`, `String`, `List<String>`, `List<String[]>`, `List<Date>`, `List<Boolean>`, or `List<Boolean[]>` member-variables and provide `set()` methods to populate these variables. For instance:

[27] http://itextpdf.com/itext7/pdfInvoice

```
public void setSellerName(String sellerName) {
    this.sellerName = sellerName;
}
public void setSellerPostcode(String sellerPostcode) {
    this.sellerPostcode = sellerPostcode;
}
public void setSellerLineOne(String sellerLineOne) {
    this.sellerLineOne = sellerLineOne;
}
public void setSellerLineTwo(String sellerLineTwo) {
    this.sellerLineTwo = sellerLineTwo;
}
public void setSellerCityName(String sellerCityName) {
    this.sellerCityName = sellerCityName;
}
public void setSellerCountryID(String sellerCountryID) {
    this.sellerCountryID = sellerCountryID;
}
public void addSellerTaxRegistration(String schemeID, String taxId) {
    sellerTaxRegistrationSchemeID.add(schemeID);
    sellerTaxRegistrationID.add(taxId);
}
```

The provided values are used to implement the getter methods defined in the interface:

```
public String getSellerName() {
    return sellerName;
}
public String getSellerPostcode() {
    return sellerPostcode;
}
public String getSellerLineOne() {
    return sellerLineOne;
}
public String getSellerLineTwo() {
    return sellerLineTwo;
}
public String getSellerCityName() {
    return sellerCityName;
}
public String getSellerCountryID() {
    return sellerCountryID;
```

```
18  }
19  public String[] getSellerTaxRegistrationID() {
20      return to1DArray(sellerTaxRegistrationID);
21  }
22  public String[] getSellerTaxRegistrationSchemeID() {
23      return to1DArray(sellerTaxRegistrationSchemeID);
24  }
```

Not all the getters need to be fully implemented. Some data is optional in the Basic and Comfort profile. For instance, it is perfectly OK to implement some of the methods like this:

```
1   public Date getBillingStartDateTime() {
2       return null;
3   }
4   public String getBillingStartDateTimeFormat() {
5       return null;
6   }
7   public Date getBillingEndDateTime() {
8       return null;
9   }
10  public String getBillingEndDateTimeFormat() {
11      return null;
12  }
```

How you implement these interfaces will largely depend on the CRM you're using. You'll have to query its database and use the results of that query to implement the methods of the interface corresponding with the profile you want to support.

Getting and setting the data

In this tutorial, we'll use `BasicProfileImp` and `ComfortProfileImp` (the `IBasicProfile` and `IComfortProfile` implementations that ship with iText) to store the information from the database we've discussed in chapter 3. See the `InvoiceData`[28] class that we'll use in the next handful of examples.

[28] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/data/InvoiceData.java

```
public InvoiceData() {
}
public IBasicProfile createBasicProfileData(Invoice invoice) {
    BasicProfileImp profileImp = new BasicProfileImp();
    importData(profileImp, invoice);
    importBasicData(profileImp, invoice);
    return profileImp;
}
public IComfortProfile createComfortProfileData(Invoice invoice) {
    ComfortProfileImp profileImp = new ComfortProfileImp();
    importData(profileImp, invoice);
    importComfortData(profileImp, invoice);
    return profileImp;
}
```

We don't pass any data to the `InvoiceData` constructor. Instead, we pass our `Invoice` POJO to the `createBasicProfileData()` or the `createComfortProfile()` method to obtain an `IBasicProfile` or `IComfortProfile` implementation.

Internally, the default `BasicProfileImp` or `ComfortProfileImp` implementations are used. These data containers are populated using the `importData()`, `importBasicData()` and `importComfortData()` methods.

Basic and Comfort profile

The `importData()` method deals with data that is relevant for both the Basic and the Comfort profile:

```
public void importData(BasicProfileImp profileImp, Invoice invoice) {
    profileImp.setTest(true);
    profileImp.setId(String.format("I/%05d", invoice.getId()));
    profileImp.setName("INVOICE");
    profileImp.setTypeCode(DocumentTypeCode.COMMERCIAL_INVOICE);
    profileImp.setDate(invoice.getInvoiceDate(), DateFormatCode.YYYYMMDD);
    profileImp.setSellerName("Das Company");
    profileImp.setSellerLineOne("ZUG Business Center");
    profileImp.setSellerLineTwo("Highway 1");
    profileImp.setSellerPostcode("9000");
    profileImp.setSellerCityName("Ghent");
    profileImp.setSellerCountryID("BE");
    profileImp.addSellerTaxRegistration(
        TaxIDTypeCode.FISCAL_NUMBER, "201/113/40209");
    profileImp.addSellerTaxRegistration(TaxIDTypeCode.VAT, "BE123456789");
```

```
    Customer customer = invoice.getCustomer();
    profileImp.setBuyerName(String.format("%s, %s",
        customer.getLastName(), customer.getFirstName()));
    profileImp.setBuyerPostcode(customer.getPostalcode());
    profileImp.setBuyerLineOne(customer.getStreet());
    profileImp.setBuyerCityName(customer.getCity());
    profileImp.setBuyerCountryID(customer.getCountryId());
    profileImp.setPaymentReference(String.format("%09d", invoice.getId()));
    profileImp.setInvoiceCurrencyCode("EUR");
}
```

As you can see, we add some of the data, such as the name of the seller, in a hard-coded way. Obviously, this should be avoided in a real-world implementation.

We also use some constants such as `DocumentTypeCode.COMMERCIAL_INVOICE`, `DateFormatCode.YYYYMMDD`, and `TaxIDTypeCode.VAT`. `DocumentTypeCode`, `DateFormatCode`, `TaxIDTypeCode` and many other code list classes can be found in the `com.itextpdf.text.zugferd.checkers` packages. They all implement the `CodeValidation` class. This abstract class contains a `check()` method that throws an `InvalidCodeException` if the wrong code is provided. We'll look at these checkers in more detail in a moment.

Basic profile

All data that is necessary for the Basic profile is also necessary for the Comfort profile, but due to some differences between Basic and Comfort, the implementation of these profiles requires different setters in iText.

```
public void importBasicData(BasicProfileImp profileImp, Invoice invoice) {
    profileImp.addNote(new String[]{
        "This is a test invoice.\nNothing on this invoice is real."
        + "\nThis invoice is part of a tutorial."});
    profileImp.addPaymentMeans("", "", "BE 41 7360 0661 9710",
        "", "", "KREDBEBB", "", "KBC");
    profileImp.addPaymentMeans("", "", "BE 56 0015 4298 7888",
        "", "", "GEBABEBB", "", "BNP Paribas");
    Map<Double,Double> taxes = new TreeMap<Double, Double>();
    double tax;
    for (Item item : invoice.getItems()) {
        tax = item.getProduct().getVat();
        if (taxes.containsKey(tax)) {
            taxes.put(tax, taxes.get(tax) + item.getCost());
        }
```

```
16          else {
17              taxes.put(tax, item.getCost());
18          }
19          profileImp.addIncludedSupplyChainTradeLineItem(
20              format4dec(item.getQuantity()), "C62", item.getProduct().getName());
21      }
22      double total, tA;
23      double ltN = 0;
24      double ttA = 0;
25      double gtA = 0;
26      for (Map.Entry<Double, Double> t : taxes.entrySet()) {
27          tax = t.getKey();
28          total = round(t.getValue());
29          gtA += total;
30          tA = round((100 * total) / (100 + tax));
31          ttA += (total - tA);
32          ltN += tA;
33          profileImp.addApplicableTradeTax(format2dec(total - tA), "EUR",
34              TaxTypeCode.VALUE_ADDED_TAX, format2dec(tA), "EUR", format2dec(tax));
35      }
36      profileImp.setMonetarySummation(format2dec(ltN), "EUR",
37          format2dec(0), "EUR",
38          format2dec(0), "EUR",
39          format2dec(ltN), "EUR",
40          format2dec(ttA), "EUR",
41          format2dec(gtA), "EUR");
42  }
```

This is typical for the Basic profile:

- The Basic profile allows free text in the header, see line 28-30: *"This is a test invoice. Nothing on this invoice is real. This invoice is part of a tutorial."*
- You don't need that much information regarding the payment means.
- You don't need that much information regarding the line items (the invoice lines).

Note that we loop over the different `Item` objects to calculate the monetary summation:

- the total amount of the line items (line 62),
- the tax basis amount (line 65), which is total of the line items after taking into account the charge total amount (line 63) and the allowance total amount (line 64), which are 0 in this case,

- the total tax amount (line 66), and
- the grand total (line 67).

Note that we use some helper methods to format numbers and percentages:

```
1  public static double round(double d) {
2      d = d * 100;
3      long tmp = Math.round(d);
4      return (double) tmp / 100;
5  }
6  public static String format2dec(double d) {
7      return String.format("%.2f", d);
8  }
9  public static String format4dec(double d) {
10     return String.format("%.4f", d);
11 }
```

In some cases, numbers need to be expressed using 2 decimals; in other cases, numbers need to be expressed using 4 decimals. Once you create the XML, iText will throw an `InvalidCodeException` if you pass a numeric value with the wrong number of decimals.

