Automated PDF Tests with Java

PDFUnit-Java

Carsten Siedentop

Table of Contents

Preface

The Current Situation of Testing PDF in Projects

These days telephone bills, insurance policies, official notifications and many types of contract are delivered as PDF documents. They are the result of a process chain consisting of programs in various programming languages using numerous libraries. Depending on the complexity of the documents to be produced, such programming is not easy. The software may have errors and statistically will have errors. So it should be tested in some of the following ways:

- Is the expected text within the expected page region?
- Is the bar code's text the expected text?
- Does the layout fulfill the requirements?
- Do the embedded ZUGFeRD data have the expected values?
- Are the values of the embedded ZUGFeRD data correspond with the visible values?
- Does a PDF comply with DIN 5008?
- Is the PDF signed? When and by whom?

It should scare developers, project managers and CEO's that until now there is almost no way of repeatedly testing PDF documents. And even the options which are available are not used as frequently as they should be. Unfortunately, manual testing is widespread. It is expensive and prone to errors.

With PDFUnit, any document, whether created using a powerful design tool, exported from MS Word or LibreOffice, processed using an API, or dropped out of an XSL-FO workflow, can be tested.

Intuitive API

The interface of PDFUnit follows the principle of "fluent builder". All names of classes and methods try to follow the English language as closely as possible and thus support general thought patterns.

The next example shows the simplicity of the API:

```
String filename = "documentUnderTest.pdf";
int leftX  =  17;  // in millimeter
int upperY =  45;
int width  =  80;
int height =  50;
PageRegion addressRegion = new PageRegion(leftX, upperY, width, height);

AssertThat.document(filename)
          .restrictedTo(FIRST_PAGE)
          .restrictedTo(addressRegion)
          .hasText()
          .containing("John Doe Ltd.")
;
```

To write successful tests it is neither necessary for a test developer to know the structure of PDF nor to know anything about the creation of the PDF document.

Time to Start

Don't gamble with the data and processes of your document processing system. Check the output of the PDF creating programs with automated tests.

Chapter 1. About this Documentation

Who Should Read it

This documentation is addressed to all people who are involved in the process of creating PDF documents. That are not only software developers who create programs which produce PDF documents, that are also members of the quality assurance staff and project managers.

Code Examples

The code examples in the following chapters are colorized similar to the Eclipse-editor to facilitate the reading of this documentation. They are combined with JUnit but you can also use PDFUnit with TestNG. A code example with TestNG is shown in the appendix. A demo project with many examples is available here: http://www.pdfunit.com/en/download/index.html.

Javadoc

The Javadoc documentation of the API is available online: http://www.pdfunit.com/api/javadoc/index.html.

Other Programming Languages

PDFUnit is available not only for Java, but also for .NET and XML. Separate documentation exists for each language.

If there are Problems

If you have problems to test a PDF, please search for a similar problem in the internet. Maybe, you find a solution. Finally, you are invited to write to info[at]pdfunit.com and describe the problem. We'll try to help you.

New Features Wanted?

Do you need other test functions? Please feel free to send your requirements to info@pdfunit.com. You are invited to influence the further development of PDFUnit.

Responsibility

Some examples in this book use PDF documents from the web. For legal reasons I make clear that I dissociate myself from their content, for instance I can not read Chinese. These documents support tests, for which I could not create my own test documents, e.g. the Chinese PDF documents.

Acknowledgement

Axel Miesen developed the Perl-API of PDFUnit and during that time he asked a lot of questions about the Java API. PDFUnit Java has profitted greatly from his input. Thank you, Axel.

Unfortunately, my English is not as good as I would like. But my colleague John Boyd-Rainey read the first version of this English documentation and corrected a huge number of misplaced commas and other typical errors. Thank you, John, for your perseverance and thoroughness. However, all remaining errors are my fault. He also asked critical questions which helped me to sharpen some descriptions.

Production of this Documentation

This documentation was created with DocBook-XML and both PDF and HTML are generated from one text source. It is well known that the layout can be improved in both formats, e.g. the pagebreaks

in PDF format. And improving the layout is already on the to-do list, but there are other tasks with higher priority.

Feedback

Any kind of feedback is welcomed. Please write to info[at]pdfunit.com.

Chapter 2. Quickstart

Quickstart

Let's assume you have a some programs that generates PDF documents and you want to make sure that the programs do what they should. Further expect that your test document has exactly one page and contains the greeting “Thank you for using our services.” and the value “30.34 Euro” for the total bill. Then it's easy to check these requirements:

```
@Test
public void hasOnePage() throws Exception {
  String filename = "quickstart/quickstartDemo_en.pdf";
  AssertThat.document(filename)
            .hasNumberOfPages(1)
  ;
}

@Test
public void letterHasExpectedRegards() throws Exception {
  String filename = "quickstart/quickstartDemo_en.pdf";
  String expectedRegards = "Thank you for using our services.";
  AssertThat.document(filename)
            .restrictedTo(LAST_PAGE)
            .hasText()
            .containing(expectedRegards)
  ;
}

@Test
public void hasExpectedCharge() throws Exception {
  String filename = "quickstart/quickstartDemo_en.pdf";
  int leftX  =  172;  // in millimeter
  int upperY =  178;
  int width  =   20;
  int height =    9;
  PageRegion regionInvoiceTotal = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(regionInvoiceTotal)
            .hasText()
            .containing("30,34 Euro") // This is really expected.
            .containing("29,89 Euro") // Let's see an error message :-)
  ;
}
```

The typical JUnit report shows the success or the failures with meaningful messages:

Class com.pdfunit.test.quickstart.QuickstartTests_en

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
QuickstartTests_en	3	1	0	0	0.021	2016-03-25T11:39:09	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
hasOnePage	Success		0.002
hasExpectedCharge	Error	Page(s) [1] of 'C:\...\quickstart\quickstartDemo_en.pdf' not containing the expected text: '29,89 Euro'. Found: '30,34 Euro'. com.pdfunit.errors.PDFUnitValidationException: Page(s) [1] of 'C:\...\quickstart\quickstartDemo_en.pdf' not containing the expected text: '29,89 Euro'. Found: '30,34 Euro'. at com.pdfunit.internal.matcher.page.TextContainingMatcher.throwExceptionWhenMismatch(TextContainingMatcher.java:63) at com.pdfunit.validators.TextValidator.containing(TextValidator.java:92) at com.pdfunit.validators.TextValidator.containing(TextValidator.java:81) at com.pdfunit.test.quickstart.QuickstartTests_en.hasExpectedCharge(QuickstartTests_en.java:71) at java.util.concurrent.FutureTask.run(FutureTask.java:266) at java.lang.Thread.run(Thread.java:745)	0.007
hasExpectedRegards	Success		0.008

That's it. The following chapters describe the features, typical test scenarios and typical problems when testing PDF documents.

Chapter 3. Test Scopes

3.1. Overview

An Introduction to the Syntax

Each test for a **single PDF document** begins with the method `AssertThat.document(..)`. The file to be tested is passed as a parameter. Then each following function opens a different test scope, e.g. content, fonts, layouts, etc.:

```
// Instantiation of PDFUnit for a single document:

AssertThat.document(filename)          13.1: “Instantiation of PDF Documents” (p. 155)
          ...
// Switch to one of many test scopes.

// Compare one PDF with a reference:
AssertThat.document(filename)          ❶
          .and(..)                     4.1: “Overview” (p. 87)
          ...
```

❶ The PDF document can be passed to the function as a `String`, `File`, `InputStream`, `URL` or `byte[]`.

It is possible to write test for a given **set of PDF documents**. Such tests start with the method `AssertThat.eachDocument(..)`:

```
// Instantiation of PDFUnit for multiple documents:
...
File[] files = {file1, file2, file3};
AssertThat.eachDocument(filename)    ❷ 5: “Folders and Multiple Documents” (p. 101)
          .hasText(..)
          .containing(..)
;
```

❷ The PDF documents can be passed to the function as a `String[]`, `File[]`, `InputStream[]`, or `URL[]`.

A test can also cover **all PDF documents in a folder**. Such tests start with the method `AssertThat.eachDocument().inFolder(..)`:

```
// Instantiation of PDFUnit for a folder:
...
File folderToCheck = new File(..);
AssertThat.eachDocument()
          .inFolder(folderToCheck) ❸ 5.3: “Validate all documents in a folder” (p. 102)
          .hasText(..)
          .containing(..)
;
```

❸ All PDF documents in this folder will be checked. PDF documents in subfolders will not be checked.

Exception

Tests that expect an exception have to catch the class `PDFUnitValidationException`.

Here is an example for a test which expects an exception:

```
@Test(expected=PDFUnitValidationException.class)
public void isSigned_DocumentNotSigned() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .isSigned()
  ;
}
```

Test Scopes

The following list gives a complete overview over the test scopes of PDFUnit. The link behind each method points to the chapter where the test scope is described:

```
// Every one of the following methods opens a new test scope:

.asRenderedPage()                 3.20: “Layout - Entire PDF Pages” (p. 48)

.containsImage(..)                3.16: “Images in PDF Documents” (p. 40)
.containsOneImageOf(..)           3.16: “Images in PDF Documents” (p. 40)

.hasAuthor()                      3.9: “Document Properties” (p. 25)
.hasBookmark()                    3.5: “Bookmarks and Named Destinations” (p. 18)
.hasBookmarks()                   3.5: “Bookmarks and Named Destinations” (p. 18)
.hasCreationDate()                3.7: “Dates” (p. 22)
.hasCreator()                     3.7: “Dates” (p. 22)
.hasEmbeddedFile(..)              3.3: “Attachments” (p. 14)
.hasEncryptionLength(..)          3.24: “Passwords” (p. 53)
.hasField(..)                     3.13: “Form Fields” (p. 32)
.hasFields(..)                    3.13: “Form Fields” (p. 32)
.hasFont()                        3.12: “Fonts” (p. 29)
.hasFonts()                       3.12: “Fonts” (p. 29)
.hasFormat(..)                    3.15: “Format” (p. 39)
.hasImage(..)                     3.16: “Images in PDF Documents” (p. 40)
.hasJavaScript()                  3.17: “JavaScript” (p. 43)
.hasJavaScriptAction()            3.2: “Actions” (p. 12)
.hasKeywords()                    3.9: “Document Properties” (p. 25)
.hasLanguageInfo(..)              3.18: “Language” (p. 45)
.hasLayer()                       3.19: “Layers” (p. 46)
.hasLayers()                      3.19: “Layers” (p. 46)
.hasLessPagesThan()               3.23: “Page Numbers as Objectives” (p. 52)
.hasLocalGotoAction()             3.2: “Actions” (p. 12)
.hasModificationDate()            3.7: “Dates” (p. 22)
.hasMorePagesThan()               3.23: “Page Numbers as Objectives” (p. 52)
.hasNamedDestination()            3.5: “Bookmarks and Named Destinations” (p. 18)

.hasNoAuthor()                    3.9: “Document Properties” (p. 25)
.hasNoCreationDate()              3.7: “Dates” (p. 22)
.hasNoCreator()                   3.9: “Document Properties” (p. 25)
.hasNoImage()                     3.16: “Images in PDF Documents” (p. 40)
.hasNoKeywords()                  3.9: “Document Properties” (p. 25)
.hasNoLanguageInfo()              3.18: “Language” (p. 45)
.hasNoModificationDate()          3.7: “Dates” (p. 22)
.hasNoProducer()                  3.9: “Document Properties” (p. 25)
.hasNoProperty(..)                3.9: “Document Properties” (p. 25)
.hasNoSubject()                   3.9: “Document Properties” (p. 25)
.hasNoText()                      3.30: “Text” (p. 63)
.hasNoTitle()                     3.9: “Document Properties” (p. 25)
.hasNoXFAData()                   3.37: “XFA Data” (p. 76)
.hasNoXMPData()                   3.38: “XMP Data” (p. 79)

.hasNumberOf...()                 3.22: “Number of PDF Elements” (p. 51)

.hasOCG()                         3.19: “Layers” (p. 46)
.hasOCGs()                        3.19: “Layers” (p. 46)
.hasOwnerPassword(..)             3.24: “Passwords” (p. 53)
.hasPermission()                  3.26: “Permissions” (p. 55)
.hasProducer()                    3.9: “Document Properties” (p. 25)
.hasProperty(..)                  3.9: “Document Properties” (p. 25)
.hasSignatureField(..)            3.28: “Signed PDF” (p. 59)
.hasSignatureFields()             3.28: “Signed PDF” (p. 59)
.hasSubject()                     3.9: “Document Properties” (p. 25)
.hasText()                        3.30: “Text” (p. 63)
.hasTitle()                       3.9: “Document Properties” (p. 25)
.hasUserPassword(..)              3.24: “Passwords” (p. 53)
.hasVersion()                     3.36: “Version Info” (p. 75)
.hasXFAData()                     3.37: “XFA Data” (p. 76)
.hasXMPData()                     3.38: “XMP Data” (p. 79)
.hasZugferdData()                 3.39: “ZUGFeRD” (p. 82)

.haveSame...()                    4.1: “Overview” (p. 87)

.isCertified()                    3.6: “Certified PDF” (p. 21)
.isCertifiedFor(..)               3.6: “Certified PDF” (p. 21)
.isLinearizedForFastWebView()     3.11: “Fast Web View” (p. 29)
.isSigned()                       3.28: “Signed PDF” (p. 59)
.isSignedBy(..)                   3.28: “Signed PDF” (p. 59)
.isTagged()                       3.29: “Tagged Documents” (p. 62)

.restrictedTo(..)                 13.2: “Page Selection” (p. 155)
```

Sometimes, two methods are needed to enter a test scope:

```
// Validation of bar code and QR code:
.hasImage().withBarcode()                 3.4: “Bar Code” (p. 16)
.hasImage().withQRcode()                  3.27: “QR Code” (p. 56)

// Validation based on Excel files:
.compliesWith().constraints(excelRules)  3.10: “Excel Files for Validation Constraints” (p. 28)

// Validation of DIN5008 constraints:
.compliesWith().din5008FormA()            3.8: “DIN 5008” (p. 24)
.compliesWith().din5008FormB()            3.8: “DIN 5008” (p. 24)
.compliesWith().pdfStandard()             3.25: “PDF/A” (p. 54)

// Validation around ZUGFeRD:
.compliesWith().zugferdSpecification(..) 3.39: “ZUGFeRD” (p. 82)
```

PDFUnit is continuously being improved and the manual kept up to date. Wishes and requests for new functions are appreciated. Please, send them to info[at]pdfunit.com.

3.2. Actions

Overview

“Actions” make PDF documents interactive and more complex. “Complex” means that they should be tested, especially when interactive documents are part of a workflow.

An “action” is a dictionary object inside PDF containing the keys `/S` and `/Type`. The key `/Type` always maps to the value “Action”. And the the key `/S` (Subtype) has different values:

```
// Types of actions:

GoTo:        Set the focus to a destination in the current PDF document
GoToR:       Set the focus to a destination in another PDF document
GoToE:       Go to a destination inside an embedded file
GoTo3DView:  Set the view to a 3D annotation
Hide:        Set the hidden flag of the specified annotation
ImportData:  Import data from a file to the current document
JavaScript:  Execute JavaScript code
Movie:       Play a specified movie
Named:       Execute an action, which is predefined by the PDF viewer
Rendition:   Control the playing of multimedia content
ResetForm:   Set the values of form fields to default
SetOCGState: Set the state of an OCG
Sound:       Play a specified sound
SubmitForm:  Send the form data to an URL
Launch:      Execute an application
Thread:      Set the viewer to the beginning of a specified article
Trans:       Update the display of a document, using a transition dictionary
URI:         Go to the remote URI
```

For a few of these actions PDFUnit provides test methods. Future releases will have more tests methods.

```
// Simple tests:

.hasNumberOfActions(..)

// Action specific tests:

.hasJavaScriptAction()
.hasJavaScriptAction().containing(..)
.hasJavaScriptAction().containingSource(..)
.hasJavaScriptAction().equalsTo(..)
.hasJavaScriptAction().equalsToSource(..)
.hasJavaScriptAction().matchingRegex(..)

.hasLocalGotoAction()
.hasLocalGotoAction().toDestination(..)
.hasLocalGotoAction().toDestination(.., pageNumber)
```

The following sections show examples for these test methods.

JavaScript-Actions

Since JavaScript code is generally quite long, it makes sense to read the expected text for a JavaScript-Action test from a file:

```
@Test
public void hasJavaScriptAction_WithWhitespaceProcessing() throws Exception {
  String filename = "documentUnderTest.pdf";
  String scriptFileName = "javascript/javascriptAction_OneSimpleAlert.js";
  Reader scriptFileAsReader = new FileReader(scriptFileName);

  AssertThat.document(filename)
            .hasJavaScriptAction()
            .equalsToSource(scriptFileAsReader, WhitespaceProcessing.IGNORE)
  ;
}
```

The parameter of the method `equalsToSource(..)` (and also `containingSource(..)`) can be of type `java.io.Reader`, `java.io.InputStream` or `java.lang.String`.

The content of the JavaScript file is completely compared with the content of the JavaScript action. White spaces are ignored.

But parts of a JavaScript code can be checked as well:

```
@Test
public void hasJavaScript_ContainingText_FunctionNames() throws Exception {
  String filename = "javaScriptClock.pdf";

  AssertThat.document(filename)
            .hasJavaScript()
            .containing("StopWatchProc")
            .containing("SetFldEnable")
            .containing("DoTimers")
            .containing("ClockProc")
            .containing("CountDownProc")
            .containing("CDEnables")
            .containing("SWSetEnables")
  ;
}
```

Chapter 13.5: “Whitespace Processing” (p. 159) explains the flexible handling of whitespaces.

Goto-Actions

Goto-Actions need a destination in the same PDF document:

```
@Test
public void hasGotoAction_ToNamedDestination() throws Exception {
  String filename = "documentUnderTest.pdf";
  String destinationName21 = "destination2.1";

  AssertThat.document(filename)
            .hasLocalGotoAction()
            .toDestination(destinationName21)
  ;
}
```

This test is successful if the tested PDF contains the expected destination “destination2.1”. Chapter 4.11: “Comparing Named Destinations” (p. 95) discusses how to test destinations together with bookmarks.

It is also possible to check whether a destination is located on a given page:

```
@Test
public void hasGotoAction_ToDestinationWithPage_Page3() throws Exception {
  String filename = "documentUnderTest.pdf";
  String destinationName21 = "destination2.1";
  int page3 = 3;

  AssertThat.document(filename)
            .hasLocalGotoAction()
            .toDestination(destinationName21, page3)
  ;
}
```

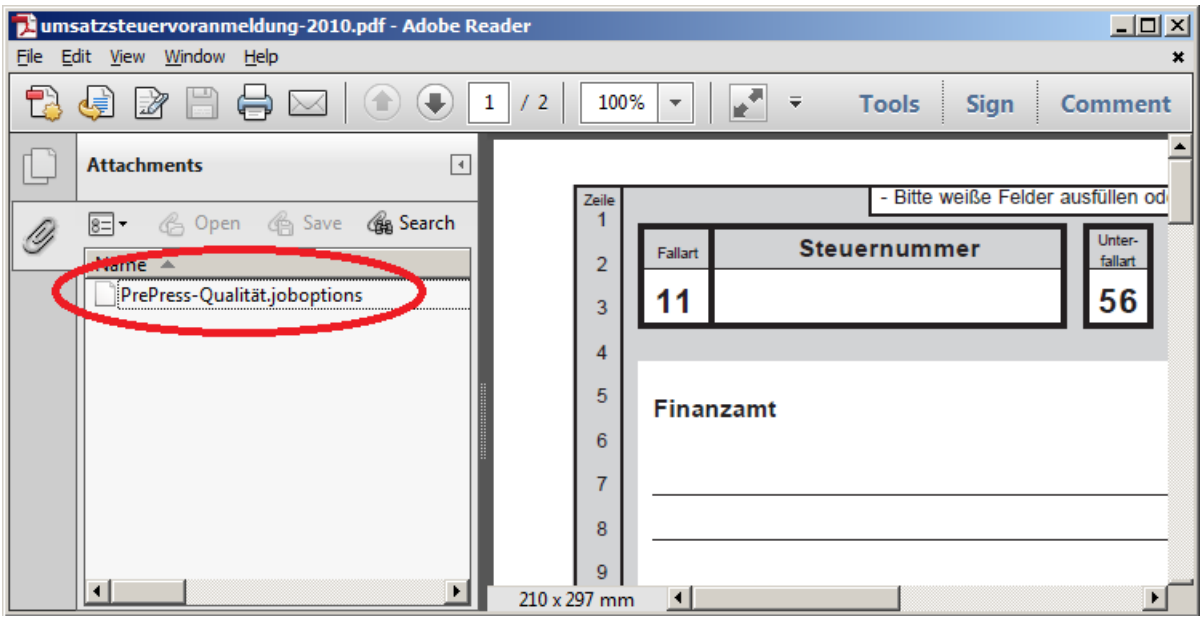
3.3. Attachments

Overview

Files that are embedded in PDF documents, often play an important role in post-processing steps. Therefore PDFUnit provides test methods for these attachments:

```
// Simple tests:
.hasEmbeddedFile()
.hasNumberOfEmbeddedFiles(..)

// More detailed tests:
.hasEmbeddedFile().withContent(..)
.hasEmbeddedFile().withName(..)
```

The following tests are using “umsatzsteuervoranmeldung-2010.pdf”, a PDF form for the German sales tax return of 2010. It contains a file named “PrePress-Qualität.joboptions”.



Existence of Attachments

A very simple test is to check whether an embedded file exists:

```
@Test
public void hasEmbeddedFile() throws Exception {
  String filename = "umsatzsteuervoranmeldung-2010.pdf";

  AssertThat.document(filename)
            .hasEmbeddedFile()
  ;
}
```

Number of Attachments

The next test verifies the expected number of embedded files:

```
@Test
public void hasNumberOfEmbeddedFiles() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfEmbeddedFiles(1)
  ;
}
```

Filename

Also, the names of embedded files can be tested:

```
@Test
public void hasEmbeddedFile_WithName() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasEmbeddedFile().withName("PrePress-Qualität.joboptions")
  ;
}
```

Content

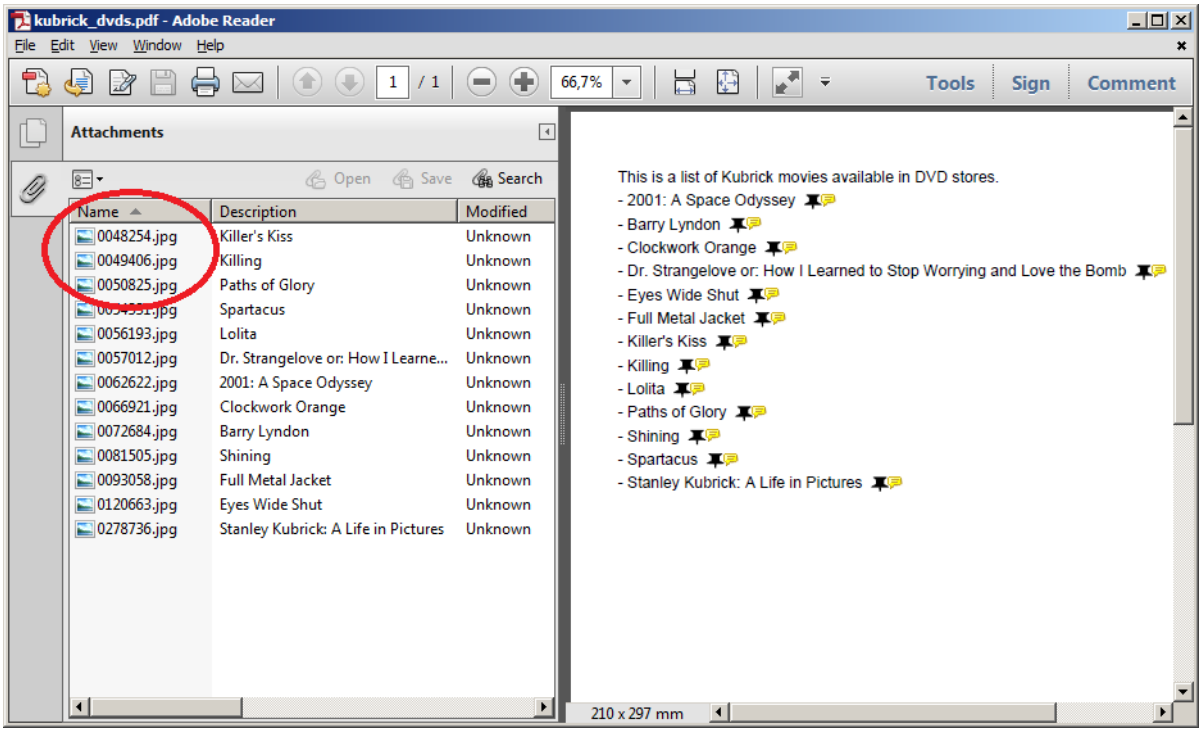
And finally, the content of an embedded file can be compared with the content of an external file:

```
@Test
public void hasEmbeddedFile_WithContent() throws Exception {
  String filename = "documentUnderTest.pdf";
  String embeddedFileName = "embeddedfiles/PrePress-Qualität.joboptions";

  AssertThat.document(filename)
            .hasEmbeddedFile().withContent(embeddedFileName)
  ;
}
```

The comparison is carried out byte-wise. The parameter can be a filename or an instance of `java.util.File`.

Multiple Attachments

The next example refers to the file “kubrick_dvds.pdf”, an iText sample. Adobe Reader® shows the attachments:



You can check for multiple attachments in one test. But select a better name for such a test than the name of the test in the next example:

```
@Test
public void hasEmbeddedFile_MultipleInvocation() throws Exception {
  String filename = "kubrick_dvds.pdf";
  String embeddedFileName1 = "0048254.jpg";
  String embeddedFileName2 = "0049406.jpg";
  String embeddedFileName3 = "0050825.jpg";

  AssertThat.document(filename)
            .hasEmbeddedFile().withName(embeddedFileName1)
            .hasEmbeddedFile().withName(embeddedFileName2)
            .hasEmbeddedFile().withName(embeddedFileName3)
  ;
}
```

If embedded files are not available as separate files, they can be extracted from an existing PDF with the utility “ExtractEmbeddedFiles”. This program is described in detail in chapter 9.4: “Extract Attachments” (p. 123):

3.4. Bar Code

Overview

PDFUnit can validate bar codes within PDF documents. It uses ZXing as a parser. Detailed information about ZXing can be found on the project's home page https://github.com/zxing/zxing.

The text of a bar code can be validated using the following methods:

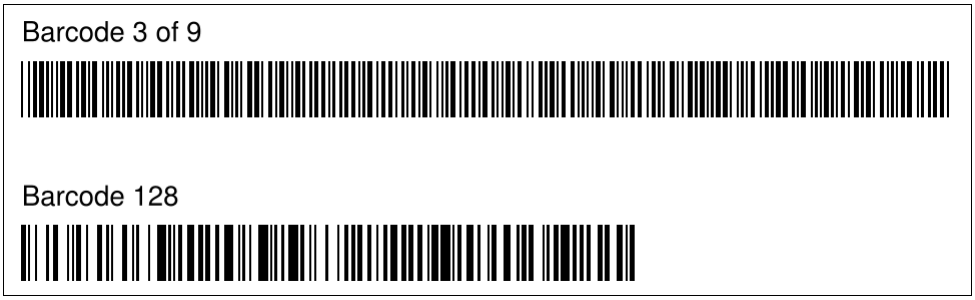
```
// Entry to all bar code validations:
.hasImage().withBarcode()

// Validate text in bar codes:
...withBarcode().containing(..)
...withBarcode().containing(.., WhitespaceProcessing)
...withBarcode().endingWith(..)
...withBarcode().equalsTo(..)
...withBarcode().equalsTo(.., WhitespaceProcessing)
...withBarcode().matchingRegex(..)
...withBarcode().startingWith(..)

// Validate text in bar code in image region:
...withBarcodeInRegion(imageRegion).containing(..)
...withBarcodeInRegion(imageRegion).containing(.., WhitespaceProcessing)
...withBarcodeInRegion(imageRegion).endingWith(..)
...withBarcodeInRegion(imageRegion).equalsTo(..)
...withBarcodeInRegion(imageRegion).equalsTo(.., WhitespaceProcessing)
...withBarcodeInRegion(imageRegion).matchingRegex(..)
...withBarcodeInRegion(imageRegion).startingWith(..)

// Compare with another bar code:
...withBarcode().matchingImage(..)
```

ZXing detects bar code formats automatically. The following formats can be used in PDFUnit tests:

```
// 1D bar codes, supported by PDFUnit/ZXing:
CODE_128
CODE_39
CODE_93
EAN_13
EAN_8
CODABAR
UPC_A
UPC_E
UPC_EAN_EXTENSION
ITF
```

Example - Compare Bar Code with Text

The next examples use these two bar code samples:



The first bar code contains the text 'TESTING BARCODES WITH PDFUNIT'. The second contains the text 'hello, world - 1234567890'.

```
@Test
public void hasBarCode_Code3of9() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  =   0;
  int upperY =  70;
  int width  = 210;
  int height =  30;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .withBarcode()
            .containing("TESTING BARCODES WITH PDFUNIT")
  ;
}
```

```
@Test
public void hasBarCode_Code128() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  =   0;
  int upperY = 105;
  int width  = 210;
  int height =  30;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .withBarcode()
            .containing("hello, world")
  ;
}
```

When multiple images exist in a defined region, each image must pass the test. Default whitespace processing when searching text in bar code is NORMALIZE, but whitespace processing can be controlled using a method parameter.

The internal used bar code parser ZXing supports many, but not all, bar code formats. So PDFUnit provides an external interface to plug in customer specific bar code parsers. This interface is documented separately. You can request it by writing an email to info[at]pdfunit.com.

Example - Compare a Bar Code with an Image

A bar code within a PDF document can also be compared with a bar code image from a file:

```
@Test
public void hasBarCodeMatchingImage() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  =   0;
  int upperY = 105;
  int width  = 210;
  int height =  30;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  String expectedImageFilename = "images/barcode-128.png";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .withBarcode()
            .matchingImage(expectedImageFilename);
  ;
}
```

Important: For the image comparison to work, the type of the external image (PNG or TIFF, for example) must match the type of the bar code image in the PDF document.

3.5. Bookmarks and Named Destinations

Overview

Bookmarks are essential for a quick navigation in large PDF documents. The value of a book drops dramatically when the chapters are not available via bookmarks. Use the following tests to ensure that the bookmarks are generated correctly.

```
// Simple methods:
.hasNumberOfBookmarks(..)
.hasBookmark()     // Test for one bookmark
.hasBookmarks()    // Test for all bookmarks

// Tests for one bookmark:
.hasBookmark().withLabel(..)
.hasBookmark().withLabelLinkingToPage(..)
.hasBookmark().withLinkToName(..)
.hasBookmark().withLinkToPage(..)
.hasBookmark().withLinkToURI(..)

// Tests for all bookmarks:
.hasBookmarks().withDestinations()
.hasBookmarks().withoutDuplicateNames()
```

We can see bookmarks as starting points and named destinations as the landing points. “Named destinations” (landing points) can be used by bookmarks and also by HTML links. So, you can jump from a website directly to a specific location within a PDF document.

For named destinations, the following test methods are available:

```
.hasNamedDestination()
.hasNamedDestination().withName(..)
```

Named Destinations

The names of named destinations can be tested easily:

```
@Test
public void hasNamedDestination_WithName() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNamedDestination().withName("Seventies")
            .hasNamedDestination().withName("Eighties")
            .hasNamedDestination().withName("1999")
            .hasNamedDestination().withName("2000")
  ;
}
```

Because a name also has to work with external links, it may not contain spaces. For example, if a document in LibreOffice has a label `"Export to PDF"` (which contains spaces) then LibreOffice creates a destination with the label `"First2520Bookmark"` when exporting it to PDF. A test has to use the escaped value:

```
@Test
public void hasNamedDestination_ContainingBlanks() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNamedDestination().withName("First2520Bookmark")  ❶
  ;
}
```

❶ `'2520'` stands for `'%20'` and that corresponds to a space.

Existence of Bookmarks

It is easy test to verify the existence of bookmarks:

```
@Test
public void hasBookmarks() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasBookmarks()
  ;
}
```

Number of Bookmarks

After testing whether a document contains bookmarks at all, it is worth verifying the number of bookmarks:

```
@Test
public void hasNumberOfBookmarks() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfBookmarks(19)
  ;
}
```

Label of a Bookmark

An important property of a bookmark is its label. That is what the reader sees. So, you should test that an expected bookmark has the expected label:

```
@Test
public void hasBookmark_WithLabel() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasBookmark().withLabel("Content on page 3.")
  ;
}
```

Destinations of Bookmarks

Bookmarks can have different kinds of destinations. A suitable test method is provided for each kind.

Does a **particular bookmark** point to the expected page number:

```
@Test
public void hasBookmark_WithLabelLinkingToPage() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasBookmark().withLabelLinkingToPage("Content on first page.", 1)
  ;
}
```

Is there **any bookmark** pointing to an expected page number:

```
@Test
public void hasBookmark_WithLinkToPage() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasBookmark().withLinkToPage(1)
  ;
}
```

Does a bookmark exist which points to an expected destination:

```
@Test
public void hasBookmark_WithLinkToName() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasBookmark().withLinkToName("Destination on Page 1")
  ;
}
```

Is there a bookmark pointing to a URI:

```
@Test
public void hasBookmark_WithLinkToURI() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasBookmark().withLinkToURI("http://www.wikipedia.org/")
  ;
}
```

PDFUnit does not accesses websites, it checks the existence of a link.

And finally, we can check that every bookmark has a destination:

```
@Test
public void hasBookmarkWithDestinations() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasBookmarks().withDestinations()
  ;
}
```

3.6. Certified PDF

Overview

A “certified PDF” is a regular PDF with additional information. It contains information about how it can be changed.

For certified PDF PDFUnit provides these test methods:

```
// Tests for certified PDF:
.isCertified()
.isCertifiedFor(FORM_FILLING)
.isCertifiedFor(FORM_FILLING_AND_ANNOTATIONS)
.isCertifiedFor(NO_CHANGES_ALLOWED)
```

Examples

First you can check that a document is certified at all:

```
@Test
public void isCertified() throws Exception {
  String filename = "sampleCertifiedPDF.pdf";

  AssertThat.document(filename)
            .isCertified()
  ;
}
```

Next you can check the level of certification:

```
@Test
public void isCertifiedFor_NoChangesAllowed() throws Exception {
  String filename = "sampleCertifiedPDF.pdf";

  AssertThat.document(filename)
            .isCertifiedFor(NO_CHANGES_ALLOWED)
  ;
}
```

Certification Level

PDFUnit provides constants for the certification level:

```
com.pdfunit.Constants.NO_CHANGES_ALLOWED
com.pdfunit.Constants.FORM_FILLING
com.pdfunit.Constants.FORM_FILLING_AND_ANNOTATIONS
```

3.7. Dates

Overview

It's unusual to have to test a PDF document's creation or modification date. But when you do it, it's not easy because dates can be formatted in many different ways. PDFUnit tries to hide the complexity and provides a wide range of functions:

```
// Date existence tests:
.hasCreationDate()
.hasNoCreationDate()
.hasModificationDate()
.hasNoModificationDate()

// Date value tests:
.hasCreationDate().after(..)
.hasCreationDate().before(..)
.hasCreationDate().equalsTo(..)
.hasModificationDate().after(..)
.hasModificationDate().before(..)
.hasModificationDate().equalsTo(..)
```

The following listings only show tests for the creation date because tests for the modification date have exactly the same syntax.

Existence of a Date

The first test checks that a creation date exists:

```
@Test
public void hasCreationDate() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasCreationDate()
  ;
}
```

You can verify that your PDF document has **no** creation date like this:

```
@Test
public void hasCreationDate_NoDateInPDF() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNoCreationDate()
  ;
}
```

The next chapter compares a date value with an expected date.

Date Resolution

The expected date has to be an instance of `java.util.Calendar`. Additionally, the date resolution has to be declared, saying which parts of an existing date are parts to be compared with the expected date.

Using the enumeration `DateResolution.DATE` day, month and year are used for comparison. When using the `DateResolution.DATETIME` hours, minutes and seconds are also compared. Both enumerations exist as constants:

```
// Constants for date resolution:

com.pdfunit.Constants.AS_DATE
com.pdfunit.Constants.DateResolution AS_DATETIME
```

A test looks like this:

```
@Test
public void hasCreationDate_WithValue() throws Exception {
  String filename = "documentUnderTest.pdf";
  Calendar expectedCreationDate =
           DateHelper.getCalendar("20131027_17:24:17", "yyyyMMdd_HH:mm:ss"); ❶

  AssertThat.document(filename)
            .hasCreationDate()
            .equalsTo(expectedCreationDate, AS_DATE)   ❷
  ;
}
```

❶ The utility `com.pdfunit.util.DateHelper` can be used to create an instance of `java.util.Calender` for the expected date.

❷ The constants ard defined in the enumeration `com.pdfunit.DateResolution`.

Formatted dates are generally difficult to handle by programs. That's why PDFUnit uses an instance of `java.util.Calendar`. That makes the date processing independent from formats. If PDFUnit has problems converting the PDF-internal date value into an instance of the class Calendar, you can check the date as follows:

```
@Test
public void hasCreationDate() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasProperty("CreationDate").startingWith("D:20131027")
            .hasProperty("CreationDate").equalsTo("D:20131027172417+01'00'")
  ;
}
```

You can find the date format of an existing PDF document when you open it with a simple text editor. Search for the string "CreationDate".

Date Tests with Upper and Lower Limit

You can check that a PDF document's creation date is later or earlier than a given date:

```
@Test
public void hasCreationDate_Before() throws Exception {
  String filename = "documentUnderTest.pdf";
  Calendar creationDateUpperLimit = DateHelper.getCalendar("20991231", "yyyyMMdd");

  AssertThat.document(filename)
            .hasCreationDate()
            .before(creationDateUpperLimit, AS_DATE) ❶
  ;
}
```

```
@Test
public void hasCreationDate_After() throws Exception {
  String filename = "documentUnderTest.pdf";
  Calendar creationDateLowerLimit = DateHelper.getCalendar("19990101", "yyyyMMdd");

  AssertThat.document(filename)
            .hasCreationDate()
            .after(creationDateLowerLimit, AS_DATE) ❷
  ;
}
```

❶❷ The lower- or upper-limits are not included in the expected date range.

Creation Date of a Signature

When a PDF document is signed, the date of the signature can be tested. Chapter 3.28: “Signed PDF” (p. 59) covers tests for that.

3.8. DIN 5008

Overview

DIN 5008 defines rules for formating letters and also some rules for text in letters. PDFUnit can check for compliance with these rules. A lot of rules are defined by the standard as a recommendation only, so companies have to select which rules they want to follow. The selected rules must be written into an Excel file. PDFUnit reads this Excel file and applies all the rules to all related PDF documents.

The following methods can be used to validate documents against DIN 5008:

```
// Methods to validate DIN 5008 constraints:
.compliesWith().din5008FormA()
.compliesWith().din5008FormB()
```

Information about DIN 5008 can be found online at https://en.wikipedia.org/wiki/DIN_5008. Chapter 3.10: “Excel Files for Validation Constraints” (p. 28) describes how the rules must be written in Excel files.

Example - Validate a Single Document

In the next example only the PDF document is given to the testing method, not the Excel file with the rules. The name and the directory of the Excel file are instead declared in the config file `pdfunit.config`:

```
#
################################################################################
#
# Definition of PDF validation files.
#
################################################################################
file.din5008.forma = src/main/resources/din5008/ValidationRules-DIN5008-FormA.xls
file.din5008.formb = src/main/resources/din5008/ValidationRules-DIN5008-FormB.xls
```

The directory can be either an absolute or a relative path.

```
@Test
public void compliesWithDin5008B() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .compliesWith()
            .din5008FormB()
  ;
}
```

If multiple Excel files exist, the above syntax is not enough. The name of the Excel file has to be given to the testing method:

```
@Test
public void din5008FormB() throws Exception {
  String filename = "documentUnderTest.pdf";
  String rulesAsExcelFile = PATH_TO_RULES + "din5008-formB_letter-portrait.xls";
  PDFValidationConstraints excelRules = new PDFValidationConstraints(rulesAsExcelFile);

  AssertThat.document(filename)
            .compliesWith()
            .constraints(excelRules)
  ;
}
```

Example - Validate Each Document in a Folder

The last two examples can be used with a folder instead of a single document. Then each PDF file in the folder will be validated:

```
@Test
public void compliesWithDin5008BInFolder() throws Exception {
  String filename = "documentUnderTest.pdf";
  File folderWithPDF = new File(folderName);

  AssertThat.eachDocument()
            .inFolder(folderWithPDF)
            .compliesWith()
            .din5008FormB()
  ;
}
```

3.9. Document Properties

Overview

PDF documents contain information about title, author, keywords and other properties. These standard properties can be extended by individual key-value data. Such metadata are playing an ever increasing role in the context of search engines and archive systems, so PDF document properties should be set wisely. PDFUnit provides some test to verify them.

An example of very poor document properties is a PDF document entitled “jfqd231.tmp” (that really is its title). Nobody will ever search for that and therefore it will never be found. It is a typewriter document by an U.S. government organization that was scanned in 1993. But not only is the title useless, also the file name lacks any meaning. So, the benefit of this document is only marginally greater than if it didn't exist at all.

The following methods are available to verify document properties:

```
// Testing document properties:

.hasAuthor()
.hasCreator()
.hasKeywords()
.hasProducer()
.hasProperty(..)
.hasSubject()
.hasTitle()

.hasNoAuthor()
.hasNoCreator()
.hasNoKeywords()
.hasNoProducer()
.hasNoProperty(..)
.hasNoSubject()
.hasNoTitle()

.hasCreationDate()       ❶
.hasModificationDate()   ❷
.hasNoCreationDate()
.hasNoModificationDate()
```

❶❷ Tests for creation date and modification date are described in chapter 3.7: “Dates” (p. 22) because they differ from tests for the other document properties.

Document properties of a test document can also be compared with the properties of a another document. Such tests are described in chapter 4.5: “Comparing Document Properties” (p. 89).