Comfort profile

When we look at the `importComfortData()` method, we see that much more data can be provided. For instance: the notes in the header need to consist of qualified text. This means that we have to describe what the note is about using a code. In this case, we used `FreeTextSubjectCode.REGULATORY_INFORMATION` (see line 5).

```
1  public void importComfortData(ComfortProfileImp profileImp, Invoice invoice) {
2      profileImp.addNote(new String[]{
3          "This is a test invoice.\nNothing on this invoice is real."
4          + "\nThis invoice is part of a tutorial."},
5          FreeTextSubjectCode.REGULATORY_INFORMATION);
6      profileImp.addPaymentMeans(
7          PaymentMeansCode.PAYMENT_TO_BANK_ACCOUNT,
8          new String[]{"This is the preferred bank account."},
9          "", "",
10         "", "",
11         "BE 41 7360 0661 9710", "", "",
12         "", "", "",
13         "KREDBEBB", "", "KBC");
```

```
14      profileImp.addPaymentMeans(
15          PaymentMeansCode.PAYMENT_TO_BANK_ACCOUNT,
16          new String[]{"Use this as an alternative account."},
17          "", "",
18          "", "",
19          "BE 56 0015 4298 7888", "", "",
20          "", "", "",
21          "GEBABEBB", "", "BNP Paribas");
22      Map<Double,Double> taxes = new TreeMap<Double, Double>();
23      double tax;
24      int counter = 0;
25      for (Item item : invoice.getItems()) {
26          counter++;
27          tax = item.getProduct().getVat();
28          if (taxes.containsKey(tax)) {
29              taxes.put(tax, taxes.get(tax) + item.getCost());
30          }
31          else {
32              taxes.put(tax, item.getCost());
33          }
34          profileImp.addIncludedSupplyChainTradeLineItem(
35              String.valueOf(counter),
36              null,
37              format4dec(item.getProduct().getPrice()), "EUR", null, null,
38              null, null, null, null,
39              null, null, null, null,
40              format4dec(item.getQuantity()), "C62",
41              new String[]{TaxTypeCode.VALUE_ADDED_TAX},
42              new String[1],
43              new String[]{TaxCategoryCode.STANDARD_RATE},
44              new String[]{format2dec(item.getProduct().getVat())},
45              format2dec(item.getCost()), "EUR",
46              null, null,
47              String.valueOf(item.getProduct().getId()), null,
48              item.getProduct().getName(), null
49          );
50      }
51      double total, tA;
52      double ltN = 0;
53      double ttA = 0;
54      double gtA = 0;
55      for (Map.Entry<Double, Double> t : taxes.entrySet()) {
```

```
56          tax = t.getKey();
57          total = round(t.getValue());
58          gtA += total;
59          tA = round((100 * total) / (100 + tax));
60          ttA += (total - tA);
61          ltN += tA;
62          profileImp.addApplicableTradeTax(
63              format2dec(total - tA), "EUR", TaxTypeCode.VALUE_ADDED_TAX,
64              null, format2dec(tA), "EUR",
65              TaxCategoryCode.STANDARD_RATE, format2dec(tax));
66      }
67      profileImp.setMonetarySummation(format2dec(ltN), "EUR",
68          format2dec(0), "EUR",
69          format2dec(0), "EUR",
70          format2dec(ltN), "EUR",
71          format2dec(ttA), "EUR",
72          format2dec(gtA), "EUR");
73  }
```

When browsing the code, you see that we can leave certain values empty (`""`) or pass `null` as a value. This isn't always true. iText will throw a `DataIncompleteException` when you try to make an XML based on incomplete data. As explained earlier, an `InvalidDataException` is thrown when the wrong data is provided, although iText doesn't check all the codes you pass.

Validation of the data

Table 4.1 shows an overview of the checker classes that will be used once iText creates an XML file based on your implementation of the `IBasicProfile` or the `IComfortProfile` interface:

Table 4.1: Checker classes for the Basic and Comfort profile

Checker class	Description	Profile
`NumberChecker`	Can be used to check if a number is an integer or a decimal; if a decimal, checks if it has two or four decimals.	Basic and higher
`CountryCode`	Just checks if the code consists of two uppercase letters (no numbers). It doesn't check if the country code actually exists. That's *your* responsibility.	Basic and higher
`CurrencyCode`	Just checks if the code consists of three uppercase letters (no numbers). It doesn't check if the currency code actually exists. That's *your* responsibility.	Basic and higher

Table 4.1: Checker classes for the Basic and Comfort profile

Checker class	Description	Profile
`DateFormatCode`	Contains the three acceptable date formats `YYYYMMDD` (code 102), `YYYYMM` (code 610) and `YYYYWW` (code 616). This class also allows you to convert a `Date` to a `String` when given a format, and vice-versa.	Basic and higher
`DocumentTypeCode`	Could be `COMMERCIAL_INVOICE` (code 380), `DEBIT_NOTE_FINANCIAL_ADJUSTMENT` (code 38) and `SELF_BILLED_INVOICE` (code 389). iText will check if you're using the right profile for the document type.	Basic and higher
`LanguageCode`	Just checks if the code consists of two lowercase letters (no numbers). It doesn't check if the language code actually exists. That's *your* responsibility.	Basic and higher
`MeasurementUnitCode`	Contains constants for every possible measurement unit and checks if a code that was provided is one of these values.	Basic and higher
`TaxIDTypeCode`	Contains the codes for two types of tax ids (`VAT` and `FISCAL_NUMBER`) and checks if the code that was provided is one of these values.	Basic and higher
`TaxTypeCode`	Contains the codes for three types of tax (`VALUE_ADDED_TAX`, `INSURANCE_TAX` and `TAX_ON_REPLACEMENT_PART`) and checks if the code that was provided is one of these values	Basic and higher
`FreeTextSubjectCode`	Contains constants for every possible free text subject (to make it qualified text) and checks if the code that was provided is one of these values.	Comfort and higher
`GlobalIdentifierCode`	Contains a handful of frequently used codes, but only checks if the value consists of four numeric values.	Comfort and higher
`PaymentMeansCode`	Contains constants for all the possible payment means and checks if a code that was provided is one of these values.	Comfort and higher
`TaxCategoryCode`	Contains the codes for different tax categories and checks if a code that was provided is one of these values.	Comfort and higher

Text also has checkers that are only important in the context of the Extended profile, but iText doesn't generate XMLs using the Extended profile. Companies that need the Extended profile are assumed to already use specialized EDI software that creates XML that complies with the requirements of the Extended profile.

Let's finish this chapter by creating a series of XML files that comply with the Comfort profile.

Creating an XML file with iText

Once you have an implementation of the `IBasicProfile` or the `IComfortProfile` interface, you can use it to create an `InvoiceDOM` object. The actual XML is created as a `byte` array when you use the `toXML()` method. This is shown in the XmlInvoicesComfort[29] example:

```
1  public static void main(String[] args)
2      throws SQLException, ParserConfigurationException, SAXException, IOException,
3          TransformerException, DataIncompleteException, InvalidCodeException {
4      LicenseKey.loadLicenseFile(
5          System.getenv("ITEXT7_LICENSEKEY")
6          + "/itextkey-html2pdf_typography.xml");
7      File file = new File(DEST);
8      file.getParentFile().mkdirs();
9      PojoFactory factory = PojoFactory.getInstance();
10     List<Invoice> invoices = factory.getInvoices();
11     InvoiceData invoiceData = new InvoiceData();
12     IBasicProfile comfort;
13     InvoiceDOM dom;
14     for (Invoice invoice : invoices) {
15         comfort = invoiceData.createComfortProfileData(invoice);
16         dom = new InvoiceDOM(comfort);
17         byte[] xml = dom.toXML();
18         FileOutputStream fos = new FileOutputStream(
19             String.format(DEST, invoice.getId()));
20         fos.write(xml);
21         fos.flush();
22         fos.close();
23     }
24     factory.close();
25 }
```

Observe that we have to load a license key for the pdfInvoice add-on[30] (line 4). If you don't have a license key, you should get a trial license[31].

Just like in the example from chapter 3 where we tested the database, we use the `PojoFactory` to get a list of invoices and we loop over every `Invoice` object. We use the `InvoiceData` class that was already discussed to create a `IComfortProfile`. We pass this profile to the `InvoiceDOM` constructor as if it were a `IBasicProfile` instance, but `InvoiceDOM` is smart enough to see that it's actually a `IComfortProfile` instance.

[29] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/XmlInvoicesComfort.java
[30] http://itextpdf.com/itext7/pdfInvoice
[31] http://pages.itextpdf.com/iText-7-Free-Trial-Landing-Page-1.html

In this case, we want the XML as a file, so we write the `byte[]` to a `FileOutputStream`. Figures 4.1, 4.2 and 4.3 show what such an XML looks like.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<rsm:CrossIndustryDocument xmlns:ram="urn:un:unece:uncefact:data:standard:Reusab
    <rsm:SpecifiedExchangedDocumentContext>
        <ram:TestIndicator>
            <udt:Indicator>true</udt:Indicator>
        </ram:TestIndicator>
        <ram:GuidelineSpecifiedDocumentContextParameter>
            <ram:ID>urn:ferd:CrossIndustryDocument:invoice:1p0:comfort</ram:ID>
        </ram:GuidelineSpecifiedDocumentContextParameter>
    </rsm:SpecifiedExchangedDocumentContext>
    <rsm:HeaderExchangedDocument>
        <ram:ID>I/00004</ram:ID>
        <ram:Name>INVOICE</ram:Name>
        <ram:TypeCode>380</ram:TypeCode>
        <ram:IssueDateTime>
            <udt:DateTimeString format="102">20150401</udt:DateTimeString>
        </ram:IssueDateTime>
        <ram:IncludedNote>
            <ram:Content>This is a test invoice.
Nothing on this invoice is real.
This invoice is part of a tutorial.</ram:Content>
            <ram:SubjectCode>REG</ram:SubjectCode>
        </ram:IncludedNote>
    </rsm:HeaderExchangedDocument>
    <rsm:SpecifiedSupplyChainTradeTransaction>
        <ram:ApplicableSupplyChainTradeAgreement>
            <ram:SellerTradeParty>
                <ram:Name>Das Company</ram:Name>
```