Testing the Author ...

You can verify the author of a document manually with any PDF reader, but an automated test is quicker.

It is very simple to check whether a document has **any value** for the property “author”:

```
@Test
public void hasAuthor() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasAuthor()
  ;
}
```

Use the method `hasNoAuthor()` to verify that the document property “author” does **not exist**:

```
@Test
public void hasNoAuthor() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNoAuthor()
  ;
}
```

The next test verifies the value of the property “author”:

```
@Test
public void hasAuthor() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasAuthor()
            .equalsTo("PDFUnit.com")
  ;
}
```

There are several methods to compare an expected property value with the actual one. The names are self-explanatory:

```
// Comparing text for author, creator, keywords, producer, subject, title:
.containing(..)
.endingWith(..)
.equalsTo(..)
.matchingRegex(..)
.notContaining(..)
.notMatchingRegex(..)
.startingWith(..)
```

Whitespaces are not changed by these methods. Typically property values are short, so the test-developer has to use whitespaces in a correct way.

All test methods are is case sensitive.

The method `matchingRegex()` follows the rules of `java.util.regex.Pattern`.

... and Creator, Keywords, Producer, Subject and Title

Tests on the content of creator, keywords, producer, subject and title work just like those for “Author” above.

Concatenating of Validation Methods

Of course, methods can be concatenated:

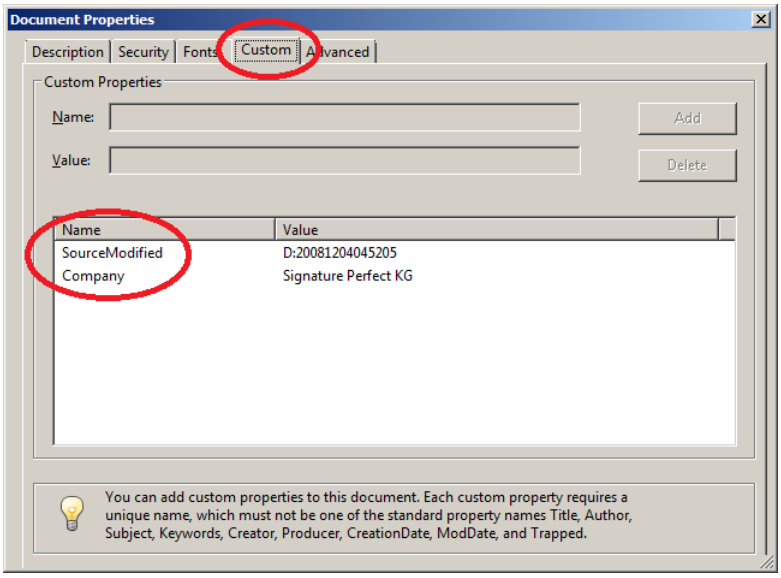
```
@Test
public void hasKeywords_allTextComparingMethods() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasKeywords().notContaining("--")
            .hasKeywords().matchingRegex(".*key.*")
            .hasKeywords().startingWith("PDFUnit")
  ;
}
```

Common Validation as a Key-Value Pair

All tests for document properties shown in the previous sections can also be implemented with the general method `hasProperty(..)`. The method validates any document property as a key-value pair:

```
@Test
public void hasProperty_StandardProperties() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasProperty("Title")
            .equalsTo("PDFUnit sample - Demo for Document Infos")
            .hasProperty("Subject").equalsTo("Demo for Document Infos")
            .hasProperty("CreationDate").equalsTo("D:20131027172417+01'00'")
            .hasProperty("ModDate").equalsTo("D:20131027172417+01'00'")
  ;
}
```

The PDF document in the following example has two custom properties as can be seen with Adobe Reader®:



And this is the test for custom properties:

```
@Test
public void hasProperty_CustomProperties() throws Exception {
  String filename = "documentUnderTest.pdf";
  String key1 = "Company";
  String expectedValue1 = "Signature Perfect KG";
  String key2 = "SourceModified";
  String expectedValue2 = "D:20081204045205";

  AssertThat.document(filename)
            .hasProperty(key1).equalsTo(expectedValue1)
            .hasProperty(key2).equalsTo(expectedValue2)
  ;
}
```

To ensure that a property does **not exist**, use the following method:

```
@Test
public void hasNoProperty() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNoProperty("OldProperty_ShouldNotExist")
  ;
}
```

PDF documents of version PDF-1.4 or higher can have metadata as XML (Extensible Metadata Platform, XMP). Chapter 3.38: “XMP Data” (p. 79) explains that in detail.

3.10. Excel Files for Validation Constraints

Overview

It might seem surprisingly to store validation rules in Excel and use them in automated test. But it has the advantage of enabling non-programmers to create tests which can be executed by programs created by programmers.

The Excel files can be used by the Java-API of PDFUnit and by the PDFUnit-Monitor. The monitor is also created for non-programmers. Separate documentation exists for the monitor and can be accessed at . A short description can be found in chapter 8: “PDFUnit-Monitor” (p. 117).

A detailed description of the structure of an Excel file and the available test methods is given in chapter 10: “Validation Constraints in Excel Files” (p. 137). That chapter is not necessary for understanding how the Excel files are used, so you can read it later.

There is only one method for using Excel files:

```
// Validation method, using Excel-based constraints:

.compliesWith().constraints(excelRules)
```

This method can be applied to a single document, multiple documents, or all documents in a folder:

```
// Validation of one, many or all PDF documents:

AssertThat.document(filename)
          .compliesWith()
          .constraints(excelRules)

AssertThat.eachDocument()
          .inFolder(folder)
          .compliesWith()
          .constraints(excelRules)

AssertThat.eachDocument(files)
          .compliesWith()
          .constraints(excelRules)
```

Example

In the next example, a PDF document is validated against constraints which are located in the Excel file 'din5008-formA_letter-portrait.xls':

```
@Test
public void singleDocumentCompliesWithExcelRules() throws Exception {
  String filename = "documentUnderTest.pdf";
  String rulesAsExcelFile = PATH_TO_RULES + "din5008-formA_letter-portrait.xls";
  PDFValidationConstraints excelRules = new PDFValidationConstraints(rulesAsExcelFile);

  AssertThat.document(filename)
            .compliesWith()
            .constraints(excelRules)
  ;
}
```

If the test fails, an error message shows all constraint violations. A test does not stop after the first detected error.

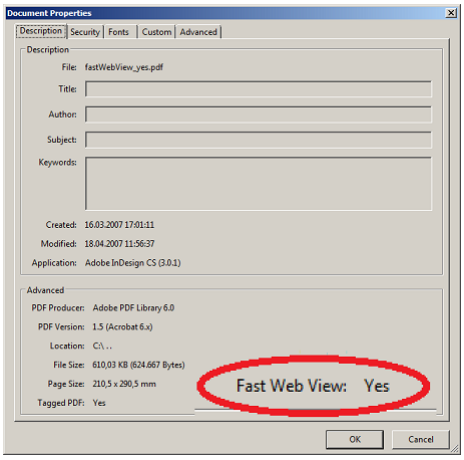
The structure of an Excel file is described in chapter 10: “Validation Constraints in Excel Files” (p. 137).

3.11. Fast Web View

Overview

The term “Fast Web View” means that a server can deliver a PDF document to a client one page at a time. The ability to do this, is a function of the server, but the PDF document itself has to support this ability. Objects that are needed to render the first PDF page have to be stored at the beginning of the file. And also the cross reference table of the objects must be placed at the beginning of a PDF document.

“Fast Web View” can be seen in the properties dialog with Adobe Reader®.



There is one test method for linearized documents:

```
.isLinearizedForFastWebView()
```

In the current release PDFUnit does not cover all of the above-named aspects. If there is a problem when testing linearized documents, please send a mail to info[at]pdfunit.com.

Example

```
@Test
public void isLinearizedForFastWebView() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .isLinearizedForFastWebView()
  ;
}
```

3.12. Fonts

Overview

Fonts are a difficult topic in PDF documents. The PDF standard defines 14 fonts, but does your document use any others? Fonts are also important for archiving. PDFUnit provides several test methods for different requirements.

```
// Simple tests:
.hasFont()
.hasFonts()           // identical with hasFont()
.hasNumberOfFonts(..)

// Tests for one font:
.hasFont().withNameContaining(..)
.hasFont().withNameNotContaining(..)

// Tests for many fonts:
.hasFonts().ofThisTypeOnly(..)
```

Number of Fonts

What is a font? Should a subset of a font count as a separate font? In most situations this question is irrelevant for developers, but for a testing tool the question has to be answered.

Fonts have different aspects (name, type, size). So, it's difficult to say when two fonts are equal. In PDFUnit, two fonts are 'equal' if the selected comparison criteria have the same values. The criteria are defined by the following constants:

```
// Constants to identify fonts:

com.pdfunit.Constants.IDENTIFIEDBY_ALLPROPERTIES
com.pdfunit.Constants.IDENTIFIEDBY_BASENAME
com.pdfunit.Constants.IDENTIFIEDBY_EMBEDDED
com.pdfunit.Constants.IDENTIFIEDBY_NAME
com.pdfunit.Constants.IDENTIFIEDBY_NAME_TYPE
com.pdfunit.Constants.IDENTIFIEDBY_TYPE
```

The following list explains the available criteria to compare fonts.

Constant	Description
ALLPROPERTIES	All properties of a font are used to identify a font. Two fonts having the same values for all properties considered equal.
BASENAME	Fonts are different when they have different base fonts.
EMBEDDED	This filter counts only fonts that are embedded.
NAME	Only the fonts' names are relevant to the test.
NAME_TYPE	Only the font name and font type are used to compare fonts.
TYPE	Only the types of the fonts are considered in the comparison.

The following example shows all possible filters:

```
@Test
public void hasNumberOfFonts_Japanese() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfFonts(11, IDENTIFIEDBY_ALLPROPERTIES)
            .hasNumberOfFonts( 9, IDENTIFIEDBY_BASENAME)
            .hasNumberOfFonts( 8, IDENTIFIEDBY_EMBEDDED)
            .hasNumberOfFonts( 9, IDENTIFIEDBY_NAME)
            .hasNumberOfFonts( 9, IDENTIFIEDBY_NAME_TYPE)
            .hasNumberOfFonts( 2, IDENTIFIEDBY_TYPE)
  ;
}
```

Font Names

Testing the names of fonts is easy:

```
@Test
public void hasFont_WithNameContaining() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFont().withNameContaining("Arial")
  ;
}
```

Sometimes, font names in a PDF document have a prefix, e.g. `FGNNPL+ArialMT`. Because this prefix is worthless for tests, PDFUnit only checks whether the desired font name is a **substring** of the existing font names.

Of course, you can chain multiple methods:

```
@Test
public void hasFont_WithNameContaining_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFont().withNameContaining("Arial")
            .hasFont().withNameContaining("Georgia")
            .hasFont().withNameContaining("Tahoma")
            .hasFont().withNameContaining("TimesNewRoman")
            .hasFont().withNameContaining("Verdana")
            .hasFont().withNameContaining("Verdana-BoldItalic")
  ;
}
```

Because it is sometimes interesting to know that a particular font is **not** included in a document, PDFUnit provides a suitable test method for it:

```
@Test
public void hasFont_WithNameNotContaining() throws Exception {
  String filename = "documentUnderTest.pdf";
  String wrongFontnameIntended = "ComicSansMS";

  AssertThat.document(filename)
            .hasFont().withNameNotContaining(wrongFontnameIntended)
  ;
}
```

Font Types

You can check that **all** fonts used in a PDF document are of a certain type:

```
@Test
public void hasFonts_OfThisTypeOnly_TrueType() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFonts()
            .ofThisTypeOnly(FONTTYPE_TRUETYPE)
  ;
}
```

Predefined font types are:

```
// Constants for font types:
com.pdfunit.Constants.FONTTYPE_CID
com.pdfunit.Constants.FONTTYPE_CID_TYPE0
com.pdfunit.Constants.FONTTYPE_CID_TYPE2
com.pdfunit.Constants.FONTTYPE_MMTYPE1
com.pdfunit.Constants.FONTTYPE_OPENTYPE
com.pdfunit.Constants.FONTTYPE_TRUETYPE
com.pdfunit.Constants.FONTTYPE_TYPE0
com.pdfunit.Constants.FONTTYPE_TYPE1
com.pdfunit.Constants.FONTTYPE_TYPE3
```

3.13. Form Fields

Overview

It is often the content of form fields which is processed when PDF documents are part of a workflow. To avoid problems the fields should be created properly. So, field names should be unique and some field properties should be set right.

All information about form fields can be extracted into an XML file by using the utility `ExtractFieldInfo`. All properties in the XML file can be validated.

The following sections describe a lot of tests for field properties, size and content. Depending on the application context one of the following tags and attributes may be useful to you:

```
// Simple tests:
.hasField(..)
.hasField(..).ofType(..)
.hasField(..).withHeight()
.hasField(..).withWidth()
.hasFields()
.hasFields(..)
.hasNumberOfFields(..)
.hasSignatureField(..)
.hasSignatureFields()            ❶

// Tests belonging to all fields:
.hasFields().withoutDuplicateNames()
.hasFields().allWithoutTextOverflow()   ❷

// Content of a field:
.hasField(..).withText().containing()
.hasField(..).withText().endingWith()
.hasField(..).withText().equalsTo()
.hasField(..).withText().matchingRegex()
.hasField(..).withText().notContaining()
.hasField(..).withText().notMatchintRegex()
.hasField(..).withText().startingWith()

// JavaScript associated to a field:
.hasField(..).withJavaScript().containing(...)

// Field properties:
.hasField(..).withProperty().checked()
.hasField(..).withProperty().editable()
.hasField(..).withProperty().exportable()
.hasField(..).withProperty().multiLine()
.hasField(..).withProperty().multiSelect()
.hasField(..).withProperty().notExportable()
.hasField(..).withProperty().notSigned()
.hasField(..).withProperty().notVisibleInPrint()
.hasField(..).withProperty().notVisibleOnScreen()
.hasField(..).withProperty().optional()
.hasField(..).withProperty().passwordProtected()
.hasField(..).withProperty().readOnly()
.hasField(..).withProperty().required()
.hasField(..).withProperty().signed()
.hasField(..).withProperty().singleLine()
.hasField(..).withProperty().singleSelect()
.hasField(..).withProperty().unchecked()
.hasField(..).withProperty().visibleInPrint()
.hasField(..).withProperty().visibleOnScreen()
.hasField(..).withProperty().visibleOnScreenAndInPrint()
```

❶ This test is described separately in chapter 3.28: “Signed PDF” (p. 59)

❷ This test is described separately in chapter 3.14: “Form Fields - Text Overflow” (p. 38):

Existence of Fields

The following test verifies whether or not fields exist:

```
@Test
public void hasFields() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFields()  // throws an exception when no fields exist
  ;
}
```

Name of Fields

Because fields are accessed by their names to get their content, you could check that the names exist:

```
@Test
public void hasField_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname1 = "name";
  String fieldname2 = "address";
  String fieldname3 = "postal_code";
  String fieldname4 = "email";

  AssertThat.document(filename)
            .hasField(fieldname1)
            .hasField(fieldname2)
            .hasField(fieldname3)
            .hasField(fieldname4)
  ;
}
```

The same result can be achieved using an array of field names and the method `hasFields(..)`:

```
@Test
public void hasFields() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname1 = "name";
  String fieldname2 = "address";
  String fieldname3 = "postal_code";
  String fieldname4 = "email";

  AssertThat.document(filename)
            .hasFields(fieldname1, fieldname2, fieldname3, fieldname4)
  ;
}
```

Duplicate field names are allowed by the PDF specification, but they are probably a source of surprises in the later workflow. Thus PDFUnit provides a method to check the absence of duplicate names.

```
@Test
public void hasFields_WithoutDuplicateNames() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFields()
            .withoutDuplicateNames()
  ;
}
```

Number of Fields

If you only need to verify the number of fields, you can use the method `hasNumberOfFields(..)`:

```
@Test
public void hasNumberOfFields() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfFields(4)
  ;
}
```

Perhaps it might also be interesting to ensure that a PDF document has **no fields**:

```
@Test
public void hasNumberOfFields_NoFieldsAvailable() throws Exception {
  String filename = "documentUnderTest.pdf";
  int zeroExpected = 0;

  AssertThat.document(filename)
            .hasNumberOfFields(zeroExpected)
  ;
}
```

Content of Fields

It is very simple to verify that a given field contains data:

```
@Test
public void hasField_WithAnyValue() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "ageField";

  AssertThat.document(filename)
            .hasField(fieldname)
            .withText()
  ;
}
```

To verify the actual content of fields with an expected string, the following methods are available:

```
.containing(..)
.endingWith(..)
.equalsTo(..)
.matchingRegex(..)
.notContaining(..)
.notMatchingRegex(..)
.startingWith(..)
```

The following examples should give you some ideas about how to use these methods:

```
@Test
public void hasField_EqualsTo() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Text 1";
  String expectedValue = "Single Line Text";

  AssertThat.document(filename)
            .hasField(fieldname)
            .equalsTo(expectedValue)
  ;
}
```

```
/**
 * This is a small test to protect fields against SQL-Injection.
 */
@Test
public void hasField_NotContaining_SQLComment() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Text 1";
  String sqlCommentSequence = "--";

  AssertThat.document(filename)
            .hasField(fieldname)
            .notContaining(sqlCommentSequence)
  ;
}
```

Whitespaces will be normalized when comparing expected and actual field content.

Type of Fields

Each field has a type. Although a field type is not as important as the name, it can be tested with a special method:

```
@Test
public void hasField_WithType_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasField("Text 25")       .ofType(TEXT)
            .hasField("Check Box 7")   .ofType(CHECKBOX)
            .hasField("Radio Button 4").ofType(RADIOBUTTON)
            .hasField("Button 19")     .ofType(PUSHBUTTON)
            .hasField("List Box 1")    .ofType(LIST)
            .hasField("List Box 1")    .ofType(CHOICE)
            .hasField("Combo Box 5")   .ofType(CHOICE)
            .hasField("Combo Box 5")   .ofType(COMBO)
  ;
}
```

The previous program listing shows all testable fields except for a signature field, because that document has no signature field. The document of the next listing has a signature field and that can be tested:

```
@Test
public void hasField_WithType_Signature() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasField("Signature2").withType(SIGNATURE)
  ;
}
```

Datailed tests for signatures are described in chapter 3.28: “Signed PDF” (p. 59):

Available field types are defined as constants in `com.pdfunit.Constants`. The names of the constants correspond to the typical names of visible elements of a graphical user interface. But the PDF standard uses other names for the types. The following list shows the association between PDFUnit constants and PDF internal constants. These may appear in error messages:

```
// Mapping between PDFUnit-Constants and PDF-internal types.

PDFUnit-Constant    PDF-intern
-------------------------------
CHOICE          ->  "Ch"
COMBO           ->  "Ch"
LIST            ->  "Ch"
CHECKBOX        ->  "Btn"
PUSHBUTTON      ->  "Btn"
RADIOBUTTON     ->  "Btn"
SIGNATURE       ->  "Sig"
TEXT            ->  "Tx"
```

Field Size

If the size of form fields is important, methods can be used to verify width and height:

```
@Test
public void hasField_WidthAndHeight() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Title of 'someField'";
  int allowedDeltaForMillis = 2;
  AssertThat.document(filename)
            .hasField(fieldname)
            .withWidth(159, MILLIMETERS, allowedDeltaForMillis)
            .withHeight(11, MILLIMETERS, allowedDeltaForMillis)
  ;
}
```

Both methods can be invoked with different pre-defined measuring units: points or millimeters. Because rounding is necessary, rounding tolerance must be given as a third parameter. The default is set to the unit points with a tolerance of zero.

```
@Test
public void hasField_Width() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Title of 'someField'";
  int allowedDeltaForPoints = 0;
  int allowedDeltaForMillis = 2;
  AssertThat.document(filename)
            .hasField(fieldname)
            .withWidth(450, POINTS, allowedDeltaForPoints)
            .withWidth(159, MILLIMETERS, allowedDeltaForMillis)
            .withWidth(450) // default is POINTS
  ;
}
```

When you are creating a test you probably do not know the dimensions of a field. That is not a problem. Use any value for width and height and run the test. The resulting error message returns the real field size in millimeters

Whether a text fits into a field or not is not predictable by calculation using font size and field size. In addition to the font size the words at the end of each line determine the required number of rows and the required height. And the calculation has to consider hyphenation. Chapter 3.14: “Form Fields - Text Overflow” (p. 38) deals with this subject in detail.

Field Properties

Fields have more properties than just the size, for example `editable` and `required`. Since most of the properties can not be tested manually, appropriate test methods have to be part of every PDF testing tool. The following example shows the principle.

```
@Test
public void hasField_Editable() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldnameEditable = "Combo Box 4";

  AssertThat.document(filename)
            .hasField(fieldnameEditable)
            .withProperty()
            .editable()
  ;
}
```

These are the available attributes for verifing properties of form fields:

```
// Check field properties

// All methods following .withProperty():
.checked()                .unchecked()
.editable(),              .readOnly()
.exportable(),            .notExportable()
.multiLine(),             .singleLine()
.multiSelect(),           .singleSelect()
.optional(),              .required()
.signed(),                .notSigned()
.visibleInPrint(),        .notVisibleInPrint()
.visibleOnScreen(),       .notVisibleOnScreen()

.visibleOnScreenAndInPrint()
.passwordProtected()
```

JavaScript Actions for Fields

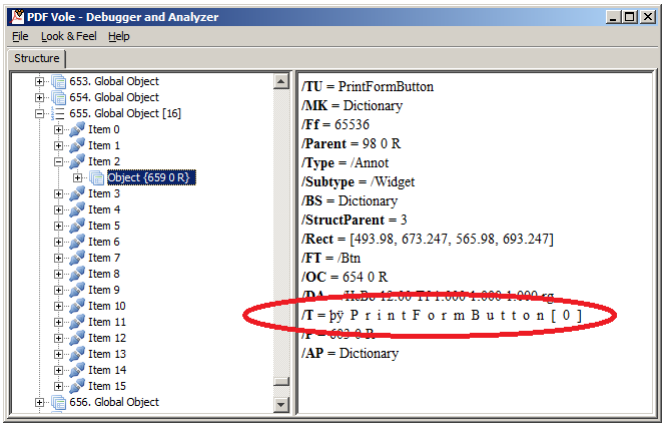
Assuming that PDF documents are processed in a workflow, the input into fields is typically validated with constraints implemented in JavaScript. That prevents incorrect input.

PDFUnit can verify whether JavaScript is associated with a field. The expected content of JavaScript can be validated by the method 'containing()'. Whitespaces are ignored:

```
@Test
public void hasFieldWithJavaScript() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Calc1_A";

  String scriptText = "AFNumber_Keystroke";
  AssertThat.document(filename)
            .hasField(fieldname)
            .withJavaScript()
            .containing(scriptText)
  ;
}
```

Unicode

When tools for creating PDF do not handle Unicode sequences properly, it is difficult to test those sequences. But difficult does not mean impossible. The following picture shows the name of a field in the encoding UTF-16BE with a Byte Order Mark (BOM) at the beginning:



Although it is tricky, the name of this field can be tested as a Java Unicode sequence:

```
@Test
public void hasField_nameContainingUnicode_UTF16() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldName =
  //                F           o           r           m           R
     "\u00fe\u00ff\u0000\u0046\u0000\u006f\u0000\u0072\u0000\u006d\u0000\u0052" +
  //          o           o           t           [           0           ]
     "\u0000\u006f\u0000\u006f\u0000\u0074\u0000\u005b\u0000\u0030\u0000\u005d" +
  //
     "\u002e"                                                                  +
  //                P           a           g           e           1
     "\u00fe\u00ff\u0000\u0050\u0000\u0061\u0000\u0067\u0000\u0065\u0000\u0031" +
  //          [           0           ]
     "\u0000\u005b\u0000\u0030\u0000\u005d"                                    +
  //
     "\u002e"                                                                  +
  //                P           r           i           n           t
     "\u00fe\u00ff\u0000\u0050\u0000\u0072\u0000\u0069\u0000\u006e\u0000\u0074" +
  //          F           o           r           m           B           u
     "\u0000\u0046\u0000\u006f\u0000\u0072\u0000\u006d\u0000\u0042\u0000\u0075" +
  //          t           t           o           n           [           0
     "\u0000\u0074\u0000\u0074\u0000\u006f\u0000\u006e\u0000\u005b\u0000\u0030" +
  //          ]
     "\u0000\u005d";

  AssertThat.document(fileName)
            .hasField(fieldName)
  ;
}
```

More information about Unicode and Byte-Order-Mark can be found in Wikipedia.

3.14. Form Fields - Text Overflow

Overview

Some projects create PDF documents with empty fields as placeholders which are later filled with text. But if the text is longer than the available size of the field, the excessive text is placed outside the field and can't be seen on the page. Decreasing the font size is rarely an acceptable solution for that problem.

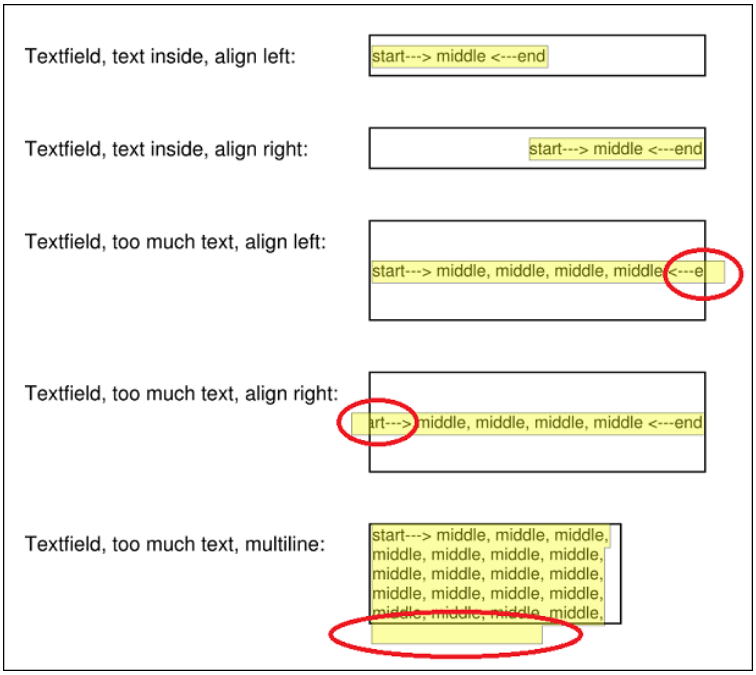
Verifying if text fits in a field is not easy because the required space depends not only on the font size, but also on the length of words at the end of each line and whether hyphenation is used. Following requests by various users, PDFUnit now provides these two methods:

```
// Fit(nes)-test for one field:
.hasField(..).withoutTextOverflow()

// Verifying that all fields are big enough:
.hasFields().allWithoutTextOverflow()
```

Important: only text fields are checked. Buttons, lists, combo boxes, signature fields and password fields are not validated.

Correct Size of a Field

The screenshot shows the form fields of a PDF document and their content used in the next example. It is easy to see that the text in the last three fields does not fit:



And this is the test for the last field of the document from the previous picture:

```
@Test(expected=PDFUnitValidationException.class)
public void hasField_WithoutTextOverflow_Fieldname() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Textfield, too much text, multiline:";

  AssertThat.document(filename)
            .hasField(fieldname)
            .withoutTextOverflow()
  ;
}
```

If you do not declare the expected exception, the following message appears: `Content of field 'Textfield, too much text, multiline:' of 'C:\...\fieldSizeAndText.pdf' does not fit in the available space.`

When the method `hasField(..)` is used for non-text fields, no exception is thrown.

Correct Size of all Fields

If a document has many fields, it would be time-consuming to write a test for every single field. Therefore, all fields can be checked for text overflow using one method:

```
@Test
public void hasFields_AllWithoutTextOverflow() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFields()
            .allWithoutTextOverflow()
  ;
}
```

Also for this test: only textfields are checked, but not button fields, lists, combo-boxes, signature- or password fields.

3.15. Format

Overview

You need to check that the intended format was created successfully: A4-landscape, Letter-portrait or a special format for a poster? You think testing such simple thing is a waste of time? But have you ever printed a “LETTER”-formatted document on an “A4”-printer? It is possible, but... So, PDFUnit provides the following test method:

Deshalb stellt PDFUnit eine Testmethode für Formattests zur Verfügung:

PDFUnit provides one flexible method to test document formats:

```
// Simple tests for page formats:
.hasFormat(..)
```

Documents with Same Page Format

You can use predefined constants to verify the format of conventional PDF documents, each page having the same size:

```
import static com.pdfunit.Constants.*;
...
@Test
public void hasFormat_A4Landscape() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFormat(A4_LANDSCAPE)    ❶
  ;
}
```

```
import static com.pdfunit.Constants.*;
...
@Test
public void hasFormat_LetterPortrait() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFormat(LETTER_PORTRAIT) ❷
  ;
}
```

❶❷ Many popular formats are defined in `com.pdfunit.Constants`

You can also verify individual formats:

```
@Test
public void hasFormat_FreeFormat_1117x836_mm() throws Exception {
  String filename = "documentUnderTest.pdf";
  int heightMM = 1117;
  int widthMM = 863;
  DocumentFormat formatMM = new DocumentFormatMillis(widthMM, heightMM);

  AssertThat.document(filename)
            .hasFormat(formatMM)
  ;
}
```

It is recommended to use the unit millimeters for tests with document formats, although a class DocumentFormatPoints exists. When the unit points is used, the real size of a page depends on the resolution. Two documents with the same number of points can have different page sizes if one document has a resolution of 72 DPI (72 dots per inch) and the other has a resolution of 150 DPI.

All size values are of type integer. If you want to check formats having tenths of a millimeter, round the values. Please note that PDFUnit uses the tolerance defined in DIN 476 when comparing two size values.

Information about paper formats can be found at http://en.wikipedia.org/wiki/Paper_size. PDFUnit uses the stricter tolerances of DIN 476 when comparing page sizes, although the ISO 216 standard defines larger tolerances.

Formats on Single Pages

A document with different page sizes can also be checked. The next example checks the format of page 3 only:

```
@Test
public void hasFormatOnPage3() throws Exception {
  String filename = "documentUnderTest.pdf";
  PagesToUse page3 = PagesToUse.getPage(3);

  AssertThat.document(filename)
            .restrictedTo(page3)
            .hasFormat(A5_PORTRAIT)
  ;
}
```

The restriction of tests to individual pages or page ranges is described in chapter 13.2: “Page Selection” (p. 155):

3.16. Images in PDF Documents

Overview

Images in a PDF document are seldom optical decorations of minor importance. More often, they transfer information which can have contractual meaning. Typical errors with images include:

- Does an image appear on the expected page?
- Is an image missing in the document because it was not found during document creation?
- Does a letter show the new logo and not the old one?
- If an image contains rendered text, is it the expected text?
- Is the content of a bar code or a QR code the expected one?

All errors can be detected with these test methods:

```
// Testing images in PDF:
.hasImage().matching(..)
.hasImage().matchingOneOf(..)

.hasImage().withBarcode()          3.4: “Bar Code” (p. 16)
.hasImage().withBarcodeInRegion()  3.4: “Bar Code” (p. 16)
.hasImage().withText().xxx(..)     3.31: “Text - in Images (OCR)” (p. 68)
.hasImage().withTextInRegion()     3.31: “Text - in Images (OCR)” (p. 68)
.hasImage().withQRCode()           3.27: “QR Code” (p. 56)
.hasImage().withQRCodeInRegion()   3.27: “QR Code” (p. 56)
.hasImage().withHeightInPixel()
.hasImage().withWidthInPixel()

.hasNoImage()

.hasNumberOfDifferentImages(..)
.hasNumberOfVisibleImages(..)
```

Number of different Images inside PDF

The number of images inside a PDF document is typically not the same as the number of images you can see when it is printed. A logo visible on 10 pages is stored only once within the document. So, PDFUnit provides two test methods. The method `hasNumberOfDifferentImages(..)` validates the number of images **stored internally** and the method `hasNumberOfVisibleImages(..)` validates the number of **visible** images.

The following listing shows the syntax for verifing the number of images **internally stored** in PDF:

```
@Test
public void hasNumberOfDifferentImages() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfDifferentImages(2)
  ;
}
```

How do you know in this example that “2” is the right number? How do you know which images are stored internally for a given PDF? The answer to both questions is given by the utility program `ExtractImages`. You can use it to extract all images from a document into separate files. Chapter 9.7: “Extract Images from PDF” (p. 127) describes this topic in detail.

Number of visible Images inside a PDF

The next example validates the number of **visible images**:

```
@Test
public void hasNumberOfVisibleImages() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfVisibleImages(8)
  ;
}
```

The sample document has 8 images on 6 pages, but 2 images on page 3, no image on page 4 and 3 images on page 6.

The test for the visual images can be limited to specified pages. In the following example, only the images in a defined region on page 6 are counted:

```
@Test
public void numberOfVisibleImages() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  = 14; // in millimeter
  int upperY = 91;
  int width  = 96;
  int height = 43;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);
  PagesToUse page6 = PagesToUse.getPage(6);

  AssertThat.document(filename)
            .restrictedTo(page6)
            .restrictedTo(pageRegion)
            .hasNumberOfVisibleImages(1)
  ;
}
```

The same image shown twice on a page is counted twice.

The possibilities for limitting tests to specified pages are described in chapter 13.2: “Page Selection” (p. 155).

Validate the Existence of an Expected Image

After counting images you might need to test the images themselves. In the following example, PDFUnit verifies that a given image is part of a PDF document:

```
@Test
public void hasImage() throws Exception {
  String filename = "documentUnderTest.pdf";
  String imageFile = "images/apache-software-foundation-logo.png";

  AssertThat.document(filename)
            .restrictedTo(ANY_PAGE)
            .hasImage()
            .matching(imageFile)
  ;
}
```

The result of a comparison of two images depends on their file formats. PDFUnit can handle all image formats which can be converted into `java.awt.image.BufferedImage`: `JPEG, PNG, GIF, BMP` and `WBMP`. The images are compared byte by byte. Therefore, BMP and PNG versions of an image are not recognized as equal.

The picture may pass to the method in different types:

```
// Types for images:

.hasImage().matching(BufferedImage image);
.hasImage().matching(String imageFileName);
.hasImage().matching(File imageFile);
.hasImage().matching(InputStream imageStream);
.hasImage().matching(URL imageURL);
```

A tool which generates PDF files may do a format conversion when importing images from a file because not all image formats are supported in PDF. This might make it impossible for PDFUnit to successfully compare an image from the PDF file with the original image file. If you have encounter this problem, extract the images of the PDF under test into new image files and use them for the tests. Validate them 'by eye' first.

All images in a PDF document can be compared to the images of a referenced PDF. Those tests are described in chapter 4.8: “Comparing Images” (p. 92).

Use an Array of Images for Comparison

It might be, that a PDF document contains one of three possible signature images. Use the method `matchingOneOf(..)` to test such a situation:

```
@Test
public void containsOneOfManyImages() throws Exception {
  BufferedImage signatureAlex = ImageHelper.getAsImage("images/signature-alex.png");
  BufferedImage signatureBob  = ImageHelper.getAsImage("images/signature-bob.png");
  BufferedImage[] allPossibleImages = {signatureAlex, signatureBob};

  String documentSignedByAlex = "letter-signed-by-alex.pdf";
  AssertThat.document(documentSignedByAlex)
            .restrictedTo(LAST_PAGE)
            .matchingOneOf(allPossibleImages)
  ;

  String documentSignedByBob = "letter-signed-by-bob.pdf";
  AssertThat.document(documentSignedByBob)
            .restrictedTo(LAST_PAGE)
            .matchingOneOf(allPossibleImages)
  ;
}
```

This test can also refer to several sides of a document, as the following section shows.

Validate Images on Specified Pages

The tests for images can be restricted to single pages, multiple individual or multiple contiguous pages. All possibilities are described in chapter 13.2: “Page Selection” (p. 155).

Here is an example:

```
@Test
public void containsImage_OnAllPagesAfter5() throws Exception {
  String filename = "documentUnderTest.pdf";
  String imageFileName = "images/apache-ant-logo.jpg";
  File imageFile = new File(imageFileName);

  AssertThat.document(filename)
            .restrictedTo(EVERY_PAGE.after(5))
            .hasImage()
            .matching(imageFile)
  ;
}
```

Validate the Absences of Images

Some pages or page regions are meant to be empty. That can also be tested. The following example validates that the page body - without header and footer - does not have images or text:

```
@Test
public void lastPageBodyShouldBeEmpty() throws Exception {
  String pdfUnderTest = "documentUnderTest.pdf";
  PageRegion textBody = createBodyRegion();

  AssertThat.document(pdfUnderTest)
            .restrictedTo(LAST_PAGE)
            .restrictedTo(textBody)
            .hasNoImage()
            .hasNoText()
  ;
}
```

3.17. JavaScript

Overview

If JavaScript exists in your PDF documents it is probably important. Often JavaScript plays an active role within document workflows.

PDFUnit's ability to test JavaScript does not replace specialized JavaScript testing tools such as “Google JS Test” , but it is not easy to test the JavaScript in a PDF document using these tools. The following validation methods are provided:

```
// Methods to validate JavaScript:
.hasJavaScript()
.hasJavaScript().containing(..)
.hasJavaScript().equalsTo(..)
.hasJavaScript().equalsToSource(..)
```

Existence of JavaScript

The following method checks whether a document contains JavaScript at all:

```
@Test
public void hasJavaScript() throws Exception {
  String filename = "javaScriptClock.pdf";

  AssertThat.document(filename)
            .hasJavaScript()
  ;
}
```

Comparison against Expected Text

The expected JavaScript can be read from a file and compared with the JavaScript in a PDF document. The utility `ExtractJavaScript` extracts JavaScript into a text file, which can then be used for tests:

```
@Test
public void hasJavaScript_ScriptFromFile() throws Exception {
  String filename = "javaScriptClock.pdf";
  File file = new File("javascript/javascriptClock.js");

  AssertThat.document(filename)
            .hasJavaScript()
            .equalsToSource(file)  ❶
  ;
}
```

❶ The file name parameter can be of type `java.io.File`, `java.io.Reader`, `java.io.InputStream` and `java.lang.String`.

The expected JavaScript need not necessarily be read from a file. It can be passed to the method as a string:

```
@Test
public void hasJavaScript_ComparedToString() throws Exception {
  String filename = "javaScriptClock.pdf";
  String scriptFile = "javascript/javascriptClock.js";
  String scriptContent = IOHelper.getContentAsString(scriptFile);

  AssertThat.document(filename)
            .hasJavaScript()
            .equalsTo(scriptContent)
  ;
}
```

Comparing Substrings

In the previous tests complete JavaScript code was used. But also small parts of a JavaScript code can test:

```
public void hasJavaScript_ContainingText() throws Exception {
  String filename = "javaScriptClock.pdf";

  String javascriptFunction =  "function DoTimers() "
                             + "{ "
                             + "   var nCurTime = (new Date()).getTime(); "
                             + "   ClockProc(nCurTime); "
                             + "   StopWatchProc(nCurTime); "
                             + "   CountDownProc(nCurTime); "
                             + "   this.dirty = false; "
                             + "}"
                             ;

  AssertThat.document(filename)
            .hasJavaScript()
            .containing(javascriptFunction)
  ;
}
```

```
@Test
public void hasJavaScript_ContainingText_FunctionNames() throws Exception {
  String filename = "javaScriptClock.pdf";

  AssertThat.document(filename)
            .hasJavaScript()
            .containing("StopWatchProc")
            .containing("SetFldEnable")
            .containing("DoTimers")
            .containing("ClockProc")
            .containing("CountDownProc")
            .containing("CDEnables")
            .containing("SWSetEnables")
  ;
}
```

Whitespaces are ignored when comparing JavaScript.

3.18. Language

Overview

Screen readers can read PDF documents out loud for visually handicapped users. It helps if a document can inform the screen readers of its language:

These methods are available for tests:

```
// Tests for PDF language:

.hasLanguageInfo(..)
.hasNoLanguageInfo()
```

Examples

The following example verifies that the document language is set to the English language for Great Britain:

```
@Test
public void hasLocale_CaseInsensitive() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasLanguageInfo("en-gb")  ❶
  ;
  AssertThat.document(filename)
            .hasLanguageInfo("en_GB")  ❷
  ;
}
```

❶ Notation typical for PDF
❷ Notation typical for Java

The string for the language is treated case independent. Underscore and hyphen are equivalent.

You can also use `java.util.Locale` directly:

```
@Test
public void hasLocale_LocaleInstance_GERMANY() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasLanguageInfo(Locale.GERMANY)
  ;
}
```

```
@Test
public void hasLocale_LocaleInstance_GERMAN() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasLanguageInfo(Locale.GERMAN)
  ;
}
```

A PDF document with the actual locale `"en_GB"` is tested successfully when using the locale `Locale.en`. In the opposite case, a document with the actual locale `"en"` fails when it is tested against the expected locale `Locale.UK`.

You can also check that a PDF document does **not** have a country code:

```
@Test
public void hasNoLanguageInfo() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNoLanguageInfo()
  ;
}
```

3.19. Layers

Overview

The content of a PDF document can be arranged in multiple layers. Section 8.11.2.1 of the PDF specification "PDF 32000-1:2008" says: "An optional content group is a dictionary representing a collection of graphics that can be made visible or invisible dynamically by users of conforming readers.".

Adobe Reader® uses the term "Layer" and the specification uses the term "OCG". They are equivalent.

PDFUnit provides the following methods to test layers:

```
// Simple methods:

.hasNumberOfLayers(..)  // 'Layer' and ...
.hasNumberOfOCGs(..)    // ...'OCG' are always used the same way

.hasLayer()
.hasOCG()
.hasOCGs()
.hasLayers()

// Methods for layer names:
.hasLayer().withName().containing(..)
.hasLayer().withName().equalsTo(..)
.hasLayer().withName().startingWith(..)
.hasOCG().withName().containing(..)
.hasOCG().withName().equalsTo(..)
.hasOCG().withName().startingWith(..)

// see the plural form:
.hasLayers().allWithoutDuplicateNames()
.hasOCG().allWithoutDuplicateNames()
```

A test method `matchingRegex()` is not provided because layer names are usually short and well known.