```
                    <ram:Name>BNP Paribas</ram:Name>
                </ram:PayeeSpecifiedCreditorFinancialInstitution>
            </ram:SpecifiedTradeSettlementPaymentMeans>
            <ram:ApplicableTradeTax>
                <ram:CalculatedAmount currencyID="EUR">0.96</ram:CalculatedAmount>
                <ram:TypeCode>VAT</ram:TypeCode>
                <ram:BasisAmount currencyID="EUR">16.04</ram:BasisAmount>
                <ram:CategoryCode>S</ram:CategoryCode>
                <ram:ApplicablePercent>6.00</ram:ApplicablePercent>
            </ram:ApplicableTradeTax>
            <ram:ApplicableTradeTax>
                <ram:CalculatedAmount currencyID="EUR">258.60</ram:CalculatedAmount>
                <ram:TypeCode>VAT</ram:TypeCode>
                <ram:BasisAmount currencyID="EUR">1231.40</ram:BasisAmount>
                <ram:CategoryCode>S</ram:CategoryCode>
                <ram:ApplicablePercent>21.00</ram:ApplicablePercent>
            </ram:ApplicableTradeTax>
            <ram:SpecifiedTradeSettlementMonetarySummation>
                <ram:LineTotalAmount currencyID="EUR">1247.44</ram:LineTotalAmount>
                <ram:ChargeTotalAmount currencyID="EUR">0.00</ram:ChargeTotalAmount>
                <ram:AllowanceTotalAmount currencyID="EUR">0.00</ram:AllowanceTotalAmount>
                <ram:TaxBasisTotalAmount currencyID="EUR">1247.44</ram:TaxBasisTotalAmount>
                <ram:TaxTotalAmount currencyID="EUR">259.56</ram:TaxTotalAmount>
                <ram:GrandTotalAmount currencyID="EUR">1507.00</ram:GrandTotalAmount>
            </ram:SpecifiedTradeSettlementMonetarySummation>
        </ram:ApplicableSupplyChainTradeSettlement>
        <ram:IncludedSupplyChainTradeLineItem>
            <ram:AssociatedDocumentLineDocument>
                <ram:LineID>1</ram:LineID>
```

```
234         <ram:IncludedSupplyChainTradeLineItem>
235             <ram:AssociatedDocumentLineDocument>
236                 <ram:LineID>6</ram:LineID>
237             </ram:AssociatedDocumentLineDocument>
238             <ram:SpecifiedSupplyChainTradeAgreement>
239                 <ram:GrossPriceProductTradePrice>
240                     <ram:ChargeAmount currencyID="EUR">80.0000</ram:ChargeAmount>
241                 </ram:GrossPriceProductTradePrice>
242             </ram:SpecifiedSupplyChainTradeAgreement>
243             <ram:SpecifiedSupplyChainTradeDelivery>
244                 <ram:BilledQuantity unitCode="C62">2.0000</ram:BilledQuantity>
245             </ram:SpecifiedSupplyChainTradeDelivery>
246             <ram:SpecifiedSupplyChainTradeSettlement>
247                 <ram:ApplicableTradeTax>
248                     <ram:TypeCode>VAT</ram:TypeCode>
249                     <ram:CategoryCode>S</ram:CategoryCode>
250                     <ram:ApplicablePercent>21.00</ram:ApplicablePercent>
251                 </ram:ApplicableTradeTax>
252                 <ram:SpecifiedTradeSettlementMonetarySummation>
253                     <ram:LineTotalAmount currencyID="EUR">160.00</ram:LineTotalAmount>
254                 </ram:SpecifiedTradeSettlementMonetarySummation>
255             </ram:SpecifiedSupplyChainTradeSettlement>
256             <ram:SpecifiedTradeProduct>
257                 <ram:SellerAssignedID>37</ram:SellerAssignedID>
258                 <ram:Name>Golf kit</ram:Name>
259             </ram:SpecifiedTradeProduct>
260         </ram:IncludedSupplyChainTradeLineItem>
261     </rsm:SpecifiedSupplyChainTradeTransaction>
262 </rsm:CrossIndustryDocument>
```

We're halfway creating a ZUGFeRD invoice: we already have the XML, now we need to create the PDF. That's what chapter 5 is about.!

5. Creating PDF invoices (Basic profile)

To create a ZUGFeRD invoice, we now have to combine what we've learned in chapter 2 "Creating PDF/A files with iText" with what we've learned in chapter 4 "Creating XML Invoices with iText".

Creating PDF from scratch

In this chapter, we'll discuss a single example: PdfInvoicesBasic[32]. In this example we'll create a PDF document for every invoice stored in our database. The PDF documents will be ZUGFeRD invoices using the Basic profile.

Let's start with the different constants defined in this class:

```
public static final String DEST = "results/zugferd/pdf/basic%05d.pdf";
public static final String ICC = "resources/color/sRGB_CS_profile.icm";
public static final String REGULAR = "resources/fonts/OpenSans-Regular.ttf";
public static final String BOLD = "resources/fonts/OpenSans-Bold.ttf";
public static final String NEWLINE = "\n";
```

We recognize the pattern for the destination files, the color profile (as discussed in chapter 2[33]), two font files, and a string with a newline character.

The main method of this example is very straightforward:

```
public static void main(String[] args)
    throws IOException, ParserConfigurationException, SQLException,
        SAXException, TransformerException, ParseException
        DataIncompleteException, InvalidCodeException {
    LicenseKey.loadLicenseFile(
        System.getenv("ITEXT7_LICENSEKEY")
        + "/itextkey-html2pdf_typography.xml");
    File file = new File(DEST);
    file.getParentFile().mkdirs();
    PdfInvoicesBasic app = new PdfInvoicesBasic();
    PojoFactory factory = PojoFactory.getInstance();
    List<Invoice> invoices = factory.getInvoices();
```

[32] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/PdfInvoicesBasic.java
[33] http://developers.itextpdf.com/content/zugferd-future-invoicing/2-creating-pdfa-files-itext

```
13      for (Invoice invoice : invoices) {
14          app.createPdf(invoice);
15      }
16      factory.close();
17  }
```

In this method, we create an instance of the `PdfInvoicesBasic`[34] class; we get a `List` of `Invoice`[35] objects from the `PojoFactory`[36]; for every invoice, we call the `createPdf()` method.

```
1   public void createPdf(Invoice invoice)
2       throws ParserConfigurationException, SAXException, TransformerException,
3       IOException, ParseException, DataIncompleteException, InvalidCodeException {
4
5       String dest = String.format(DEST, invoice.getId());
6
7           // Create the XML
8       InvoiceData invoiceData = new InvoiceData();
9       IBasicProfile basic = invoiceData.createBasicProfileData(invoice);
10      InvoiceDOM dom = new InvoiceDOM(basic);
11
12      // Create the ZUGFeRD document
13      ZugferdDocument pdfDocument = new ZugferdDocument(
14              new PdfWriter(dest), ZugferdConformanceLevel.ZUGFeRDBasic,
15              new PdfOutputIntent("Custom", "", "http://www.color.org",
16              "sRGB IEC61966-2.1", new FileInputStream(ICC)));
17      pdfDocument.addFileAttachment(
18          "ZUGFeRD invoice", dom.toXML(), "ZUGFeRD-invoice.xml",
19          PdfName.ApplicationXml, new PdfDictionary(), PdfName.Alternative);
20
21      // Create the document
22      Document document = new Document(pdfDocument);
23      document.setFont(PdfFontFactory.createFont(REGULAR, true))
24                          .setFontSize(12);
25      PdfFont bold = PdfFontFactory.createFont(BOLD, true);
26
27      // Add the header
28      document.add(
29          new Paragraph()
30          .setTextAlignment(TextAlignment.RIGHT)
```

[34] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/PdfInvoicesBasic.java
[35] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/Invoice.java
[36] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/pojo/PojoFactory.java

```
31          .setMultipliedLeading(1)
32                  .add(new Text(String.format("%s %s\n", basic.getName(), basic.getId()))
33                      .setFont(bold).setFontSize(14))
34                  .add(convertDate(basic.getDateTime(), "MMM dd, yyyy")));
35      // Add the seller and buyer address
36      document.add(getAddressTable(basic, bold));
37      // Add the line items
38      document.add(getLineItemTable(invoice, bold));
39      // Add the grand totals
40      document.add(getTotalsTable(
41          basic.getTaxBasisTotalAmount(), basic.getTaxTotalAmount(),
42          basic.getGrandTotalAmount(), basic.getGrandTotalAmountCurrencyID(),
43          basic.getTaxTypeCode(), basic.getTaxApplicablePercent(),
44          basic.getTaxBasisAmount(), basic.getTaxCalculatedAmount(),
45          basic.getTaxCalculatedAmountCurrencyID(), bold));
46      // Add the payment info
47      document.add(getPaymentInfo(basic.getPaymentReference(),
48          basic.getPaymentMeansPayeeFinancialInstitutionBIC(),
49          basic.getPaymentMeansPayeeAccountIBAN()));
50
51      document.close();
52  }
```

This `createPdf()` method consists of different parts:

- We construct a file name (line 5) and we use the `InvoiceData`[37] class (line 8) that was discussed in chapter 4[38] to create an `IBasicProfile` instance (line 9). We use this `IBasicProfile` instance to create an `InvoiceDOM` object (line 8). `InvoiceDOM` is one of the classes available in the pdfInvoice add-on.
- We construct a `ZugferdDocument` instance (line 13) and we set the conformance level to `ZUGFeRDBasic` for the basic profile (line 14). We also add the output intent (line 15-16). We add the XML invoice as an attachment (line 17-19).
- Then we create the document (line 22), and we add the content: a header (line 28-34), the address information of the seller and the buyer (line 36), the line items (line 38), the grand total and the tax information (line 40-45), and the payment information (line 47-49).

We create the header paragraph in the `createPdf()` method (lines 28-34), but we're using helper methods for the other content. Let's take a closer look at those methods.

[37] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/data/InvoiceData.java
[38] http://developers.itextpdf.com/content/zugferd-future-invoicing/4-creating-xml-invoices-itext

Adding the seller and buyer addresses

Creating a PDF from scratch using iText is easy, but not trivial. It's easy, because you can use many different high-level objects, but it's not trivial as you have to create a design by writing code. In chapter 6[39], we'll see an alternative way to create PDF invoices that doesn't require you to write much code.

Most of the data that needs to be rendered is available through the `IBasicProfile` interface. See for instance the `getAddressTable()` method:

```
1  public Table getAddressTable(IBasicProfile basic, PdfFont bold) {
2      Table table = new Table(new UnitValue[]{
3          new UnitValue(UnitValue.PERCENT, 50),
4          new UnitValue(UnitValue.PERCENT, 50)})
5              .setWidthPercent(100);
6      table.addCell(getPartyAddress("From:",
7          basic.getSellerName(),
8          basic.getSellerLineOne(),
9          basic.getSellerLineTwo(),
10         basic.getSellerCountryID(),
11         basic.getSellerPostcode(),
12         basic.getSellerCityName(),
13         bold));
14     table.addCell(getPartyAddress("To:",
15         basic.getBuyerName(),
16         basic.getBuyerLineOne(),
17         basic.getBuyerLineTwo(),
18         basic.getBuyerCountryID(),
19         basic.getBuyerPostcode(),
20         basic.getBuyerCityName(),
21         bold));
22     table.addCell(getPartyTax(basic.getSellerTaxRegistrationID(),
23         basic.getSellerTaxRegistrationSchemeID(), bold));
24     table.addCell(getPartyTax(basic.getBuyerTaxRegistrationID(),
25         basic.getBuyerTaxRegistrationSchemeID(), bold));
26     return table;
27 }
```

We create a table with two columns, and we use convenience methods to create the `Cell` instances:

[39] http://developers.itextpdf.com/content/zugferd-future-invoicing/6-creating-html-invoices

```
public Cell getPartyAddress(String who, String name,
    String line1, String line2, String countryID,
    String postcode, String city, PdfFont bold) {
        Paragraph p = new Paragraph()
        .setMultipliedLeading(1.0f)
        .add(new Text(who).setFont(bold)).add(NEWLINE)
        .add(name).add(NEWLINE)
        .add(line1).add(NEWLINE)
        .add(line2).add(NEWLINE)
        .add(String.format("%s-%s %s", countryID, postcode, city));
    Cell cell = new Cell()
        .setBorder(Border.NO_BORDER)
        .add(p);
    return cell;
}
public Cell getPartyTax(String[] taxId, String[] taxSchema, PdfFont bold) {
    Paragraph p = new Paragraph()
        .setFontSize(10).setMultipliedLeading(1.0f)
        .add(new Text("Tax ID(s):").setFont(bold));
    if (taxId.length == 0) {
        p.add("\nNot applicable");
    }
    else {
        int n = taxId.length;
        for (int i = 0; i < n; i++) {
            p.add(NEWLINE)
            .add(String.format("%s: %s", taxSchema[i], taxId[i]));
        }
    }
    return new Cell().setBorder(Border.NO_BORDER).add(p);
}
```

With these helper methods, we already have the “upper part” of the invoice as shown in Figure 5.1.

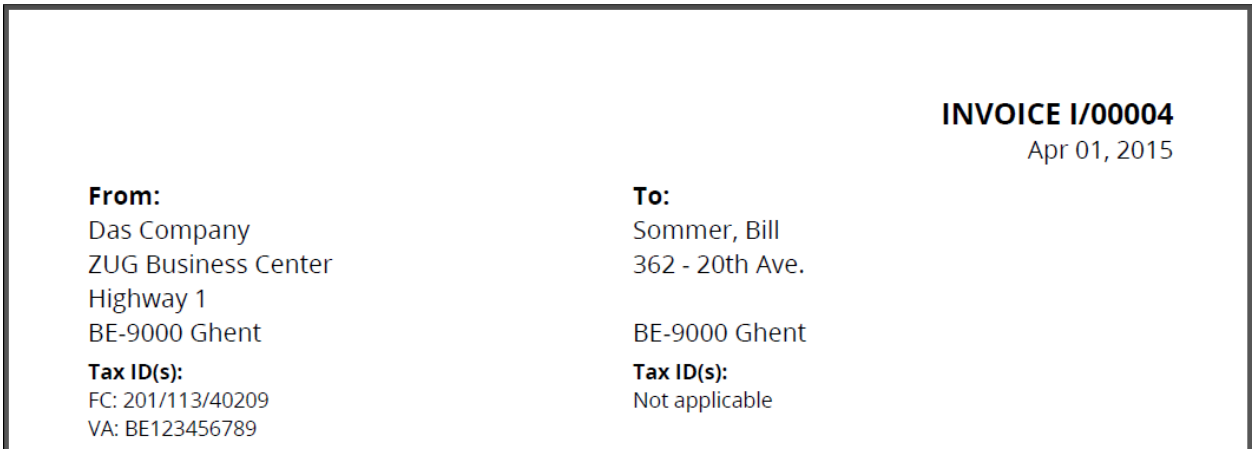


Figure 5.1: first part of the invoice

We'll also use a `Table` to render the invoice lines.

Adding invoice lines

The part with the invoice lines is a `Table` with six columns that is created like this:

```
1  public Table getLineItemTable(Invoice invoice, PdfFont bold) {
2      Table table = new Table(
3          new UnitValue[]{
4              new UnitValue(UnitValue.PERCENT, 43.75f),
5              new UnitValue(UnitValue.PERCENT, 12.5f),
6              new UnitValue(UnitValue.PERCENT, 6.25f),
7              new UnitValue(UnitValue.PERCENT, 12.5f),
8              new UnitValue(UnitValue.PERCENT, 12.5f),
9              new UnitValue(UnitValue.PERCENT, 12.5f)})
10         .setWidthPercent(100)
11     .setMarginTop(10).setMarginBottom(10);
12     table.addHeaderCell(createCell("Item:", bold));
13     table.addHeaderCell(createCell("Price:", bold));
14     table.addHeaderCell(createCell("Qty:", bold));
15     table.addHeaderCell(createCell("Subtotal:", bold));
16     table.addHeaderCell(createCell("VAT:", bold));
17     table.addHeaderCell(createCell("Total:", bold));
18     Product product;
19     for (Item item : invoice.getItems()) {
20         product = item.getProduct();
21         table.addCell(createCell(product.getName()));
22         table.addCell(createCell(
23             InvoiceData.format2dec(InvoiceData.round(product.getPrice()))))
```

```
                .setTextAlignment(TextAlignment.RIGHT));
        table.addCell(createCell(String.valueOf(item.getQuantity()))
            .setTextAlignment(TextAlignment.RIGHT));
        table.addCell(createCell(
            InvoiceData.format2dec(InvoiceData.round(item.getCost())))
            .setTextAlignment(TextAlignment.RIGHT));
        table.addCell(createCell(
            InvoiceData.format2dec(InvoiceData.round(product.getVat())))
            .setTextAlignment(TextAlignment.RIGHT));
        table.addCell(createCell(
            InvoiceData.format2dec(InvoiceData.round(
                item.getCost() + ((item.getCost() * product.getVat()) / 100))))
            .setTextAlignment(TextAlignment.RIGHT));
    }
    return table;
}
```

Again we use some helper methods to create `Cell` objects:

```
public Cell createCell(String text) {
    return new Cell().setPadding(0.8f)
        .add(new Paragraph(text)
            .setMultipliedLeading(1));
}
public Cell createCell(String text, PdfFont font) {
    return new Cell().setPadding(0.8f)
        .add(new Paragraph(text)
            .setFont(font).setMultipliedLeading(1));
}
```

The result looks like figure 5.2:

Item:	Price:	Qty:	Subtotal:	VAT:	Total:
Running jersey	8.00	9	72.00	21.00	87.12
Golf polo	8.00	1	8.00	21.00	9.68
Threadmill	600.00	2	1200.00	21.00	1452.00
Pro steel dartboard	25.00	2	50.00	21.00	60.50
My First Cookbook	17.00	1	17.00	6.00	18.02
Golf kit	80.00	2	160.00	21.00	193.60

Figure 5.2: rendering the line items of an invoice

We need yet another table for the tax totals and the grand total.

Adding the totals

We add a table with all the totals like this:

```
public Table getTotalsTable(String tBase, String tTax, String tTotal,
    String tCurrency, String[] type, String[] percentage, String base[],
    String tax[], String currency[], PdfFont bold) {
    Table table = new Table(
        new UnitValue[]{
            new UnitValue(UnitValue.PERCENT, 8.33f),
            new UnitValue(UnitValue.PERCENT, 8.33f),
            new UnitValue(UnitValue.PERCENT, 25f),
            new UnitValue(UnitValue.PERCENT, 25f),
            new UnitValue(UnitValue.PERCENT, 25f),
            new UnitValue(UnitValue.PERCENT, 8.34f)})
        .setWidthPercent(100);
    table.addCell(createCell("TAX:", bold));
    table.addCell(createCell("%", bold)
        .setTextAlignment(TextAlignment.RIGHT));
    table.addCell(createCell("Base amount:", bold));
    table.addCell(createCell("Tax amount:", bold));
    table.addCell(createCell("Total:", bold));
    table.addCell(createCell("Curr.:", bold));
    int n = type.length;
    for (int i = 0; i < n; i++) {
        table.addCell(createCell(type[i])
            .setTextAlignment(TextAlignment.RIGHT));
        table.addCell(createCell(percentage[i])
            .setTextAlignment(TextAlignment.RIGHT));
        table.addCell(createCell(base[i])
            .setTextAlignment(TextAlignment.RIGHT));
        table.addCell(createCell(tax[i])
            .setTextAlignment(TextAlignment.RIGHT));
        double total = Double.parseDouble(base[i]) + Double.parseDouble(tax[i]);
        table.addCell(createCell(
            InvoiceData.format2dec(InvoiceData.round(total)))
            .setTextAlignment(TextAlignment.RIGHT));
        table.addCell(createCell(currency[i]));
    }
    table.addCell(new Cell(1, 2).setBorder(Border.NO_BORDER));
    table.addCell(createCell(tBase, bold)
            .setTextAlignment(TextAlignment.RIGHT));
    table.addCell(createCell(tTax, bold)
```

```
40              .setTextAlignment(TextAlignment.RIGHT));
41      table.addCell(createCell(tTotal, bold)
42              .setTextAlignment(TextAlignment.RIGHT));
43      table.addCell(createCell(tCurrency, bold));
44      return table;
45  }
```

The code is very similar to what we did for the line items table. The only thing that is out of the ordinary, is that we create a cell that spans two columns in line 24-25. Figure 5.3 shows the result.

TAX:	%	Base amount:	Tax amount:	Total:	Curr.:
VAT	6.00	16.04	0.96	17.00	EUR
VAT	21.00	1231.40	258.60	1490.00	EUR
		1247.44	**259.56**	**1507.00**	**EUR**

Figure 5.3: rendering the totals

We're almost there. There only one piece of content missing.

Adding the payment info

Adding the payment info is just a matter of creating a `Paragraph`:

```
1   public Paragraph getPaymentInfo(String ref, String[] bic, String[] iban) {
2       Paragraph p = new Paragraph(String.format(
3           "Please wire the amount due to our bank account using "
4           + " the following reference: %s",
5           ref));
6       int n = bic.length;
7       for (int i = 0; i < n; i++) {
8           p.add(NEWLINE).add(String.format("BIC: %s - IBAN: %s", bic[i], iban[i]));
9       }
10      return p;
11  }
```

Note that we made some assumptions about the payment means. ZUGFeRD allows for different payment types, but we assumed that payments have to be done by bank wire. The result is shown in figure 5.4:

Please wire the amount due to our bank account using the following reference: 000000004

BIC: KREDBEBB - IBAN: BE 41 7360 0661 9710

BIC: GEBABEBB - IBAN: BE 56 0015 4298 7888

Figure 5.4: payment info

In the `createPdf()` method, we combined all this content with a PDF invoice as result.

The final result

The goal of this example was to create a proof of concept, and Figure 5.5 shows the final result.

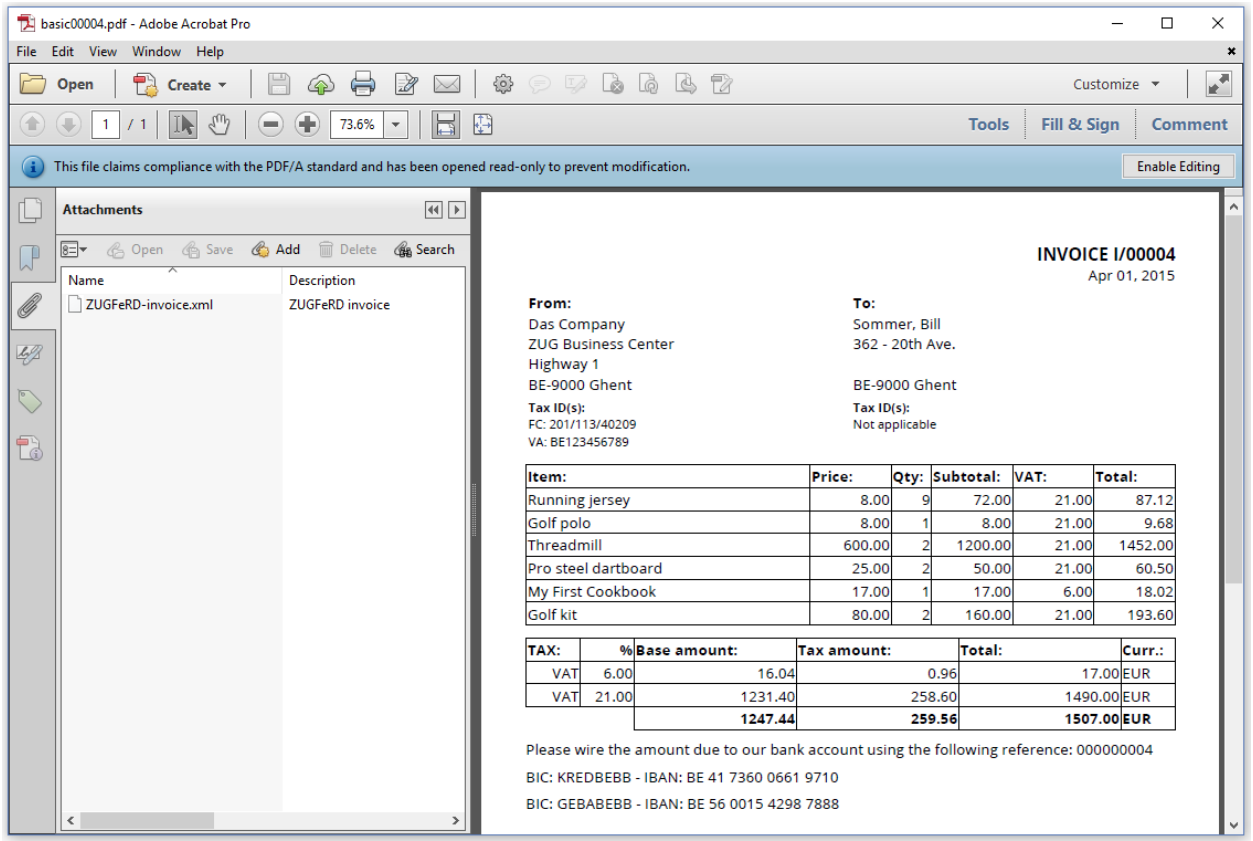


Figure 5.5: the resulting invoice

People who want to process the invoice manually, can do so; if they don't open the attachment panel, they won't even notice that there's an XML attachment inside. People who want to process the invoice automatically, can extract the XML or have their software extract the XML for processing.

I am aware that this invoice doesn't look very sexy. We could create tables with rounded borders, introduce a logo and some colors, we could add an extra sheet with terms-of-use, and so on, but that would lead us too far. In chapter 7, we'll discover that there is an alternative way to create PDF invoices, but let's take a look at some HTML first.

6. Creating HTML invoices

Please allow me to introduce a short intermezzo. The ZUGFeRD standard doesn't discuss HTML anywhere, yet I think it's useful to dedicate a chapter to the creating of HTML based on the ZUGFeRD data model. In the next chapter, you'll discover why.

An XSL for Comfort XMLs

As explained in chapter 5, the XMLs created to comply with the Basic profile don't contain sufficient information to create the visual appearance of the corresponding invoice. The XML doesn't contain all the information that is needed if we want to render the line items.

This means that we could create an XSL file that takes the info from the ZUGFeRD XML and converts it into HTML. We'll do that in the HtmlInvoicesComfort[40] example. For instance:

```
<xsl:template match="/rsm:CrossIndustryDocument">
    <html>
    <head><link rel="stylesheet" type="text/css" href="invoice.css" /></head>
    <body>
        <img src="logo.png" alt="Das Company - logo" /><br />
        <xsl:apply-templates /></body>
    </html>
</xsl:template>
```

In this snippet, we match the root tag of the ZUGFeRD XML and we introduce `<html>`, `<head>` and `<body>`. We'll use a CSS to apply some styles and colors and an `<img>` tag to introduce a logo.

The `<xsl:apply-templates />` instruction will deal with all the other components. For instance:

```
<xsl:template match="rsm:HeaderExchangedDocument">
    <h1 id="header"><xsl:value-of select="ram:Name" />
    <xsl:text> </xsl:text><xsl:value-of select="ram:ID" /></h1>
    <xsl:apply-templates select="ram:IssueDateTime" />
</xsl:template>
```

This XSL snippet in turn applies the templates for what's inside the `<ram:IssueDateTime>` tag. In this case, the following template is called:

[40] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/HtmlInvoicesComfort.java

```
<xsl:template match="udt:DateTimeString">
   <h2 id="date"><xsl:choose>
       <xsl:when test="@format='610'">
           <xsl:call-template name="YYYYMM">
               <xsl:with-param name="date" select="." />
           </xsl:call-template>
       </xsl:when>
       <xsl:when test="@format='616'">
           <xsl:call-template name="YYYYWW">
               <xsl:with-param name="date" select="." />
           </xsl:call-template>
       </xsl:when>
       <xsl:otherwise>
           <xsl:call-template name="YYYYMMDD">
               <xsl:with-param name="date" select="." />
           </xsl:call-template>
       </xsl:otherwise>
   </xsl:choose></h2>
</xsl:template>
```

This is what a `switch` looks like in XSL. Depending on the date format, one of the following templates is called:

```
<xsl:template name="YYYYMMDD">
    <xsl:param name="date" />
    <xsl:value-of select="substring($date,1,4)"
    />-<xsl:value-of select="substring($date,5,2)"
    />-<xsl:value-of select="substring($date,7,2)" />
</xsl:template>
<xsl:template name="YYYYMM">
    <xsl:param name="date" />
    <xsl:value-of select="substring($date,1,4)"
    />-<xsl:value-of select="substring($date,5,2)" />
</xsl:template>
<xsl:template name="YYYYWW">
    <xsl:param name="date" />
    <xsl:value-of select="substring($date,1,4)"
    />; week <xsl:value-of select="substring($date,5,2)" />
</xsl:template>
```

We won't go into further detail, but it shouldn't be difficult to extend the XSL so that it covers all the data you want to visualize. You can get some inspiration by looking at the XSL I wrote for this example and that results in invoices such as the one shown in figure 6.1.

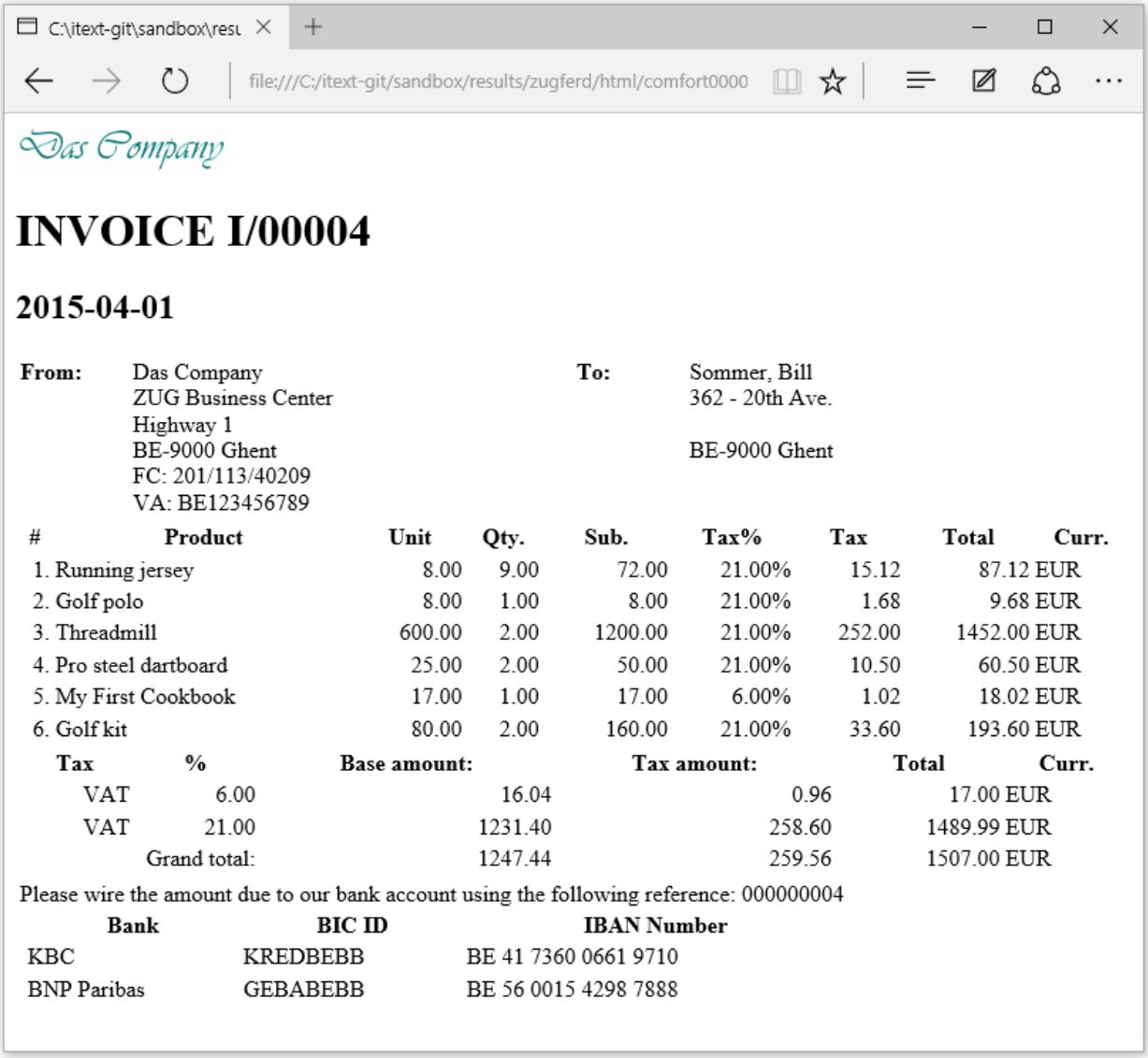


Figure 6.1: HTML version of an invoice

That's not a very attractive invoice yet, so let's introduce some styles and some color. We'll do that using CSS.

Adding some simple CSS

Please compare the result shown in figure 6.1 with the result shown in figure 6.2.

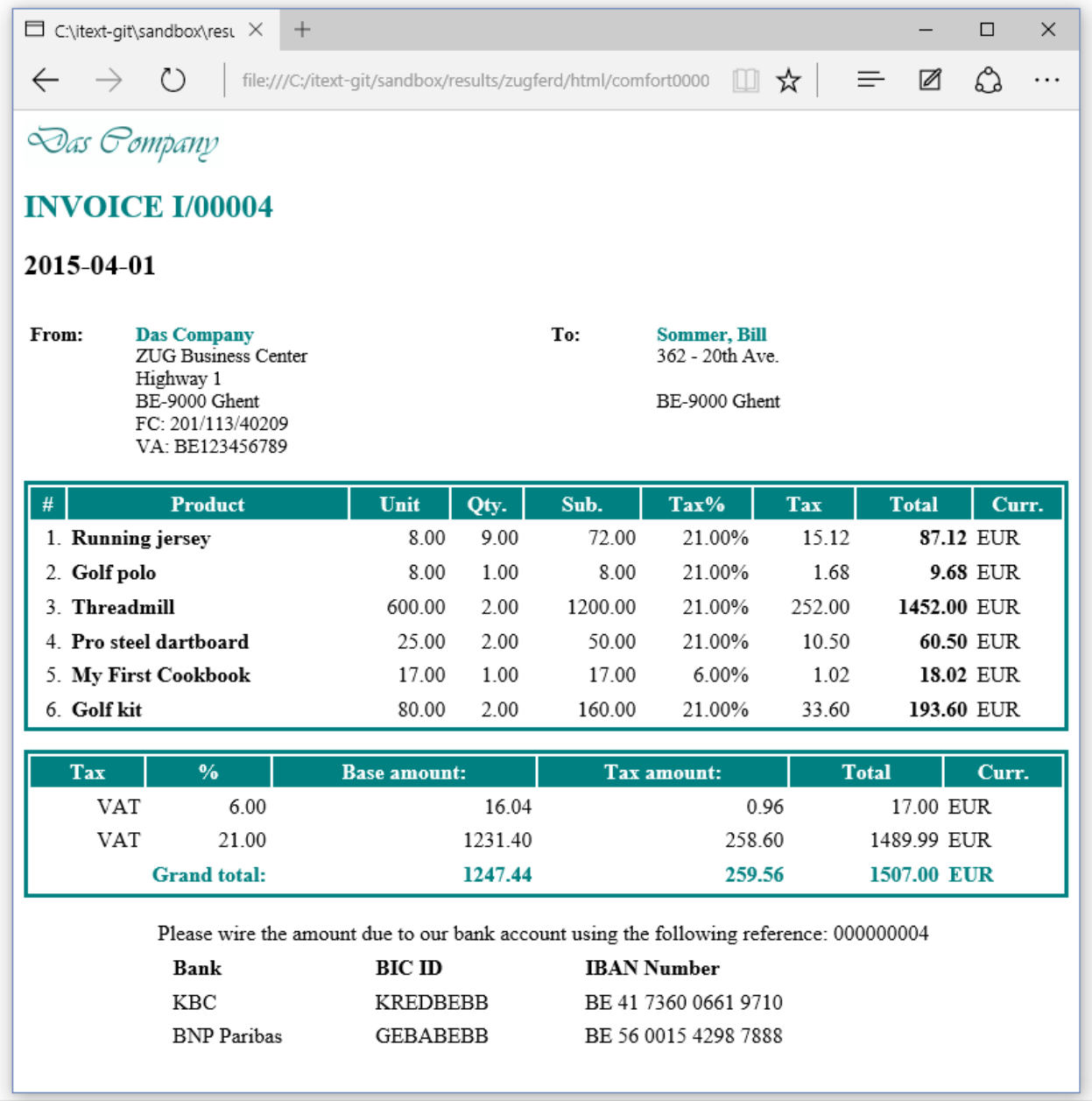


Figure 6.2: HTML + CSS version of an invoice

Now we're getting somewhere, aren't we? In our XSL, we've added some `id` and some `class` attributes, so that we can refer to these elements from a simple CSS file:

```
body { font-family: FreeSans; }
#header { color: #008080; font-size: 18pt; font-weight: bold; }
#date { font-size: 16pt; }
#addresses { margin-top: 20pt; font-size: 11pt; }
#products { margin-top: 10pt; border: 3px solid #008080; }
#totals { margin-top: 10pt; border: 3px solid #008080; }
#wireinfo { margin-top: 10pt; margin-left: 72pt; }
.name { font-weight: bold; color: #008080; }
.total { font-weight: bold; color: #008080; }
.headerrow { background-color: #008080; color: #FFFFFF; }
.bold { font-weight: bold; }
.wireheader { font-weight: bold; text-align: left; }
th { padding: 2pt; font-weight: bold; text-align: center; }
td { padding: 2pt; }
```

Let's take a look at the Java code that was used to produce these HTML pages.

Transforming XML to HTML using XSL and Java

We start by defining some constants, more specifically the pattern for the paths of the resulting HTML files, and the paths to the XSL, CSS, and logo file.

```
public static final String DEST = "results/zugferd/html/comfort%05d.html";
public static final String XSL = "resources/zugferd/invoice.xsl";
public static final String CSS = "resources/zugferd/invoice.css";
public static final String LOGO = "resources/zugferd/logo.png";
```

In the main method, we copy the resources (the CSS and the image) to the directory where we'll generate our HTML. The rest of the code is similar to what we had in chapter 5: we loop over the invoices obtained from the `PojoFactory` and we call a `createHtml()` method.

```
public static void main(String[] args)
    throws SQLException, IOException,
        ParserConfigurationException, SAXException, TransformerException,
        DataIncompleteException, InvalidCodeException {
    LicenseKey.loadLicenseFile(
        System.getenv("ITEXT7_LICENSEKEY")
        + "/itextkey-html2pdf_typography.xml");
    File file = new File(DEST);
    file.getParentFile().mkdirs();
    File css = new File(CSS);
```

```
11      copyFile(css, new File(file.getParentFile(), css.getName()));
12      File logo = new File(LOGO);
13      copyFile(logo, new File(file.getParentFile(), logo.getName()));
14      HtmlInvoicesComfort app = new HtmlInvoicesComfort();
15      PojoFactory factory = PojoFactory.getInstance();
16      List<Invoice> invoices = factory.getInvoices();
17      for (Invoice invoice : invoices) {
18          app.createHtml(invoice,
19              new FileWriter(String.format(DEST, invoice.getId())));
20      }
21      factory.close();
22  }
```

In the `createHtml()` method, we use the `InvoiceData` class to create an `IComfortProfile` instance. We then use standard Java code to convert XML using XSL.