Number of Layers

The first tests check the number of existing layers (OCGs):

```
@Test
public void hasNumberOfOCGs() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfOCGs(40)     ❶
            .hasNumberOfLayers(40)   ❷
  ;
}
```

❶❷ “Layer” and “Optional Content Group” are functionally the same. For ease to use, both terms are available as equivalent tags.

Layer Names

The next example tests the name of a layer:

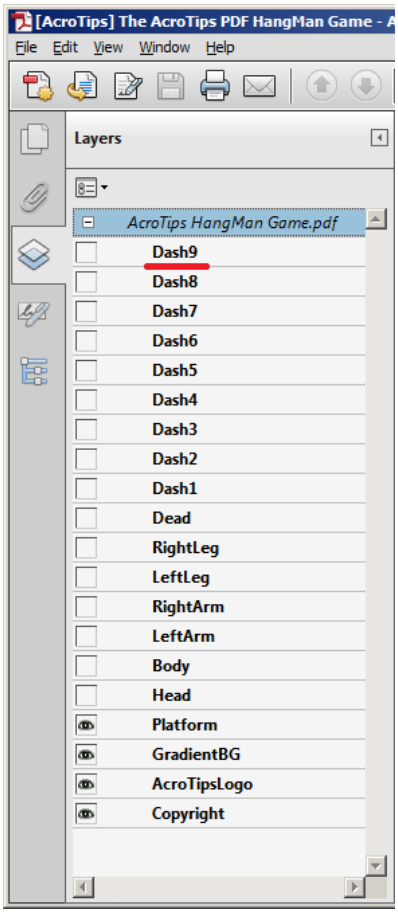
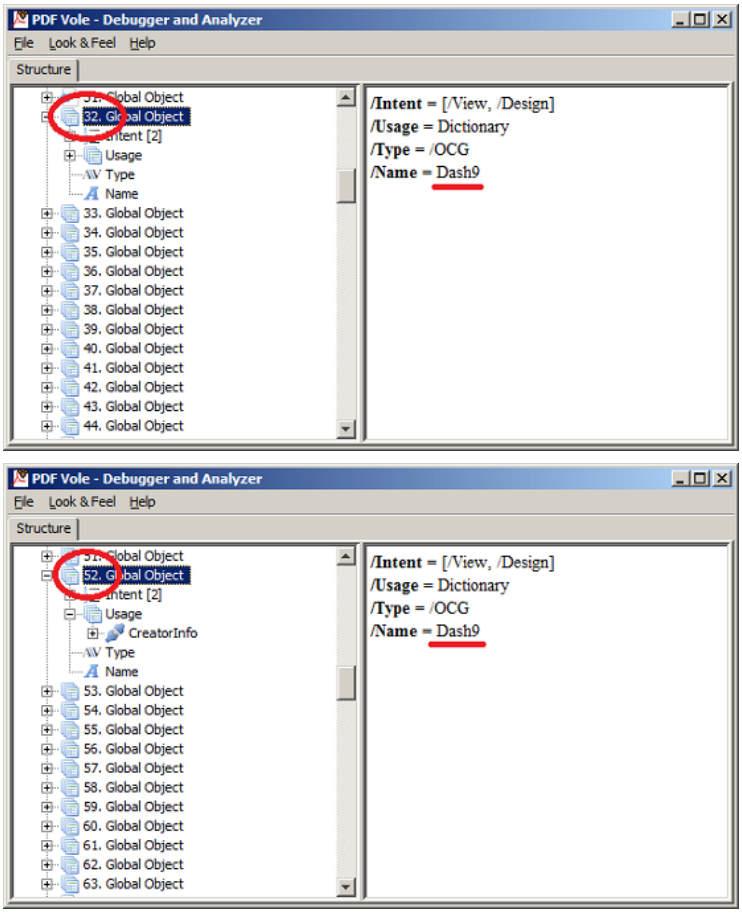
```
@Test
public void hasLayer_WithName() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasLayer()
            .withName().equalsTo("Parent Layer")
  ;
}
```

A layer name can be compared using the following methods:

```
.hasLayer().withName().equalsTo(layerName1)
.hasLayer().withName().startingWith(layerName2)
.hasLayer().withName().containing(layerName3)
```

Duplicate Layer Names

According to the PDF standard, layer names are not necessarily unique. The document of the next example contains duplicate layer names. They can not be seen with Adobe Reader®. Use “PDFVole” instead. shows them:

Layer Names in Adobe Reader®	Layer Objects in PDFVole
	

You can see clearly that the layer objects with the numbers 32 and 52 have the same name “Dash9”.

PDFUnit provides a method to verify that a document has **no duplicate** layer names:

```
@Test
public void hasLayers_AllWithoutDuplicateNames() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasLayers().allWithoutDuplicateNames()
            .hasOCGs().allWithoutDuplicateNames()    // hasOCGs() is equal to hasLayers()
  ;
}
```

In the current release 2016.05 PDFUnit does not provides functions to verify the content of a single layer.

3.20. Layout - Entire PDF Pages

Overview

Text in a PDF document has properties such as font size, font color and lines which should be correct before sending it to the customer. Also, paragraphs, alignment of text, images and image descriptions are important parts of the layout. PDFUnit tests these aspects, first rendering the document page by page and then comparing each page:

- ... with an existing image file. PDFUnit's utility program `RenderPdfToImages` creates images of one or more PDF pages. Chapter 9.14: “Render Pages to PNG” (p. 134) explains how to use it.

- ... with a rendered page of a referenced document. Chapter 4.10: “Comparing Layout as Rendered Pages” (p. 93) describes tests with referenced documents.

The following test methods are provided:

```
// Compare rendered page(s) with a given image.
// The left-upper corner is 0/0:
.asRenderedPage(..).isEqualToImage(BufferedImage)
.asRenderedPage(..).isEqualToImage(File)
.asRenderedPage(..).isEqualToImage(String fileName)
.asRenderedPage(..).isEqualToImages(BufferedImage[] images)
.asRenderedPage(..).isEqualToImages(File[] imageFiles)
.asRenderedPage(..).isEqualToImages(String[] fileNames)

// Compare rendered page(s) with a given image.
// The left-upper corner is given: 3.21: “Layout - in Page Regions” (p. 49)
.asRenderedPage(..).isEqualToImage(upperLeftX, upperLeftY, FormatUnit, BufferedImage)
.asRenderedPage(..).isEqualToImage(upperLeftX, upperLeftY, FormatUnit, File)
.asRenderedPage(..).isEqualToImage(upperLeftX, upperLeftY, FormatUnit, imageFileName)
.asRenderedPage(..).isEqualToImages(upperLeftX, upperLeftY, FormatUnit, BufferedImage)
.asRenderedPage(..).isEqualToImages(upperLeftX, upperLeftY, FormatUnit, File)
.asRenderedPage(..).isEqualToImages(upperLeftX, upperLeftY, FormatUnit, imageFileName)
```

You can choose a page to be compared. This is described in chapter 13.2: “Page Selection” (p. 155). Also, a comparison can be limited to a region of a page. That is described in chapter 3.21: “Layout - in Page Regions” (p. 49).

Example - Compare Preselected Pages as Rendered Images

The following example checks that the pages 1, 3 and 4 look the same as referenced image files:

```
@Test
public void compareAsRenderedPage_MultipleImages() throws Exception {
  String filename = "documentUnderTest.pdf";

  String imagePage1 = "images/documentUnderTest_page1.png";
  String imagePage3 = "images/documentUnderTest_page3.png";
  String imagePage4 = "images/documentUnderTest_page4.png";
  PagesToUse pages134 = PagesToUse.getPages(1, 3, 4);

  AssertThat.document(filename)
            .restrictedTo(pages134)
            .asRenderedPage()
            .isEqualToImages(imagePage1, imagePage3, imagePage4)
  ;
}
```

All image formats which are supported by `java.awt.image.BufferedImage` can be used, i.e. `GIF`, `PNG`, `JPEG`, `BMP` and `WBMP`.

3.21. Layout - in Page Regions

Overview

Comparing entire pages as rendered images will cause problems if a page contains variable content. A date is a typical example of content that changes frequently.

The syntax for comparing sections of a rendered page is very similiar to the syntax for comparing entire pages. The method `isEqualToImage(..)` is extended with the x/y values of the upper left corner used to position the image on the page. Only the area corresponding to the size of the image.

```
// Compare rendered page(s) with a given image.
// The left-upper corner is defined by the given x/y values.
.asRenderedPage(..).isEqualToImage(upperLeftX, upperLeftY, FormatUnit, BufferedImage)
.asRenderedPage(..).isEqualToImage(upperLeftX, upperLeftY, FormatUnit, File)
.asRenderedPage(..).isEqualToImage(upperLeftX, upperLeftY, FormatUnit, imageFileName)
.asRenderedPage(..).isEqualToImages(upperLeftX, upperLeftY, FormatUnit, BufferedImage)
.asRenderedPage(..).isEqualToImages(upperLeftX, upperLeftY, FormatUnit, File)
.asRenderedPage(..).isEqualToImages(upperLeftX, upperLeftY, FormatUnit, imageFileName)
```

Example - Left Margin on Every Page

If you want to check that the left margin of each page is empty for at least 2 cm, then you can write this test:

```
@Test
public void compareAsRenderedPage_LeftMargin() throws Exception {
  String filename = "documentUnderTest.pdf";

  String fullImage2cmWidthFromLeft = "images/marginFullHeight2cmWidth.png";
  int leftX  = 0;
  int upperY = 0;

  AssertThat.document(filename)
            .restrictedTo(EVERY_PAGE)
            .asRenderedPage()
            .isEqualToImage(leftX, upperY, MILLIMETERS, fullImage2cmWidthFromLeft)
  ;
}
```

The image is 2 cm wide and as high as the page. It contains the background color of the PDF pages. So, the example verifies that the margin of each page has the same background color. That means the margin is “empty”.

Every section needs an x/y position within the PDF page. The values 0/0 correspond to the upper left corner of a page.

The test assumes that all pages of the PDF have the same size. If you want to check left margins for pages of different formats in a single PDF document, you have to write multiple tests, each for pages of the same format.

Example - Logo on Page 1 and 2

The next example verifies that the company logo is placed at an expected position on pages 1 and 2:

```
@Test
public void verifyLogoOnEachPage() throws Exception {
  String filename = "documentUnderTest.pdf";
  String logo = "images/logo.png";

  int leftX  = 135;                                          ❶
  int upperY =  35;
  PagesToUse pages12 = PagesToUse.getPages(1, 2);            ❷

  AssertThat.document(filename)
            .restrictedTo(pages12)
            .asRenderedPage()
            .isEqualToImage(leftX, upperY, MILLIMETERS, logo) ❸
  ;
}
```

❶ Set the x/y position of the upper left corner of the region
❷ Specify the pages, see chapter 13.2: “Page Selection” (p. 155)
❸ Invoke the test method

Multiple Comparisons

Multiple pages can be compared with multiple images in a single test:

```
@Test
public void compareAsRenderedPage_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";

  String fullImage2cmWidthFromLeft = "images/marginFullHeight2cmWidth.png";
  int ulX_Image1 = 0;  // upper left X
  int ulY_Image1 = 0;  // upper left Y

  String subImagePage3And4 = "images/subImage_page3-page4.png";
  int ulX_Image2 = 480;
  int ulY_Image2 = 765;

  PagesToUse pages34 = PagesToUse.getPages(3, 4);

  AssertThat.document(filename)
            .asRenderedPage(pages34)
            .isEqualToImage(ulX_Image1, ulY_Image1, POINTS, fullImage2cmWidthFromLeft)
            .isEqualToImage(ulX_Image2, ulY_Image2, POINTS, subImagePage3And4)
  ;
}
```

However, you should consider whether it is better to write two tests for this. The decisive argument for separate tests is that you can choose two different names. The name chosen here is not good enough for a real project.

3.22. Number of PDF Elements

Overview

Not only the number of pages can be a test goal, also any kind of countable items in a PDF document, e.g. form fields and bookmarks. The following list shows the items that are countable and therefore testable:

```
// Test counting parts of a PDF:
.hasNumberOfActions(..)
.hasNumberOfBookmarks(..)
.hasNumberOfDifferentImages(..)    ❶
.hasNumberOfEmbeddedFiles(..)
.hasNumberOfFields(..)
.hasNumberOfFonts(..)
.hasNumberOfLayers(..)
.hasNumberOfOCGs(..)
.hasNumberOfPages(..)              ❷
.hasNumberOfSignatures(..)
.hasNumberOfVisibleImages(..)      ❸
```

❶❸ Tests for the number of images are described in chapter 3.16: “Images in PDF Documents” (p. 40).

❷ Tests for the number of pages are described in chapter 3.23: “Page Numbers as Objectives” (p. 52).

Examples

Validating the number of items in PDF documents works identically for all items. So, only two of them are shown as examples:

```
@Test
public void hasNumberOfFields() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfFields(4)
  ;
}
```

```
@Test
public void hasNumberOfBookmarks() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfBookmarks(19)
  ;
}
```

All tests can be concatenated:

```
@Test
public void testHugeDocument_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfPages(1370)
            .hasNumberOfBookmarks(565)
            .hasNumberOfActions(4814)
            .hasNumberOfEmbeddedFiles(0)
  ;
}
```

Fortunately, this test takes only 2.5 seconds for a document with 1370 pages on a contemporary notebook.

3.23. Page Numbers as Objectives

Overview

It is sometimes useful to check if a generated PDF document has exactly one page. Or maybe you want to ensure that a document has less than 6 pages, because otherwise you have to pay higher postage. PDFUnit provides suitable test methods:

```
// Method for tests with pages:
.hasNumberOfPages(..)
.hasLessPagesThan(..)
.hasMorePagesThan(..)
```

Examples

You can check the number of pages like this:

```
@Test
public void hasNumberOfPages() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNumberOfPages(1)
  ;
}
```

Tests are also possible with a minimum or maximum number of pages.

```
@Test
public void hasLessPagesThan() throws Exception {
  String filename = "documentUnderTest.pdf";
  int upperLimitExclusive = 6; // The document has 5 pages

  AssertThat.document(filename)
            .hasLessPagesThan(upperLimitExclusive)  ❶
  ;
}
```

```
@Test
public void hasMorePagesThan() throws Exception {
  String filename = "documentUnderTest.pdf";
  int lowerLimitExclusive = 2; // The document has 5 pages

  AssertThat.document(filename)
            .hasMorePagesThan(lowerLimitExclusive)  ❷
  ;
}
```

❶❷ The values for upper- and lower limits are exclusive.

Of course methods can be concatenated:

```
@Test
public void hasNumberOfPages_InRange() throws Exception {
  String filename = "documentUnderTest.pdf";
  // The current document has 5 pages
  int lowerLimit_2 = 2;  // the limit is exclusive
  int upperLimit_8 = 8;  // the limit is exclusive

  AssertThat.document(filename)
            .hasMorePagesThan(lowerLimit_2)
            .hasLessPagesThan(upperLimit_8)
  ;
}
```

Don't omit tests with page numbers just because you might think they are too simple. Experience shows that you can find errors in the context of a primitive test that you would not have found without the test.

3.24. Passwords

Overview

In general, you can perform all tests with both unprotected and password protected PDF documents. The syntax differs slightly in the instantiation method, a second parameter is required representing the password:

```
// Access to password protected PDF
AssertThat.document(filename, ownerPassword) ❶ ❷

// Test methods:
.hasEncryptionLength(..)
.hasOwnerPassword(..)
.hasUserPassword(..)
```

❶ If the document is not protected, use only the first parameter.

❷ If the second parameter is used, the document is considered protected.

This syntax is the same for “user password” and “owner password”.

Verifying Both Passwords

Opening a document with a password is already a test. But you can verify the second password with the methods `hasOwnerPassword(..)` or `hasUserPassword(..)`:

```
// Verify the owner-password of the document:
@Test
public void hasOwnerPassword() throws Exception {
  String filename = "documentUnderTest.pdf";
  String userPassword = "user-password";

  AssertThat.document(filename, userPassword)    ❶
            .hasOwnerPassword("owner-password")  ❷
  ;
}
```

```
// Verify the user-password of the document:
@Test
public void hasUserPassword() throws Exception {
  String filename = "documentUnderTest.pdf";
  String ownerPassword = "owner-password";

  AssertThat.document(filename, ownerPassword)   ❸
            .hasUserPassword("user-password")    ❹
  ;
}
```

❶❸ Open the file with one password
❷❹ Verify the other password

Usually it's bad practice to hard code passwords in the source code, but it's OK for test passwords in test environments.

Verifying the Password Encryption Length

This example shows how to verify the encryption length:

```
@Test
public void hasEncryptionLength() throws Exception {
  String filename = "documentUnderTest.pdf";
  String userPassword = "user-password";

  AssertThat.document(filename, userPassword)
            .hasEncryptionLength(128)
  ;
}
```

3.25. PDF/A

Overview

There are many tools for checking whether a PDF document complies with one of the PDF standards (PDF/A, PDF/X etc.). But only a few of them can be used for automated tests. PDFUnit uses the PDFBox's 'Preflight' parser internally to check the compliance with PDF/A.

Information on the 'Preflight' parser can be found on the project web site https://pdfbox.apache.org/1.8/cookbook/pdfavalidation.html.

A PDF/A validation starts with the following method:

```
// Validate PDF/A-1a and PDF/A-1b:
.compliesWith().pdfStandard(Standard)
```

Example

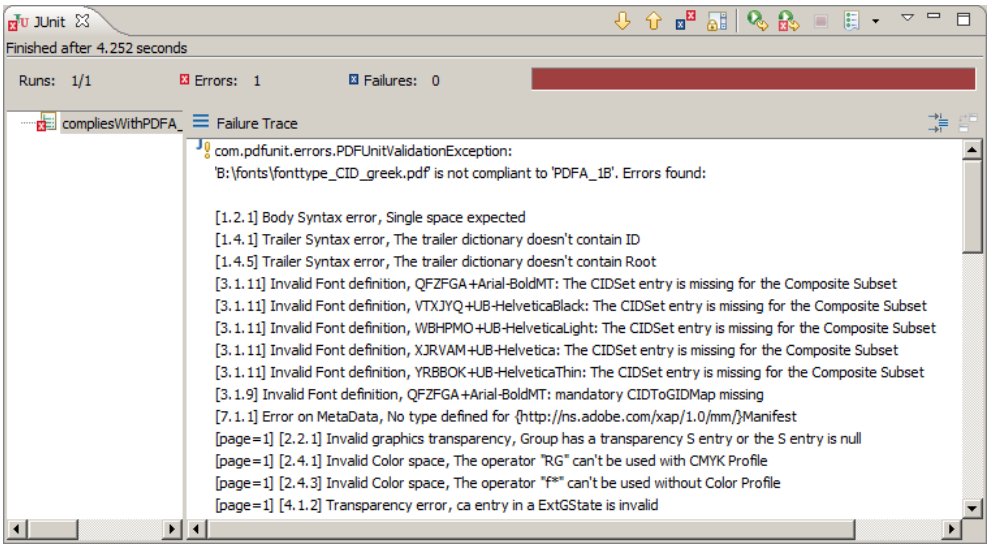
The next example checks compliance with PDF/A-1a:

```
@Test
public void compliesWithPDFA() throws Exception {
  String fileName = "pdf_a/documentUnderTest.pdf"; ❶

  AssertThat.document(fileName)
            .compliesWith()
            .pdfStandard(PDFA_1A)                  ❷
  ;
}
```

❶ Files can be tested but not byte-arrays or streams.

❷ The validation can be parameterized using one of the two constants PDFA_1A and PDFA_1B.

The error messages from the Preflight parser are passed through to PDFUnit. The images below show examples in Eclipse and in an HTML browser:




```
[1.2.1] Body Syntax error, Single space expected
[1.4.1] Trailer Syntax error, The trailer dictionary doesn't contain ID
[1.4.5] Trailer Syntax error, The trailer dictionary doesn't contain Root
[3.1.11] Invalid Font definition, QFZFGA+Arial-BoldMT: The CIDSet entry is missing for the Composite Subset
[3.1.11] Invalid Font definition, VTXJYQ+UB-HelveticaBlack: The CIDSet entry is missing for the Composite Subset
[3.1.11] Invalid Font definition, WBHPMO+UB-HelveticaLight: The CIDSet entry is missing for the Composite Subset
[3.1.11] Invalid Font definition, XJRVAM+UB-Helvetica: The CIDSet entry is missing for the Composite Subset
[3.1.11] Invalid Font definition, YRBBOK+UB-HelveticaThin: The CIDSet entry is missing for the Composite Subset
[3.1.9] Invalid Font definition, QFZFGA+Arial-BoldMT: mandatory CIDToGIDMap missing
[7.1.1] Error on MetaData, No type defined for {http://ns.adobe.com/xap/1.0/mm/}Manifest
[page=1] [2.2.1] Invalid graphics transparency, Group has a transparency S entry or the S entry is null
[page=1] [2.4.1] Invalid Color space, The operator "RG" can't be used with CMYK Profile
[page=1] [2.4.3] Invalid Color space, The operator "f*" can't be used without Color Profile
[page=1] [4.1.2] Transparency error, ca entry in a ExtGState is invalid
[page=2] [2.2.1] Invalid graphics transparency, Group has a transparency S entry or the S entry is null
[page=2] [2.4.1] Invalid Color space, The operator "RG" can't be used with CMYK Profile
[page=2] [2.4.3] Invalid Color space, The operator "f*" can't be used without Color Profile
[page=2] [4.1.2] Transparency error, ca entry in a ExtGState is invalid
[page=3] [2.2.1] Invalid graphics transparency, Group has a transparency S entry or the S entry is null
[page=3] [2.4.1] Invalid Color space, The operator "RG" can't be used with CMYK Profile
[page=3] [2.4.3] Invalid Color space, The operator "f*" can't be used without Color Profile
[page=3] [4.1.2] Transparency error, ca entry in a ExtGState is invalid
[page=4] [2.2.1] Invalid graphics transparency, Group has a transparency S entry or the S entry is null
[page=4] [2.4.1] Invalid Color space, The operator "RG" can't be used with CMYK Profile
```

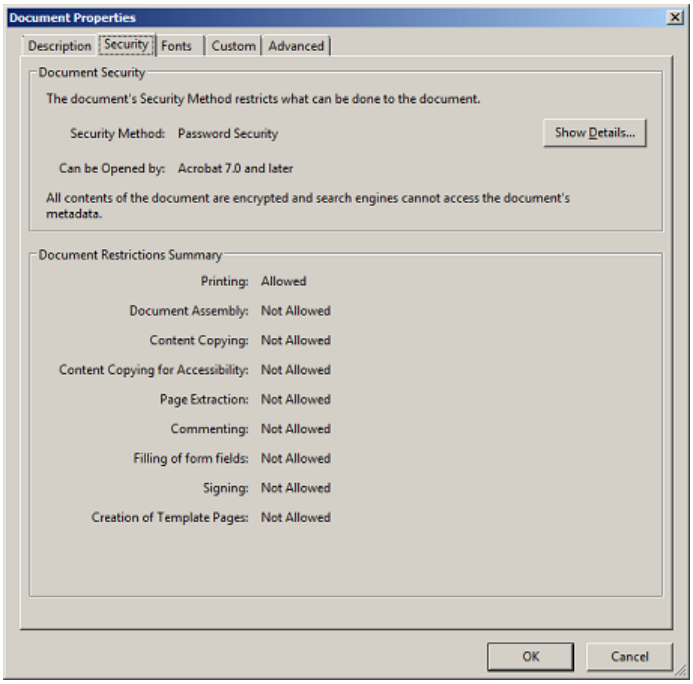
A PDF/A validation can also be applied to all PDF documents in a given folder:

```
@Test
public void compliesWithPDFA_InFolder() throws Exception {
  File foldertoWatch = new File("pdf_a-1b");
  FilenameFilter filenameFilter = new FilenameContainingFilter("tn0001");

  AssertThat.eachDocument()
            .inFolder(foldertoWatch)
            .passedFilter(filenameFilter)
            .compliesWith()
            .pdfStandard(PDFA_1B)
  ;
}
```

3.26. Permissions

Overview

If you expect your workflow to create copy protected PDF documents, you should test that. You can see the permissions using Adobe Reader®, but that is a poor “test”:



You can test permissions using the following methods. All methods have one typed parameter with the values `true` or `false`:

```
// Testing permissions:
.toAllowScreenReaders(..)
.toAssembleDocument(..)
.toExtractContent(..)
.toFillInFields(..)
.toModifyAnnotations(..)
.toModifyContent(..)
.toPrintInDegradedQuality(..)
.toPrintInHighQuality(..)
```

Example

```
@Test
public void hasPermission_ScreenReadersAllowed() throws Exception {
  String filename = "documentUnderTest.pdf";
  AssertThat.document(filename)
            .hasPermission()
            .toAllowScreenReaders(true)
  ;
}
```

The document permissions of a protected document differ depending on whether the document is opened with the owner password or with the user password.

3.27. QR Code

Overview

A QR code is a 2-dimensional code and can be tested similar to the 1-dimensional bar code. Internally, PDFUnit uses ZXing to parse QR code. Information about ZXing is available on the project's home page: https://github.com/zxing/zxing.

The following methods can be used to validate QR codes inside PDF files:

```
// Entry to all QR code validations:
.hasImage().withQRCode()

// Validate text in barcode:
...withQRCode().containing(..)
...withQRCode().containing(.., WhitespaceProcessing)
...withQRCode().endingWith(..)
...withQRCode().equalsTo(..)
...withQRCode().equalsTo(.., WhitespaceProcessing)
...withQRCode().matchingRegex(..)
...withQRCode().startingWith(..)

// Validate text in QR code in image region:
...withQRCodeInRegion(imageRegion).containing(..)
...withQRCodeInRegion(imageRegion).containing(.., WhitespaceProcessing)
...withQRCodeInRegion(imageRegion).endingWith(..)
...withQRCodeInRegion(imageRegion).equalsTo(..)
...withQRCodeInRegion(imageRegion).equalsTo(.., WhitespaceProcessing)
...withQRCodeInRegion(imageRegion).matchingRegex(..)
...withQRCodeInRegion(imageRegion).startingWith(..)

// Compare with another QR code:
...withQRCode().matchingImage(..)
```

ZXing detects QR code formats automatically. The following formats can be used in PDFUnit tests:

```
// 2D codes, supported by PDFUnit/ZXing:
AZTEC
DATA_MATRIX
QR_CODE
MAXICODE

// Stacked barcode, support by PDFUnit/ZXing:
PDF_417
RSS_14
RSS_EXPANDED
```

Example - Validate Text from QR Code

The QR code used in the next examples contains text from Herman Melville's novel 'Moby-Dick'.

```
@Test
public void hasQRCodeWithText_MobyDick() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  =  10;
  int upperY = 175;
  int width  =  70;
  int height =  70;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  String expectedText = "Some years ago--never mind how long precisely";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .withQRCode()
            .containing(expectedText)
  ;
}
```

When multiple images occupy a defined region, each image must pass the test. Default whitespace processing when searching text in QR code is NORMALIZE, but whitespace processing can be controlled using a method parameter.

PDFUnit's internal QR code parser ZXing supports many, but not all, formats. So PDFUnit provides an external interface to plug in customer specific QR code parsers. This interface is documented separately. You can requesting it by writing an email to info[at]pdfunit.com.

Example - QR Code as Part of an Image

The next example uses an image that contains two QR codes. To let the test focus on only one QR code, an image region has to be defined. The reference point of such a region is the upper-left corner of the image. Note: the unit for image size values is points, the unit for page size values is millimeter.

```
@Test
public void hasQRCodeInRegion_MobyDick() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  =   0;
  int upperY =   0;
  int width  = 209;
  int height = 297;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  int imgLeftX  = 120; // pixel
  int imgUpperY =   0;
  int imgWidth  = 140;
  int imgHeight = 140;
  ImageRegion imageRegion = new ImageRegion(imgLeftX, imgUpperY, imgWidth, imgHeight);

  String expectedText = "Some years ago--never mind how long precisely";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .withQRCodeInRegion(imageRegion)
            .containing(expectedText)
  ;
}
```

Example - Compare QR Code with an External Image

In the next example, a QR code from a PDF is compared to a QR code image from a file:

```
@Test
public void hasQRCodeMatchingFile_OnAnyPage() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  = 10;
  int upperY = 65;
  int width  = 50;
  int height = 50;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  String expectedQrImage = "images/hello.qrgen.png";
  AssertThat.document(filename)
            .restrictedTo(ANY_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .withQRCode()
            .matchingImage(expectedQrImage)
  ;
}
```

Important: For the test to work, the type of the external image file (PNG or TIFF, for example) must match the type of the QR code image of the PDF document.

3.28. Signed PDF

Overview

If contractual information is sent as a PDF document, you must check that the data really was sent by the person they claim to be. Certificates allow this. A certificate confirms the authenticity of personal or corporate data. It confirms your signature when you “sign” the content of a document in a special formular field.

PDFUnit provides these test methods for signatures:

```
// Simple methods for signatures:
.isSigned()
.isSignedBy(..)
.hasNumberOfSignatures(..)
.hasSignatureField(..)
.hasSignatureFields()

// Detailed tests for one signature:
.hasSignatureField(..).withSignature(..).coveringWholeDocument()
.hasSignatureField(..).withSignature(..).signedBy(name)
.hasSignatureField(..).withSignature(..).signedOn(date)
.hasSignatureField(..).withSignature(..).withReason(..)
.hasSignatureField(..).withoutSignature(..)

// Tests covering all signature fields:
.hasSignatureFields().allSigned()
.hasSignatureFields().allUnSigned()

// Other tests with signatures:
.hasField(..).ofType(SIGNATURE)
.hasField(..).withProperty().signed()
```

A “signed” PDF must not be confused with a “certified” PDF. A “certified” PDF guarantees the compliance with certain properties which are needed to process a document in further workflow steps. Tests for certified PDF documents are described in chapter 3.6: “Certified PDF” (p. 21).

Existence of Signatures

The simplest test is to check whether a document is actually signed:

```
@Test
public void isSigned() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";

  AssertThat.document(filename)
            .isSigned()
  ;
}
```

Multiple signature fields can be verified together:

```
@Test
public void allFieldsSigned() throws Exception {
  String filename = "documentUnderTest.pdf";
  AssertThat.document(filename)
            .hasSignatureFields()
            .allSigned()
  ;
}
```

The method `.allUnsigned()` validates the absence of all signatures in all fields.

PDFUnit can check, whether a particular signature field contains a signature:

```
@Test
public void hasField_Signed() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";
  String fieldnameSignedField = "Signature2";

  AssertThat.document(filename)
            .hasField(fieldnameSignedField)
            .withProperty()
            .signed()
  ;
}
```

The next example shows how to check whether there are any specific signature fields:

```
@Test
public void hasSignatureFields() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";

  AssertThat.document(filename)
            .hasSignatureField("Signature of Seller")
            .hasSignatureField("Signature of Buyer")
  ;
}
```

The previous example can be extended to check whether specific signature fields are signed:

```
@Test
public void hasSignatureFieldsWithSignature() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";

  AssertThat.document(filename)
            .hasSignatureField("Signature of Seller").withSignature()
            .hasSignatureField("Signature of Buyer").withSignature()
  ;
}
```

The function `.hasSignatureField("name").withoutSignature()` checks that a signature field is unsigned.

Number of Signatures

Because a document may contain multiple signatures, a test exists to check the number of the signatures:

```
@Test
public void hasNumberOfSignatures() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";

  AssertThat.document(filename)
            .hasNumberOfSignatures(1)
  ;
}
```

Validity Date

Sometimes, you need to know when a PDF document was signed:

```
@Test
public void hasSignature_WithSigningDate() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";
  Calendar signingDate = DateHelper.getCalendar("2009-07-16", "yyyy-MM-dd");

  AssertThat.document(filename)
            .hasSignatureField("Signature2")
            .signedOn(signingDate)  ❶
  ;
}
```

❶ The comparison is allways made on the base of year-month-day.

Reason for a Signature

Maybe, it's not interesting enough to validate the signature-reason. But if it turns out to be, it can be checked:

```
@Test
public void hasSignature_WithReason() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";

  AssertThat.document(filename)
            .hasSignatureField("Signature2")
            .withSignature()
            .withReason("I am the author of this document")
  ;
}
```

Whitespaces will be normalized before string comparison.

Name of the Signatory

The name of the person who signed a PDF document can be checked as well, even if such a test is more interesting for validation in a productive work flow than in a testing scenario.

```
@Test
public void hasSignature_WithSigningName() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";

  AssertThat.document(filename)
            .hasSignatureField("Signature2")
            .withSigningName("John B Harris")
  ;
}
```

The following test checks that a document is signed by the expected person. This test is unrelated to a signature field.

```
@Test
public void isSignedBy() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";

  AssertThat.document(filename)
            .isSignedBy("John B Harris")
  ;
}
```

Scope of the Signature

The PDF standard allows a signature to cover only a part of a document. Thus, one may need to test whether a signature covers the complete document:

```
@Test
public void hasSignature_CoveringWholeDocument() throws Exception {
  String filename = "sampleSignedPDFDocument.pdf";

  AssertThat.document(filename)
            .hasSignatureField("Signature2")
            .coveringWholeDocument()
  ;
}
```

Multiple Invocation

Of course, test methods can be chained when they are related to one signature field:

```
@Test
public void differentAspectsAroundSignature() throws Exception {
  String filename = "helloWorld_signed.pdf";
  Calendar signingDate = DateHelper.getCalendar("2007-10-14", "yyyy-MM-dd");

  AssertThat.document(filename)
            .hasSignatureField("sign_rbl")
            .signedBy("Raymond Berthou")
            .signedOn(signingDate)
            .coveringWholeDocument()
  ;
}
```

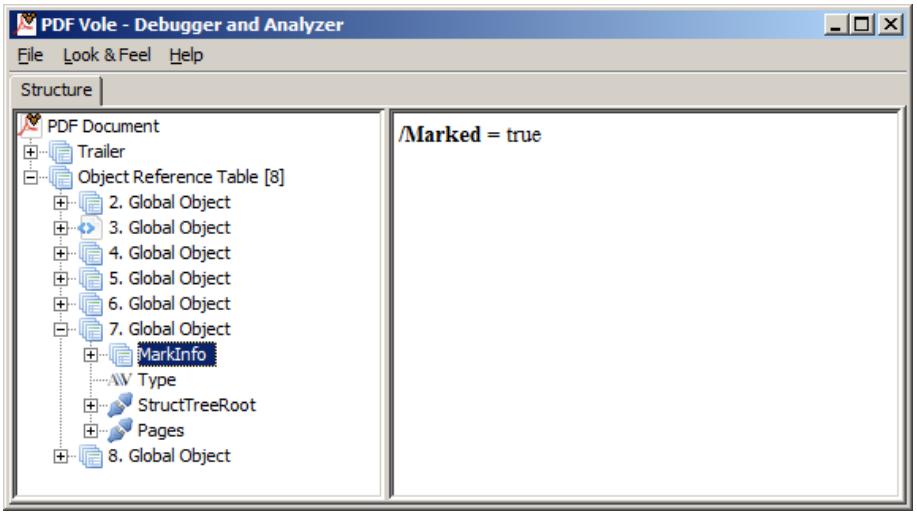
But think of a better name for this test. It would be better to split it into several tests with specific names.

3.29. Tagged Documents

Overview

The PDF standard “ISO 32000-1:2008” says in chapter 14.8.1 “A Tagged PDF document shall also contain a mark information dictionary (see Table 321) with a value of true for the Marked entry.” (Cited from: .)

Although the standard says “shall”, PDFUnit looks in a PDF document for a dictionary with the name `/MarkInfo`. And if that dictionary contains the key `/Marked` with the value `true`, PDFUnit identifies the PDF document as “tagged”.



Following test methods are available:

```
// Simple tests:
.isTagged()

// Tag value tests:
.isTagged().with(..)
.isTagged().with(..).andValue(..)
```

Examples

The simplest test checks whether a document is tagged.

```
@Test
public void isTagged() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .isTagged()
  ;
}
```

Further tests verify the existence of a particular tag.

```
@Test
public void isTagged_WithKey() throws Exception {
  String filename = "documentUnderTest.pdf";
  String tagName = "LetterspaceFlags";

  AssertThat.document(filename)
            .isTagged()
            .with(tagName)
  ;
}
```

And finally you can verify values of tags:

```
@Test
public void isTagged_WithKeyAnValue_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .isTagged()
            .with("Marked").andValue("true")
            .with("LetterspaceFlags").andValue("0")
  ;
}
```

3.30. Text

Overview

The most common test case for PDF documents is probably to check the presence of expected text. Various methods can be used:

```
// Testing page content:
.hasText()      // pages has to be specified before

// Validating expected text:
.hasText().containing(..)
.hasText().containing(.., WhitespaceProcessing)    ❶
.hasText().endingWith(..)
.hasText().endingWith(.., WhitespaceProcessing)
.hasText().equalsTo(..)
.hasText().equalsTo(.., WhitespaceProcessing)
.hasText().matchingRegex(..)
.hasText().startingWith(..)

// Prove the absence of defined text:
.hasText().notContaining(..)
.hasText().notContaining(.., WhitespaceProcessing)
.hasText().notEndingWith(..)
.hasText().notMatchingRegex(..)
.hasText().notStartingWith(..)

// Validate multiple text in an expected order:
.hasText().inOrder(..)
.hasText().containingFirst(..).then(..)

// Comparing visible text with ZUGFeRD data:
.hasText.containingZugferdData(..)                 ❷
```

❶ Chapter 13.5: “Whitespace Processing” (p. 159) describes the different options to handle whitespaces.

❷ Chapter 3.39: “ZUGFeRD” (p. 82) describes how to compare the visible content of a PDF document with the invisible ZUGFeRD data.

Text on Individual Pages

If you are looking for a text on the first page of a letter, test it this way:

```
@Test
public void hasText_OnFirstPage() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing("Content on first page.")
  ;
}
```

The next example searches a text on the last page:

```
@Test
public void hasText_OnLastPage() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(LAST_PAGE)
            .hasText()
            .containing("Content on last page.")
  ;
}
```

Also, you can tests individual pages:

```
@Test
public void hasText_OnIndividualPages() throws Exception {
  String filename = "documentUnderTest.pdf";
  PagesToUse pages23 = PagesToUse.getPages(2, 3);  ❶

  AssertThat.document(filename)
            .restrictedTo(pages23)
            .hasText()
            .containing("Content on")
  ;
}
```

❶ Using the method `getPages(Integer[])` any individual combination of pages can be used. For a single page you can use the method `PagesToUse.getPage(int)`.

Several constants are available for typical pages, e.g. `FIRST_PAGE`, `LAST_PAGE`, `EVEN_PAGES` und `ODD_PAGES`. Chapter 13.2: “Page Selection” (p. 155) describes more constants and how to use them.

Text on All Pages

There are three constants for searching text on multiple pages: `ANY_PAGE`, `EACH_PAGE` and `EVERY_PAGE`. The last two are functionally identical. Together they enable a higher linguistic flexibility.

```
@Test
public void hasText_OnEveryPage() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .retricted(EVERY_PAGE)
            .hasText()
            .startingWith("PDFUnit")
  ;
}
```

```
@Test
public void hasText_OnAnyPage() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(ANY_PAGE)
            .hasText()
            .containing("Page # 3")
  ;
}
```

The constants `EVERY_PAGE` and `EACH_PAGE` require that the text really exists **on each page**. When you use the constant `ANY_PAGE`, a test is successful if the expected text exists **on one or more pages**.

Text in Parts of a Page

Text can be searched not only on whole pages, but also in a region of a page. Chapter 3.32: “Text - in Page Regions” (p. 71) describes that topic.

Individual Pages with Upper and Lower Limit

Do you need to know that an expected text can be found on every page except the first page? Such a test looks like this:

```
@Test
public void hasText_OnAllPagesAfter3() throws Exception {
  String filename = "documentUnderTest.pdf";
  PagesToUse pagesAfter3 = ON_EVERY_PAGE.after(3);    ❶

  AssertThat.document(filename)
            .restrictedTo(pagesAfter3)
            .hasText()
            .containing("Content")
  ;
}
```

❶ The value '3' is an exclusive lower bound. Validation starts at page 4.

Page numbers start from “1”.

Invalid page limits are not necessarily an error. In the following example, the text is searched for on all pages between 1 and 99 (exclusive). Although the document has only 4 pages, the test ends successfully because the expected string is found on page 1:

```
/**
 * Attention: The document has the search token on page 1.
 * And '1' is before '99'. So, this test ends successfully.
 */
@Test
public void hasText_OnAnyPageBefore_WrongUpperLimit() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(AnyPage.before(99))
            .hasText()
            .containing("Content on")
  ;
}
```

Validate Text That Spans Over Multiple Pages

Text to be validated may extend over 2 or more pages, but be interrupted by the header and footer on each page. Such text can be tested as follows:

```
@Test
public void hasText_SpanningOver2Pages() throws Exception {
  String filename = "documentUnderTest.pdf";
  String textOnPage1 = "Text starts  on page 1 and ";
  String textOnPage2 = "continues on page 2";
  String expectedText = textOnPage1 + textOnPage2;
  PagesToUse pages1to2 = PagesToUse.spanningFrom(1).to(2);

  // Define the section without header and footer:
  int leftX  = 18;
  int upperY = 30;
  int width  = 182;
  int height = 238;
  PageRegion regionWithoutHeaderAndFooter = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filename)
            .restrictedTo(pages1to2)
            .restrictedTo(regionWithoutHeaderAndFooter)
            .hasText()
            .containing(expectedText)
  ;
}
```

Other methods can be used instead of the method `.containing()` to compare text.

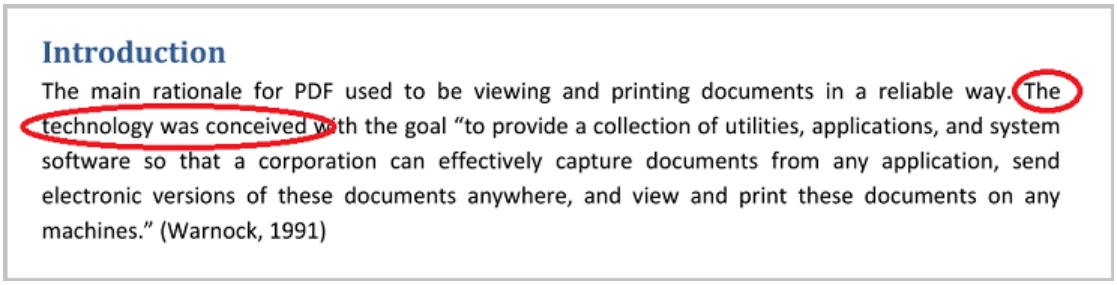
Negated Search

The absence of text can also be a test objective, particularly in a certain region of a page. Tests for that are follow common speech patterns:

```
@Test
public void hasText_NotMatchingRegex() throws Exception {
  String filename = "documentUnderTest.pdf";

  PagesToUse page2 = PagesToUse.getPage(2);
  PageRegion region = new PageRegion(70, 80, 90, 60);
  AssertThat.document(filename)
            .restrictedTo(page2)
            .restrictedTo(region)
            .hasNoText()
  ;
}
```

Line Breaks in Text

When searching text, line breaks and other whitespaces are ignored in the expected text as well as in the text being tested. In the following example the text to be searched belongs to the document "Digital Signatures for PDF Documents" from Bruno Lowagie (iText). The first chapter has some line breaks:



The following tests for the marked text use different line breaks. They both succeed because white-spaces will be normalized by default:

```
/**
 * The expected search string does not contain a line break.
 */
@Test
public void hasText_LineBreakInPDF() throws Exception {
  String filename = "digitalsignatures20121017.pdf";
  String text = "The technology was conceived";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(text)
  ;
}
```

```
/**
 * The expected search string intentionally contains other line breaks.
 */
@Test
public void hasText_LineBreakInExpectedString() throws Exception {
  String filename = "digitalsignatures20121017.pdf";
  String text = "The " +
                "\n " +
                "technology " +
                "\n " +
                "was " +
                "\n " +
                "conceived";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(text)
  ;
}
```

If a normalization of whitespace is not desired, most methods allow an additional parameter defining the intended whitespace handling. The following constants are available:

```
// Constants to define whitespace processing:
WhitespaceProcessing.IGNORE
WhitespaceProcessing.NORMALIZE
WhitespaceProcessing.KEEP
```

No Empty Pages

You can verify that your PDF document does not have empty pages:

```
@Test
public void hasText_AnyPageEmpty() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasText()
  ;
}
```

Multiple Search Tokens

It is annoying to write a separate test for every expected text. So, it is possible to invoke the methods `containing(..)` and `notContaining(..)` with an array of expected texts:

```
@Test
public void hasText_Containing_MultipleTokens() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(ODD_PAGES)
            .hasText()
            .containing("on", "page", "odd pagenumber") // multiple search tokens
  ;
}
```

```
@Test
public void hasText_NotContaining_MultipleTokens() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .notContaining("even pagenumber", "Page #2")
  ;
}
```

The first example is successful when **all** expected tokens are **found**, and the second test is successful when **none** of the expected tokens are **found**.

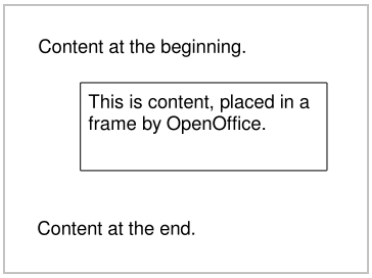
Concatenation of Methods

All chained test methods refer to the same pages:

```
@Test
public void hasText_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(ANY_PAGE)
            .hasText()
            .startingWith("PDFUnit")
            .containing("Content on last page.")
            .matchingRegex(".*[Cc]ontent.*")
            .endingWith("of 4")
  ;
}
```

Visible Text Order - Potential Problem

The visible sequence of text on a PDF page does not necessarily correspond to the text sequence within the PDF document. The next screenshot shows a text frame which is not part of the 'normal' text of the page body. That is the reason why the next test succeeds:




```
@Test
public void hasText_TextNotInVisibleOrder() throws Exception {
  String filename = "documentUnderTest.pdf";

  String firstAndLastLine = "Content at the beginning. Content at the end.";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(firstAndLastLine)

  ;
}
```

If you imagine removing the border of the frame, it seems that the text inside the frame is the middle part of one text block. However, a test expecting the complete text would fail.

3.31. Text - in Images (OCR)

Overview

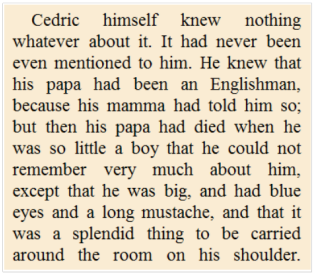
PDFUnit can extract text from images and can validate this text in the same way as normal text. The syntax of these OCR tests follows natural language as much as possible. Each OCR tests starts with the methods `hasImage().withText()` or `hasImage().withTextInRegion()`. The following methods can be used to validate text in images:

```
// Tests for text in images:
.hasImage().withText().containing(..)
.hasImage().withText().endingWith(..)
.hasImage().withText().equalsTo(..)
.hasImage().withText().matchesRegex(..)
.hasImage().withText().startingWith(..)

// Tests for text in parts of an image:
.hasImage().withTextInRegion(imageRegion).containing(..)
.hasImage().withTextInRegion(imageRegion).endingWith(..)
.hasImage().withTextInRegion(imageRegion).equalsTo(..)
.hasImage().withTextInRegion(imageRegion).matchesRegex(..)
.hasImage().withTextInRegion(imageRegion).startingWith(..)
```

The text recognition function uses the OCR-processor Tesseract.

Example - Validate Text from Images

The following example uses a PDF file which contains an image showing the text of the novel 'Little Lord Fauntleroy'. The image has a slightly coloured background.


Cedric himself knew nothing whatever about it. It had never been even mentioned to him. He knew that his papa had been an Englishman, because his mamma had told him so; but then his papa had died when he was so little a boy that he could not remember very much about him, except that he was big, and had blue eyes and a long mustache, and that it was a splendid thing to be carried around the room on his shoulder.

```
@Test
public void hasImageWithText() throws Exception {
  String filename = "ocr_little-lord-fauntleroy.pdf";
  int leftX  =  10; // millimeter
  int upperY =  35;
  int width  = 160;
  int height = 135;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  String expectedText = "Cedric himself knew nothing whatever about it.";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .withText()
            .containing(expectedText)
  ;
}
```

Normalization of OCR Text

If you look at the image, you can see the line break after the word 'nothing'. Despite this line break, the test is successful, because all whitespaces are eliminated by PDFUnit before comparing the OCR text with the expected text.

Steps in the normalization of OCR Text:

- Characters are converted to lower-case
- All whitespaces are deleted
- 12 different hyphen/dash characters are deleted
- 10 different underscore characters are deleted
- Punctuation characters are deleted

The result of text recognition can be improved by "training" the OCR-processor. Language specific training data can be downloaded from https://github.com/tesseract-ocr/tessdata.

Example - Text in Image Regions

Sometimes an expected text should be located in a certain region of an image. You can define an image region to handle such a requirement:

```
@Test
public void hasImageWithTextInRegion() throws Exception {
  String filename = "ocr_little-lord-fauntleroy.pdf";

  int leftX  =  10; // millimeter
  int upperY =  35;
  int width  = 160;
  int height = 135;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  int imgLeftX  = 250; // pixel
  int imgUpperY =  90;
  int imgWidth  = 130;
  int imgHeight =  30;
  ImageRegion imageRegion = new ImageRegion(imgLeftX, imgUpperY, imgWidth, imgHeight);

  String expectedText = "Englishman";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .withTextInRegion(imageRegion)
            .containing(expectedText)
  ;
}
```

The unit for image size values is always pixel, since images in PDFs may be scaled. This means that using the unit millimeter might lead to incorrect measurements. To find the right values for an image region, extract all images from the PDF and use a simple image processing tool to get the values for the desired region. PDFUnit provides the tool `ExtractImages` to extract images. Chapter 9.7: “Extract Images from PDF” (p. 127) explains how to use it.

Example - Rotated and Flipped Text in Images

Water marks and some other text in images may be intentionally rotated or flipped. Such text can be validated using the following methods:

```
// Method to rotate and flip images before OCR processing:

.hasImage().flipped(FlipDirection).withText()...
.hasImage().rotatedBy(Rotation).withText()...
```

The horribly mangled text in the next image can be validated.

The text in this image is rotated 270 degrees and flipped vertically. If you know these data, the text can be checked:

```
@Test
public void testFlippedAndRotated() throws Exception {
  String filename = "image-with-rotated-and-flipped-text.pdf";
  int leftX  = 80; // in millimeter
  int upperY = 65;
  int width  = 50;
  int height = 75;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  String expectedText = "text rotated 270 and flipped vertically";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasImage()
            .rotatedBy(Rotation.DEGREES_270)
            .flipped(FlipDirection.VERTICAL)
            .withText()
            .equalsTo(expectedText)
  ;
}
```

Allowed values for rotation or flipping are:

```
Rotation.DEGREES_0
Rotation.DEGREES_90
Rotation.DEGREES_180
Rotation.DEGREES_270

FlipDirection.NONE
FlipDirection.HORIZONTAL
FlipDirection.VERTICAL
```

3.32. Text - in Page Regions

Overview

You might find that a certain text exists more than once on a page, but only one of the occurrences has to be tested. This requires you to narrow the search to a section of a page. The syntax is simple:

```
// Reducing the validation to a page region:
.restrictedTo(PageRegion)
// The following validations are limited to the regions.
```

Example

The following example shows the definition and the usage of a page region:

```
@Test
public void hasTextOnFirstPageInPageRegion() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  =  17;  // in millimeter
  int upperY =  45;
  int width  =  60;
  int height =   9;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasText()
            .startingWith("Content")
            .containing("on first")
            .endingWith("page.")
  ;
}
```

When comparing text in page areas, all the test methods are available which are available when comparing text on entire pages. These methods are described in section 13.4: “Comparing Text” (p. 158).

A page region can also be used when testing images.

3.33. Text - Ordered Text

Overview

Sometimes it is necessary to check that certain text in a PDF document comes **before** other text. PDFUnit provides the following methods for such cases:

```
// Testing ordered text:
.hasText().first(text1).then(text2)...
.hasText().first(text1, WhitespaceProcessing).then(text2)...

.hasText().inOrder(text1, text2, ...)
.hasText().inOrder(WhitespaceProcessing, text1, text2, ...)
.hasText().inOrder(WhitespaceProcessing, text[])
```

Example - hasText() ... first() ... then()

The following example checks the existence of three texts in order: first the title of chapter 5, then the body of chapter 5 and lastly the title of chapter 6. Validation is restricted to page 2:

```
@Test
public void hasTextInOrder_FirstThen() throws Exception {
  String filename = "documentUnderTest.pdf";
  String titleChapter5 = "Chapter 5";
  String textChapter5 = "Yours truly, Huck Finn";
  String titleChapter6 = "Chapter 6";
  PagesToUse page2 = PagesToUse.getPage(2);

  int leftX  =  18;  // in millimeter
  int upperY =  80;
  int width  = 180;
  int height = 100;
  PageRegion regionWithoutHeaderAndFooter = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filename)
            .restrictedTo(page2)
            .restrictedTo(regionWithoutHeaderAndFooter)
            .hasText()
            .first(titleChapter5)
            .then(textChapter5)
            .then(titleChapter6)
  ;
}
```

Internally PDFUnit uses the method `containing(..)` to look for the text. Whitespaces are normalized prior to comparison, unless a whitespace-processing parameter is passed to the method `first(expectedText, WhitespaceProcessing)`. All subsequent methods `then()` use the same kind of whitespace processing.

The method `then(..)` can be repeated multiple times.

Example - hasText().inOrder(..)

The next example fulfills the same test objective, but the expected texts are passed to a method as an array:

```
@Test
public void hasTextInOrder_WithStringArray() throws Exception {
  String filename = "documentUnderTest.pdf";

  String titleChapter2 = "Chapter 2";
  String textChapter2 = "Call me Ishmael";
  String titleChapter3 = "Chapter 3";

  AssertThat.document(filename)
            .hasText()
            .inOrder(titleChapter2, textChapter2, titleChapter3)
  ;
}
```

The order of the search string parameters must correspond to the order of the strings in the PDF document.

Again, PDFUnit uses the method `contains(..)` internally and whitespaces will be normalized.

Different whitespace handling can be achieved by giving the desired whitespace-processing as a parameter to the method `inOrder(WhitespaceProcessing, expectedText[])`.

Search strings can also be given to the methods as a string array.

3.34. Text - Right to Left (RTL)

Overview

Tests with RTL-text do not differ from tests with LTR-text, so all methods for comparing text can be used:

```
// Testing page content:
.hasText()                // pages and regions has to be specified before

// Validating expected text:
.hasText().containing(..)
.hasText().containing(.., WhitespaceProcessing)
.hasText().endingWith(..)
.hasText().endingWith(.., WhitespaceProcessing)
.hasText().equalsTo(..)
.hasText().equalsTo(.., WhitespaceProcessing)
.hasText().matchingRegex(..)
.hasText().startingWith(..)

// Prove the absence of defined text:
.hasText().notContaining(..)
.hasText().notContaining(.., WhitespaceProcessing)
.hasText().notEndingWith(..)
.hasText().notMatchingRegex(..)
.hasText().notStartingWith(..)

// Validate multiple text in an expected order:
.hasText().inOrder(..)
.hasText().containingFirst(..).then(..)
```

Example - 'hello, world' from right to left

The next examples use two PDF documents which contain the text 'hello, world' in Arabic and in Hebrew:


```
// Testing RTL text:
@Test
public void hasRTLText_HelloWorld_Arabic() throws Exception {
  String filename = "helloworld_ar.pdf";
  String rtlHelloWorld = "مرحبا، العالم";  // english: 'hello, world!'

  int leftX  = 97;
  int upperY = 69;
  int width  = 69;
  int height = 16;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasText()
            .startingWith(rtlHelloWorld)
  ;
}
```

```
// Testing RTL text:
@Test
public void hasRTLText_HelloWorld_Hebrew() throws Exception {
  String filename = "helloworld_iw.pdf";
  String rtlHelloWorld = "שלום, עולם";   // english: 'hello, world!'

  int leftX  = 97;
  int upperY = 69;
  int width  = 69;
  int height = 16;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasText()
            .endingWith(rtlHelloWorld)
  ;
}
```

It's interesting that the Java-editor in Eclipse can handle text with both text directions. Here is a screen-shot of the Java code from the previous example:

```
@Test
public void hasRTLText_HelloWorld_Arabic() throws Exception {
  String filename = PATH + "textDirection/helloworld_ma.pdf";
  String rtlHelloWorld = "مرحبا, العالم";   // english: 'hello, world!'
```

Internally, PDFUnit uses the PDF-Parser PDFBox. PDFBox parses RTL-text and converts it into a Java string without the need for any special method calls. Congratulations to the development team for such an achievement!

3.35. Text - Rotated and Overhead

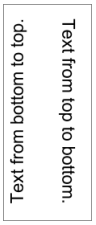
Overview

Some documents have a vertical text in the margin which is needed for identification in subsequent steps of the post-processing. Also, table headers with vertical text are used occasionally.

To test text which is rotated by an arbitrary angle, the same test methods are available as for “normal” text.

Example

The document used by the next example contains two vertical texts:



And this is the test using a page region for the vertical text:

```
// Comparing upright text in a part of a PDF page
@Test
public void hasRotatedTextInRegion() throws Exception {
  String filename = "verticalText.pdf";

  int leftX  =  40;
  int upperY =  45;
  int width  =  45;
  int height = 110;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  String textBottomToTop = "Text from bottom to top.";
  String textTopToBottom = "Text from top to bottom.";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasText()
            .containing(textBottomToTop, WhitespaceProcessing.IGNORE)
            .containing(textTopToBottom, WhitespaceProcessing.IGNORE)
  ;
}
```

The parser extracts vertical text using a lot of line breaks. Therefore, it is absolutely necessarry to use `WhitespaceProcessing.IGNORE`.

Note that the vertical text is still text with the writing direction LTR (left-to-right). PDFUnit also supports RTL (right-to-left) text; see chapter 3.34: “Text - Right to Left (RTL)” (p. 73).

3.36. Version Info

Overview

Sometimes, generated PDF documents need to have a particular version number before they can be processed in a workflow. That can be tested with the following methods:

```
// Simple tests:
.hasVersion().matching(..)

// Tests for version ranges:
.hasVersion().greaterThan(..)
.hasVersion().lessThan(..)
```

One Distinct Version

For popular PDF versions constants can be used:

```
// Constants for PDF versions:

com.pdfunit.Constants.PDFVERSION_11
com.pdfunit.Constants.PDFVERSION_12
com.pdfunit.Constants.PDFVERSION_13
com.pdfunit.Constants.PDFVERSION_14
com.pdfunit.Constants.PDFVERSION_15
com.pdfunit.Constants.PDFVERSION_16
com.pdfunit.Constants.PDFVERSION_17
```

Here an example to check for version “1.4”

```
@Test
public void hasVersion_v14() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasVersion()
            .matching(PDFVERSION_14)
  ;
}
```

Ranges of Versions

The existing constants can be used to verify version ranges:

```
@Test
public void hasVersion_GreaterThanLessThan() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasVersion()
            .greaterThan(PDFVERSION_13) ❶
            .lessThan(PDFVERSION_17)    ❷
  ;
}
```

❶❷ Upper and lower limit values are exclusive.

Also, upcoming versions can be tested. The expected version must be given as a string of the format "%1.1f".

```
@Test
public void hasVersion_LessThanFutureVersion() throws Exception {
  String filename = "documentUnderTest.pdf";
  PDFVersion futureVersion = PDFVersion.withValue("2.0");

  AssertThat.document(filename)
            .hasVersion()
            .lessThan(futureVersion)
  ;
}
```

3.37. XFA Data

Overview

The “XML Forms Architecture, (XFA)” is an extension of the PDF structure using XML information. Its goal is to integrate PDF forms better into workflow processes.

XFA forms are not compatible with “Acro Forms”. Therefore, tests for acroforms cannot be used for XFA data. Tests for XFA data are mainly based on XPath.

```
// Methods around XFA data:
.hasXFAData()
.hasXFAData().matchingXPath(..)
.hasXFAData().withNode(..)

.hasNoXFAData()
```

Existence and Absence of XFA

The first test focuses on the existence of XFA data:

```
@Test
public void hasXFAData() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasXFAData()
  ;
}
```

You can also check that a PDF document does **not contain** XFA data:

```
@Test
public void hasNoXFAData() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNoXFAData()
  ;
}
```

Validate Single XML-Tags

The next examples use the following XFA data (extract):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
  ...
  <x:xmpmeta xmlns:x="adobe:ns:meta/"
             x:xmptk="Adobe XMP Core 4.2.1-c041 52.337767, 2008/04/13-15:41:00"
  >
    <config xmlns="http://www.xfa.org/schema/xci/2.6/">
      ...
      <log xmlns="http://www.xfa.org/schema/xci/2.6/">
        <to>memory</to>
        <mode>overwrite</mode>
      </log>
      ...
    </config>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      ...
      <rdf:Description xmlns:xmp="http://ns.adobe.com/xap/1.0/" >
        <xmp:MetadataDate>2009-12-03T17:50:52Z</xmp:MetadataDate>
      </rdf:Description>
      ...
    </rdf:RDF>
  </x:xmpmeta>
 ...
</xdp:xdp>
```

To test a particular node, an instance of `com.pdfunit.XMLNode` has to be created. The constructor needs an XPath expression and the expected value of the node:

```
@Test
public void hasXFAData_WithNode() throws Exception {
  String filename = "documentUnderTest.pdf";
  XMLNode xmpNode = new XMLNode("xmp:MetadataDate", "2009-12-03T17:50:52Z");  ❶

  AssertThat.document(filename)
            .hasXFAData()
            .withNode(xmpNode)
  ;
}
```

❶ PDFUnit analyzes the XFA data from the current PDF document and determines the namespaces automatically. Only the default namespace has to be specified.

When processing the XPath expression PDFUnit internally adds the path element `"//"` to the given XPath expression. For this reason the expression need not contain the document root `"/"`.

If the XPath expression evaluates to a node set, the first node is used.

If the XMLNode instance contains the expected value, this value will be used to compare it against the actual node value. If the XMLNode instance does not have an expected value, PDFUnit checks only for the existence of the node in the XFA data.

Tests on attribute nodes are of course also possible:

```
@Test
public void hasXFAData_WithNode_NamespaceDD() throws Exception {
  String filename = "documentUnderTest.pdf";
  XMLNode ddNode = new XMLNode("dd:dataDescription/@dd:name", "movie");

  AssertThat.document(filename)
            .hasXFAData()
            .withNode(ddNode)
  ;
}
```

XPath based XFA Tests

To take advantage of the full power of XPath, the method `matchingXPath(..)` is provided. The following two examples help give an idea of what is possible:

```
@Test
public void hasXFAData_MatchingXPath_FunctionStartsWith() throws Exception {
  String filename = "documentUnderTest.pdf";
  String xpathString = "starts-with(//dd:dataDescription/@dd:name, 'mov')";
  XPathExpression expressionWithFunction = new XPathExpression(xpathString);

  AssertThat.document(filename)
            .hasXFAData()
            .matchingXPath(expressionWithFunction)
  ;
}
```

```
@Test
public void hasXFAData_MatchingXPath_FunctionCount_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";

  String xpathProducer = "//pdf:Producer[. ='Adobe LiveCycle Designer ES 8.2']";
  String xpathPI       = "count(//processing-instruction()) = 30";

  XPathExpression exprPI       = new XPathExpression(xpathPI);
  XPathExpression exprProducer = new XPathExpression(xpathProducer);

  AssertThat.document(filename)
            .hasXFAData()
            .matchingXPath(exprProducer)
            .matchingXPath(exprPI)
  ;

  // The same test in a different style:
  AssertThat.document(filename)
            .hasXFAData().matchingXPath(exprProducer)
            .hasXFAData().matchingXPath(exprPI)
  ;
}
```

One limitation has to be mentioned. The evaluation of the XPath expressions depends on the implemented features of the XPath engine you are using. By default PDFUnit uses the JAXP implementation of the your JDK. So, the XPath compatibility also depends on the version of your JDK.

Chapter 13.12: “JAXP-Configuration” (p. 167) explains how to use any XPath-Engine, for example from the Xerces-project.

Default Namespaces in XPath

XML namespaces are detected automatically, but the default namespace has to be declared explicitly using an instance of `DefaultNamespace`. This instance must have a prefix. Any value can be chosen for the prefix:

```
@Test
public void hasXFAData_WithDefaultNamespace_XPathExpression() throws Exception {
  String filename = "documentUnderTest.pdf";

  String namespaceURI  = "http://www.xfa.org/schema/xfa-template/2.6/";
  String xpathSubform  = "count(//default:subform[@name ='movie']//default:field) = 5";

  DefaultNamespace defaultNS   = new DefaultNamespace(namespaceURI);
  XPathExpression exprSubform  = new XPathExpression(xpathSubform, defaultNS);

  AssertThat.document(filename)
            .hasXFAData()
            .matchingXPath(exprSubform)
  ;
}
```

The default namespace must be used not only with the class `XPathExpression`, but also with the class `XMLNode`:

```
/**
 * The default namespace has to be declared,
 * but any alias can be used for it.
 */
@Test
public void hasXFAData_WithDefaultNamespace_XMLNode() throws Exception {
  String filename = "documentUnderTest.pdf";

  String namespaceXCI = "http://www.xfa.org/schema/xci/2.6/";
  DefaultNamespace defaultNS = new DefaultNamespace(namespaceXCI);
  XMLNode aliasFoo = new XMLNode("foo:log/foo:to", "memory", defaultNS);

  AssertThat.document(filename)
            .hasXFAData()
            .withNode(aliasFoo)
  ;
}
```

3.38. XMP Data

Overview

XMP is the abbreviation for “Extensible Metadata Platform”, an open standard initiated by Adobe to embed metadata into files. Not only PDF documents are able to embed data, but also images. For example, metadata can be location and time.

The metadata in a PDF file can be important when processing a document, so they should be correct. PDFUnit provides the same test methods for XMP data as for XFA data:

```
// Methods to test XMP data:
.hasXMPData()
.hasXMPData().matchingXPath(..)
.hasXMPData().withNode(..)

.hasNoXMPData()
```

Existence and Absence of XMP

The following examples show how to verify the existence and absence of XMP data:

```
@Test
public void hasXMPData() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasXMPData()
  ;
}
```

```
@Test
public void hasNoXMPData() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasNoXMPData()
  ;
}
```

Validate Single XML-Tags

Tests can check a single node of the XMP data and its value. The next example is based on the following XML-snippet:

```
<x:xmpmeta xmlns:x="adobe:ns:meta/">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    ...
    <rdf:Description rdf:about="" xmlns:xmp="http://ns.adobe.com/xap/1.0/">
      <xmp:CreateDate>2011-02-08T15:04:19+01:00</xmp:CreateDate>
      <xmp:ModifyDate>2011-02-08T15:04:19+01:00</xmp:ModifyDate>
      <xmp:CreatorTool>My program using iText</xmp:CreatorTool>
    </rdf:Description>
    ...
  </rdf:RDF>
</x:xmpmeta>
```

With the utility `ExtractXMPData` you can extract the XMP data from a PDF document into an XML file. Chapter 9.12: “Extract XMP Data to XML” (p. 132) describes how to use the utility.

In the example the existence of XML-nodes are validated. The method `withNode(..)`needs an instance of `com.pdfunit.XMLNode` as a parameter:

```
@Test
public void hasXMPData_WithNode_ValidateExistence() throws Exception {
  String filename = "documentUnderTest.pdf";
  XMLNode nodeCreateDate = new XMLNode("xmp:CreateDate");
  XMLNode nodeModifyDate = new XMLNode("xmp:ModifyDate");

  AssertThat.document(filename)
            .hasXMPData()
            .withNode(nodeCreateDate)
            .withNode(nodeModifyDate)
  ;
}
```

When you want to verify the value of a node, you also have to pass the expected value to the constructor of `XMLNode`:

```
@Test
public void hasXMPData_WithNodeAndValue() throws Exception {
  String filename = "documentUnderTest.pdf";
  XMLNode nodeCreateDate = new XMLNode("xmp:CreateDate", "2011-02-08T15:04:19+01:00");
  XMLNode nodeModifyDate = new XMLNode("xmp:ModifyDate", "2011-02-08T15:04:19+01:00");

  AssertThat.document(filename)
            .hasXMPData()
            .withNode(nodeCreateDate)
            .withNode(nodeModifyDate)
  ;
}
```

The XPath expression may not start with the document root, because PDFUnit adds `//` internally.

If an expected node exists multiple times within the XMP data, the first match is used.

Of course, the node may also be an attribute node.

XPath based XMP Tests

To take advantage of the full power of XPath, the method `matchingXPath(..)` is provided. The following two examples help give an idea of what is possible:

```
@Test
public void hasXMPData_MatchingXPath() throws Exception {
  String filename = "documentUnderTest.pdf";
  String xpathString = "//xmp:CreateDate[node() = '2011-02-08T15:04:19+01:00']";
  XPathExpression expression = new XPathExpression(xpathString);

  AssertThat.document(filename)
            .hasXMPData()
            .matchingXPath(expression)
  ;
}
```

```
@Test
public void hasXMPData_MatchingXPath_MultipleInvocation() throws Exception {
  String filename = "documentUnderTest.pdf";

  String xpathDateExists = "count(//xmp:CreateDate) = 1";
  String xpathDateValue = "//xmp:CreateDate[node()='2011-02-08T15:04:19+01:00']";

  XPathExpression exprDateExists = new XPathExpression(xpathDateExists);
  XPathExpression exprDateValue = new XPathExpression(xpathDateValue);

  AssertThat.document(filename)
            .hasXMPData()
            .matchingXPath(exprDateValue)
            .matchingXPath(exprDateExists)
  ;

  // The same test in a different style:
  AssertThat.document(filename)
            .hasXMPData().matchingXPath(exprDateValue)
            .hasXMPData().matchingXPath(exprDateExists)
  ;
}
```

The capability to evaluate XPath expressions depends on the XML parser or more exactly the XPath engine. By default PDFUnit uses the parser in the JDK/JRE.

Chapter 13.12: “JAXP-Configuration” (p. 167) explains how to use any other XML-Parser:

Default Namespaces in XPath

XML namespaces are detected automatically, but the default namespace has to be declared explicitly using an instance of `DefaultNamespace`. This instance must have a prefix. Any value can be chosen for the prefix:

```
@Test
public void hasXMPData_MatchingXPath_WithDefaultNamespace() throws Exception {
  String filename = "documentUnderTest.pdf";

  String xpathAsString = "//foo:format = 'application/pdf'";
  String stringDefaultNS = "http://purl.org/dc/elements/1.1/";
  DefaultNamespace defaultNS = new DefaultNamespace(stringDefaultNS);
  XPathExpression expression = new XPathExpression(xpathAsString, defaultNS);

  AssertThat.document(filename)
            .hasXMPData()
            .matchingXPath(expression)
  ;
}
```

The default namespace must be used not only with the class `XPathExpression`, but also with the class `XMLNode`:

```
@Test
public void hasXMPData_WithDefaultNamespace_SpecialNode() throws Exception {
  String filename = "documentUnderTest.pdf";

  String stringDefaultNS = "http://ns.adobe.com/xap/1.0/";
  DefaultNamespace defaultNS = new DefaultNamespace(stringDefaultNS);
  String nodeName  = "foo:ModifyDate";
  String nodeValue = "2011-02-08T15:04:19+01:00";
  XMLNode nodeModifyDate = new XMLNode(nodeName, nodeValue, defaultNS);

  AssertThat.document(filename)
            .hasXMPData()
            .withNode(nodeModifyDate)
  ;
}
```

3.39. ZUGFeRD

Overview

On 25.06.2014, the 'Forum elektronische Rechnung Deutschland' (FeRD), an association of organizations and companies of the industry and the public sector, published version 1.0 of an XML format for electronic invoices. The specification is called ZUGFeRD (Zentraler User Guide des Forums elektronische Rechnung Deutschland). Detailed information is available online at Wikipedia (ZUGFeRD, german) , at 'FeRD' (german) and in a publication of the PDF-Association 'ZUGFeRD 1.0 – English Version'.

Many validation tools check whether the XML data comply with the XML-Schema specification, but do not check whether the invisible XML data are in accordance with the visible data of the printed PDF document. That is easy to do with PDFUnit, if you know the page region of the data to be checked.

PDFUnit provides the following methods for testing ZUGFeRD data:

```
// Methods to test ZUGFeRD data:

.hasZugferdData().matchingXPath(xpathExpression)
.hasZugferdData().withNode(xmlNode)

.compliesWith().zugferdSpecification()
```

The next examples refer to the document 'ZUGFeRD_1p0_BASIC_Einfach.pdf' which is provided together with the specification files of the ZUGFeRD standard version 1.0. Each example shows first the Java code, then the related part of the printed PDF document, and lastly the corresponding XML data.

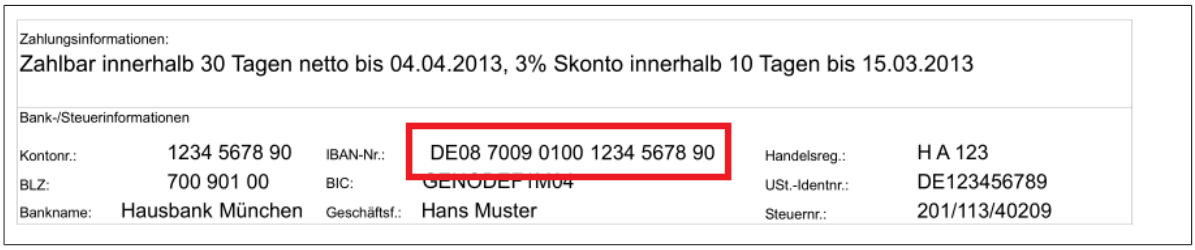
If you want to make the ZUGFeRD data visible, simply open the PDF with Adobe Reader® and save the file 'ZUGFeRD-invoice.xml'.

Validate Content of PDF with Content from ZUGFeRD - IBAN

In this example, an IBAN value is expected both in the XML data and in the PDF text. The test is performed using two `AssertThat` statements. To point the test to the expected regions, a page region (`regionIBAN`) is defined for the visible PDF data and an XML node (`nodeIBAN`) is defined for the ZUGFeRD data.

```
@Test
public void validateIBAN() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";
  String expectedIBAN = "DE87000901001234567890";

  XMLNode nodeIBAN = new XMLNode("ram:IBANID", expectedIBAN);
  PageRegion regionIBAN = createRegionIBAN();

  AssertThat.document(filename)
            .hasZugferdData()
            .withNode(nodeIBAN)
  ;
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(regionIBAN)
            .hasText()
            .containing(expectedIBAN, WhitespaceProcessing.IGNORE)
  ;
}
```

Section of the PDF page:



Section of the ZUGFeRD data:

```
<ram:SpecifiedTradeSettlementPaymentMeans>
    <ram:PayeePartyCreditorFinancialAccount>
        <ram:IBANID>DE08700901001234567890</ram:IBANID>
    </ram:PayeePartyCreditorFinancialAccount>
    <ram:PayeeSpecifiedCreditorFinancialInstitution>
        <ram:BICID>GENODEF1M04</ram:BICID>
    </ram:PayeeSpecifiedCreditorFinancialInstitution>
</ram:SpecifiedTradeSettlementPaymentMeans>
```

Validate Content of PDF with Content from ZUGFeRD - simplified

But the previous test can be simplified by using the method `hasText().containingZugferdData(xmlNode)`. Internally, that method first extracts the text from the ZUGFeRD data and then compares it with the visible text of the given page region. For that comparison, the method `containing()` is used. This means that the ZUGFeRD data must exist somewhere within the given page region. The page region may also contain additional text.

```
@Test
public void validateIBAN_simplified() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";

  XMLNode nodeIBAN = new XMLNode("ram:IBANID");
  PageRegion regionIBAN = createRegionIBAN();

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(regionIBAN)
            .hasText()
            .containingZugferdData(nodeIBAN)
  ;
}
```

Important: The whitespaces are removed before the two values are compared. This is necessary because line breaks and formatting spaces have different meanings in XML and PDF.

Validate Content of PDF with Content from ZUGFeRD - Billing Address

The simplified check in the previous example works fine if the XML data represent the visible data. But in the following example the ZIP code in the ZUGFeRD data is separated from the city name. So two invocations of `AssertThat` are needed to validate the complete address.

The XPath expression uses the XPath function `contains()` because the expected value is a part of the complete node value. The node value ends with 'DE' which is not part of the visible text.

```
@Test
public void validatePostalTradeAddress() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";
  String expectedAddressPDF = "Hans Muster "
                            + "Kundenstraße 15 "
                            + "69876 Frankfurt";
  String expectedAddressXML = "Hans Muster "
                            + "Kundenstraße 15 "
                            + "Frankfurt";
  String addressXMLNormalized = WhitespaceProcessing.NORMALIZE.process(expectedAddressXML);
  String xpathWithPlaceholder =
        "ram:BuyerTradeParty/ram:PostalTradeAddress[contains(normalize-space(.), '%s')]";
  String xpathPostalTradeAddress = String.format(xpathWithPlaceholder, addressXMLNormalized);

  XMLNode nodePostalTradeAddress = new XMLNode(xpathPostalTradeAddress);
  PageRegion regionPostalTradeAddress = createRegionPostalAddress();

  AssertThat.document(filename)
            .hasZugferdData()
            .withNode(nodePostalTradeAddress)
  ;
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(regionPostalTradeAddress)
            .hasText()
            .containing(expectedAddressPDF)
  ;
}
```

The whitespaces in the PDF document differ from those in the XML data. Therefore, the XPath function `normalize-space()` is used.

Section of the PDF page:

Section of the ZUGFeRD data:

```
<ram:BuyerTradeParty>
  <ram:Name>Kunden AG Mitte</ram:Name>
  <ram:PostalTradeAddress>
    <ram:PostcodeCode>69876</ram:PostcodeCode>
    <ram:LineOne>Hans Muster</ram:LineOne>
    <ram:LineTwo>Kundenstraße 15</ram:LineTwo>
    <ram:CityName>Frankfurt</ram:CityName>
    <ram:CountryID>DE</ram:CountryID>
  </ram:PostalTradeAddress>
</ram:BuyerTradeParty>
```

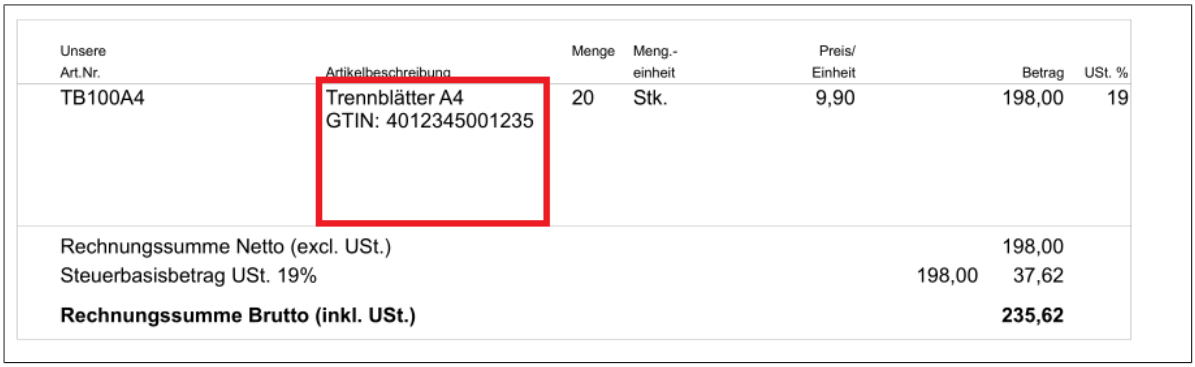
Validate Content of PDF with Content from ZUGFeRD - Product

But text in a PDF file does not always correspond with the text of a node in the ZUGFeRD data. That's why in the next example the string 'Trennblätter A4 GTIN: 4012345001235' cannot be validated. Only

the String 'GTIN: 4012345001235' can be validated. The XPath expression thus needs to use the function `contains()` and the PDFUnit method needs to use `hasText().containing()`.

```
@Test
public void validateTradeProduct() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";
  String expectedTradeProduct = "GTIN: 4012345001235";
  String xpathWithPlaceholder =
         "ram:SpecifiedTradeProduct/ram:Name[contains(., '%s')]";
  String xpathTradeProduct = String.format(xpathWithPlaceholder, expectedTradeProduct);

  XMLNode nodeTradeProduct = new XMLNode(xpathTradeProduct);
  PageRegion regionTradeProduct = createRegionTradeProduct();

  AssertThat.document(filename)
            .hasZugferdData()
            .withNode(nodeTradeProduct)
  ;
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(regionTradeProduct)
            .hasText()
            .containing(expectedTradeProduct)
  ;
}
```

Section of the PDF page:



Unsere Art.Nr.	Artikelbeschreibung	Menge	Meng.-einheit	Preis/ Einheit	Betrag	USt. %
TB100A4	Trennblätter A4 GTIN: 4012345001235	20	Stk.	9,90	198,00	19
Rechnungssumme Netto (excl. USt.)					198,00	
Steuerbasisbetrag USt. 19%				198,00	37,62	
Rechnungssumme Brutto (inkl. USt.)					**235,62**	

Section of the ZUGFeRD data:

```
<ram:SpecifiedTradeProduct>
    <ram:GlobalID schemeID="0160">4012345001235</ram:GlobalID>
    <ram:SellerAssignedID>TB100A4</ram:SellerAssignedID>
    <ram:Name>GTIN: 4012345001235
      Unsere Art.-Nr.: TB100A4
      Trennblätter A4
    </ram:Name>
</ram:SpecifiedTradeProduct>
```

Complex Validations of ZUGFeRD Data

For more complex validations, PDFUnit provides the method `matchingXPath(..)`. This method makes it possible to use the full power of XPath in combination with ZUGFeRD data.

The next example checks that the number of traded articles is exactly '1'.

```
@Test
public void hasZugferdDataMatchingXPath() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";
  String xpathNumberOfTradeItems = "count(//ram:IncludedSupplyChainTradeLineItem) = 1";
  XPathExpression exprNumberOfTradeItems = new XPathExpression(xpathNumberOfTradeItems);
  AssertThat.document(filename)
            .hasZugferdData()
            .matchingXPath(exprNumberOfTradeItems)
  ;
}
```

Section of the ZUGFeRD data:

```
<ram:IncludedSupplyChainTradeLineItem>
    <ram:AssociatedDocumentLineDocument />
    <ram:SpecifiedSupplyChainTradeDelivery>
      <ram:BilledQuantity unitCode="C62">20.0000</ram:BilledQuantity>
    </ram:SpecifiedSupplyChainTradeDelivery>
    <ram:SpecifiedSupplyChainTradeSettlement>
      <ram:SpecifiedTradeSettlementMonetarySummation>
        <ram:LineTotalAmount currencyID="EUR">198.00</ram:LineTotalAmount>
      </ram:SpecifiedTradeSettlementMonetarySummation>
    </ram:SpecifiedSupplyChainTradeSettlement>
    <ram:SpecifiedTradeProduct>
      <ram:GlobalID schemeID="0160">4012345001235</ram:GlobalID>
      <ram:SellerAssignedID>TB100A4</ram:SellerAssignedID>
      <ram:Name>GTIN: 4012345001235
        Unsere Art.-Nr.: TB100A4
        Trennblätter A4
      </ram:Name>
    </ram:SpecifiedTradeProduct>
</ram:IncludedSupplyChainTradeLineItem>
```

It is even more challenging to check that the summed up prices of all articles are equal to the given sum which is stored separately in the ZUGFeRD data. But with a little XPath the test can be performed as follows:

```
@Test
public void hasZugferdData_TotalAmountWithoutTax() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";
  String xpTotalAmount = "sum(//ram:IncludedSupplyChainTradeLineItem//ram:LineTotalAmount)"
                         + " = "
                         + "sum(//ram:TaxBasisTotalAmount)";
  XPathExpression exprTotalAmount = new XPathExpression(xpTotalAmount);

  AssertThat.document(filename)
            .hasZugferdData()
            .matchingXPath(exprTotalAmount)
  ;
}
```

You can develop such a test only if you have direct access to the XML data. The ZUGFeRD data can either be exported from the PDF by using Adobe Reader® (right mouse button) or extracted using the utility program `ExtractZugferdData` in PDFUnit. This utility is described in chapter 9.15: “Extract ZUGFeRD Data” (p. 136).

ZUGFeRD Data Validation against Specification

Lastly, it should be mentioned that PDFUnit can also be used to check the compliance of your ZUGFeRD data with the XML-Schema specification.