```
1   public void createHtml(Invoice invoice, Writer writer)
2           throws IOException, ParserConfigurationException, SAXException,
3           DataIncompleteException, InvalidCodeException, TransformerException {
4       IComfortProfile comfort =
5           new InvoiceData().createComfortProfileData(invoice);
6       InvoiceDOM dom = new InvoiceDOM(comfort);
7       StreamSource xml = new StreamSource(new ByteArrayInputStream(dom.toXML()));
8       StreamSource xsl = new StreamSource(new File(XSL));
9       TransformerFactory factory = TransformerFactory.newInstance();
10      Transformer transformer = factory.newTransformer(xsl);
11      transformer.transform(xml, new StreamResult(writer));
12      writer.flush();
13      writer.close();
14  }
```

I'm adding the `copyFile()` method for the sake of completeness. We're copying the files from the resources to the output directory because we defined the paths to the CSS and the image using relative links.

```
private static void copyFile(File source, File dest) throws IOException {
    InputStream input = new FileInputStream(source);
    OutputStream output = new FileOutputStream(dest);
    byte[] buf = new byte[1024];
    int bytesRead;
    while ((bytesRead = input.read(buf)) > 0) {
        output.write(buf, 0, bytesRead);
    }
    input.close();
    output.close();
}
```

We could tweak the XSL and the CSS to produce an even nicer HTML document, but as I explained: invoices in HTML are outside the scope of ZUGFeRD. Let's move on to the next chapter to find out why we introduced this intermezzo.

7. Creating PDF invoices (Comfort)

iText 7 is a PDF library with several add-ons, one of which is called pdfHTML[41]. The pdfHTML[42] add-on is a powerful HTML to PDF conversion tool. I takes an HTML or HTML5 file, and converts it to PDF, taking into account CSS and media queries. Incidentally, we have just created a batch of invoices in the HTML format. I wonder if we could convert those to ZUGFeRD invoices…

Converting XML to HTML, and HTML to PDF

Please take a look at the PdfInvoicesComfort[43] example. We'll reuse some of the code from the example in chapter 6[44], but instead of merely a `createHtml()` method, we'll now call a `createPdf()` method:

```
public static void main(String[] args)
        throws SQLException, IOException,
        ParserConfigurationException, SAXException, TransformerException,
        DataIncompleteException, InvalidCodeException {
    LicenseKey.loadLicenseFile(
        System.getenv("ITEXT7_LICENSEKEY")
        + "/itextkey-html2pdf_typography.xml");
    File file = new File(DEST);
    file.getParentFile().mkdirs();
    PdfInvoicesComfort app = new PdfInvoicesComfort();
    PojoFactory factory = PojoFactory.getInstance();
    List<Invoice> invoices = factory.getInvoices();
    for (Invoice invoice : invoices) {
            app.createPdf(invoice,
                new FileOutputStream(String.format(DEST, invoice.getId())));
    }
    factory.close();
}
```

That `createPdf()` method will create an HTML file first, and then convert it to PDF:

[41] http://itextpdf.com/itext7/pdfHTML
[42] http://itextpdf.com/itext7/pdfHTML
[43] https://git.itextsupport.com/projects/I7JS/repos/zugferd/browse/src/main/java/com/itextpdf/zugferd/PdfInvoicesComfort.java
[44] http://developers.itextpdf.com/content/zugferd-future-invoicing/6-creating-html-invoices

```
1  public void createPdf(Invoice invoice, FileOutputStream fos)
2      throws IOException, ParserConfigurationException,
3      SAXException, TransformerException,
4      DataIncompleteException, InvalidCodeException {
5      IComfortProfile comfort =
6          new InvoiceData().createComfortProfileData(invoice);
7      InvoiceDOM dom = new InvoiceDOM(comfort);
8      StreamSource xml = new StreamSource(
9          new ByteArrayInputStream(dom.toXML()));
10     StreamSource xsl = new StreamSource(new File(XSL));
11     TransformerFactory factory = TransformerFactory.newInstance();
12     Transformer transformer = factory.newTransformer(xsl);
13     ByteArrayOutputStream baos = new ByteArrayOutputStream();
14     Writer htmlWriter = new OutputStreamWriter(baos);
15     transformer.transform(xml, new StreamResult(htmlWriter));
16     htmlWriter.flush();
17     htmlWriter.close();
18     byte[] html = baos.toByteArray();
19
20     ZugferdDocument pdfDocument = new ZugferdDocument(
21         new PdfWriter(fos), ZugferdConformanceLevel.ZUGFeRDComfort,
22         new PdfOutputIntent("Custom", "", "http://www.color.org",
23             "sRGB IEC61966-2.1", new FileInputStream(INTENT)));
24     pdfDocument.addFileAttachment(
25         "ZUGFeRD invoice", dom.toXML(), "ZUGFeRD-invoice.xml",
26         PdfName.ApplicationXml, new PdfDictionary(), PdfName.Alternative);
27     pdfDocument.setTagged();
28
29     HtmlConverter.convertToPdf(
30         new ByteArrayInputStream(html), pdfDocument, getProperties());
31 }
```

The body of this method consists of three parts:

- Line 5-18 are copied from the `createHtml()` method we created in the previous chapter, but as you can see in line 19, we don't create an HTML file that is stored on disk. Instead, we keep the HTML in memory.
- Line 21-28 start the same way as our example in chapter 5. We create a `ZugferdDocument` and we add the XML as an attachment. The line where we tell iText to tag the document isn't strictly necessary, but it's good practice.
- Line 30-31 is new: in chapter 5, we created `Paragraph` and `Table` objects, and we added those object to a `Document` instance. In this case, we'll leave the creation of building blocks to the

pdfHTML[45] add-on. The add-on will convert the content of `<p>` tags to `Paragraph` objects, the content of `<table>` tags to `Table` objects, and so on.

There are a couple of caveats, though. As we created the HTML in memory, the relative links to resources such as images and CSS can't be resolved. That's why we define a `ConverterProperties` instance. That instance is obtained through the `getProperties()` method:

```
1  public ConverterProperties getProperties() {
2      if (properties == null) {
3          properties = new ConverterProperties()
4              .setBaseUri("resources/zugferd/");
5      }
6      return properties;
7  }
```

In the converter properties, we define a base URI. The `resources/zugferd/` directory contains the `logo.png` and the `invoice.css` file. Without the base URI, the HTML to PDF conversion process would never know where to look for `./logo.png` or `./css.html`.

The `ConverterProperties` object can also be used to define other properties, such as the `FontProvider`. All ZUGFeRD documents are also PDF/A document, which means that all fonts need to be embedded. In this case, we used FreeSans as font, as defined in the first line of our CSS file: `body { font-family: FreeSans; }` Different fonts of the FreeSans font family are shipped with the pdfHTML add-on, and the default `FontProvider` knows where to find that font. If you want to use another font, you may need to create a custom `FontProvider` that tells iText where to find the fonts you need.

The final result

Figure 7.1 shows the resulting PDF. It looks much nicer than the invoices we produced in chapter 5, doesn't it?

[45] http://itextpdf.com/itext7/pdfHTML

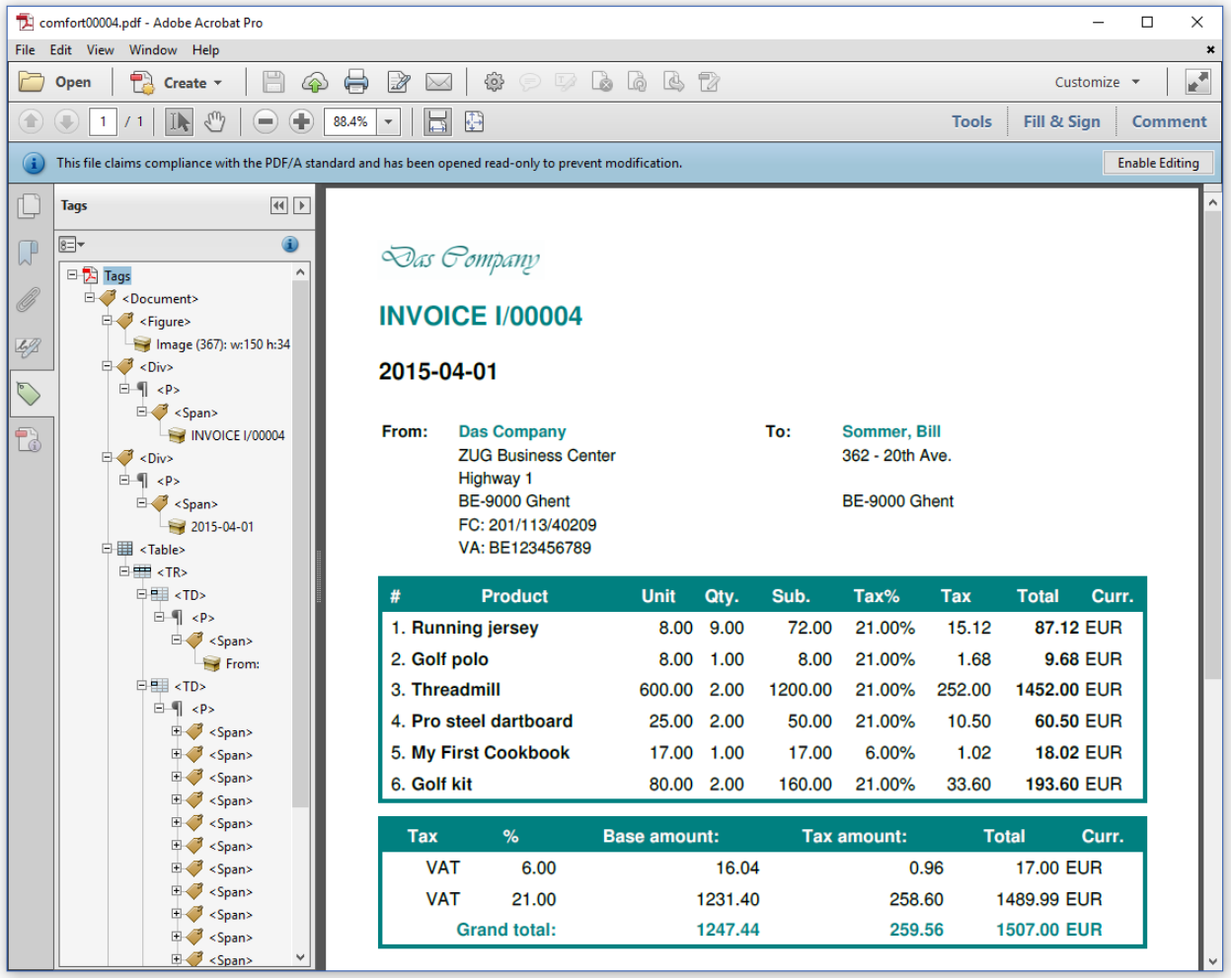


Figure 7.1: a ZUGFeRD invoice created from HTML

If you look at the panel to the left, you can see the effect of the extra line we smuggled into our code (`pdfDocument.setTagged();`). As a result, the invoice is now also accessible. The HTML tables can now also be interpreted as tables by a PDF processor that understands tagged PDF. This is an example of a future-proof, archivable invoice that can be read by humans (including people who are visually impaired) as well as by machines.

Before we close, let's also take a look inside the PDF document. In figure 7.2, we see the root object of the PDF document, aka the catalog. We also see the `/AF` (Associated Files) and the `/Names` entry We recognize the `/EmbeddedFiles` name tree that has a single element named `"ZUGFeRD invoice"`. This refers to a dictionary of filetype `/Filespec` that is also referred to from the `/AF` array. The `/AFRelationship` is `"Altermative"`, meaning that the PDF document and the attached XML are alternative presentations of the same content. This is required by the ZUGFeRD standard.

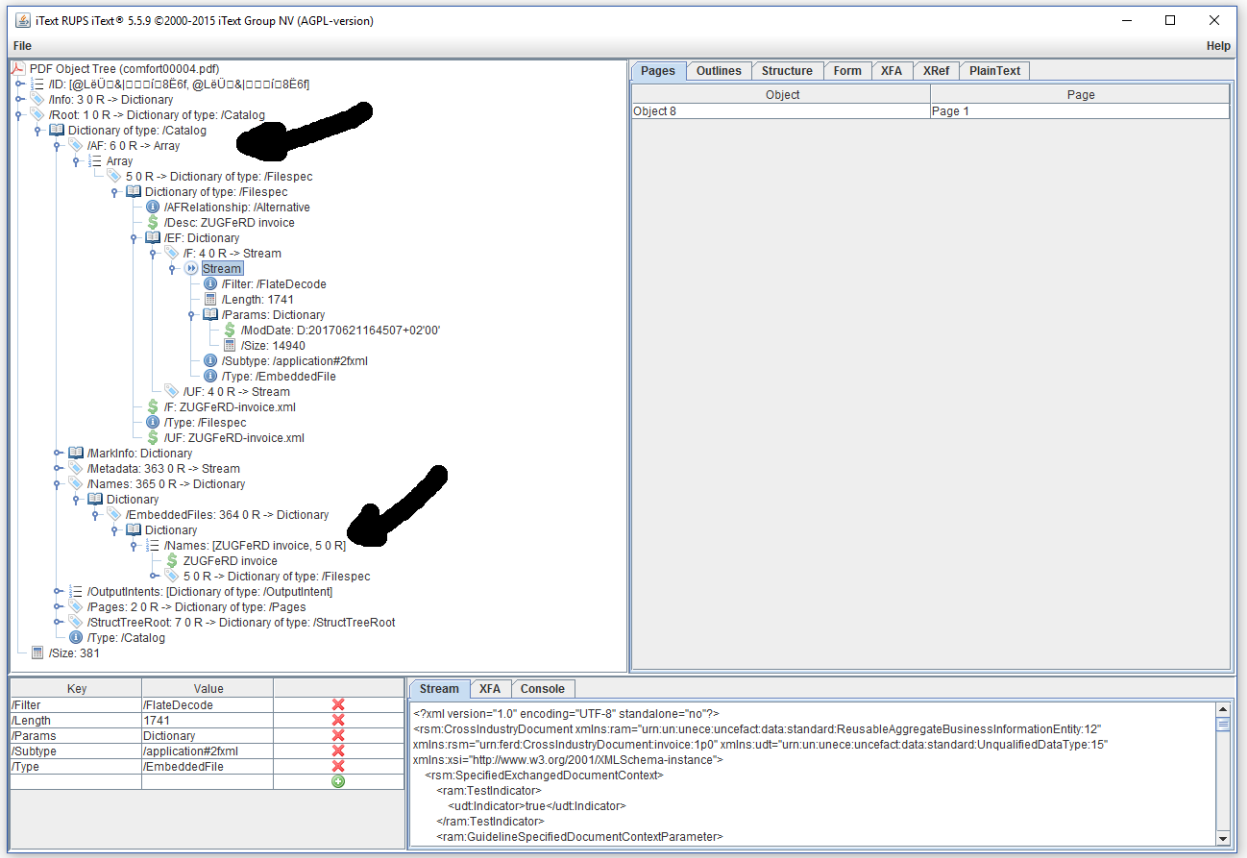


Figure 7.2: The Associated File

Figure 7.3 shows the XMP metadata. The document knows that it's a PDF/A-3B document because of the presence of `pdfaid:part="3"` and `pdfaod:conformance="B"` attributes in the `rdf:Description`. There are also a number of entries in the `zf` namespace, that define the profile (`zf:ConformanceLevel="COMFORT"`), the name of the XML attachment (`zf:DocumentFileName="ZUGFeRD-invoice.xml"`), the document type (`zf:DocumentType="INVOICE"`), and the version of the XML schema for the invoice data (`zf:Version="1.0"`).

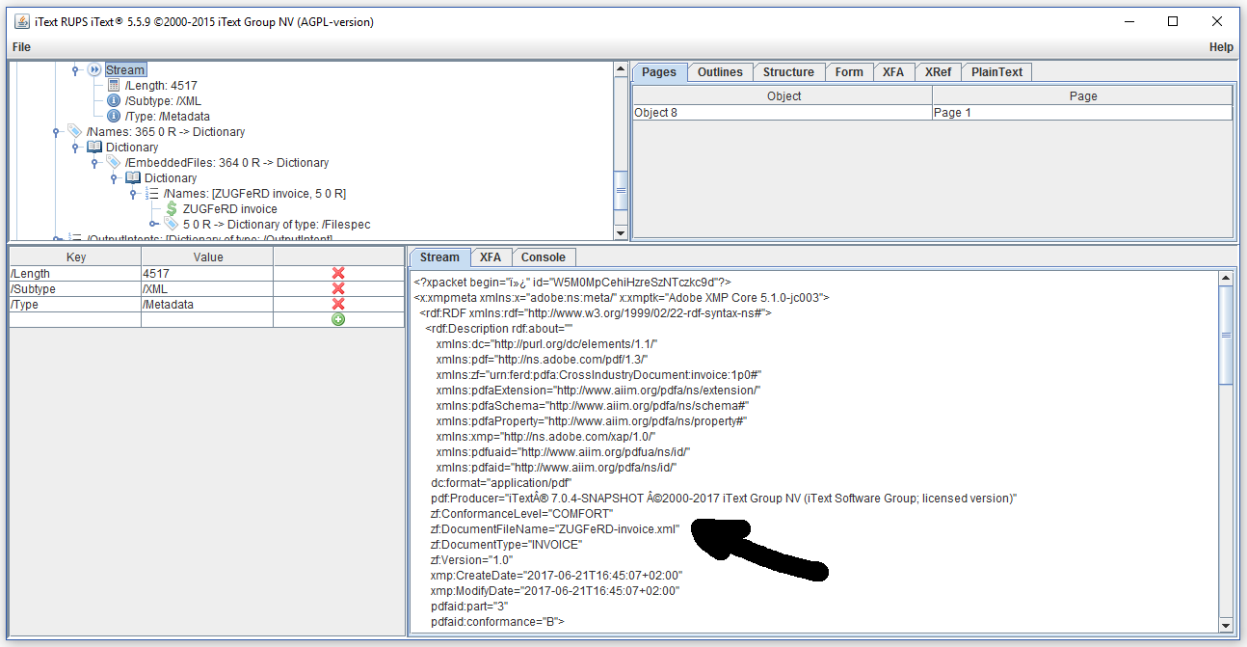


Figure 7.3: The XMP Metadata

You don't have to worry about these Metadata values or the embedded XMP stream. The pdfInvoice add-on[46] automatically takes care of creating this metadata.

[46] http://itextpdf.com/itext7/pdfInvoice

Conclusion

I started this tutorial by explaining that I'm not always happy when I receive an invoice.

In the first chapter of this book, I explained the issues I have with many invoices:

- They aren't future proof,
- They aren't accessible,
- Either they can be read by humans but not by machines,
- or the can be read by machines but not by humans.

In the second chapter, I explained how we can fix the first two problems: we can use iText to create invoices in the PDF/A format so that they are future proof; we can introduce Tagged PDF and conform with PDF/A level A to make our invoices accessible.

Chapter three introduced a simple database. We have used this database in all the following chapters.

In chapter four, we created invoices in the XML format so that the can be correctly interpreted by machines. These XML invoices conformed to the Comfort level of the data model of the ZUGFeRD standard.

In chapter 5 we created invoices that can be read by humans. We added the XML version of the invoice as an attachment.

Chapter 6 started as a side-track: we wrote some XSL to convert ZUGFeRD XML files to HTML, and we used CSS to introduce some styles and colors.

We used the HTML files in chapter 7 to create ZUGFeRD invoices with XML Worker.

We hope that this tutorial shows that it's not that difficult to create ZUGFeRD invoices. First you implement the `ComfortProfile` interface. You do so to provide all the information that needs to be on the invoice. Then you create a design for your invoices using XSL that converts the ZUGFeRD data model into an HTML file. Optionally, you can add some CSS to define colors and styles. Finally you use iText and XML Worker to combine all these different elements into a finalized ZUGFeRD invoice.

This is enterprise-grade technology that every company can afford, whether it's a large, medium or even a small business. There is no reason not to adopt ZUGFeRD as *your* standard for invoices. You can help realize world-wide implementation of the ZUGFeRD standard. This will yield financial, technical and operational benefits across the entire economy and across all borders. What are you waiting for?