```
@Test
public void compliesWithZugferdSpecification() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";
  AssertThat.document(filename)
            .compliesWith()
            .zugferdSpecification(ZugferdVersion.VERSION10)
  ;
}
```

Chapter 4. Comparing a Test PDF with a Reference

4.1. Overview

Many tests follow the principle of comparing a newly created test document with a PDF document which has already been validated. Such tests are useful if the process that creates the PDF has to be changed, but the output should be the same.

Initialization

The instantiation of a reference document is done with the method `and(..)`:

```
@Test
public void testInstantiationWithReference() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";
  String passwordTest = "owner-password";

  AssertThat.document(filenameTest, passwordTest) ❶
            .and(filenameReference)               ❷
  ;
}
```

❶ If the test document is password protected, the second parameter is needed.

❷ If the referenced document is not password protected, only the file name is needed. Otherwise a password has be used passed to the method.

Passwords are “only” used to open the documents. They do not influence the tests.

Overview

The following list gives a complete overview of all tests which compare two PDF files. Links after each tag refer to the chapter which describes it in detail.

```
// Methods to compare two PDF documents:

.haveSameAccessPermission()          4.12: “Comparing Permission” (p. 95)
.haveSameAccessPermission(..)        4.12: “Comparing Permission” (p. 95)
.haveSameAppearance()                4.10: “Comparing Layout as Rendered Pages” (p. 93)
.haveSameAuthor()                    4.5: “Comparing Document Properties” (p. 89)
.haveSameBookmarks()                 4.3: “Comparing Bookmarks” (p. 88)
.haveSameCreationDate()              4.4: “Comparing Date Values” (p. 89)
.haveSameCreator()                   4.5: “Comparing Document Properties” (p. 89)
.haveSameEmbeddedFiles(..)           4.2: “Comparing Attachments” (p. 88)
.haveSameFieldsByName()              4.7: “Comparing Form Fields” (p. 91)
.haveSameFieldsByValue()             4.7: “Comparing Form Fields” (p. 91)
.haveSameFormat()                    4.6: “Comparing Format” (p. 90)
.haveSameImages()                    4.8: “Comparing Images” (p. 92)
.haveSameJavaScript()                4.17: “More Comparisons” (p. 99)
.haveSameKeywords()                  4.17: “More Comparisons” (p. 99)
.haveSameLanguageInfo()              4.17: “More Comparisons” (p. 99)
.haveSameLayerNames()                4.17: “More Comparisons” (p. 99)
.haveSameModificationDate()          4.4: “Comparing Date Values” (p. 89)
.haveSameNamedDestinations()         4.11: “Comparing Named Destinations” (p. 95)
.haveSameNumberOfBookmarks()         4.3: “Comparing Bookmarks” (p. 88)
.haveSameNumberOfEmbeddedFiles()     4.2: “Comparing Attachments” (p. 88)
.haveSameNumberOfFields()            4.7: “Comparing Form Fields” (p. 91)
.haveSameNumberOfImages()            4.8: “Comparing Images” (p. 92)
.haveSameNumberOfLayers()            4.13: “Comparing Quantities of PDF Elements” (p. 96)
.haveSameNumberOfNamedDestinations() 4.11: “Comparing Named Destinations” (p. 95)
.haveSameNumberOfPages()             4.13: “Comparing Quantities of PDF Elements” (p. 96)

...  continued
```

```
... continuation:

.haveSameProducer()                 4.5: “Comparing Document Properties” (p. 89)
.haveSameProperties()               4.5: “Comparing Document Properties” (p. 89)
.haveSameProperty(..)               4.5: “Comparing Document Properties” (p. 89)
.haveSameSubject()                  4.5: “Comparing Document Properties” (p. 89)
.haveSameTaggingInfo()              4.17: “More Comparisons” (p. 99)
.haveSameText()                     4.14: “Comparing Text” (p. 97)
.haveSameTitle()                    4.5: “Comparing Document Properties” (p. 89)
.haveSameXFAData()                  4.15: “Comparing XFA Data” (p. 98)
.haveSameXMPData()                  4.16: “Comparing XMP Data” (p. 99)
```

4.2. Comparing Attachments

Quantity

Compare the number of attachments in two documents:

```
@Test
public void haveSameNumberOfEmbeddedFiles() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfEmbeddedFiles()
  ;
}
```

Name and Content

There is a parameterized test method to compare the attachments by name or by content:

```
@Test
public void haveSameEmbeddedFiles() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameEmbeddedFiles(COMPARED_BY_NAME)
            .haveSameEmbeddedFiles(COMPARED_BY_CONTENT)  ❶
  ;
}
```

❶ The attachments are compared byte-by-byte. So, two files of any type can be compared.

The two constants are defined in `com.pdfunit.Constants`:

```
// Constants defining the kind comparing embedded files:

com.pdfunit.Constants.COMPARED_BY_CONTENT
com.pdfunit.Constants.COMPARED_BY_NAME
```

You can use the utility `ExtractEmbeddedFiles` to extract the attachments. See chapter 9.4: “Extract Attachments” (p. 123).

4.3. Comparing Bookmarks

Quantity

The simplest comparison related to bookmarks is to compare the number of bookmarks in two documents:

```
@Test
public void haveSameNumberOfBookmarks() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfBookmarks()
  ;
}
```

Bookmarks with Properties

Next, the bookmarks and their properties are compared. Bookmarks of two PDF documents are 'equal' if the following attributes have the same values:

- label (title)
- destination (URI)
- destination (related page)
- destination (link name)

```
@Test
public void haveSameBookmarks() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameBookmarks()
  ;
}
```

If you are uncertain about the bookmarks, all bookmark data can be extracted into an XML file using the utility `ExtractBookmarks`. This file can easily be analyzed. See chapter 9.5: “Extract Bookmarks to XML” (p. 125).

4.4. Comparing Date Values

It rarely makes sense to compare date values of two PDF documents, but if it is really needed, you can use the following methods:

```
// Methods comparing dates:

.haveSameCreationDate()
.haveSameModificationDate()
```

In the next example, the modification dates of two documents are compared:

```
@Test
public void haveSameModificationDate() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameModificationDate()
  ;
}
```

Comparing two dates is always carried out with the time resolution `DateResolution.DATE`.

4.5. Comparing Document Properties

You might want to compare the title or other document information of two PDF documents. Use the following methods:

```
// Comparing document properties:

.haveSameAuthor()
.haveSameCreationDate()
.haveSameCreator()
.haveSameKeywords()
.haveSameLanguageInfo()
.haveSameModificationDate()
.haveSameProducer()
.haveSameProperties()
.haveSameProperty(String)
.haveSameSubject()
.haveSameTitle()
```

As an example of comparing any document property we compare the “author”:

```
@Test
public void haveSameAuthor() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAuthor()
  ;
}
```

Custom properties can be compared using the method `haveSameProperty(..)`:

```
@Test
public void haveSameCustomProperty() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameProperty("Company")
            .haveSameProperty("SourceModified")
  ;
}
```

Of course, you can use this method to compare all standard properties.

If you want to compare **all** properties of two documents, you can use the general method `haveSameProperties()`:

```
@Test
public void haveSameProperties_AllProperties() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameProperties()
  ;
}
```

4.6. Comparing Format

Two documents have the same page format if width and height of all pages have the same values. The tolerance defined by DIN 476 is taken into account:

```
@Test
public void haveSameFormat() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameFormat()
  ;
}
```

The comparison can be restricted to selected pages:

```
@Test
public void haveSameFormat_OnPage2() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";
  PagesToUse page2 = PagesToUse.getPage(2);

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(page2)
            .haveSameFormat()
  ;
}
```

```
@Test
public void haveSameFormat_OnEveryPageAfter() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";
  PagesToUse pagesBefore2 = ON_EVERY_PAGE.before(2);

  AssertThat.document(filename)
            .and(filenameReference)
            .haveSameFormat(pagesBefore2)
  ;
}
```

All possibilities to select pages are explained in chapter 13.2: “Page Selection” (p. 155).

4.7. Comparing Form Fields

Quantity

The first and simplest test is to check that a document has the same number of form fields as the referenced document:

```
@Test
public void haveSameNumberOfFields() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfFields()
  ;
}
```

Field Names

The next test checks that the number of fields and their names are equal in both PDF documents:

```
@Test
public void haveSameFields_ByName() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameFieldsByName()
  ;
}
```

Field Content

And finally, you can compare the contents in form fields in two documents using the method `haveSameFieldsByValue()`. Fields with the same name must have the same contents:

```
@Test
public void haveSameFields_ByValue() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameFieldsByValue() ❶
  ;
}
```

❶ Whitespaces are “normalized”, see chapter 13.5: “Whitespace Processing” (p. 159).

Concatenated Tests

Multiple and different test methods can be concatenated in one test, but such a test is not recommended because it's hard to find a good name:

```
@Test
public void compareManyItems() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAppearance()
            .haveSameText
  ;
}
```

4.8. Comparing Images

Quantity

The first test compares the number of images in two PDF documents:

```
@Test
public void haveSameNumberOfImages() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfImages()
  ;
}
```

The comparison can be limited to selected pages:

```
@Test
public void haveSameNumberOfImages_OnPage2() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";
  PagesToUse page2 = PagesToUse.getPage(2);

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(page2)
            .haveSameNumberOfImages()
  ;
}
```

All possibilities to select pages are described in chapter 13.2: “Page Selection” (p. 155).

Content of Images

The images stored in a test PDF can be compared with those of a reference PDF. They are identified as equal when they are equal byte-by-byte.

```
/**
 * The method haveSameImages() does not consider the order of the images.
 */
@Test
public void haveSameImages() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameImages()
  ;
}
```

This test method does not care about which pages the images appear on or how often they are used.

You can restrict the comparison of reference to individual pages:

```
@Test
public void haveSameImages_OnPage2() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";
  PagesToUse page2 = PagesToUse.getPage(2);

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(page2)
            .haveSameImages()            ❶
  ;
}
```

```
@Test
public void haveSameImages_BeforePage2() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";
  PagesToUse pagesBefore2 = ON_EVERY_PAGE.before(2);

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(pagesBefore2)
            .haveSameImages()            ❷
  ;
}
```

❶❷ The order of images is irrelevant for the comparison.

If there are any doubts about the images in a PDF document all images can be extracted using the utility `ExtractImages`. See chapter 9.7: “Extract Images from PDF” (p. 127).

4.9. Comparing JavaScript

Two PDF documents can have the “same” JavaScript. The comparison is done byte-wise using the method `haveSameJavaScript()`. Whitespaces are ignored.

```
@Test
public void haveSameJavaScript() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameJavaScript()
  ;
}
```

If you want to see the JavaScript code, you can extract it with the utility `ExtractJavaScript` as described in chapter 9.8: “Extract JavaScript to a Text File” (p. 128).

4.10. Comparing Layout as Rendered Pages

PDF documents can be compared as rendered images (PNG) to ensure that a test document and a referenced document have the same layout.

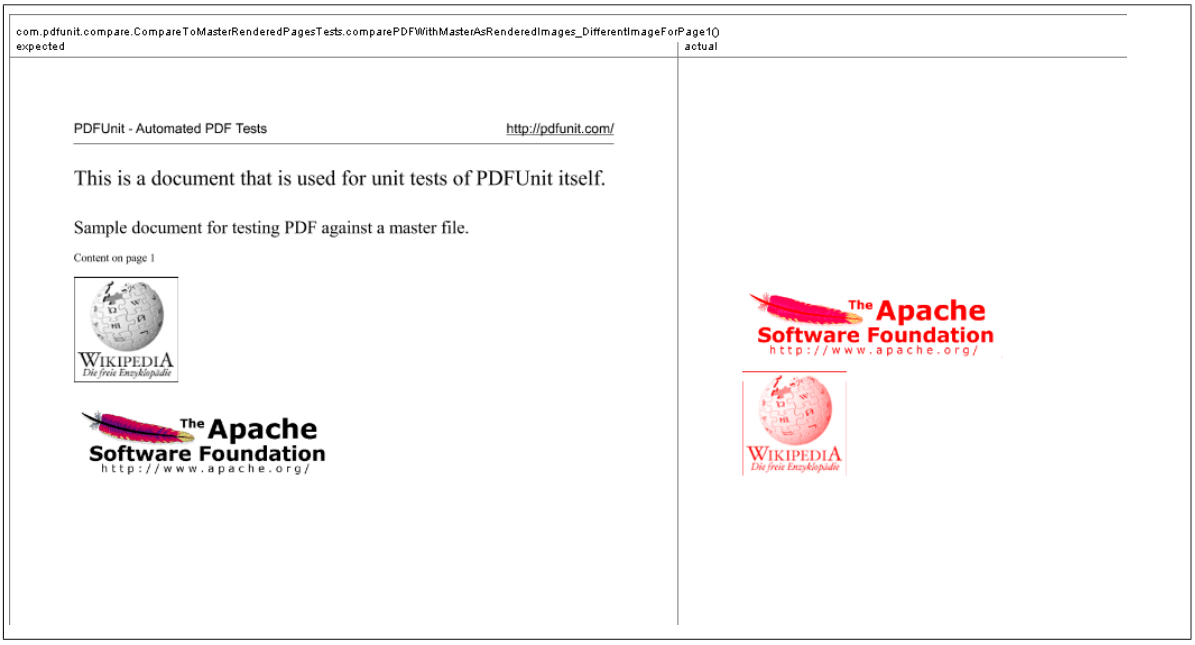
```
@Test
public void haveSameAppearanceCompleteDocument() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(EVERY_PAGE)
            .haveSameAppearance()
  ;
}
```

You can select individual pages in many ways. All possibilities are described in chapter 13.2: “Page Selection” (p. 155).

You can compare the layout of entire pages or you can restrict the comparison to sections of a page:

```
@Test
public void haveSameAppearanceInPageRegion() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  int leftX  =   0;
  int upperY =   0;
  int width  = 210;
  int height =  50;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filenameTest)
            .and(filenameMaster)
            .restrictedTo(EVERY_PAGE)
            .restrictedTo(pageRegion)
            .haveSameAppearance()
  ;
}
```

In case of a test error, PDFUnit creates a **diff image**.



The diff image contains the name of the test in the header. And the name of the diff image is shown in the error message. That allows a cross reference between test and diff image.

Type
'C:\daten\p...aster\compareToMaster_sameImagesDifferentOrder.pdf' differs to 'C:\daten\p...ents\pdf\used-for-tests\master\compareToMaster.pdf' as rendered page for page 1. Reason: See report-image 'C:\daten\p...ents\pdf\used-for-tests\master\compareToMaster_sameImagesDifferentOrder.pdf.20140526-203522929.out.png'.

The name of the diff-image file has following parts:

- The full name of the test file.
- A formatted date in the format 'yyyyMMdd-HHmmssSSS'.
- The last part of the file name is the string '.out.png'.

The folder for the diff-images can be defined using the key `output.path.diffimages` in the config file `pdfunit.config`.

If you need to analyze the layout of two PDF documents after a test fails, you can use the very powerful program `DiffPDF`. Information about this Open-Source application by Mark Summerfield is available on the project site http://soft.rubypdf.com/software/diffpdf. You can install the program under Linux with a package manager, for example `apt-get install diffpdf`. On Windows you can use it as a 'portable application' available from http://portableapps.com/apps/utilities/diffpdf_portable.

4.11. Comparing Named Destinations

“Named Destinations” are seldom a test goal, because until now no test tool is been available which could compare named destinations. With PDFUnit you can verify that two documents have the same named destinations.

Quantity

A simple test is to compare the number of “Named Destinations” in two documents:

```
@Test
public void compareNumberOfNamedDestinations() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfNamedDestinations()
  ;
}
```

Names and Internal Position

If the names of 'Named Destinations' have to be equal for two documents, the following test can be used:

```
@Test
public void compareNamedDestinations() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNamedDestinations()
  ;
}
```

4.12. Comparing Permission

PDFUnit can compare the access permission of two PDF documents. The following example compares **all permission**:

```
@Test
public void compareAllPermission() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAccessPermission()
  ;
}
```

If you want to compare a **single permission** you can parameterize the test method with predefined constants:

```
@Test
public void haveSamePermission_MultipleInvocation() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAccessPermission(COPY)
            .haveSameAccessPermission(EXTRACT_CONTENT)
            .haveSameAccessPermission(MODIFY_CONTENT)
  ;
}
```

The following constants are available:

```
// Available permissions:

com.pdfunit.Constants.ASSEMBLE_DOCUMENTS
com.pdfunit.Constants.EXTRACT_CONTENT
com.pdfunit.Constants.FILL_IN
com.pdfunit.Constants.MODIFY_ANNOTATIONS
com.pdfunit.Constants.MODIFY_CONTENT
com.pdfunit.Constants.PRINT_IN_HIGHQUALITY
com.pdfunit.Constants.PRINT_DEGRADED_ONLY
com.pdfunit.Constants.ALLOW_SCREENREADERS
```

4.13. Comparing Quantities of PDF Elements

The number of various items in a test document can be compared with the number of the same items in a referenced document.

Even if some of these tests are already described in other chapters, the following list gives an overview of all comparing methods for countable components:

```
// Overview of counting the number of parts of a PDF document:

.haveSameNumberOfBookmarks()
.haveSameNumberOfEmbeddedFiles()
.haveSameNumberOfFields()
.haveSameNumberOfImages()
.haveSameNumberOfLayers()
.haveSameNumberOfNamedDestinations()
.haveSameNumberOfPages()
.haveSameNumberOfTaggingInfo()
```

Here are some examples which are not shown in other chapters:

```
@Test
public void haveSameNumberOfPages() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfPages()
  ;
}
```

```
@Test
public void haveSameNumberOfLayers() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfLayers()
  ;
}
```

4.14. Comparing Text

PDFUnit can compare text on any page of a test document with the corresponding page of a referenced document. The following simple example compares the first and last page of two documents. Please note that whitespaces are normalized:

```
@Test
public void haveSameText_OnSinglePage() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(FIRST_PAGE)
            .haveSameText()
  ;

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(LAST_PAGE)
            .haveSameText()
  ;
}
```

You can restrict the test to selected pages which is explained in chapter 13.2: “Page Selection” (p. 155):

And you can restrict the comparison to a section of a page:

```
@Test
public void haveSameText_InPageRegion() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  int leftX   =  17;
  int upperY  = 254;
  int width   =  53;
  int height  =  11;
  PageRegion region = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(EVERY_PAGE)
            .restrictedTo(region)
            .haveSameText()
  ;
}
```

The treatment of white space can be controlled in the same kind as in other tests:

```
@Test
public void haveSameText_IgnoreWhitespace() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  int leftX   =  17;
  int upperY  = 254;
  int width   =  53;
  int height  =  11;
  PageRegion region = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(region)
            .haveSameText(WhitespaceProcessing.IGNORE)
  ;
}
```

4.15. Comparing XFA Data

It does not make sense to compare the entire XFA data of two PDF documents, because often they contain a creation date and a modification data. For this reason PDFUnit provides an XPath based test method for XFA data.

Overview

Only one powerful method is provided:

```
// Method for tests with XMP data

.haveSameXFAData().matchingXPath(XPathExpression)
```

If you are in doubt about the XFA data, you can extract it with the utility `ExtractXFAData` into an XML file. For more information see chapter 9.11: “Extract XFA Data to XML” (p. 131).

Example with multiple XML nodes in the XFA data

In the following example, two nodes of the XFA data defined by XPath expressions will be compared. The result of the XPath expression has to be the same for both PDF documents.

```
@Test
public void haveSameXFAData_MultipleDefaultNamespaces() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  String nsStringXFATemplate = "http://www.xfa.org/schema/xfa-template/2.6/";
  String nsStringXFALocale   = "http://www.xfa.org/schema/xfa-locale-set/2.7/";

  DefaultNamespace nsXFATemplate = new DefaultNamespace(nsStringXFATemplate);
  DefaultNamespace nsXFALocale   = new DefaultNamespace(nsStringXFALocale);

  String xpathSubform = "//default:subform/@name[.='movie']";
  String xpathLocale  = "//default:locale/@name[.='nl_BE']";

  XPathExpression exprXFATemplate = new XPathExpression(xpathSubform, nsXFATemplate);
  XPathExpression exprXFALocale   = new XPathExpression(xpathLocale, nsXFALocale);

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameXFAData()
            .matchingXPath(exprXFATemplate)
            .matchingXPath(exprXFALocale)
  ;
}
```

Be careful using the default namespace. It has to be declared in the attribute `<haveSameXFAData />`. Other namespaces will be detected automatically.

You can use the method `matchingXPath(..)` more than once in a test. But it would be better to split the test.

PDFUnit throws an exception if two documents without any XFA data are compared. It makes no sense to compare things that don't exist. A detailed description of how to work with XPath in PDFUnit can be found in chapter 13.11: “Using XPath” (p. 165).

Example - XPath Expression with XPath Function

The XPath-expressions may contain XPath functions:

```
@Test
public void haveSameXFAData() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  DefaultNamespace defaultNS
                = new DefaultNamespace("http://www.xfa.org/schema/xfa-template/2.6/");
  XPathExpression  expression
                = new XPathExpression("count(//default:field) = 3", defaultNS);

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameXFAData()
            .matchingXPath(expression)
  ;
}
```

4.16. Comparing XMP Data

You can compare the XMP data in two PDF documents using XPath. Comparing XFA and XMP data are basically the same. And because the previous chapter 3.37: “XFA Data” (p. 76) describes the tests in detail, this section is very short.

If you are in doubt about the structure and values of the XMP data you can extract them with the program `ExtractXMPData`. See chapter 9.12: “Extract XMP Data to XML” (p. 132).

Overview

PDFUnit provides the following method:

```
// Method for tests with XMP data:

.haveSameXMPData().matchingXPath(XPathExpression)
```

Example

```
@Test
public void haveSameXMPData() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";
  XPathExpression expression = new XPathExpression("//pdf:Producer");

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameXMPData()
            .matchingXPath(expression)
  ;
}
```

If the XMP data is compared in two documents, but neither contains XMP data, PDFUnit throws an exception.

4.17. More Comparisons

In the previous chapters a lot of examples show how to compare various parts of two PDF documents. But PDFUnit provides more methods for comparing PDF. The following list shows the remaining methods. They should be self explanatory:

```
// Various methods, comparing PDF. Not described before:

.haveSameJavaScript()
.haveSameKeywords()
.haveSameLanguageInfo()
.haveSameLayerNames()
.haveSameTaggingInfo()
```

Concatenation of Test Methods

All methods can be concatenated:

```
@Test
public void haveSameAuthorTitle() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAuthor()
            .haveSameTitle()
  ;
}
```

This example is only a syntax demo. In real projects you should separate the test into two tests, each with a meaningful name.

Chapter 5. Folders and Multiple Documents

5.1. Overview

For such tests almost all test methods are available, which exist for tests with a single PDF document. The following list shows the available methods. Links refer to the description of each test.

```
// Methods to validate a set of PDF documents:

.compliesWith()
   .constraints(..)              3.10: “Excel Files for Validation Constraints” (p. 28)
   .din5008FormA()               3.8: “DIN 5008” (p. 24)
   .din5008FormB()               3.8: “DIN 5008” (p. 24)
   .pdfStandard()                3.25: “PDF/A” (p. 54)
   .zugferdSpecification(..)     3.39: “ZUGFeRD” (p. 82)

.containsOneImageOf(..)          3.16: “Images in PDF Documents” (p. 40)
.hasAuthor()                     3.9: “Document Properties” (p. 25)
.hasBookmark()                   3.5: “Bookmarks and Named Destinations” (p. 18)
.hasBookmarks()                  3.5: “Bookmarks and Named Destinations” (p. 18)
.hasEncryptionLength(..)         3.24: “Passwords” (p. 53)
.hasField(..)                    3.13: “Form Fields” (p. 32)
.hasFields()                     3.13: “Form Fields” (p. 32)
.hasFont()                       3.12: “Fonts” (p. 29)
.hasFonts()                      3.12: “Fonts” (p. 29)
.hasFormat(..)                   3.15: “Format” (p. 39)
.hasImage()                      3.16: “Images in PDF Documents” (p. 40)
   .withBarcode()                3.4: “Bar Code” (p. 16)
   .withQRcode()                 3.27: “QR Code” (p. 56)
.hasJavaScript()                 3.17: “JavaScript” (p. 43)
.hasKeywords()                   3.9: “Document Properties” (p. 25)
.hasLanguageInfo(..)             3.18: “Language” (p. 45)
.hasNoAuthor()                   3.9: “Document Properties” (p. 25)
.hasNoImage()                    3.16: “Images in PDF Documents” (p. 40)
.hasNoKeywords()                 3.9: “Document Properties” (p. 25)
.hasNoLanguageInfo()             3.18: “Language” (p. 45)
.hasNoProperty()                 3.9: “Document Properties” (p. 25)
.hasNoSubject()                  3.9: “Document Properties” (p. 25)
.hasNoText()                     3.30: “Text” (p. 63)
.hasNoTitle()                    3.9: “Document Properties” (p. 25)
.hasNoXFAData()                  3.37: “XFA Data” (p. 76)
.hasNoXMPData()                  3.38: “XMP Data” (p. 79)

.hasNumberOf...()                3.22: “Number of PDF Elements” (p. 51)

.hasOwnerPassword(..)            3.24: “Passwords” (p. 53)
.hasPermission()                 3.26: “Permissions” (p. 55)
.hasProperty(..)                 3.9: “Document Properties” (p. 25)
.hasSignatureField(..)           3.28: “Signed PDF” (p. 59)
.hasSignatureFields()            3.28: “Signed PDF” (p. 59)
.hasSubject()                    3.9: “Document Properties” (p. 25)
.hasText(..)                     3.30: “Text” (p. 63)
.hasTitle()                      3.9: “Document Properties” (p. 25)
.hasUserPassword(..)             3.24: “Passwords” (p. 53)
.hasVersion()                    3.36: “Version Info” (p. 75)
.hasXFAData()                    3.37: “XFA Data” (p. 76)
.hasXMPData()                    3.38: “XMP Data” (p. 79)
.hasZugferdData()                3.39: “ZUGFeRD” (p. 82)
.isCertified()                   3.6: “Certified PDF” (p. 21)
.isCertifiedFor(..)              3.6: “Certified PDF” (p. 21)
.isLinearizedForFastWebView()    3.11: “Fast Web View” (p. 29)
.isSigned()                      3.28: “Signed PDF” (p. 59)
.isSignedBy(..)                  3.28: “Signed PDF” (p. 59)
.isTagged()                      3.29: “Tagged Documents” (p. 62)

.passedFilter(..)                5.3: “Validate all documents in a folder” (p. 102)
```

A test with multiple documents or folders stops at the first detected error.

The next two chapters show tests using multiple documents and a folder.

5.2. Test Multiple PDF Documents

The following code shows how to use multiple documents in one test.

```
@Test
public void textInMultipleDocuments() throws Exception {
  String fileName1 = "document_en.pdf";
  String fileName2 = "document_es.pdf";
  String fileName3 = "document_de.pdf";
  File file1 = new File(fileName1);
  File file2 = new File(fileName2);
  File file3 = new File(fileName3);
  File[] files = {file1, file2, file3};

  String expectedDate = "28.09.2014";
  String expectedDocumentID = "XX-123";

  AssertThat.eachDocument(files)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(expectedDate)
            .containing(expectedDocumentID)
  ;
}
```

The PDF-documents are passed to the method `eachDocument()` as a `String[]`. The types `File[]`, `InputStream[]`, and `URL[]` can also be used.

5.3. Validate all documents in a folder

All tests on file folders start with the following methods:

```
File folder = new File(folderName);
AssertThat.eachDocument()                    ❶
          .inFolder(folder)                  ❷
          .passedFilter(filter)              ❸
          ...
;
```

❶❷ These two methods are the entry into any test which uses all files in the given folder that have the '.pdf' file type extension. The extension comparison is case-insensitive.

❸ An optional filter can be used to reduce the number of documents, and multiple filters can be concatenated. Without a filter, all PDF documents in the folder will be checked.

Once one document in the folder fails a test, the test ends for all documents in that folder.

If the used filters filter out every document, an error message is thrown. An error message is also thrown if a folder does not contain any PDF documents.

Example - Validate all Documents in a Folder

The following example checks whether all PDF documents in the given folder comply with the ZUGFeRD specification version 1.0:

```
@Test
public void filesInFolderCompliesZugferdSpecification() throws Exception {
  String folderName = PATH + "zugferd10";
  File folder = new File(folderName);
  AssertThat.eachDocument()
            .inFolder(folder)
            .compliesWith()
            .zugferdSpecification(VERSION10)
  ;
}
```

Example - Filter Out Documents by Name

In the next example all PDF documents with 'pdfunit-perl' in their names are selected. The title of these documents is then validated:

```
@Test
public void validateFilteredFilesInFolder() throws Exception {
  File folder = new File(PATH);
  FilenameFilter allPdfunitFiles = new FilenameMatchingFilter(".*pdfunit-perl.*");
  AssertThat.eachDocument()
            .inFolder(folder)
            .passedFilter(allPdfunitFiles)
            .hasProperty("Title").equalsTo("PDFUnit - Automated PDF Tests")
  ;
}
```

There are two kinds of filters. The filter `FilenameContainingFilter` checks whether a file name (including the path) **contains** a substring, and the filter `FilenameMatchingFilter` evaluates whether it **matches** a regular expression. The regular expression should always begin with '`.*`'.

Chapter 6. Experience from Practice

6.1. Text in Page Header after Page 2

Initial Situation

Say your company sends electronic invoices, and each page after the first page of the invoice should have a link to the home page of your company.

Problem

The header of the first page differs from the header of the next pages. And the link itself is also given in the page body. That fact must not influence the test for the link in the header.

Solution Approach

You have to define the relevant pages and also the region for the header. The following code shows how simple it is:

Solution

```
@Test
public void hasLinkInHeaderAfterPage2() throws Exception {
  String filename = "documentUnderTest.pdf";
  String linkToHomepage = "http://pdfunit.com/";
  PagesToUse pagesAfter1 = ON_EVERY_PAGE.after(1);
  PageRegion headerRegion = createHeaderRegion();

  AssertThat.document(filename)
            .restrictedTo(pagesAfter1)
            .restrictedTo(headerRegion)
            .hasText()
            .containing(linkToHomepage)
  ;
}

private PageRegion createHeaderRegion() {
  int leftX  =   0;  // in millimeter
  int upperY =   0;
  int width  = 210;
  int height =  30;
  PageRegion headerRegion = new PageRegion(leftX, upperY, width, height);
  return headerRegion;
}
```

6.2. Does Content Fit in Predefined Form Fields?

Initial Situation

A PDF document is created using a document template with empty fields as place holders for text, e.g. the address of a customer in an advertising letter. At runtime the fields are filled with text.

Problem

The text could be larger than the available size of the field.

Solution Approach

PDFUnit provides methods to detect **text overflow**.

Solution

```
@Test
public void noTextOverflow_AllFields() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasFields()
            .allWithoutTextOverflow()
  ;
}
```

A similar test can be done with one field:

```
@Test
public void noTextOverflow_OneField() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Textfield, text inside, align left:";

  AssertThat.document(filename)
            .hasField(fieldname)
            .withoutTextOverflow()
  ;
}
```

Chapter 3.14: “Form Fields - Text Overflow” (p. 38) describes this subject in detail.

6.3. Name of the Former CEO

Initial Situation

The board of executives has changed.

Problem

The name of the former CEO must be changed in the header of newly created PDF documents.

Solution Approach

PDFUnit provides methods to check documents for the **non-existence** of text.

Solution

```
@Test
public void verifyOldCEONotPresent() throws Exception {
  String filename = "documentUnderTest.pdf";
  String oldCEO = "NameOfOldCEO";
  PageRegion header = createHeaderRegion();

  AssertThat.document(filename)
            .restrictedTo(header)
            .hasText()
            .notContaining(oldCEO)
  ;
}
```

6.4. Authorized Signature of the new CEO

Initial Situation

The board of executives has changed.

Problem

PDF documents which are used for contracts need to have a valid (visible) signature. So, the signature of the new CEO must be used.

The new and the old signature exist as files, but you have to give them the same file name. Otherwise, all programs have to be recompiled the next time the CEO changes.

Solution Approach

The image file is compared byte-wise with the signature image from the PDF documents.

Solution

```
@Test
public void verifyNewSignatureOnLastPage() throws Exception {
  String filename = "documentUnderTest.pdf";
  String newSignatureImage = "images/CEO-signature.png";

  AssertThat.document(filename)
            .restrictedTo(LAST_PAGE)
            .containsImage(newSignatureImage)
  ;
}
```

6.5. New Logo on each Page

Initial Situation

Two companies merge.

Problem

Some documents need a new logo. This should be visible on each page.

Solution Approach

The new logo exists as an image file. PDFUnit uses this file for tests.

Solution

```
@Test
public void verifyNewLogoOnEveryPage() throws Exception {
  String filename = "documentUnderTest.pdf";
  String newLogoImage = "images/newLogo.png";

  AssertThat.document(filename)
            .containsImage(newLogoImage)
  ;
}
```

6.6. Do PDF Documents comply with Company Rules?

Initial Situation

Your company has created internal rules for the layout and the content design of business letters.

Problem

It is generally too expensive - and too error-prone - to check each document manually.

Solution Approach

So the rules are catalogued in an Excel file and that file is used by the automated tests.

Solution

```
@Test
public void validateCompanyRules() throws Exception {
  String folderName = "folder-with-testdocuments";
  File folder = new File(folderName);
  String companyRules = PATH_TO_RULES + "letters/companyRules.xls";
  PDFValidationConstraints excelRules = new PDFValidationConstraints(companyRules);
  AssertThat.eachDocument()
            .inFolder(folder)
            .compliesWith()
            .constraints(excelRules)
  ;
}
```

6.7. Compare ZUGFeRD Data with Visual Content

Initial Situation

Your company sends electronic invoices having embedded ZUGFeRD data.

Problem

The problem is not that ZUGFeRD data are created in programms and pass through multiple steps of complex work flows. The real problem is that ZUGFeRD data are invisible. That makes it difficult to ensure that they are correct. No document should have any difference between the visible data and the invisible data. But how can you check that?

Solution Approach

PDFUnit reads values from the ZUGFeRD data and compares them with the data of a PDF page or a page region:

Solution

```
@Test
public void validateZUGFeRD_CustomerNameInvoiceIdIBAN() throws Exception {
  String filename = "zugferd10/ZUGFeRD_1p0_BASIC_Einfach.pdf";

  XMLNode nodeBuyerName = new XMLNode("ram:BuyerTradeParty/ram:Name");
  XMLNode nodeInvoiceId = new XMLNode("rsm:HeaderExchangedDocument/ram:ID");

  PageRegion regionPostalTradeAddress = createRegionPostalAddress();
  PageRegion regionInvoiceId = createRegionInvoiceId();

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(regionPostalTradeAddress)
            .hasText()
            .containingZugferdData(nodeBuyerName)
  ;

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(regionInvoiceId)
            .hasText()
            .containingZugferdData(nodeInvoiceId)
  ;
}
```

Sometimes the XML data and the visible data are not compatible with each other. One example is the value of 'invoice total' in the document 'ZUGFeRD_1p0_BASIC_Einfach.pdf'. The PDF page contains the value with a German **decimal comma** and the ZUGFeRD data contain it with an English **decimal point**. In such a case more sophisticated test methods may help, such as those described in chapter 3.39: “ZUGFeRD” (p. 82).

6.8. PDF Documents on Web Sites

Initial Situation

You are providing generated PDF documents on your web pages.

Problem

The PDF can only be tested in the context of the web pages. So, the input data for a test has to be entered to the browser frontend and the generated PDF has to be selected also through the web page.

Solution Approach

Selenium provides a good way to select a PDF document from a web page. This document will be opened by PDFUnit as a Stream.

Solution

The following lines represent the main part of the test:

```
/**
 * When the URL of the pdf document inside an HTML page is generated dynamically,
 * you have to find the link (href) first.
 * Input data for the web page can also be typed with Selenium (not shown here).
 */
@Test
public void verifyPDF_LoadedBySeleniumWebdriver() throws Exception {
  // arrange, navigate to web site:
  String startURL = "http://www.unicode.org/charts/";
  driver.get(startURL);
  WebElement element = driver.findElement(By.linkText("Basic Latin (ASCII)"));
  String hrefValue = element.getAttribute("href");

  // act, load PDF web site:
  URL url = new URL(hrefValue);

  // assert, validate PDF:
  String expectedTitle  = "The Unicode Standard, Version 6.3";

  AssertThat.document(url)
            .hasTitle().equalsTo(expectedTitle)
  ;
  AssertThat.document(url)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing("0000", "007F")
  ;
}
```

The remaining lines are:

```
/**
 * This sample shows how to test a PDF document with Selenium and PDFUnit.
 * See the previous code listing for the '@Test' method.
 *
 * @author Carsten Siedentop, March 2012
 */
public class PDFFromWebsiteTest {

  private WebDriver driver;

  @Before
  public void createDriver() throws Exception {
    driver = new HtmlUnitDriver();
    Logger htmlunitLogger = Logger.getLogger("com.gargoylesoftware.htmlunit");
    htmlunitLogger.setLevel(java.util.logging.Level.SEVERE);
  }

  @After
  public void closeAll() throws Exception {
    driver.close();
  }

  // @Test
  // public void verifyPDF_LoadedBySeleniumWebdriver()...

}
```

For more information about Selenium contact the project site http://seleniumhq.org/.

6.9. HTML2PDF - Does the Rendering Tool Work Correct?

Initial Situation

A web application creates dynamic web pages and additionally it provides the possibility to download the current web page as a PDF document. The PDF generation is done by rendering the HTML page with an appropriate tool.

Problem

How do you know that the complete content of the HTML page is also present in the generated PDF? Do you know the boundary conditions required by the rendering tool? Does your HTML meets these requirements?

Solution Approach

The test starts by reading the HTML page using Selenium. Text is extracted with Selenium and stored in local variables.

Then the PDF generation is started through the web page and the resulting document is selected again with Selenium. Finally the stored text can be used in PDFUnit test methods.

Solution

```
/**
 * This sample shows how to test an HTML page with Selenium, then let it be rendered
 * by the server to PDF and verify that content also appears in PDF.
 *
 * @author Carsten Siedentop, February 2013
 */
public class Html2PDFTest {

  private WebDriver driver;

  @Test
  public void testHtml2PDFRenderer_WikipediaSeleniumEnglish() throws Exception {
    String urlWikipediaSelenium = "http://en.wikipedia.org/wiki/Selenium_%28software%29";
    driver.get(urlWikipediaSelenium);

    String section1  = "History";
    String section2  = "Components";
    String section3  = "The Selenium ecosystem";
    String section4  = "References";
    String section5  = "External links";

    assertLinkPresent(section1);
    assertLinkPresent(section2);
    assertLinkPresent(section3);
    assertLinkPresent(section4);
    assertLinkPresent(section5);

    String linkName = "Download as PDF";
    URL url = loadPDF(linkName);

    AssertThat.document(url)
              .restrictedTo(ANY_PAGE)
              .hasText()
              .containing(section1,  WhitespaceProcessing.IGNORE)
              .containing(section2,  WhitespaceProcessing.IGNORE)
              .containing(section3,  WhitespaceProcessing.IGNORE)
              .containing(section4,  WhitespaceProcessing.IGNORE)
              .containing(section5,  WhitespaceProcessing.IGNORE)
    ;
  }

  private void assertLinkPresent(String partOfLinkText) {
    driver.findElement(By.xpath("//a[./span = '" + partOfLinkText + "']"));
  }

  private URL loadPDF(String linkName_LoadAsPDF) throws Exception {
    driver.findElement(By.linkText(linkName_LoadAsPDF)).click();
    String title = "Rendering finished - Wikipedia, the free encyclopedia";
    assertEquals(title, driver.getTitle());
    WebElement element = driver.findElement(By.linkText("Download the file"));
    String hrefValue = element.getAttribute("href");
    URL url = new URL(hrefValue);
    return url;
  }

  @After
  public void tearDown() throws Exception {
    driver.quit();
  }

  @Before
  public void createDriver() throws Exception {
    driver = new HtmlUnitDriver();
  }

}
```

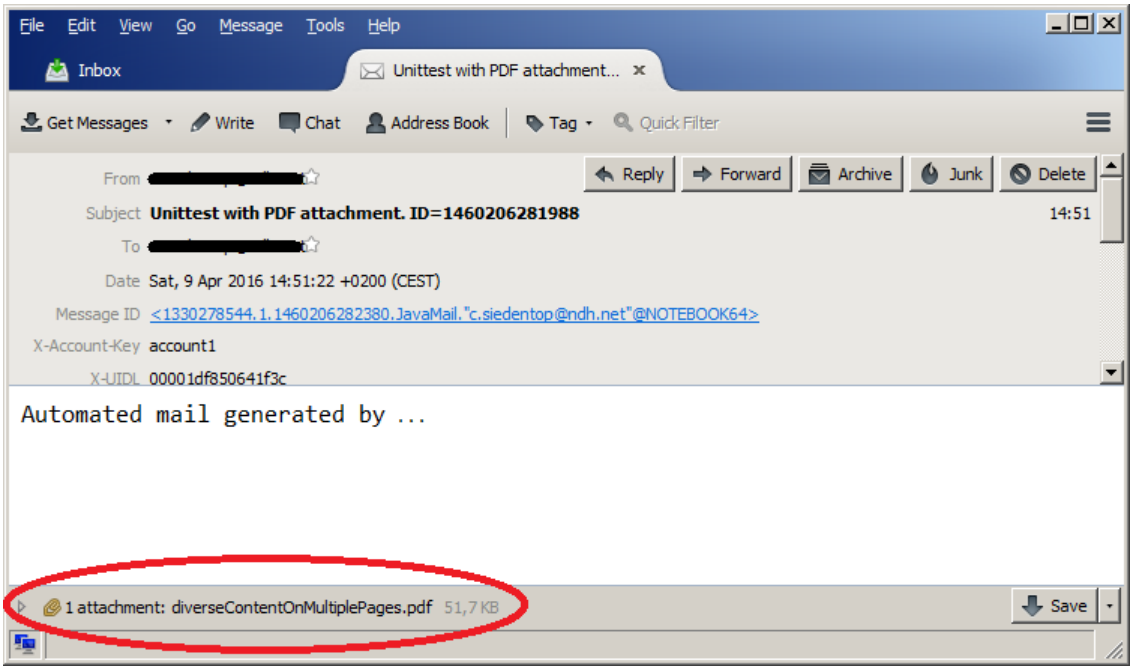
6.10. Validate PDF as Mail Attachment

Initial Situation

Your company sends monthly electronic invoices to customers via email.

Problem

How can you ensure that the PDF document in the email is valid?



There are two problems which need to be solved. First, an email has to be sent during the test. And second, the attachment of the received email has to be analyzed.

Solution Approach

The first problem can be solved using Dumbster, a Java-API for testing mail applications. The second problem can be solved using PDFUnit. The important step is that the attached PDF document is given as a byte-array to PDFUnit.

Solution

```
/**
 * This test invokes a business system which sends a mail with a PDF attachment.
 * After sending the mail the PDF file is validated using PDFUnit.
 */
@Test
public void verifyPDFReceivedByEmail() throws Exception {
  // Arrange:
  BusinessSystem myBusinessSystem = BusinessSystem.newInstance();

  // Act:
  myBusinessSystem.doSomethingImportant();
  myBusinessSystem.sendMailWithPDFAttachment();

  // Assert:
  String pdfFileName = myBusinessSystem.getAttachedPDFName();
  byte[] attachmentAsByteArray = ReceiveMailHelper.getInstance(server)
                                                  .getAttachmentFromLastMail(pdfFileName);
  DocumentValidator pdfDocument = AssertThat.document(attachmentAsByteArray);
  pdfDocument.hasNumberOfPages(4);
  pdfDocument.restrictedTo(EVERY_PAGE)
             .hasText()
             .containing("http://pdfunit.com")
  ;
}
```

Here are the remaining lines of the test:

```
/**
 * Validation of a PDF document received by email.
 * This example uses <a href="https://github.com/rjo1970/dumbster.git">dumbster</a>,
 * as a mail testing API.
 *
 * @author Carsten Siedentop, August 2014
 */
public class MailWithPDFAttachmentTest {

  private SmtpServer server;

  @Before
  public void createEmptyMailStoreDirectory() throws Exception {
    ServerOptions options = new ServerOptions();
    options.port = SendMailHelper.SMTP_PORT;
    options.threaded = false;
    server = SmtpServerFactory.startServer(options);
  }

  @After
  public void teardown() {
    server.stop();
  }

  // @Test
  // public void verifyPDFReceivedByEmail()...

}
```

In addition to this simple test, it is also possible to validate, say, invoice data from ZUGFeRD. For example, the next test checks that the IBAN value of the ZUGFeRD data is the same as the IBAN value on the first page of the document.

```
...
XMLNode nodeIBAN = new XMLNode("ram:IBANID");
PageRegion regionIBAN = createRegionIBAN();

DocumentValidator pdfDocument = AssertThat.document(pdfStream);
pdfDocument.restrictedTo(FIRST_PAGE)
           .restrictedTo(regionIBAN)
           .hasText()
           .containingZugferdData(nodeIBAN)
;
...
```

6.11. Validate PDF from Database

Initial Situation

An application stores PDF documents into a database.

Problem

How does one ensure that the PDF document in the database contains the expected data? And how should the test to be written so that it can be repeated?

Solution Approach

After the application has written the PDF into the database, PDFUnit reads the PDF from the database as an InputStream and executes your tests. DBUnit was used to reset the database each time before running the test.

Solution

```
/**
 * This tests validates a PDF document that a business program
 * has stored as a BLOB into a database.
 */
@Test
public void verifyPDFDataFromDatabase() throws Exception {
  // Arrange:
  int userID = 4711;
  BusinessSystem myBusinessSystem = BusinessSystem.newInstance(userID);

  // Act:
  myBusinessSystem.doSomethingImportantAndWritePDFToDatabase();

  // Assert - compare the data of the DB with the data of the original file:
  String referencePdfName = myBusinessSystem.getPDFName();
  InputStream actualPdfFromDB = DBHelper.readPdfFromDB(userID);
  FileInputStream pdfReferenceFromFile = new FileInputStream(referencePdfName);

  AssertThat.document(actualPdfFromDB)
            .and(pdfReferenceFromFile)
            .haveSameText()
            .haveSameAppearance()
  ;

  actualPdfFromDB.close();
  pdfReferenceFromFile.close();
}
```

This example works with every JDBC database. The complete program is part of the demo project, which can be downloaded from PDFUnit.

Information about DBUnit can be found on the project's homepage.

6.12. Caching Test Documents

Initial Situation

You have many tests - that is good.

Problem

The tests are running slowly - that is bad.

Solution Approach

Maybe it's because the PDF documents are very large, but after all it has to be tested. Instantiate the test document only once for all tests in a class.

Solution

```
public class CachedDocumentTestDemo {

  private static DocumentValidator document;

  @BeforeClass  // Document will be instantiated once for all tests:
  public static void loadTestDocument() throws Exception {
    String filename = "documentUnderTest.pdf";
    document = AssertThat.document(filename);
  }

  @Test
  public void test1() throws Exception {
    document.hasNumberOfBookmarks(4);
  }

  // ... and more tests.

}
```

Of course, you can concatenate all test methods in one statement. But how would you name the test? Names of test methods should reflect the content as exactly as possible. Otherwise, a test report with many hundreds of tests is difficult to understand. So, `testAll` is a bad name - it means nothing.

PDFUnit is stateless. But it can not be guaranteed that 3rd-party libraries are also stateless. So, if you have problems with tests using cached documents, change the annotation `@BeforeClass` into `@Before`, remove the modifier `static` and the PDF document is instantiated for each test method:

```
@Before  // Document will be instantiated for each test. No caching:
public void loadTestDocument() throws Exception {
  String filename = "documentUnderTest.pdf";
  document = AssertThat.document(filename);
}
```

Chapter 7. PDFUnit for Non-Java Systems

7.1. A quick Look at PDFUnit-NET

A 'PDFUnit-NET' version for a .NET environment is provided since December 2015.

```
[TestMethod]
public void HasAuthor()
{
  String filename = "documentUnderTest.pdf";
  AssertThat.document(filename)
            .hasAuthor()
            .matchingExact("PDFUnit.com")
  ;
}
```

```
[TestMethod]
[ExpectedException(typeof(PDFUnitValidationException))]
public void HasAuthor_StartingWith_WrongString()
{
  String filename = "documentUnderTest.pdf";
  AssertThat.document(filename)
            .hasAuthor()
            .startingWith("wrong_sequence_intended")
  ;
}
```

PDFUnit-NET is fully compatible to PDFUnit-Java because a DLL is generated from the Java version. However, this means that method names in C# begin with lowercase letters.

PDFUnit-NET comes with an own manual.

7.2. A quick Look at PDFUnit-Perl

'PDFUnit-Perl', a version of PDFUnit, contains a Perl module `PDF::PDFUnit` with all necessary scripts. Together with other CPAN modules e.g. `TEST::More` or `Test::Unit` it is easy to write automated tests, which are 100% compatible with 'PDFUnit-Java'.

It is intended to upload `PDF::PDFUnit` to the CPAN archive.

Here are two simple examples using `TEST::More` and `PDF:PDFUNIT`:

```
#
# Test hasFormat
#
ok(
  com::pdfunit::AssertThat
    ->document("documentInfo/documentInfo_allInfo.pdf")
    ->hasFormat($com::pdfunit::Constants::A4_PORTRAIT)
  , "Document does not have the expected format A4 portrait")
;
```

```
#
# Test hasAuthor_WrongValueIntended
#
throws_ok {
  com::pdfunit::AssertThat
    ->document("documentInfo/documentInfo_allInfo.pdf")
    ->hasAuthor()
    ->equalsTo("wrong-author-intended")
} 'com::pdfunit::errors::PDFUnitValidationException'
,"Test should fail. Demo test with expected exception."
;
```

A separate documentation covers PDFUnit-Perl.

7.3. A quick Look at PDFUnit-XML

It is unnecessary for testers to know Java to write tests for PDF documents. With the name 'PDFUnit-XML' a version of PDFUnit exists for an XML-based system. It contains a runtime to execute the tests, scripts to start them, XML Schema to validate the tests and stylesheets as part of the runtime. 'PDFUnit-XML' is completely compatible with 'PDFUnit-Java'.

The following examples show the idea behind PDFUnit-XML:

```
<testcase name="hasTextOnSpecifiedPages_Containing">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onPage="1, 2, 3" >
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="hasTitle_MatchingRegex">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasTitle>
      <startingWith>PDFUnit sample</startingWith>
      <matchingRegex>.*Unit.*</matchingRegex>
    </hasTitle>
  </assertThat>
</testcase>
```

```
<testcase name="compareText_InPageRegion">
  <assertThat testDocument="test/test.pdf"
              referenceDocument="reference/reference.pdf"
  >
    <haveSameText on="EVERY_PAGE" >
      <inRegion upperLeftX="50" upperLeftY="720" width="150" height="30" />
    </haveSameText>
  </assertThat>
</testcase>
```

```
<testcase name="hasField_MultipleFields">
  <assertThat testDocument="acrofields/simpleRegistrationForm.pdf">
    <hasField withName="name" />
    <hasField withName="address" />
    <hasField withName="postal_code" />
    <hasField withName="email" />
  </assertThat>
</testcase>
```

Names of the tags and attributes are mostly the same as the function names in the Java-API. They also follow the idea of 'Fluent Interfaces' (http://de.wikipedia.org/wiki/Fluent_Interface).

XML Schema exists to validate the XML syntax.

A detailed description of PDFUnit-XML is available.

Chapter 8. PDFUnit-Monitor

The PDFUnit-Monitor is an application that shows the result of PDFUnit tests. Tests are written in Excel files, so non-developers can create and run the tests.

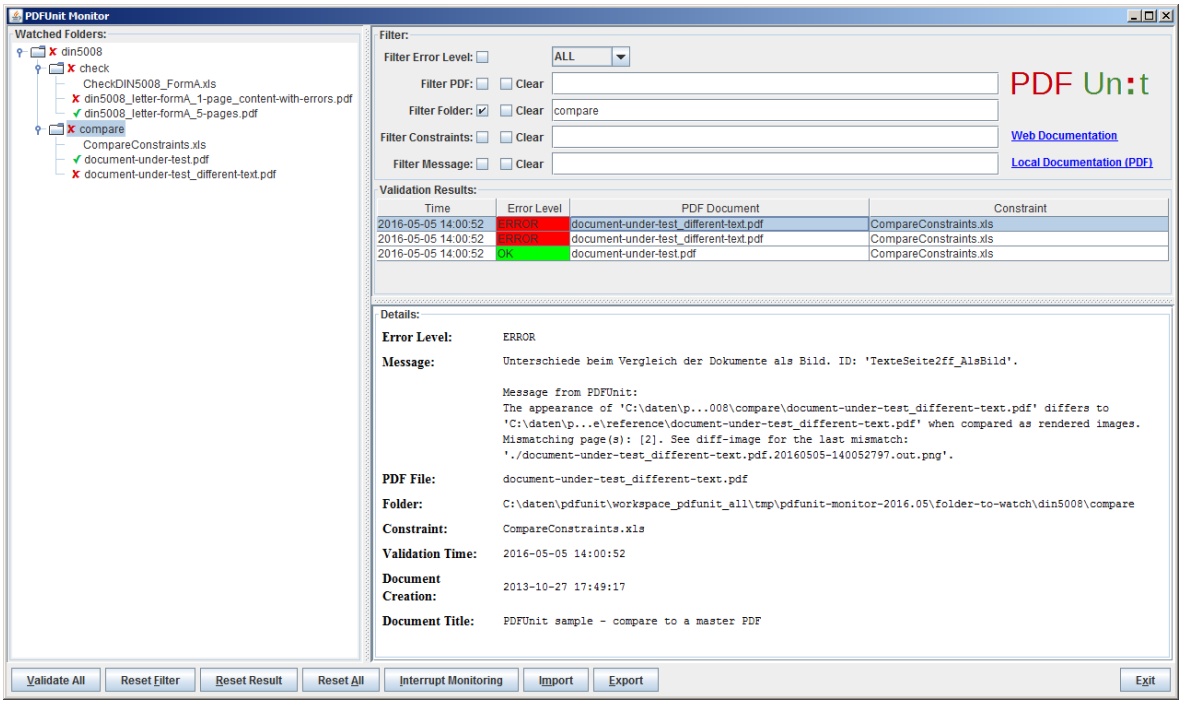
The functional scope of the PDFUnit-Monitor is large. Therefore a detailed description of it exists as a separate file, also a demonstration video is available. Both can be downloaded with this link (download). The separate documentation provides also information about the installation and configuration of the PDFUnit-Monitor. The following sections briefly describe the main features.

Monitored Folders

The PDFUnit-Monitor monitors all PDF documents in a defined directory and its subdirectories. It checks the documents against rules which are read from Excel files. The Excel files have to lay also in the monitored directories. If new PDF documents were copied into the monitored directories, the tests started automatically. A manual start is not necessary but can be done. If a PDF document complies with all rules, its name in the folder tree is decorated with a green checkmark. When a PDF test fails, all violations will be listed. Additionally its name is decorated with a red cross. This status is transferred to the directory name. The name of a folder is decorated with a green checkmark only, when the folder itself and all subdirectories contain valid PDF documents. Otherwise, the folder is decorated with a red cross.

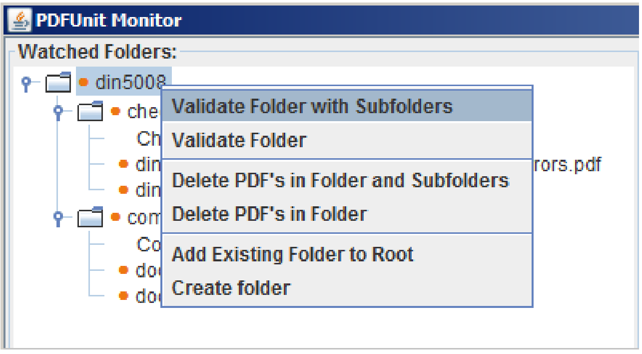
Folder with the name 'reference' are not watched. Such folders contain PDF documents which will be used as references when comparing actual documents under test.

The following picture shows the PDFUnit-Monitor. On the left side the folder structure with PDF and Excel files can be seen. The right side shows the validation results in the upper half and details of a selected message in the lower half.



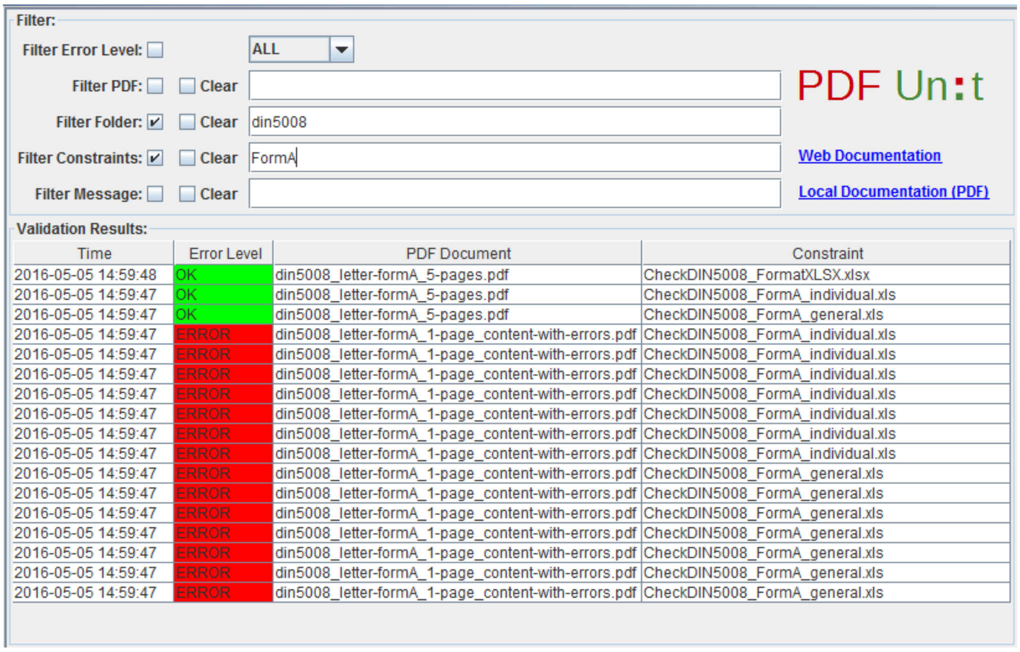
A double click on a PDF or Excel document in the folder structure or in the error list opens the standard application of the operating system for the document.

Each element of the folder structure provides a context menu with various functions. The following figure shows an example:



Overview of Test Results with Filter Options

The Monitor shows all validation results as a list in the upper part of the right-hand side. Each constraint validation is one entry in the list. The details of a validation will be shown in the lower part of the right side when a list entry is selected.



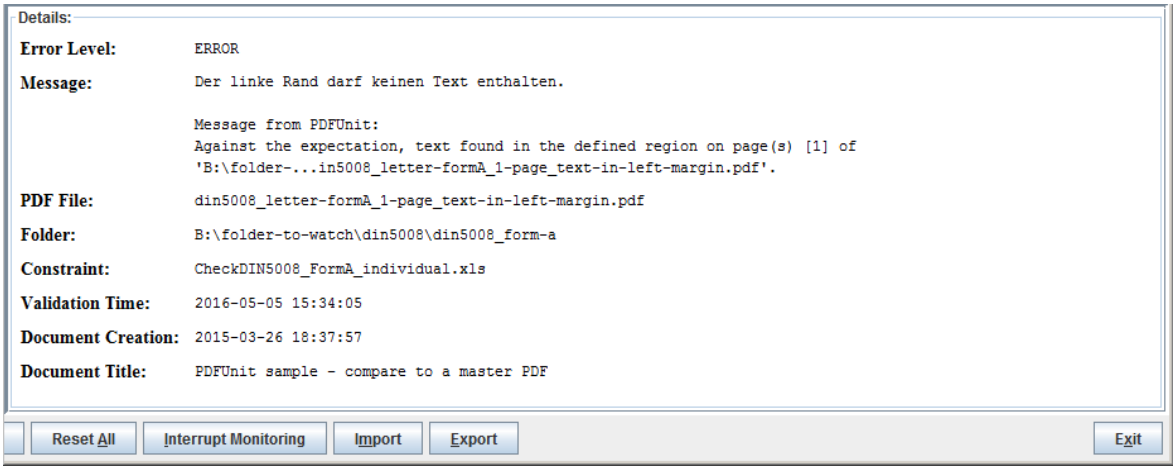
Time	Error Level	PDF Document	Constraint
2016-05-05 14:59:48	OK	din5008_letter-formA_5-pages.pdf	CheckDIN5008_FormatXLSX.xlsx
2016-05-05 14:59:47	OK	din5008_letter-formA_5-pages.pdf	CheckDIN5008_FormA_individual.xls
2016-05-05 14:59:47	OK	din5008_letter-formA_5-pages.pdf	CheckDIN5008_FormA_general.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls
2016-05-05 14:59:47	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls

The error list can be filtered by the names of PDF documents, folders, Excel files and regular expressions on error messages. The folder structure on the left side is connected with the filters on the right side. Each time, when a folder or a document is selected in the structure, a corresponding filter is set.

When a cell with a PDF or Excel document is double-clicked, the standard application of the operating system for that document type starts.

Details of an Error

When a row of the error list is selected all details of that entry will be shown in the lower part of the right side of the Monitor.



The first part of the error message was read from the Excel file. That part is designed by the person who created the tests. The next part of the message comes from PDFUnit. Additionally to the error message itself useful information about the PDF document, the constraint file and the execution time are provided.

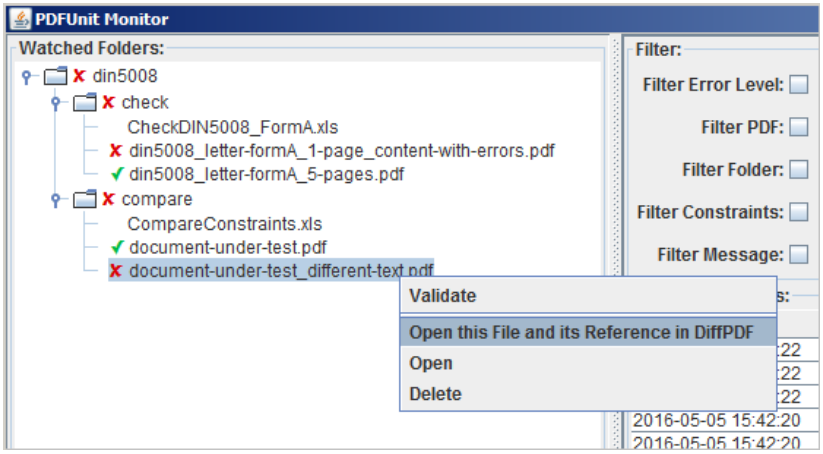
The error messages of PDFUnit are currently in English and in German. Other languages can be provided with a little work, when requested.

Comparing a PDF with a Reference

PDF document can be compared to a reference. The rules for the comparison are read from Excel files. When the PDFUnit-Monitor detects a difference between the test and the referenced PDF the name of the test document will be decorated with a red cross.

A reference document has to have the same name as the PDF under test and has to be located in a subdirectory named 'reference' below that folder which contains the PDF under test.

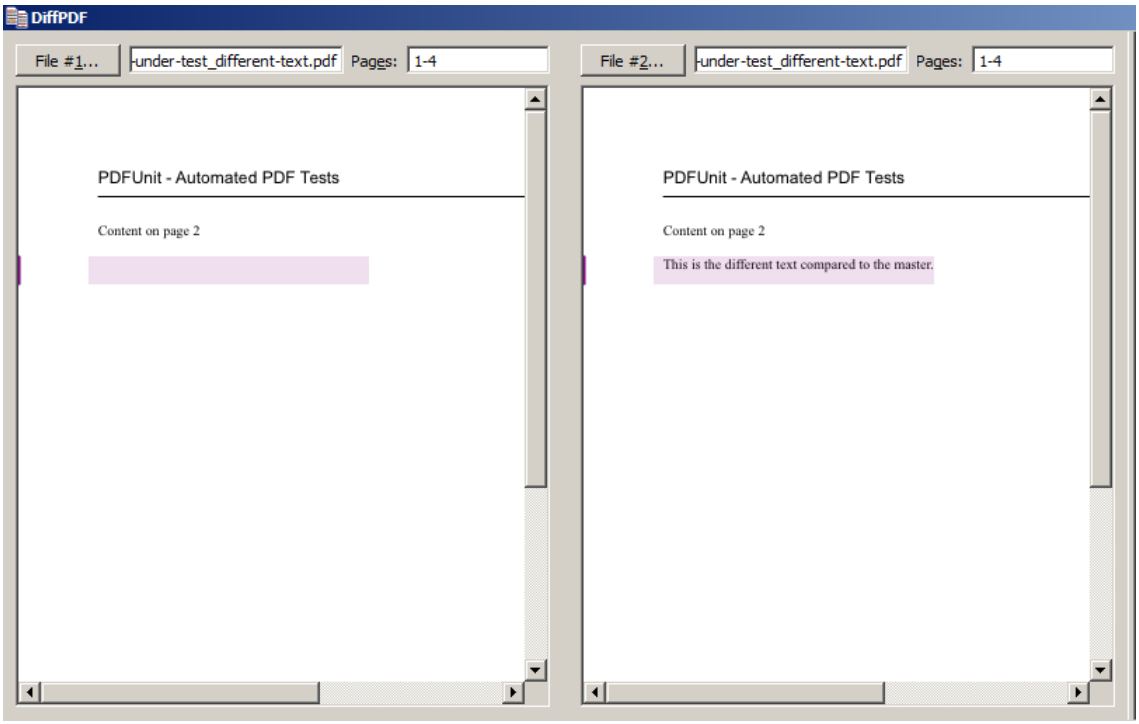
If the validation detects a difference, the program 'DiffPDF 1.5.1 portable' can be started by clicking the right mouse button. That program marks the differences very clearly:



The program was created by Mark Summerfield and is available as a 'portable app' from this link (download). DiffPDF can be used in English, German, French and Czech. Many thanks to all who are involved for their work and the great result.

The next image shows the application DiffPDF just after it was started from the PDFUnit-Monitor. The left side shows the referenced PDF and the right side the current test PDF. The application points

directly to the first difference, in this case to page 2. The differences are marked with a coloured background. The image does not show the buttons to navigate from one mismatch to the next.



Export and Import of Test Results

The test results can be exported as XML clicking the button 'Export' With XSLT stylesheets the exported files can be converted into HTML reports. With a click on the button 'Import' the can loaded again into the monitor.

Internationalization

The PDFUnit monitor is currently available for in English and in German. An extension to other languages is structurally prepared and can be realized on demand with little effort.

Chapter 9. Utility Programs

9.1. General Remarks for all Utilities

PDFUnit provides utility programs to extract several parts of a PDF document into separate files, mostly XML, which can then be used in tests. The following list gives an overview of the available programs:

```
// Utility programs belonging to PDFUnit:

ConvertUnicodeToHex          9.2: “Convert Unicode Text into Hex Code” (p. 121)
ExtractBookmarks             9.5: “Extract Bookmarks to XML” (p. 125)
ExtractEmbeddedFiles         9.4: “Extract Attachments” (p. 123)
ExtractFieldInfo             9.3: “Extract Field Information to XML” (p. 122)
ExtractFontInfo              9.6: “Extract Font Information to XML” (p. 126)
ExtractImages                9.7: “Extract Images from PDF” (p. 127)
ExtractJavaScript            9.8: “Extract JavaScript to a Text File” (p. 128)
ExtractNamedDestinations     9.9: “Extract Named Destinations to XML” (p. 129)
ExtractSignatureInfo         9.10: “Extract Signature Information to XML” (p. 130)
ExtractXFAData               9.11: “Extract XFA Data to XML” (p. 131)
ExtractXMPData               9.12: “Extract XMP Data to XML” (p. 132)
ExtractZugferdData           9.15: “Extract ZUGFeRD Data” (p. 136)
RenderPdfPageRegionToImage   9.13: “Render Page Sections to PNG” (p. 133)
RenderPdfToImages            9.14: “Render Pages to PNG” (p. 134)
```

The utility programs generate files. Their names are derived from those of the input files. The following rules are used to avoid naming conflicts with existing files:

- Generated file names start with an underscore.
- The names have two suffices. The penultimate is `.out` and the last one is the typical suffix for the kind of file type.

For example, when you extract bookmarks from `foo.pdf`, the file `_bookmarks_foo.out.xml` is created. Rename it before using it in a test, because then it is no longer an output file.

The Windows batch scripts in the following chapters demonstrate how to start the programs. These scripts are part of the PDFUnit release, but you have to adapt most of their content to your environment anyway: you need to set the classpath, input file and output directory.

When you start a program without parameters or with incorrect parameters, PDFUnit shows a message detailing the corect command line parameters.

The utilities also run on Unix. Unix developers should easily translate the Windows scripts into shell scripts. If you need assistance, please contact us at: info[at]pdfunit.com.

9.2. Convert Unicode Text into Hex Code

PDFUnit can handle Unicode. The section 11: “Unicode” (p. 142) deals with this topic in detail.

The following sections describe a utility program that converts a Unicode string into its hex code. The hex code can be used in many of your tests. If you are using a small number of Unicode characters it is easier to use hex code than to install a new font on your computer.

The utility `ConvertUnicodeToHex` converts any string into ASCII and escapes all non-ASCII characters into their corresponding Unicode hex code. For example, the Euro character is converted into `\u20AC`.

The input file can be of any encoding, but you have to define the right encoding before executing the program.

Program Start

You start the Java program with the parameter `-D`:

```
::
:: Converting Unicode content of the input file to hex code.
::

@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ConvertUnicodeToHex
set OUT_DIR=./tmp
set IN_FILE=unicode-to-hex.in.txt

java -Dfile.encoding=UTF-8 %TOOL%  %IN_FILE%  %OUT_DIR%
endlocal
```

Input

The input file `unicode-to-hex.in.txt` contains this data:

```
äöü € @
```

Output

So, the created file `_unicode-to-hex.out.txt` contains the following data:

```
#Unicode created by com.pdfunit.tools.ConvertUnicodeToHex
#Wed Jan 16 21:50:04 CET 2013
unicode-to-hex.in_as-ascii=\u00E4\u00F6\u00FC \u20AC @
```

Leading and trailing whitespaces in the input string will be trimmed! When you need them for your test, add them later by hand.

9.3. Extract Field Information to XML

The program `ExtractFieldInfo` creates an XML file with numerous items of information on many properties of all form fields. You can use the extracted data to reveal properties of the fields. The content of a field is not extracted!

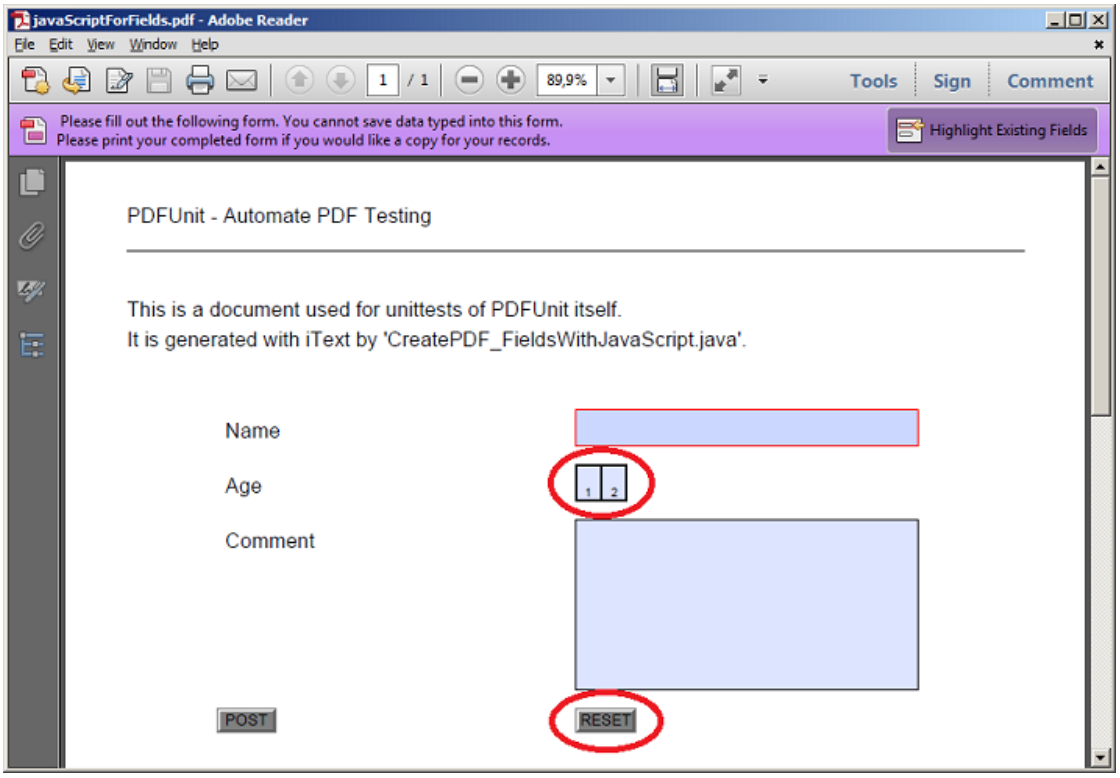
Testing field properties is described in chapter 3.13: “Form Fields” (p. 32).

Program Start

```
::
:: Extract formular fields from a PDF document into an XML file
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/commons-collections4-4.1/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractFieldInfo
set OUT_DIR=./tmp
set IN_FILE=javaScriptForFields.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The input file `javaScriptForFields.pdf` is a sample containing 3 input fields and 2 buttons:



Output

And this is a snippet of the generated file `_fieldinfo_javaScriptForFields.out.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<fields>
  <!-- Width and height values are given as millimeters, rounded to integers. -->
  <field fieldName="ageField"      type="TEXT"
         fieldHeight="8.0"         fieldWidth="11.0"
         isChecked="false"         isEditable="true"
         isExportable="true"       isMultiLineField="false"
         isMultiSelectable="false" isPasswordField="false"
         isRequired="false"        isSigned="false"
         isVisibleInPrint="true"   isVisibleOnScreen="true"
         page="1"                  positionOnPage="[x:105.0, y=59.0]"
  />
  <field fieldName="nameField"     type="TEXT"
         fieldHeight="8.0"         fieldWidth="71.0"
         isChecked="false"         isEditable="true"
         isExportable="true"       isMultiLineField="false"
         isMultiSelectable="false" isPasswordField="false"
         isRequired="true"         isSigned="false"
         isVisibleInPrint="true"   isVisibleOnScreen="true"
         page="1"                  positionOnPage="[x:105.0, y=51.0]"
  />
  <!-- 3 fields deleted for presentation -->
</fields>
```

9.4. Extract Attachments

The utility `ExtractEmbeddedFiles` creates a separate file for every attachment which is embedded in the PDF document.

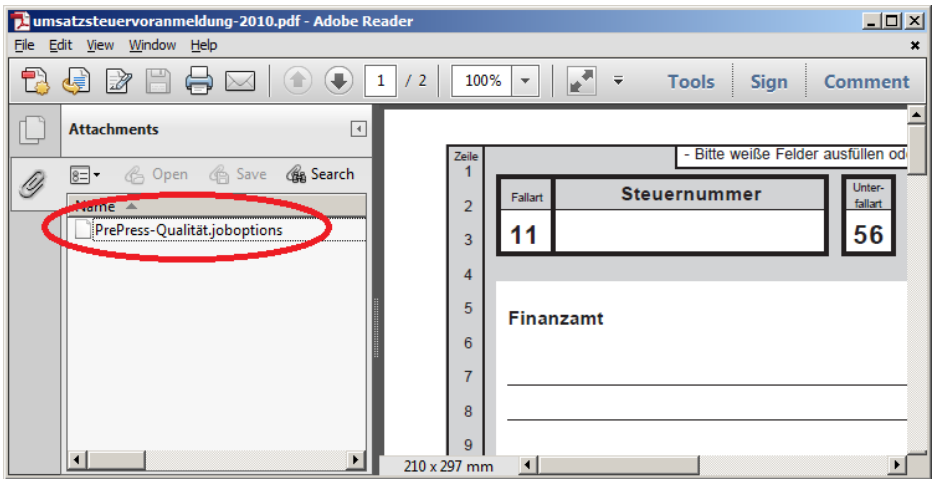
The attachments are exported byte for byte, so all file formats are supported.

Program Start

```
::
:: Extract embedded files from a PDF document. Each in a separate output file.
::

@echo off
setlocal
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractEmbeddedFiles
set OUT_DIR=./tmp
set IN_FILE=umsatzsteuervoranmeldung-2010.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

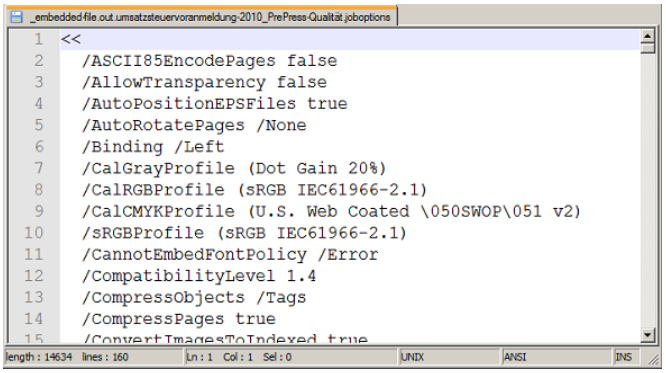
The PDF document `umsatzsteuervoranmeldung-2010.pdf` contains the file `Pre-Press-Qualität.joboptions`.



Output

The name of the generated file contains both the name of the PDF document and the name of the embedded file: `_embedded-file_umsatzsteuervoranmeldung-2010_PrePress-Qualität.joboptions.out`.

A small snippet of its content:



9.5. Extract Bookmarks to XML

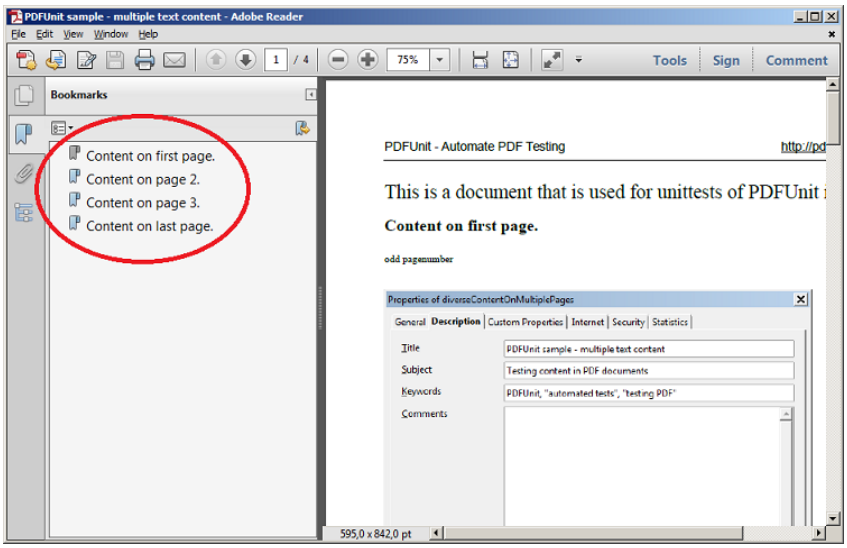
PDFUnit comes with the utility `ExtractBookmarks` which exports bookmarks from PDF documents into an XML file. Chapter 3.5: “Bookmarks and Named Destinations” (p. 18) describes how to use this created XML file for bookmark tests.

Program Start

```
::
:: Extract bookmarks from a PDF document into an XML file
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-collections4-4.1/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractBookmarks
set OUT_DIR=./tmp
set IN_FILE=diverseContentOnMultiplePages.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The file `diverseContentOnMultiplePages.pdf` contains 4 bookmarks:



Output

The output file `_bookmarks_diverseContentOnMultiplePages.out.xml` can be used a an information base for tests:

```
<?xml version="1.0" encoding="utf-8"?>
<bookmarks>
  <bookmark label="Content on first page." page="1" />
  <bookmark label="Content on page 2."     page="2" />
  <bookmark label="Content on page 3."     page="3" />
  <bookmark label="Content on last page."  page="4" />
</bookmarks>
```

9.6. Extract Font Information to XML

As described in chapter 3.12: “Fonts” (p. 29) fonts are a topic which need to be tested. Information about fonts can be extracted using the utility `ExtractFontInfo`. This generated file shows you, how PDFUnit sees the fonts.

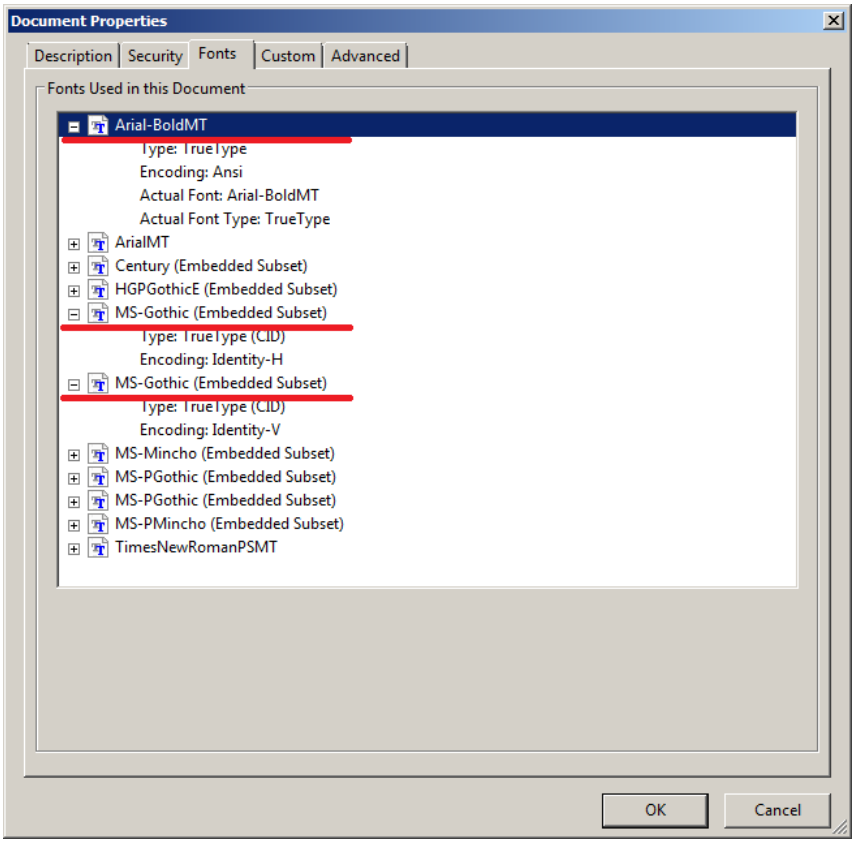
The algorithm that generates the XML file is the same as the one used by the PDFUnit tests.

Program Start

```
::
:: Extract information about fonts of a PDF document into an XML file
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-collections4-4.1/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractFontInfo
set OUT_DIR=./tmp
set IN_FILE=fonts_11_japanese.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

For the Japanese PDF document `fonts_11_japanese.pdf` the Adobe Reader® shows the following fonts:



Output

The output file `_fontinfo_fonts_11_japanese.out.xml` contains the underlined names:

```
<?xml version="1.0" encoding="UTF-8" ?>
<fonts>
  <font basename="Arial-BoldMT"       name="Arial-BoldMT"
        type="TrueType"               vertical="false"          embedded="false" />
  <font basename="ArialMT"            name="ArialMT"
        type="TrueType"               vertical="false"          embedded="false" />
  <font basename="Century"            name="MEEADE+Century"
        type="TrueType"               vertical="false"          embedded="true"  />
  <font basename="HGPGothicE"         name="MFHLHH+HGPGothicE"
        type="Type0"                  vertical="false"          embedded="true"  />
  <font basename="MS-Gothic"          name="MDOLLI+MS-Gothic"
        type="Type0"                  vertical="true"           embedded="true"  />
  <font basename="MS-Gothic"          name="MDOLLI+MS-Gothic"
        type="Type0"                  vertical="false"          embedded="true"  />
  <font basename="MS-Mincho"          name="MEOFCM+MS-Mincho"
        type="Type0"                  vertical="false"          embedded="true"  />
  <font basename="MS-PGothic"         name="MDOMCG+MS-PGothic"
        type="Type0"                  vertical="false"          embedded="true"  />
  <font basename="MS-PGothic"         name="MDOMCG+MS-PGothic"
        type="Type0"                  vertical="true"           embedded="true"  />
  <font basename="MS-PMincho"         name="MEKHMP+MS-PMincho"
        type="Type0"                  vertical="false"          embedded="true"  />
  <font basename="TimesNewRomanPSMT"  name="TimesNewRomanPSMT"
        type="TrueType"               vertical="false"          embedded="false" />
</fonts>
```

Because the XML file contains all subsets of a font it might differ from what Adobe Reader® shows.

9.7. Extract Images from PDF

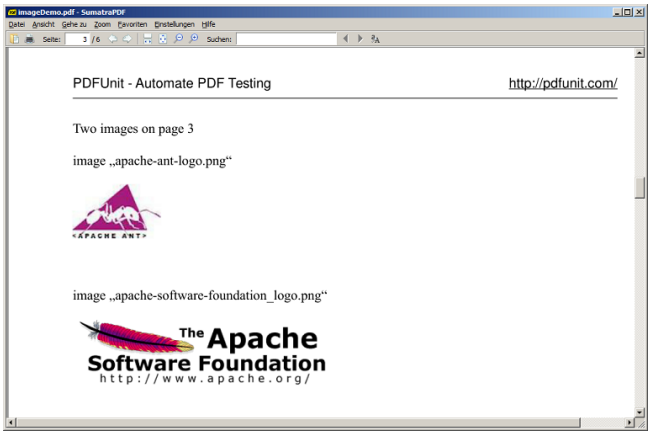
This utility extracts images imbedded in PDF document to PNG images. Each image is written to a separate file. Tests with those images are described in section 3.16: “Images in PDF Documents” (p. 40).

Program Start

```
::
:: Extract all images of a PDF document into a PNG file for each image.
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractImages
set OUT_DIR=./tmp
set IN_FILE=imageDemo.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The input file `imageDemo.pdf` contains two images:



Output

After running the utility, two files are created:

```
# created images:

.\tmp\_exported-image_imageDemo.pdf_Im4-0.out.png   ❶
.\tmp\_exported-image_imageDemo.pdf_Im12-1.out.jpg  ❷
```

❶❷ The number in the file name is the object number within the PDF document.

9.8. Extract JavaScript to a Text File

This utility extracts JavaScript from a PDF document and writes it to a text file, which can be used in PDFUnit tests as described in chapter 3.17: “JavaScript” (p. 43).

Program Start

```
::
:: Extract JavaScript from a PDF document into a text file.
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractJavaScript
set OUT_DIR=./tmp
set IN_FILE=javaScriptForFields.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The file `javaScriptForFields.pdf` used in chapter 9.3: “Extract Field Information to XML” (p. 122) contains the fields `nameField`, `ageField` and `comment` which are all associated with JavaScript.

Inside the Java program which creates the PDF document, the following JavaScript code belongs to the field `ageField`:

```
String scriptCodeCheckAge = "var ageField = this.getField('ageField');"
       + "ageField.setAction('Validate','checkAge()');"
       + ""
       + "function checkAge() {"
       + "  if(event.value < 12) {"
       + "    app.alert('Warning! Applicant\\'s age can not be younger than 12.');"
       + "    event.value = 12;"
       + "  }"
       + "}"
;
```

Output

The output file `_javascript_javaScriptForFields.out.txt` contains:

```
var nameField = this.getField('nameField');nameField.setAction('Keystroke', ...
var ageField = ...;function checkAge() {  if(event.value < 12) {...
var commentField = this.getField('commentField');commentField.setAction(...
```

You can reformat the file to make it easier to read. Added whitespaces do not affect a PDFUnit test.

Note

JavaScript is also used to implement the document actions `OPEN`, `CLOSE`, `PRINT` and SAVE. The discribed extraction utility does only extract JavaScript from document level, but no JavaScript that is bound to actions. A new utility is scheduled for the next releases.

9.9. Extract Named Destinations to XML

“Named Destinations” are landing points inside PDF documents. They are difficult to test because they aren’t displayed anywhere. But the utility `ExtractNamedDestinations` extracts all information about “Named Destinations” into a file. The data can be used in tests as described in chapter 3.5: “Bookmarks and Named Destinations” (p. 18).

And that's the extraction script:

Program Start

```
::
:: Extract information about named destinations in a PDF document into an XML file
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractNamedDestinations
set OUT_DIR=./tmp
set IN_FILE=bookmarksWithPdfOutline.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The input file in this sample, `bookmarksWithPdfOutline.pdf`, contains different named destinations.

Output

The output file `_named-destinations_bookmarksWithPdfOutline.out.xml` contains the following data:

```
<?xml version="1.0" encoding="UTF-8"?>
<destinations>
  <destination name="destination1"            page="1" />
  <destination name="destination2.1"          page="2" />
  <destination name="destination2.2"          page="2" />
  <destination name="destination2_no_blank"   page="2" />
  <destination name="destination3 with blank" page="3" />
</destinations>
```

9.10. Extract Signature Information to XML

Signatures contain a huge amount of data. Some of them reachable by PDFUnit tests. Section 3.28: “Signed PDF” (p. 59) describes tests with signatures.

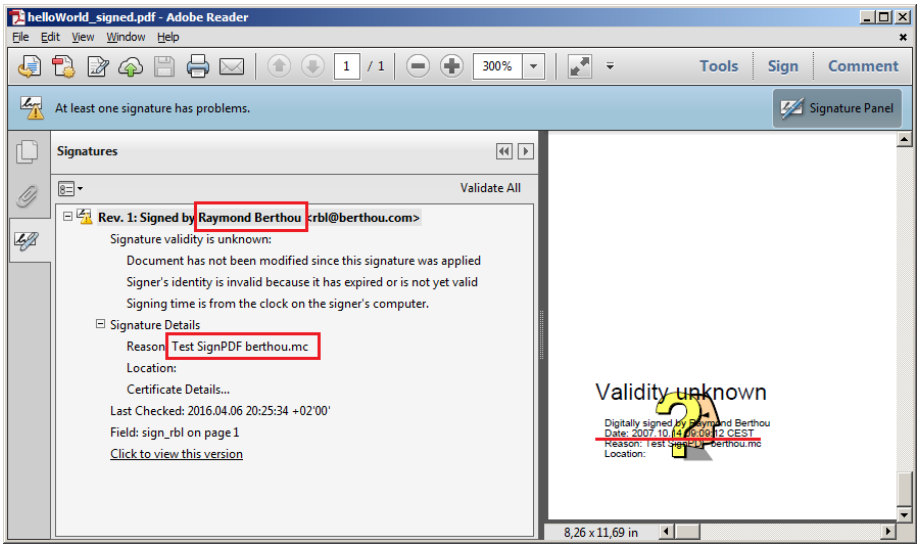
The following script will start the extraction:

Program Start

```
::
:: Extract infos about signatures of a PDF document as XML:
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractSignatureInfo
set OUT_DIR=./tmp
set IN_FILE=signed/helloWorld_signed.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

Adobe Reader® shows the signature data for `helloWorld_signed.pdf`:



Output

Here is the output file `_signatureinfo_helloWorld_signed.out.xml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<signatures>
  <signature fieldname="sign_rbl"
             signatory="Raymond Berthou"
             signingdate="2007-10-14T09:09:12"
             reason="Test SignPDF berthou.mc"
             signed="true"
             covers.whole.document="true"
  />
</signatures>
```

PFUnit will provide more features to test signatures in future releases. This may lead to changes in the XML structure. In case of a problem, please look to an actual user manual.

9.11. Extract XFA Data to XML

Using the utility `ExtractXFAData` you can export XFA data from a PDF document and use it in XPath based tests as described in section 3.37: “XFA Data” (p. 76).

Program Start

```
::
:: Extract XFA data of a PDF document as XML
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractXFAData
set OUT_DIR=./tmp
set IN_FILE=xfa-enabled.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The input file for the script is `xfa-enabled.pdf`, a sample document from iText.

Output

The output XML file `_xfadata_xfa-enabled.out.xml` is quite long. To get a better impression of the generated code, some of the XML-Tags in the next picture are folded:

9.12. Extract XMP Data to XML

The utility program `ExtractXMPData` writes the document level XMP data from a PDF document into an XML file. This file can be used for the PDFUnit tests described in section 3.38: “XMP Data” (p. 79).

XMP data can be found on other places in the PDF than just the document level. Such XMP data is currently not extracted. But it is intended to provide the extraction of all XMP data in the next release of PDFUnit.

Program Start

```
::
:: Extract XMP data from a PDF document as XML
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractXMPData
set OUT_DIR=./tmp
set IN_FILE=LXX_vocab.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The XMP data will be extracted from `LXX_vocab.pdf`.

Output

A part of the output file `_xmpdata_LXX_vocab.out.xml` is shown here:

```
<?xpacket begin='' id='W5M0MpCehiHzreSzNTczkc9d'?>
<?adobe-xap-filters esc="CRLF"?>
<x:xmpmeta xmlns:x='adobe:ns:meta/' x:xmptk='XMP toolkit 2.9.1-14, framework 1.6'>
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
            xmlns:iX='http://ns.adobe.com/iX/1.0/'
>
...
<rdf:Description rdf:about='uuid:f6a30687-f1ac-4b71-a555-34b7622eaa94'
                 xmlns:pdf='http://ns.adobe.com/pdf/1.3/'
                 pdf:Producer='Acrobat Distiller 6.0.1 (Windows)'
                 pdf:Keywords='LXX, Septuagint, vocabulary, frequency'>
</rdf:Description>
<rdf:Description rdf:about='uuid:f6a30687-f1ac-4b71-a555-34b7622eaa94'
                 xmlns:xap='http://ns.adobe.com/xap/1.0/'
                 xap:CreateDate='2006-05-02T11:35:38-04:00'
                 xap:CreatorTool='PScript5.dll Version 5.2.2'
                 xap:ModifyDate='2006-05-02T11:37:57-04:00'
                 xap:MetadataDate='2006-05-02T11:37:57-04:00'>
</rdf:Description>
...
</rdf:RDF>
</x:xmpmeta>
```

9.13. Render Page Sections to PNG

The reasons for testing a particular region of a PDF page are described in section 3.21: “Layout - in Page Regions” (p. 49). To find the coordinates of the area you want to test, PDFUnit provides the small utility `RenderPdfPageRegionToImage`. Choose the width, height and the position of the upper left corner and the corresponding region is then extracted into a file. Verify this file then “by eye” and vary the parameters until you got the right region. Once you have found the correct coordinates for your region, use those parameters in your PDFUnit test.

Program Start

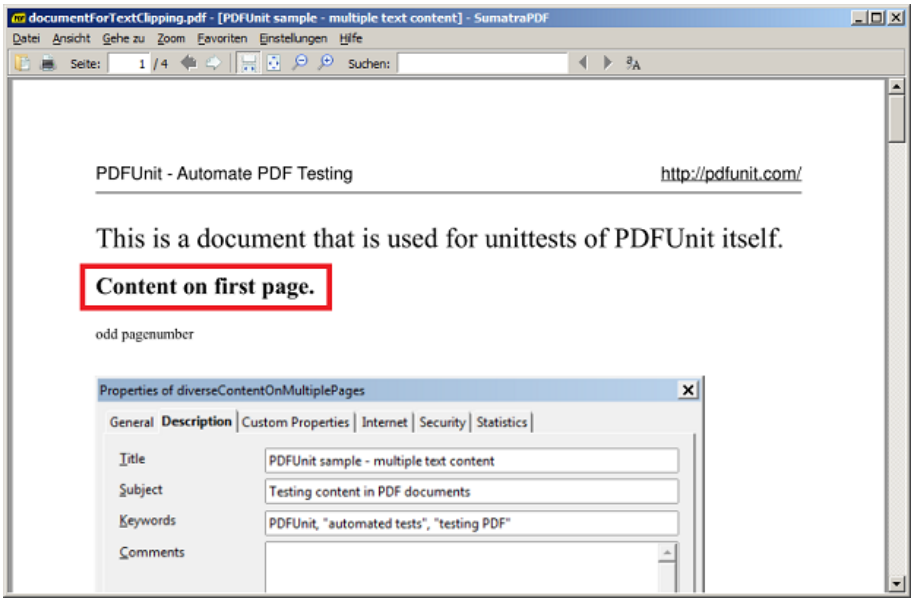
```
::
:: Render a part of a PDF page into an image file
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.RenderPdfPageRegionToImage
set OUT_DIR=./tmp
set PAGENUMBER=1
set IN_FILE=documentForTextClipping.pdf
set PASSWD=

:: Put these values into your test code:
:: Values in millimeter:
set UPPERLEFTX=17
set UPPERLEFTY=45
set WIDTH=60
set HEIGHT=9

java  %TOOL%  %IN_FILE%  %PAGENUMBER%  %OUT_DIR%     ❶
      %FORMATUNIT%  %UPPERLEFTX%  %UPPERLEFTY%  %WIDTH%  %HEIGHT%  %PASSWD%
endlocal
```

❶ The linebreak in this listing is placed here only for documentation purposes.

The 4 values that define a page region have to have the unit millimeter (mm).

Input

The upper part of the input file `documentForTextClipping.pdf` contains the text: “ `Content on first page.` ”



Output

The generated image file has to be checked.

The name of the generated PNG includes the area's coordinates. Because PDFUnit and the utility program `RenderPdfPageRegionToImage` use the same algorithm, you can use the parameter values from the script for your test. And later, you can derive them from the file name:

```
#
# Parameters from filename:
#

_rendered_documentForTextClipping_page-1_area-50-130-170-25.out.png
                                                 |  |   |   |
                                                 |  |   |   +- height
                                                 |  |   +- width
                                                 |  +- upperLeftY
                                                 +- upperLeftX
```

9.14. Render Pages to PNG

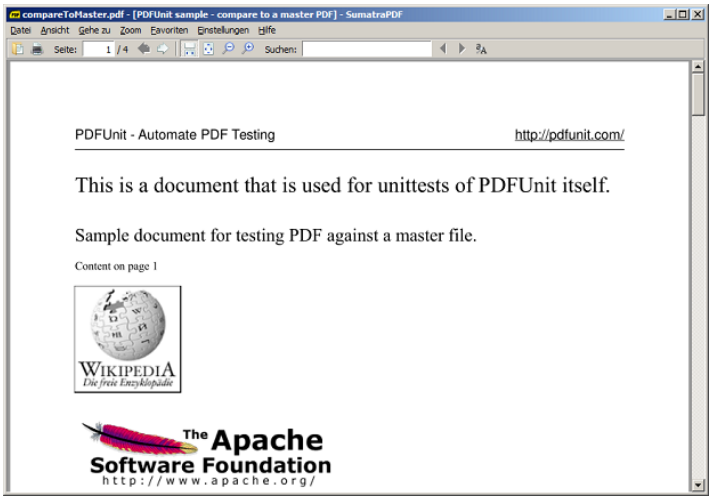
If you want to test formatted text, the only way to do it is to render a PDF page and compare the result with an image of the correctly formatted content. Section 3.20: "Layout - Entire PDF Pages" (p. 48) describes layout-tests using rendered pages. And the utility `RenderPdfToImages` renders a PDF document page by page into PNG files.

Program Start

```
::
:: Render PDF into image files. Each page as a file.
::

@echo off
setlocal
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.RenderPdfToImages
set OUT_DIR=./tmp
set IN_FILE=documentUnderTest.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The input file `documentUnderTest.pdf` consists of 4 pages with different images and text. The PDF Reader “SumatraPDF” (http://code.google.com/p/sumatrapdf) shows the first page:

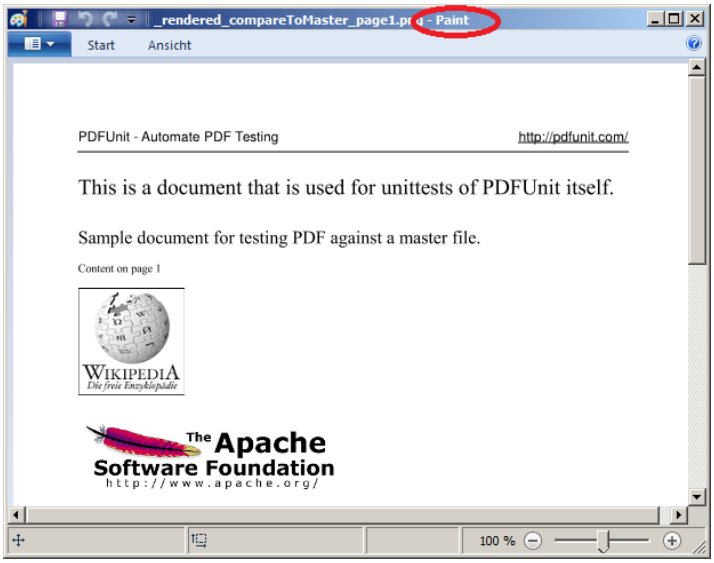


Output

After running the rendering program 4 files are created:

```
.\tmp\_rendered_documentUnderTest_page1
.\tmp\_rendered_documentUnderTest_page2
.\tmp\_rendered_documentUnderTest_page3
.\tmp\_rendered_documentUnderTest_page4
```

The first of these files looks the same as seen with the PDF Reader:



Internally, PDFUnit uses the same algorithm to render the pages as the rendering program does. So, any difference found by a test is due to a change in the PDF document.

9.15. Extract ZUGFeRD Data

The invisible embedded ZUGFeRD data are relevant for accounting, and they should have the same value as the visible data.

At least once in the development process for PDF producing programs, the ZUGFeRD data of a document should be made visible and checked. This can be done with the provided extraction utility `ExtractZugferdData`.

Program Start

The program is started using the following script:

```
::
:: Extract ZUGFeRD data from a PDF document.
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-153/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractZugferdData
set OUT_DIR=./tmp
set IN_FILE=./zugferd10/ZUGFeRD_1p0_BASIC_Einfach.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

The name of the extracted file derives from the name of the source file. In this case the file `_zugferd_ZUGFeRD_1p0_BASIC_Einfach.out.xml` will be created. Its content is the embedded ZUGFeRD file of the PDF but without comments.

Chapter 10. Validation Constraints in Excel Files

Rules for validating PDF documents can be written in Excel files. Their structure and the provided validation functions are described in the following chapters.

Structure of an Excel File

An Excel file is searched for sheets with the following names:

Excel Sheet Name	Comment
regions	Definitions of page regions
check	Definition of test cases for individual PDF documents
compare	Definition of test cases comparing the document under test with a reference document

The expected structure of all three sheets is described in the following sections. In all three sheets a star '*' in the first column indicates a commented line.

The order of the columns must not be changed. Additional columns after the expected ones are allowed. Additional sheets are also allowed.

A sheet can have empty lines. However, when a sheet has too many of them, the last lines with data may not be read. It's therefore better to use the character '*' in the first column of each empty line.

Definition of Page Regions - Sheet 'regions'

Mostly, you will need to restrict a validation rule to a region of a page, not just to the whole page. For example, it makes no sense to compare the text of full pages between two documents when the text contains a date. So, PDFUnit requires that each test case references a defined page region.

A page region is given by 4 values: the x and the y coordinates of the upper-left corner, as well as the width and the height of the region. All values are interpreted in millimeters. The values may have decimal values, but PDFUnit rounds the decimals to the next integer values. The following image shows examples:

	A	B	C	D	E	F
1	*	**Constraint file used for PDFUnit selftests.**				
2	*	**Definition of regions according to DIN5008, Form-A**				
3	*					
4	*	**id**	**x**	**y**	**width**	**height**
5		address region	25	32	80	40
6		info block	125	32	75	40
7		header	25	0	185	27
8		text body page 1	25	80	165	165
9		text body page 2ff	25	32	165	212
10		page number region	25	245	165	12
11		footer	25	260	165	30
12		left margin	0	0	25	297
13		right margin	190	70	20	227
14		spacing below header	0	72	210	9

You can see that the sheet contains the column `id` in addition to the 4 columns with numeric values. This ID has to be unique and will be referenced by the test case definitions of the Excel sheets 'check' and 'compare'.

Test Cases for Single PDF Documents - Sheet 'check'

The sheet 'check' must be used to define test cases which relate to a single PDF document. It does not cover test cases in which two documents are compared with each other. The existing columns are:

Column Name	Comment
id	Name (ID) of the test case
pages	Pages this test should be used on
region	Name of a page regions which is defined in the Excel sheet 'regions'
constraint	Kind of validation. The allowed values are described below.
expected value	The expected value, if a validation needs one.
whitespace	This column contains a value indicating how whitespace will be handled. The allowed values are described below.
message	In this column an error message with placeholders can be defined. The placeholders are also described below.

	A	B	C	D	E	F	G	H
1	*	**Constraint file used for PDFUnit selftests.**						
2	*	**This sheet declares constraints to validate PDF documents according to rules defined by DIN 5008, Form-A.**						
3	*							
4	*	**id**	**pages**	**region**	**constraint**	**expected value**	**whitespace**	**message**
5		address region page 1	1	address region	must contain	Anschrift	ignore	Text is missing. ID: {id}, region: {region}.
6		info block page 1	1	info block	must contain	Infoblock	normalize	Text is missing. ID: {id}, region: {region}.
7		header all pages	all	header	must contain	Automated PDF Tests	keep	Text is missing. ID: {id}, region: {region}.
8		text body page 1	1	text body page 1	must contain	Lorem ipsum		Text is missing. ID: {id}, region: {region}.
9	*	Textbereich Seite 2ff_ID	2...	text body page 2ff	must contain	Lorem ipsum		Text is missing. ID: {id}, region: {region}.
10		page number region all pages	all	page number region	must match	.*Seite \d von \d.*		The page number is missing. ID: {id}, region: {region}.
11		footer all pages	all	footer	must contain	Firmenangaben		Text is missing. ID: {id}, region: {region}.
12	*							
13		left margin all pages	all	left margin	must be empty			Region is not empty. ID: {id}, region: {region}.
14		right margin pages 2ff	2...	right margin	must be empty			Region is not empty. ID: {id}, region: {region}.
15		spacing below header page 1	1	spacing below header	must be empty			Region is not empty. ID: {id}, region: {region}.
16	*							
17		empty pages pages 1, 2, 3	...3	text body page 2ff	must not be empty			Region is empty. ID: {id}, region: {region}.
18		empty pages pages 4, 5	4...5	text body page 2ff	must be empty			Region should not contain text. ID: {id}, region: {region}.

Pages to which a Test Case is Restricted - Column 'page'

A test case definition is often restricted to individual pages of a document. The following list shows all available syntax elements:

Pages	Syntax in Excel
a single page	1
multiple, individual pages	1, 3, 5
all pages	all
all pages after a given page (inclusive)	2...
all pages before a given page (inclusive)	...5
all pages between (inclusive)	2...5

Two page numbers must be separated by a blank. A comma is optional.

Different Ways of Comparing Text - Column 'constraint'

Values in the column 'constraint' in the sheet 'check' are used for specifying how the actual content of a document and the expected text will be compared. The following list shows the allowed values:

Keyword	Behaviour
'must contain'	The text in the column 'expected value' must be part of the page region. Additionally, this constraint type requires whitespace handling information.

Keyword	Behaviour
'must not contain'	The text in the column 'expected value' must not exist in the page region of the document. Additionally, this constraint type requires whitespace handling information.
'must be empty'	The referenced region must not contain any text.
'must not be empty'	The referenced region must have text.
'must match'	The text in the column 'expected value' will be taken as a regular expression and executed against the text in the referenced region. At least one piece of text must match.
'must not match'	The text in the column 'expected value' will be executed as a regular expression against the text in the referenced region. The test is successful if no match is found.

The column 'constraint' must not be empty. In such a case, PDFUnit throws an error message.

The sheet 'compare' may have other values in the column 'constraint'. Those values are described later below.

Validate Signatures and Images - Column 'constraint'

The column 'constraint' in the sheet 'check' can also contain keywords for validating signatures or images in a document:

Keyword	Behaviour
'is signed'	The PDF document has to be signed.
'is signed by'	A PDF document has to be signed. The expected name of the signatory has to be put into the column 'expected value'.
'has number of images'	The number of all visible images in the referenced page region will be compared with the number of the expected images, which must be put into the column 'expected value'.

Handling whitespaces - Column 'whitespace'

Text comparisons may fail due to differences in the whitespace. For example, text which is rendered in different fonts may have line breaks at different positions.

Keyword	Behaviour
'ignore'	All whitespaces are removed from the text before two strings are compared.
'keep'	Whitespaces will be taken 'as is'.
'normalize'	Whitespace at the beginning or the end of a text are deleted. Multiple whitespaces between words are reduced to one blank.

Wrong values in the column 'whitespace' will cause an error message. If the 'whitespace' column is left blank, the program defaults to 'normalize'.

Some validations, such as comparing bookmarks, are independent of whitespaces. For such validations, any declaration of whitespace handling will be ignored.

Expected Value - Column 'expected value'

For a test case to verify that a region of a test document contains an expected value, the Excel file must provide a column for that value. This column is named 'expected value'.

If the 'constraint' column has the values 'must match' or 'must not match', the contents of the 'expected value' column are used as a regular expression. More information about regular expressions can be found online, for example on Wikipedia.

If the column 'constraint' has the value 'has number of images', then the content of the 'expected value' column will be parsed to an integer value.

Error Messages with Placeholders - Column 'message'

Individual error messages can be defined in the Excel sheets. These messages are shown in addition to PDFUnit's validation messages. An error message in the Excel sheet may have placeholders for runtime data. The following image shows some examples:

message
Text is missing. ID: {id}, region: {region}.
Text is missing. ID: {id}, region: {region}.
Text is missing. ID: {id}, region: {region}.
Text is missing. ID: {id}, region: {region}.
Text is missing. ID: {id}, region: {region}.
The page number is missing. ID: {id}, region: {region}.
Text is missing. ID: {id}, region: {region}.

The image shows clearly that placeholders in a text are enclosed with curly brackets. The following placeholders can be used:

Placeholder	Meaning
{id}	The ID of the current test case
{pages}	The page-number of the page where the error is detected
{region}	The value in the column 'region'
{constraint}	The value in the column 'constraint'

Placeholders can be used anywhere inside a text. The values of the placeholders at runtime are enclosed in single quotation marks, so their is no need to use single quotes in the error messages in the Excel sheet.

Test Cases Comparing Two Documents - Sheet 'compare'

This Excel sheet can be used to declare validation rules for comparing two PDF documents. One document is the 'document under test' and the second document is a reference document.

For comparative testing no information about an expected text need be given in the Excel sheet, so the column 'expected value' is not provided in the sheet 'compare'.

	A	B	C	D	E	F	G
1	*	**Constraint file used for PDFUnit selftests.**					
2	*	**Definition of test cases.**					
3	*						
4	*	**id**	**pages**	**region**	**constraint**	**whitespace**	**message**
5		info block page 1	1	info block	same text	normalize	Different text found. ID: {id}, region: {region}.
6		header as text	all	header	same text	keep	Different text found. ID: {id}, region: {region}.
7		header as image	all	header	same appearance		Different appearance between test PDF and the reference PDF. ID: {id}, region: {region}.
8		pages 1, 3, 4 as text	1, 3, 4	body pages 2ff	same text	normalize	Different text found. ID: {id}, region: {region}.
9		pages 1, 3, 4 as image	1, 3, 4	body pages 2ff	same appearance		Different appearance between test PDF and the reference PDF. ID: {id}, region: {region}.
10		footer from page 2	2...	footer	same text	normalize	Different text found. ID: {id}, region: {region}.

The meaning of columns is the same as described in the above sections for the sheet 'check'. But in the 'compare' sheet, other values are allowed in the 'constraint' column:

Keyword	Behaviour
'same text'	Two PDF documents must have the same text in the referenced region. Additionally, this constraint type requires whitespace handling information.
'same appearance'	The referenced regions of the two PDF documents must be identical when compared as rendered images.
'same bookmarks'	The two PDF documents must have the same bookmarks. Obviously this validation does not require page regions, but for technical reasons the column 'region' must not be empty. Instead, the value 'NO_REGION' should be supplied.

The image below shows a test comparing bookmarks of a 'PDF under test' with a reference PDF:

	A	B	C	D	E	F	G
1	*	**Constraint file used for PDFUnit selftests.**					
2	*	**Comparing bookmarks.**					
3	*						
4	*	**id**	**pages**	**region**	**constraint**	**whitespace**	**message**
5	*						
6		bookmarks	all	NO_REGION	same bookmarks		Different bookmarks found. ID: {id}.

PDFUnit searches the reference documents for a given PDF document in a subdirectory of its folder. The subdirectory has to have the name 'reference'. The filename of the reference has to be the same as the PDF under test.

Error Messages at Runtime

The validation of a PDF document does not end with the first detected failure. All defined rules of an Excel file are processed, and then an error message is created for each detected failure.

Chapter 11. Unicode

PDF Documents Containing Unicode

Would the tests described so far also run with content that is not ISO-8859-1, for example with Russian, Greek or Chinese text?

A difficult question. A lot of internal tests are done with Greek, Russian and Chinese documents, but tests are missing for Hebrew, Arabic and Japanese documents. All in all it is not 100% clear that every available test will work with every language, but it should.

The following hints may solve problems not only when working with UTF-8 files under PDFUnit. They may also be helpful in other situations.

Single Unicode Characters

Metadata and keywords can contain Unicode characters. If your operating system does not support fonts for foreign languages, you can use Unicode escape sequences in the format `\uXXXX` within strings. For example the copyright character "©" has the Unicode sequence `\u00A9`:

```
@Test
public void hasProducer_CopyrightAsUnicode() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasProducer()
            .equalsTo("txt2pdf v7.3 \u00A9 SANFACE Software 2004") // 'copyright'
  ;
}
```

Longer Unicode Text

It would be too difficult to figure out the hex code for all characters of a longer text. Therefore PDFUnit provides the small utility `ConvertUnicodeToHex`. Pass the foreign text as a string to the tool, run the program and place the generated hex code into your test. Detailed information can be found in chapter 9.2: "Convert Unicode Text into Hex Code" (p. 121). A test with a longer sequence may look like this:

```
@Test
public void hasSubject_Greek() throws Exception {
  String filename = "documentUnderTest.pdf";
  String expectedSubject = "Εργαστήριο Μηχανικής ΙΙ ΤΕΙ ΠΕΙΡΑΙΑ / Μηχανολόγοι";
//String expectedSubject =  "\u0395\u03C1\u03B3\u03B1\u03C3\u03C4\u03AE"
//                        + "\u03C1\u03B9\u03BF \u039C\u03B7\u03C7\u03B1"
//                        + "\u03BD\u03B9\u03BA\u03AE\u03C2 \u0399\u0399 "
//                        + "\u03A4\u0395\u0399 \u03A0\u0395\u0399\u03A1"
//                        + "\u0391\u0399\u0391 / \u039C\u03B7\u03C7\u03B1"
//                        + "\u03BD\u03BF\u03BB\u03CC\u03B3\u03BF\u03B9";

  AssertThat.document(filename)
            .hasSubject()
            .equalsTo(expectedSubject)
  ;
}
```

Using Unicode within XPath Expressions

Chapter 13.11: "Using XPath" (p. 165) describes how to use XPath in PDFUnit tests. You can also use Unicode sequences in XPath expressions. The following test checks that there is no XML node below the node `rsm:HeaderExchangedDocument` which contains the Unicode string `\u20AC`:

```
@Test
public void hasZugferdData_ContainingEuroSign() throws Exception {
  String filename = "ZUGFeRD_1p0_COMFORT_Kraftfahrversicherung_Bruttopreise.pdf";
  String euroSign = "\u20AC";
  String noTextInHeader =
         "count(//rsm:HeaderExchangedDocument//text()[contains(., '%s')]) = 0";
  String noEuroSignInHeader = String.format(noTextInHeader, euroSign);
  XPathExpression exprNumberOfTradeItems = new XPathExpression(noEuroSignInHeader);
  AssertThat.document(filename)
            .hasZugferdData()
            .matchingXPath(exprNumberOfTradeItems)
  ;
}
```

File Encoding UTF-8 for Shell Scripts

Pay special attention to data read from the file system. Its byte representation depends on the encoding of file system. Every Java program that processes files depends on the environment variable `file.encoding`.

There are multiple possibilities to set the environment for the current shell:

```
set _JAVA_OPTIONS=-Dfile.encoding=UTF8
set _JAVA_OPTIONS=-Dfile.encoding=UTF-8

java -Dfile.encoding=UTF8
java -Dfile.encoding=UTF-8
```

File Encoding UTF-8 for ANT

During the development of PDFUnit there were two tests which ran successfully under Eclipse, but failed with ANT due to the current encoding.

The following command did **not** solve the encoding problem:

```
// does not work for ANT:

ant -Dfile.encoding=UTF-8
```

Instead, the property had to be set using `JAVA_TOOLS_OPTIONS`:

```
// Used when developing PDFUnit:

set JAVA_TOOL_OPTIONS=-Dfile.encoding=UTF-8
```

Maven - Settings in pom.xml for UTF-8

You can configure “UTF-8” in many places in the 'pom.xml'. The following code snippets show some examples. Choose the right one for your problem:

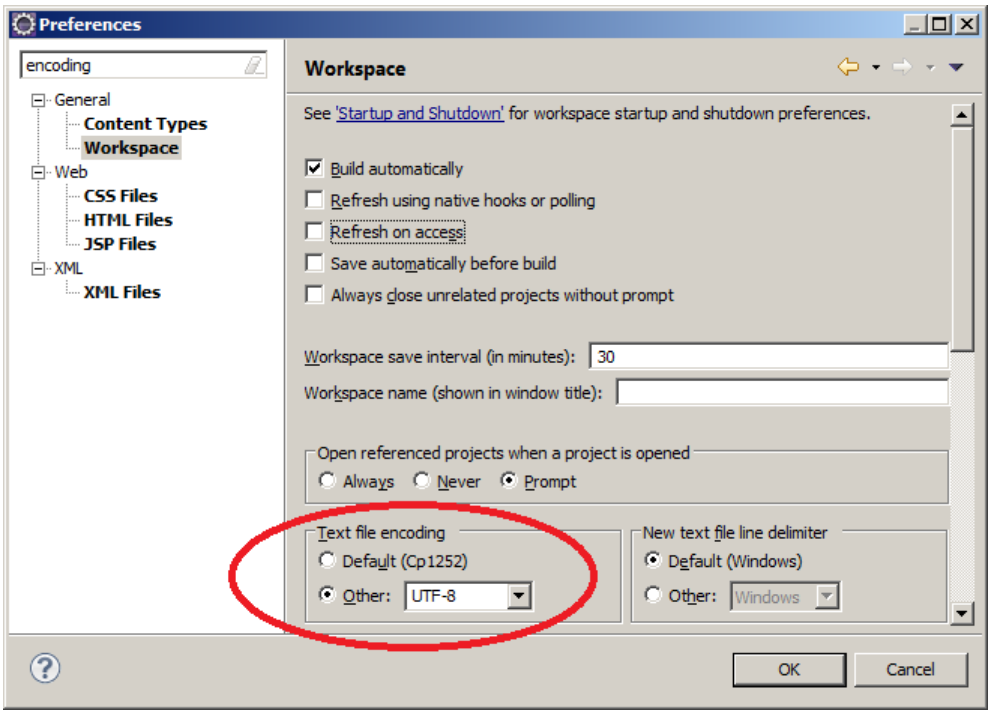
```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.5.1</version>
  <configuration>
    <encoding>UTF-8</encoding>
  </configuration>
</plugin>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <encoding>UTF-8</encoding>
  </configuration>
</plugin>
```

Configure Eclipse to UTF-8

When you are working with XML files in Eclipse, you do not need to configure Eclipse for UTF-8, because UTF-8 is the default for XML files. But the default encoding for other file types is the encoding of the file system. So, it is recommended to set the encoding for the entire workspace to UTF-8:



This default can be changed for each file.

Unicode in Error Messages

If tests of Unicode content fail, the error message may be presented incorrectly in Eclipse or in a browser. Again the file encoding is responsible for this behaviour. Configuring ANT to “UTF-8” should solve all your problems. Only characters from the encoding “UTF-16” may corrupt the presentation of the error message.

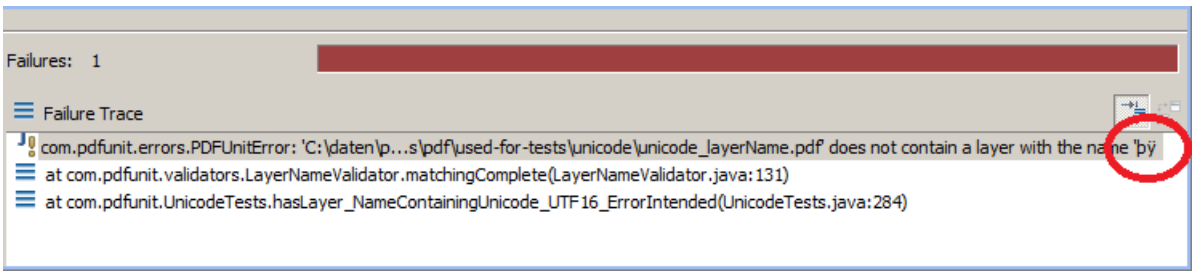
The PDF document in the next example includes a layer name containing UTF-16BE characters. To show the impact of Unicode characters in error messages, the expected layer name in the test is intentionally incorrect to produce an error message:

```
/**
 * The name of the layers consists of UTF-16BE and contains the
 * byte order mark (BOM). The error message is not complete.
 * It was corrupted by the internal Null-bytes.
 */
@Test
public void hasLayer_NameContainingUnicode_UTF16_ErrorIntended() throws Exception {
  String filename = "documentUnderTest.pdf";

//  String layername = "Ebene 1(4)"; // This is shown by Adobe Reader®,
//                     "Ebene _XXX"; // and this is the used string
  String wrongNameWithUTF16BE =
  "\u00fe\u00ff\u0000E\u0000b\u0000e\u0000n\u0000e\u0000 \u0000_XXX";

  AssertThat.document(filename)
            .hasLayer()
            .equalsTo(wrongNameWithUTF16BE);
  ;
}
```

When the tests are executed with ANT, a browser shows the complete error message including the trailing string þÿEbene _XXX:

```
'C:\daten\p...s\pdf\used-for-tests\unicode\unicode_layerName.pdf' does not
contain a layer with the name 'þÿEbene _XXX'.

junit.framework.AssertionFailedError: 'C:\daten\p...s\pdf\used-for-tests\unicode
\unicode_layerName.pdf' does not contain a layer with the name 'þÿEbene _XXX'.
at com.pdfunit.validators.LayerNameValidator.matchingComplete(LayerNameValidator.java:133)
at com.pdfunit.UnicodeTests.hasLayer_NameContainingUnicode_UTF16_ErrorIntended(UnicodeTests.java:274)
```

But the JUnit-View in Eclipse cuts the error message after the internal Byte-Order-Mark. The message `'...\unicode_layerName.pdf' does not contain a layer with the name 'þÿ'` should end with Ebene_XXX:



Unicode for invisible Characters -

A problem can occur due to a “non-breaking space”. Because at first it looks like a normal space, the comparison with a space fails. But when using the Unicode sequence of the “non-breaking space” (`\u00A0`) the test runs successfully. Here's the test:

```
@Test
public void nodeValueWithUnicodeValue() throws Exception {
  String filename = "documentUnderTest.pdf";

  DefaultNamespace defaultNS = new DefaultNamespace("http://www.w3.org/1999/xhtml");
  String nodeValue = "The code ... the button's";
  String nodeValueWithNBSP = nodeValue + "\u00A0"; // The content terminates with a NBSP.
  XMLNode nodeP7 = new XMLNode("default:p[7]", nodeValueWithNBSP, defaultNS);

  AssertThat.document(filename)
            .hasXFAData()
            .withNode(nodeP7)
  ;
}
```

Chapter 12. Installation, Configuration, Update

12.1. Technical Requirements

PDFUnit needs Java 7 or higher.

If you run PDFUnit with ANT, Maven or other tools, you have to install those tools independently from PDFUnit.

Tested Environments

PDFUnit was successfully tested in these environments:

Operating System

- Windows-7, 32 + 64 Bit
- Kubuntu Linux 12/04, 32 + 64 Bit
- Mac OS X, 64 Bit

Java Version

- Oracle JDK-1.7, 32 + 64 Bit
- Oracle JDK-1.8, Windows, 32 + 64 Bit
- IBM J9, R26_Java726_SR4, Windows 7, 64 Bit

More combinations of Java and operating systems will be tested in the future.

If you have any problems with the installation, please contact us at info[at]pdfunit.com.

12.2. Installation

Download of PDFUnit-Java

Download the file `pdfunit-java-VERSION.zip` from the project site: . If you have purchased a license, you get a new ZIP file by mail.

Unzip the file in a folder, for example into `PROJECT_HOME/lib/pdfunit-java-VERSION`. In the following text that folder is refered to as `PDFUNIT_HOME`.

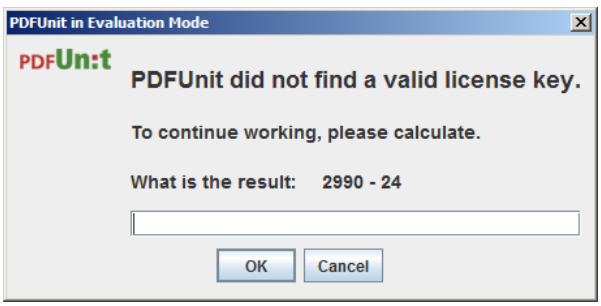
Configuring Classpath

All JAR files which are distributed by PDFUnit need to be included in the classpath. Also the file `pdfunit.config`.

If you have a licensed PDFUnit, add the license key file `license-key_pdfunit-java.lic` to the classpath too.

Using PDFUnit without a License Key

You are welcomed to evaluate PDFUnit. In this case, a message box appears showing a simple math calculation you have to solve. If you calculate successfully the test will run, otherwise you have to restart your test and calculate again.



Sometimes, the message box is covered by other applications. Then the ANT or Maven script is blocked. Minimize all applications to look for the message box.

Order a License Key

If you use PDFUnit in a commercial context you need a license. Write a mail to info[at]pdfunit.com and ask for a license. You will receive an answer as soon as possible.

The license fee is calculated individually. A small company should not pay as much as a big company, and someone testing only a few PDF documents, of course, pays less. And if you want to get a free license, give us some reasons. It is not impossible.

Use License Key

If you have ordered a license you will receive a ZIP file and a separate file `license-key_pdfunit-java.lic`. Install the content of the ZIP file as described above and include the license file in your classpath.

Any change to the license file makes it unusable. If this happens contact PDFUnit.com and ask for a new license file.

Verify the Installation

If you have a problem with the configuration start the script `verifyInstallation.bat` or `verifyInstallation.sh`. You will get a detailed problem analysis. See chapter 12.6: “Verifying the Configuration” (p. 151).

12.3. Setting Classpath in Eclipse, ANT, Maven

All development environments need a classpath with these entries:

- all JAR files delivered by PDFUnit
- the file `pdfunit.config`
- the file `license-key_pdfunit-java.lic` if PDFUnit is used with a license

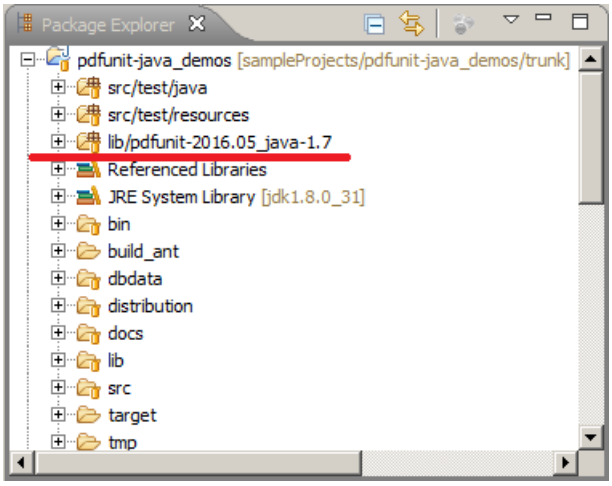
If PDFUnit does not find the files it will give error messages like these:

- `Could not find 'pdfunit.config'. Verify classpath and installation.`
- `No valid license key found. Switching to evaluation mode. Contact PDFUnit.com if you are interested in a license.`
- `A field of the license-key-file could not be parsed. Do you have the correct license-key file? Check your classpath and PDFUnit version. Please, read the documentation.`

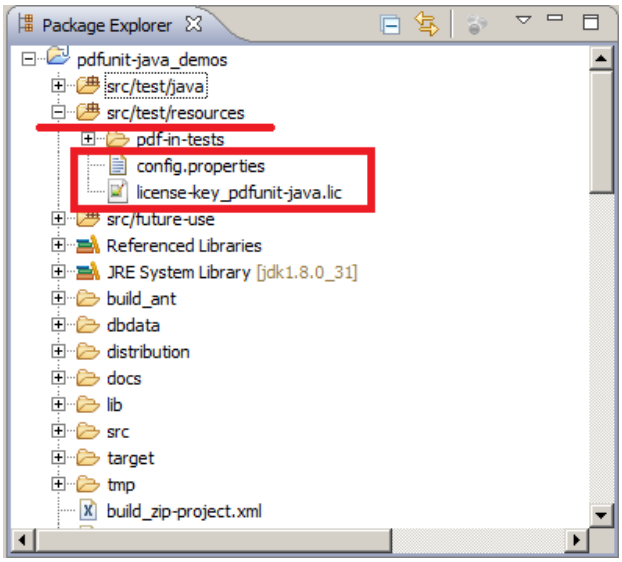
In the following examples show how to configure the classpath in different environments. Additionally chapter 12.4: “Set Paths Using System Properties” (p. 150) describes how to use system properties to declare the location of the files `pdfunit.config` and `license-key_pdfunit-java.lic`.

Configuring Eclipse

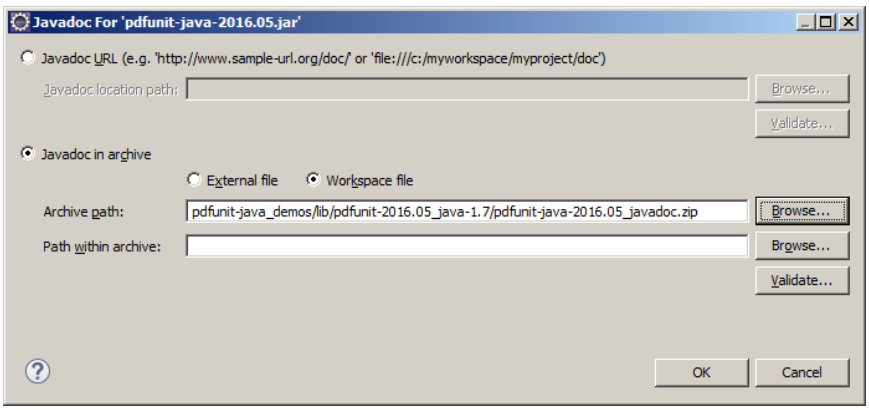
A simple way to configure Eclipse for PDFUnit is to include the installation directory `PDFUNIT_HOME` and all JAR files individually in the classpath:



Another option is to move the file `pdfunit.config` out of `PROJECT_HOME` into another folder, for example `src/test/resources`, and put that folder into the classpath.



PDFUnit includes the file `PDFUNIT_HOME/pdfunit-java-VERSION_javadoc.zip` which can be registered in Eclipse to let Eclipse show the Javadoc comments.



Configuring ANT

Many options exist to configure ANT for PDFUnit. In all options the JAR files of `PDFUNIT_HOME` and `PDFUNIT_HOME/lib/*` have to be put into the classpath. Additionally the file `pdfunit.config` must be included in classpath.

If you have not changed the `pdfunit.config`, it is simple to include `PDFUNIT_HOME` itself in the classpath additionally to the JAR files as is shown by the following listing:

```
<!--
  It is important to have the directory of PDFUnit itself in the classpath,
  because the file 'pdfunit.config' must be found.
-->
<property name="dir.build.classes"          value="build/classes" />
<property name="dir.external.tools"         value="lib-ext" />
<property name="dir.external.tools.pdfunit" value="lib-ext/pdfunit-2016.05" />

<path id="project.classpath">
  <pathelement location="${dir.external.tools.pdfunit}" />
  <pathelement location="${dir.build.classes}" />

  <!-- If there are problems with duplicate JARs, use more detailed filesets: -->
  <fileset dir="${dir.external.tools}">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

The file `pdfunit.config` can be moved to an individual folder, for example into `src/test/resources`. This way is recommended if you want to change the config file. How to configure PDFUnit is described in chapter 12.5: “Using the pdfunit.config File” (p. 150). Then the classpath in ANT looks like this:

```
<path id="project.classpath">
  <!--
  The file 'pdfunit.config' should not be located more than once in
  the classpath, because it hurts the DRY principle.
  -->
  <pathelement location="src/test/resources" />
  <pathelement location="${dir.external.tools.pdfunit}" />
  <pathelement location="${dir.build.classes}" />

  <!-- If there are problems with duplicate JARs, use more detailed fileset: -->
  <fileset dir="${dir.external.tools}">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

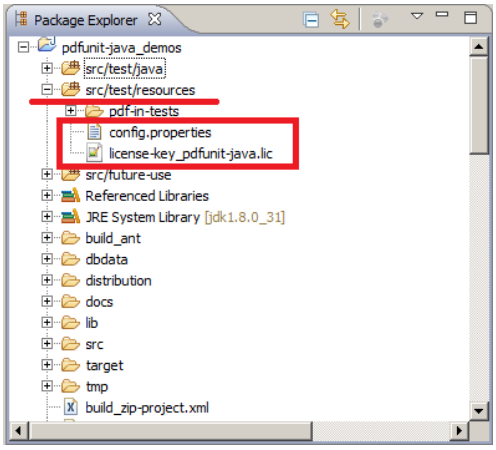
Configuring Maven

The current release of PDFUnit (2016.05) is not provided in a public Maven repository. To use it with Maven despite this fact, you have to install it into a local or company-wide repository. You can do it with the following command. Change to the directory `PDFUNIT_HOME\lib` and run this command:

```
mvn install:install-file -Dfile=<PATH_TO>pdfunit-java-VERSION.jar -DpomFile=<PATH_TO>pom.xml
```

The next step is to copy `pdfunit.config` into the directory `src/test/resources`.

The following image shows the resulting project layout:



Register the PDFUnit library to your `pom.xml`.

```
<dependency>
  <groupId>com.pdfunit</groupId>
  <artifactId>pdfunit</artifactId>
  <version>2016.05</version>
  <scope>compile</scope>
</dependency>
```

Last Step for Licensed PDFUnit

The license file `license-key_pdfunit-java.lic` must also be included in the classpath. Otherwise the message box with the math calculation is shown.

12.4. Set Paths Using System Properties

The files `pdfunit.config` and the license key file can be placed outside of the classpath. But then their location has to be declared by Java system properties. The keys of the properties are:

- `-Dpdfunit.configfile`
- `-Dpdfunit.licensekeyfile`

Dependent on your test system (Eclipse, ANT or Maven) these parameters can be set in many ways. Use the common information to see, how Java system properties are set in your specific environment. A less known option is the operating system environment variable `_JAVA_OPTIONS` which works for all test systems:

```
set _JAVA_OPTIONS=-Dpdfunit.configfile=..\myfolder\pdfunit.config
```

If you have questions about this topic, write a mail to info[at]pdfunit.com.

12.5. Using the pdfunit.config File

Typically PDFUnit does not need to be configured, but the file `pdfunit.config` gives you the chance to do so. The following sections show you how:

Locale of PDF Documents

Java needs a locale when working with date values and for the conversion of strings into lowercases too. PDFUnit reads the value of the locale from `pdfunit.config`. All the constants defined in `java.util.Locale` are allowed. The default value is `en` (English).

```
################################################################################
# Locale of PDF documents, required by some tests.
################################################################################
pdf.locale = en
#pdf.locale = de_DE
#pdf.locale = en_UK
```

You can write all values in lowercase or uppercase. PDFUnit also accepts underscore and hyphen as a delimiter between language and country.

If you delete the key `pdf.locale` from `pdfunit.config` by accident, then the default locale from the Java-runtime is taken (`Locale.getDefault()`).

Output Folder for Error Images

When a difference is detected while comparing rendered pages of two PDF documents, a diff image is created. It shows the referenced document on the left and differences to the actual test document on the right side. The differences are shown in red. The name of the test is placed above the image.

You can configure the output directory in the `pdfunit.config`. By default the diff images will be stored in the directory containing the test PDF. That might be useful for some projects. But when you want to have a fixed folder for all diff images, you can configure that behaviour by using the property `diffimage.output.path.files`:

```
################################################################################
#
# The path can be absolute or relative. The base of a relative path depends
# on the tool which starts the junit tests (Eclipse, ANT, etc.).
# The path must end with a slash. It must exist before you run the tests.
#
# If this property is not defined, the directory containing the PDF
# files is used.
#
################################################################################
diffimage.output.path.files = ./
```

12.6. Verifying the Configuration

Verifying with a Script

The installation of PDFUnit can be checked using a special program, started with the script `verifyInstallation.bat` or `verifyInstallation.sh`:

```
::
:: Verify the installation of PDFUnit
::

set CURRENTDIR=%~dp0
set PDFUNIT_HOME=%CURRENTDIR%

::
:: Change the installation directories depending on your situation:
::
set ASPECTJ_HOME=%PDFUNIT_HOME%/lib/aspectj-1.8.7
set BOUNCYCASTLE_HOME=%PDFUNIT_HOME%/lib/bouncycastle-jdk15on-153
set JAVASSIST_HOME=%PDFUNIT_HOME%/lib-ext/javassist-3.20.0-GA
set JUNIT_HOME=%PDFUNIT_HOME%/lib/junit-4.12
set COMMONSCOLLECTIONS_HOME=%PDFUNIT_HOME%/lib/commons-collections4-4.1
set COMMONSLOGGING_HOME=%PDFUNIT_HOME%/lib/commons-logging-1.2
set PDFBOX_HOME=%PDFUNIT_HOME%/lib/pdfbox-2.0.0
set TESS4J_HOME=%PDFUNIT_HOME%/lib/tess4j-3.1.0
set VIP_HOME=%PDFUNIT_HOME%/lib/vip-1.0.0
set ZXING_HOME=%PDFUNIT_HOME%/lib/zxing-core-3.2.1

set CLASSPATH=
set CLASSPATH=%ASPECTJ_HOME%/*;%CLASSPATH%
set CLASSPATH=%BOUNCYCASTLE_HOME%/*;%CLASSPATH%
set CLASSPATH=%COMMONSCOLLECTIONS_HOME%/*;%CLASSPATH%
set CLASSPATH=%COMMONSLOGGING_HOME%/*;%CLASSPATH%
set CLASSPATH=%JAVASSIST_HOME%/*;%CLASSPATH%
set CLASSPATH=%JUNIT_HOME%/*;%CLASSPATH%
set CLASSPATH=%PDFBOX_HOME%/*;%CLASSPATH%
set CLASSPATH=%TESS4J_HOME%/*;%CLASSPATH%
set CLASSPATH=%TESS4J_HOME%/lib/*;%CLASSPATH%
set CLASSPATH=%VIP_HOME%/*;%CLASSPATH%
set CLASSPATH=%ZXING_HOME%/*;%CLASSPATH%

:: The folder of PDFUnit-Java:
set CLASSPATH=%PDFUNIT_HOME%/build_ant/classes;%CLASSPATH%
:: The JAR files of PDFUnit-Java:
set CLASSPATH=%PDFUNIT_HOME%*;%CLASSPATH%


:: Run installation verification:
java org.verifyinstallation.VIPMain  --in   pdfunit_development.vip
                                     --out  verifyInstallation_result.html
                                     --xslt ./lib/vip-1.0.0/vip-java_simple.xslt
```

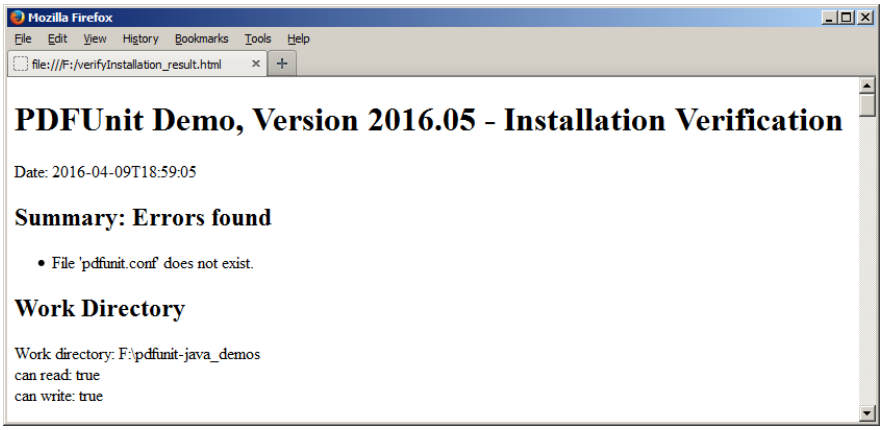
You have do edit the paths depending on your installation.

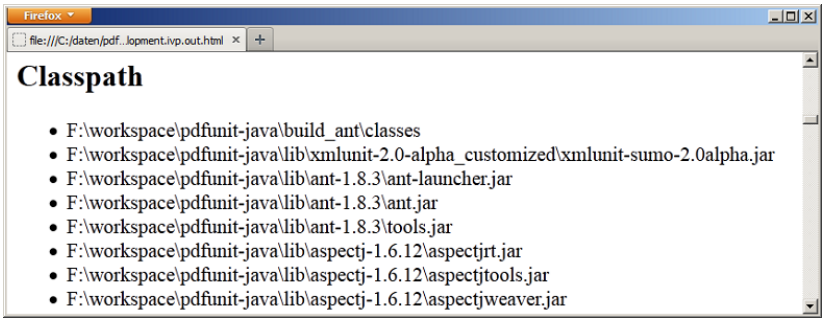
The stylesheet option is provided to use individual stylesheets. If you don't use a stylesheet, a simple one is used automatically.

The script produces the following output on the console:

```
Checking installation ...
... finished. Report created, see 'verifyInstallation_result.html'.
```

The resulting report file shows errors and information about the classpath, environment variables and other runtime related data:





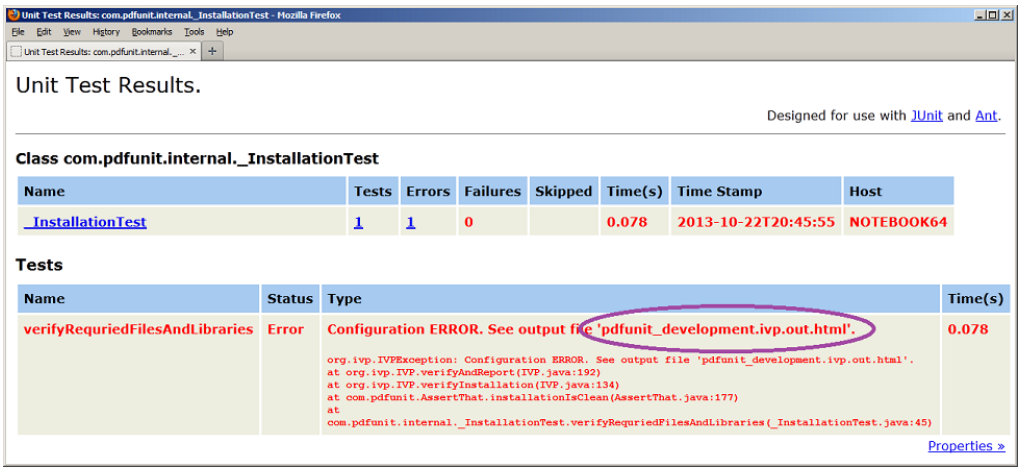
Verifying as a Unit Test

The verification of the installation can also be done as a unit test. That makes it possible to visualize the system environment of the current tests in the context of ANT, Maven or Jenkins.

Within a simple unit test you can use a special method:

```
/*
 * The method verifies that all required libraries and files are found on the
 * classpath. Additionally it logs some system properties and writes
 * all to System.out.
 */
@Test
public void verifyRequriedFilesAndLibraries() throws Exception {
  AssertThat.installationIsClean("pdfunit_development.vip");
}
```

The method performs the same internal checks as the script described above. If a configuration error exists, the test is “red” and the error message points to the created HTML file:



The report file contains the same data as when it was created by a shell script.

12.7. Installation of a New Release

The installation of a new release of PDFUnit-Java runs just like the initial installation, because PDFUnit is always delivered as a full release, never as an incremental release.

Getting the New Release

If you use PDFUnit without a license, download the new ZIP file from the internet: .

If you use PDFUnit with a license, you will receive a new release by mail with an attached ZIP file and a license file.

First Steps for all Development Environments

Before you start installing the new release, run all existing unit tests with the old release. They should be “green”.

Save your project.

Install the Update

Unzip the new release, but not into the existing project folder. In the following text the folder with your new release is called `PDFUNITJAVA_HOME_NEW` and the project folder is called `PROJECT_HOME`.

Delete the folder `PROJECT_HOME/lib/pdfunit-OLD-VERSION`.

Copy the directory `PDFUNITJAVA_HOME_NEW` to the folder, where the old release was located, for example to `PROJECT_HOME/lib/pdfunit-NEW-VERSION`.

If you had placed the file `pdfunit.config` in a different folder than the installation folder, copy the corresponding new file from `PROJECT_HOME/lib/pdfunit-NEW-VERSION` to that folder where it was located in the old release.

If you used the file `pdfunit.config` with individual settings, transfer the changes to the `pdfunit.config` of the new release.

If you use PDFUnit with a license, copy the new license file `license-key_pdfunit-java.lic` to the folder containing the old release.

Next Steps in ANT

No configuration steps are necessary for ANT if you have declared the classpath as described before in this chapter.

Next Steps in Maven

The new release has to be installed into your local or company-wide repository. Open a shell, change to the directory `PROJECT_HOME/lib/pdfunit-NEW-VERSION` and submit the following command:

```
mvn install:install-file -Dfile=pdfunit-java-VERSION.jar -DpomFile=pom.xml
```

Next Steps in Eclipse

Include all new JAR files in the build path. Remove the old JAR files from the build path. Eclipse should not show any build path error.

Register the Javadoc documentation again as described in chapter : “Configuring Eclipse” (p. 147).

Last Step

Run the existing tests with the new release. If there are no documented incompatibilities between the new and the old release, the tests should run “green” again. Otherwise read the release information.

12.8. Uninstall

PDFUnit can be uninstalled cleanly by deleting the installation directories. PDFUnit doesn't create any entries in the registry or system directories, so none need to be removed. Don't forget to remove the references to JAR files and PDFUnit directories from your own scripts.

Chapter 13. Appendix

13.1. Instantiation of PDF Documents

The following list shows the methods and data types to read PDF documents for tests:

```
// Possibilities to instantiate PDFUnit with a test PDF:
AssertThat.document(String      pdfDocument)
AssertThat.document(File        pdfDocument)
AssertThat.document(URL         pdfDocument)
AssertThat.document(InputStream pdfDocument)
AssertThat.document(byte[]      pdfDocument)

// The same with a password when the PDF is encrypted:
AssertThat.document(String      pdfDocument, String password)
AssertThat.document(File        pdfDocument, String password)
AssertThat.document(URL         pdfDocument, String password)
AssertThat.document(InputStream pdfDocument, String password)
AssertThat.document(byte[]      pdfDocument, String password)

// Instantiate a test PDF and a reference PDF:
AssertThat.document(..).and(pdfReference)                  ❶
AssertThat.document(..).and(pdfReference, String password)  ❷

// Instantiate an array of test documents:
AssertThat.eachDocument(String      pdfDocument)
AssertThat.eachDocument(File        pdfDocument)
AssertThat.eachDocument(URL         pdfDocument)
AssertThat.eachDocument(InputStream pdfDocument)

// Instantiate an array of password protected test documents:
AssertThat.eachDocument(String      pdfDocument, String password)
AssertThat.eachDocument(File        pdfDocument, String password)
AssertThat.eachDocument(URL         pdfDocument, String password)
AssertThat.eachDocument(InputStream pdfDocument, String password)

// Instantiate PDF documents in a folder:
AssertThat.eachDocument().inFolder(..)                     ❸
```

❶❷ Also, a referenced PDF document can be read from a `String`, `File`, `URL`, `InputStream` or `byte[]`.

❸ PDFUnit collects all PDF files in the given folder and runs the test with each of them. The process stops if one of the PDF files does not pass the test.

If documents are password protected, PDFUnit needs either the “user password” or the “owner password” to open the file.

13.2. Page Selection

Predefined Pages

Constants help your tests to focus on specific pages in a PDF document. Their names clearly express their intent:

```
// Possibilities to focus tests to specific pages:

com.pdfunit.Constants.ANY_PAGE      com.pdfunit.Constants.ON_ANY_PAGE
com.pdfunit.Constants.EVEN_PAGES    com.pdfunit.Constants.ON_EVEN_PAGES
com.pdfunit.Constants.EACH_PAGE     com.pdfunit.Constants.ON_EACH_PAGE
com.pdfunit.Constants.EVERY_PAGE    com.pdfunit.Constants.ON_EVERY_PAGE
com.pdfunit.Constants.FIRST_PAGE    com.pdfunit.Constants.ON_FIRST_PAGE
com.pdfunit.Constants.LAST_PAGE     com.pdfunit.Constants.ON_LAST_PAGE
com.pdfunit.Constants.ODD_PAGES     com.pdfunit.Constants.ON_ODD_PAGES
```

The constants on the left-hand side have the same meaning as those on the right-hand side. The right-hand side is supported to be compatible with older releases.

Here an example using one of the constants:

```
@Test
public void hasText_MultipleSearchTokens_EvenPages() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(EVEN_PAGES)
            .hasText()
            .containing("Content", "even pagenumber")
  ;
}
```

Individual Pages

The next example shows how individual pages can be addressed: Page numbers must be separated by commas.

```
@Test
public void hasText_OnMultiplePages() throws Exception {
  String filename = "documentUnderTest.pdf";
  PagesToUse pages123 = PagesToUse.getPages(1, 2, 3);

  AssertThat.document(filename)
            .restrictedTo(pages123)
            .hasText()
            .containing("Content on")
  ;
}
```

Two methods are available to select a single page or a set of pages:

```
// How to define individual pages:

PagesToUse.getPage(2);
PagesToUse.getPages(1, 2, 3);
```

Open Ranges

Various methods are available to define a ranges of pages at the beginning or at the end of a document:

```
// How to define open ranges:

ON_ANY_PAGE.after(2);
ON_ANY_PAGE.before(3);

ON_EVERY_PAGE.after(2);
ON_EVERY_PAGE.before(2);
```

The lower- or upper-limits are excluded from the expected date range.

The following example validates a PDF document starting with page 3.

```
@Test
public void compareFormat_OnEveryPageAfter() throws Exception {
  String filename = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";
  PagesToUse pagesAfter2 = ON_ANY_PAGE.after(2);

  AssertThat.document(filename)
            .and(filenameReference)
            .restrictedTo(pagesAfter2)
            .haveSameFormat()
  ;
}
```

Inner Ranges

And finally, it's possible to define a range of pages inside a document using `PagesToUse.spanningFrom().to()`.

```
@Test
public void hasText_SpanningOver2Pages() throws Exception {
  String filename = "documentUnderTest.pdf";
  String textOnPage1 = "Text starts  on page 1 and ";
  String textOnPage2 = "continues on page 2";
  String expectedText = textOnPage1 + textOnPage2;
  PagesToUse pages1to2 = PagesToUse.spanningFrom(1).to(2);

  // Mark the section without header and footer:
  int leftX  =  18;
  int upperY =  30;
  int width  = 182;
  int height = 238;
  PageRegion regionWithoutHeaderAndFooter = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filename)
            .restrictedTo(pages1to2)
            .restrictedTo(regionWithoutHeaderAndFooter)
            .hasText()
            .containing(expectedText)
  ;
}
```

Important Hints

- Page numbers begin with '1'.
- The page number in `before(int)` and `after(int)` are both exclusive.
- The page number in `from(int)` and `to(int)` are both inclusive.
- `ON_EVERY_PAGE` means that the expected text has to exist on each page in the given range.
- When using `ON_ANY_PAGE`, a test is successful if the expected string exists on one or more pages in the given range.
- The use of `restrictedTo(PagesToUse)` and `restrictedTo(PageRegion)` has no effect on methods applied to the full document.

13.3. Defining Page Areas

Comparing text or rendered pages can be restricted to regions of one or more pages. Such an area is defined by four values: the x/y values of the **upper left** corner, the width and the height:

```
// Instantiating a page region
public PageRegion(int leftX, int upperY, int width, int height)   ❶
public PageRegion(int leftX, int upperY, int width, int height, FormatUnit init)
```

❶ If no unit is set, PDFUnit uses the unit `MILLIMETERS`.

Page regions can be defined using millimeters or points. Further information to the units are described in chapter 13.8: “Format Units - Points and Millimeters” (p. 163).

Here's an example:

```
@Test
public void hasTextOnFirstPage_InPageRegion() throws Exception {
  String filename = "documentUnderTest.pdf";

  int leftX  =  17;  // in millimeter
  int upperY =  45;
  int width  =  60;
  int height =   9;
  PageRegion pageRegion = new PageRegion(leftX, upperY, width, height);

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .restrictedTo(pageRegion)
            .hasText()
            .containing("on first")
  ;
}
```

It's easy to use a region of a page in a test. But it might be more difficult to find the right values for the region you need. PDFUnit provides the utility `RenderPdfPageRegionToImage` to extract a page section into an image file (PNG). You can use that program using `mm`:

```
::
:: Render a part of a PDF page into an image file.
::

@echo off
setlocal
set CLASSPATH=./lib/aspectj-1.8.7/*;%CLASSPATH%
set CLASSPATH=./lib/commons-logging-1.2/*;%CLASSPATH%
set CLASSPATH=./lib/pdfbox-2.0.0/*;%CLASSPATH%
set CLASSPATH=./lib/pdfunit-2016.05/*;%CLASSPATH%
set CLASSPATH=./build_eclipse;%CLASSPATH%

set TOOL=com.pdfunit.tools.RenderPdfPageRegionToImage
set PAGENUMBER=1
set OUT_DIR=./tmp
set IN_FILE=./content/documentForTextClipping.pdf
set PASSWD=

:: All values must be millimeter:
set UPPERLEFTX=135
set UPPERLEFTY=30
set WIDTH=70
set HEIGHT=30
set PAGEHEIGHT=297

java  %TOOL%   %IN_FILE%   %PAGENUMBER%   %OUT_DIR%
      %UPPERLEFTX%   %UPPERLEFTY%   %WIDTH%   %HEIGHT%   %PAGEHEIGHT%
      %PASSWD%
endlocal
```

The generated image needs to be checked. Does it contain the section you want? If not, change the parameters until they are right. Then you can copy the four values into your test.

13.4. Comparing Text

Expected text and actual text on a PDF page can be compared using the following methods:

```
// Methods with configurable whitespace processing. Default is NORMALIZE:
.containing(searchToken                                    )   ❶
.containing(searchToken             , WhitespaceProcessing)   ❷
.containing(String[] searchTokens                          )
.containing(String[] searchTokens   , WhitespaceProcessing)
.endingWith(searchToken                                    )
.endingWith(searchToken             , WhitespaceProcessing)
.equalsTo(searchToken                                      )
.equalsTo(searchToken               , WhitespaceProcessing)
.first(searchToken                                         )
.first(searchToken                  , WhitespaceProcessing)   ❸
.notContaining(searchToken                                 )
.notContaining(searchToken          , WhitespaceProcessing)
.notContaining(String[] searchTokens                       )
.notContaining(String[] searchTokens, WhitespaceProcessing)
.startingWith(searchToken                                  )
.startingWith(searchToken           , WhitespaceProcessing)
.then(searchToken)                                             ❹

// Methods with whitespace processing NORMALIZE:
.notEndingWith(searchToken)
.notStartingWith(searchToken)

// Methods without whitespace processing:
.matchingRegex(regex)
.notMatchingRegex(regex)
```

❶ Methods **without** the second parameter normalize the whitespaces. That means whitespaces at the beginning and the end are removed and all sequences of any whitespace within a text are reduced to one space.

❷ The processing of whitespaces in these methods is controlled by the second parameter. For this parameter, the constants `IGNORE`, `NORMALIZE`, and `KEEP` exist. The constants are ex-

plained separately in section 13.5: “Whitespace Processing” (p. 159). They can be used in all methods with 'WhitespaceProcessing' as a second parameter.

❸❹ The method `then(..)` always processes whitespaces in the same way as `first(..)`.

Comparisons with regular expressions follow the rules and possibilities of the class `java.util.regex.Pattern`:

```
// Using regular expression to compare page content
@Test
public void hasText_MatchingRegex() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .matchingRegex(".*[Cc]ontent.*")
  ;
}
```

The methods `containing(String[])` and `notContaining(String[])` can be called with multiple search terms. A test with `containing(String[])` is considered successful if each expected term appears on every selected page. A test with `notContaining(String[])` is considered successful if none of the terms exist on any of the selected pages:

```
@Test
public void hasText_NotContaining_MultipleSearchTokens() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .notContaining("even pagenumber", "Page #2")
  ;
}
```

13.5. Whitespace Processing

Almost all tests compare strings. Many comparisons would fail if whitespaces remained as they are. So, you can control the way whitespaces are handled using one of the following constants. `NORMALIZE_WHITESPACES` is the default if nothing is declared:

```
// Constants for whitespace processing:

com.pdfunit.Constants.IGNORE_WHITESPACES    ❶
com.pdfunit.Constants.IGNORE                ❷

com.pdfunit.Constants.KEEP_WHITESPACES      ❸
com.pdfunit.Constants.KEEP                  ❹

com.pdfunit.Constants.NORMALIZE_WHITESPACES ❺
com.pdfunit.Constants.NORMALIZE             ❻
```

❶❷ All whitespaces are deleted before comparing two strings.

❸❹ Existing whitespaces are not changed.

❺❻ Whitespaces at the beginning and at the end of a string are deleted. Any sequences of whitespaces within a text are reduced to one space.

Each set of two constants has the same meaning. The redundancy is provided to support different linguistic preferences.

An example:

```
@Test
public void hasText_WithLineBreaks_UsingIGNORE() throws Exception {
  String filename = "documentUnderTest.pdf";

  String expected = "PDFUnit - Automated PDF Tests http://pdfunit.com/" +
                    "This is a document that is used for unit tests of PDFUnit itself." +
                    "Content on first page." +
                    "odd pagenumber" +
                    "Page # 1 of 4";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .equalsTo(expected, IGNORE_WHITESPACES)
  ;
}
```

The expected string in this example is written completely without linebreaks, although the PDF page contains many of them. However when using `IGNORE_WHITESPACES` the test runs successfully.

`NORMALIZE_WHITESPACES` is the default when nothing else is set explicitly. Test methods in which a flexible treatment of whitespaces does not make sense do not have a second parameter.

As an exception to this rule, no method involving regular expressions changes whitespaces automatically. It is up to you to integrate the whitespace processing into the regular expression, for example like this:

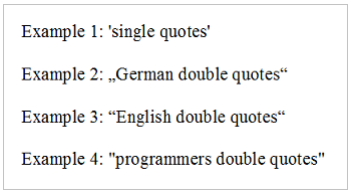
```
(?ms).*print(.*)
```

The term `(?ms)` means that the search extends over multiple lines. Line breaks are interpreted as characters.

13.6. Single and Double Quotation Marks inside Strings

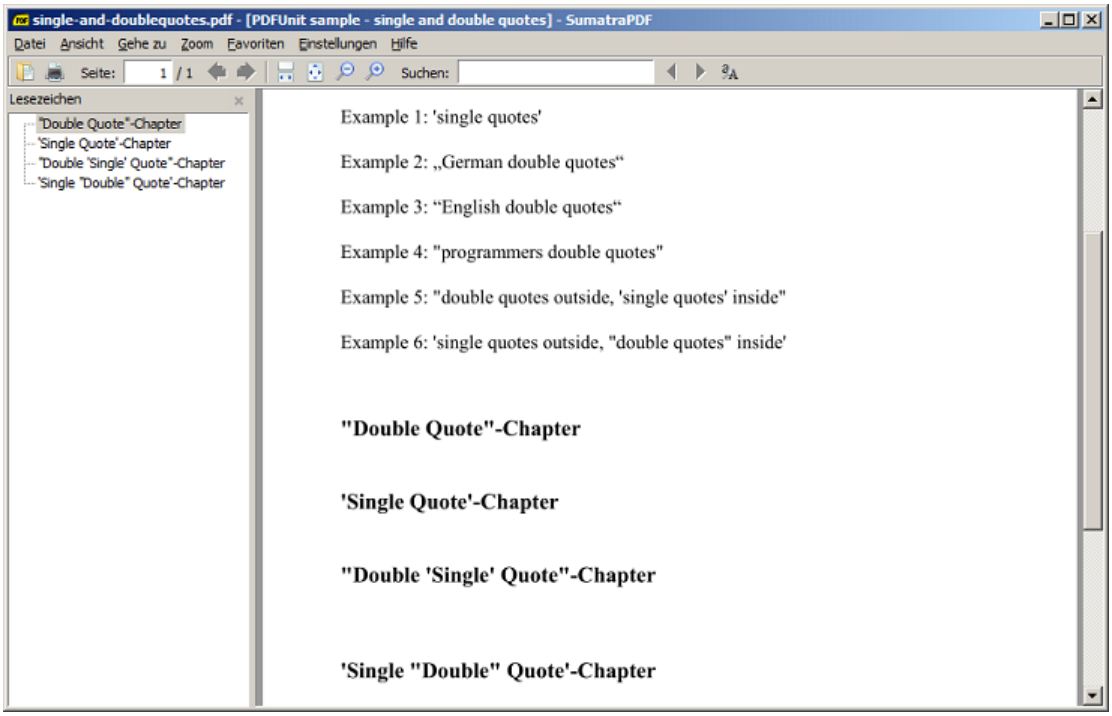
Different Types of Quotes

Important Notice: The term “Quote” has different meanings as the following picture shows:



“English“ and „German“ style quotation marks do not disturb the execution of tests. But during the creation of a test you could have problems typing them with your editor. Hint: copy the required quotation marks from a word-processor or an exisiting PDF document and paste them into your file.

The "programmers double quotes" need special attention because they are used as string delimiters in Java. The following paragraphs and examples go into detail about this. They are all based on the following document:



Valid Examples

'Single-Quotes', “English“, and „German“ quotation marks within strings cause no problems. But "Double-Quotes" have to be escaped with a backslash:

```
@Test
public void hasText_SingleQuotes() throws Exception {
  String filename = "documentUnderTest.pdf";

  String expected = "Example 1: 'single quotes'";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(expected)
  ;
}
```

```
@Test
public void hasText_GermanDoubleQuotes() throws Exception {
  String filename = "documentUnderTest.pdf";

  String expected = "Example 2: „German double quotes“";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(expected)
  ;
}
```

```
@Test
public void hasText_EnglishDoubleQuotes() throws Exception {
  String filename = "documentUnderTest.pdf";

  String expected = "Example 3: “English double quotes“";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(expected)
  ;
}
```

```
@Test
public void hasText_ProgrammersDoubleQuotes() throws Exception {
  String filename = "documentUnderTest.pdf";

  String expected = "Example 4: \"programmers double quotes\"";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(expected)
  ;
}
```

```
@Test
public void hasText_DoubleAndSingleQuotes_1() throws Exception {
  String filename = "documentUnderTest.pdf";

  String expected = "Example 5: \"double quotes outside, 'single quotes' inside\"";
  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(expected)
  ;
}
```

```
@Test
public void hasText_DoubleAndSingleQuotes_2() throws Exception {
  String filename = "documentUnderTest.pdf";
  String expected = "Example 6: 'single quotes outside, \"double quotes\" inside'";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .containing(expected)
  ;
}
```

```
@Test
public void matchingRegex_DoubleQuotes() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .restrictedTo(FIRST_PAGE)
            .hasText()
            .matchingRegex(".*\"double.*\".*")
  ;
}
```

Quotes in XPath

Strings which are used as XPath expressions must **not contain** both single and double quotes in any combination. This condition is not fulfilled in the following example:

```
@Test
public void matchingXPath_DoubleQuotes_1() throws Exception {
  String filename = "documentUnderTest.pdf";
  String xpath = "count(//Title[.='\"Double Quote\"-Chapter']) = 1";
  XPathExpression xpathExpression = new XPathExpression(xpath);

  AssertThat.document(filename)
            .hasBookmarks()
            .matchingXPath(xpathExpression)
  ;
}
```

The error can only be avoided when you improve the XPath-expression like this:

```
@Test
public void matchingXPath_DoubleQuotes_2() throws Exception {
  String filename = "documentUnderTest.pdf";
  String xpath1 = "count(//Title[contains(., 'Double Quote')]) = 1";
  String xpath2 = "count(//Title[contains(., 'Chapter')]) = 4";
  XPathExpression xpathExpression1 = new XPathExpression(xpath1);
  XPathExpression xpathExpression2 = new XPathExpression(xpath2);

  AssertThat.document(filename)
            .hasBookmarks()
            .matchingXPath(xpathExpression1)
            .matchingXPath(xpathExpression2)
  ;
}
```

13.7. Date Resolution

PDFUnit is able to compare dates (creation and modification dates) as year-month-day or additionally hour-minute-second. Two constants are available to choose between these options:

```
// Constants for date resolutions:

com.pdfunit.Constants.AS_DATE
com.pdfunit.Constants.AS_DATETIME
```

You can set a date resolution in the following methods:

```
// Date resolution in test methods:

.hasCreationDate().after(expectedDate, DateResolution)
.hasCreationDate().before(expectedDate, DateResolution)
.hasCreationDate().equalsTo(expectedDate, DateResolution)

.hasModificationDate().after(expectedDate, DateResolution)
.hasModificationDate().before(expectedDate, DateResolution)
.hasModificationDate().equalsTo(expectedDate, DateResolution)

// Internal used resolution DATE:
.hasSignatureField(..).signedOn(signingDate)

// Comparing two PDF documents, using DATE:
.haveSameCreationDate()
.haveSameModificationDate()
```

When comparing two PDF documents, date values are always compared using `DateResolution.DATE`.

13.8. Format Units - Points and Millimeters

Tests with page regions require width and height values. These values can be expressed in either millimeters or points. The following constants exist for these units:

```
// Predefined format units:

com.pdfunit.Constants.MILLIMETERS
com.pdfunit.Constants.POINTS
```

When points are converted into millimeters, 72 DPI (Dots per Inch) are used. Measurement units play a role in examples such as the following:

Format Units

```
@Test
public void hasField_Width() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Title of 'someField'";
  int allowedDeltaForMillis = 2;
  int allowedDeltaForPoints = 0;

  AssertThat.document(filename)
            .hasField(fieldname)
            .withWidth(450) // default is POINTS
            .withWidth(450, POINTS, allowedDeltaForPoints)
            .withWidth(159, MILLIMETERS, allowedDeltaForMillis)
  ;
}
```

In the previous example, 'width' refers to width as the internal property of the acro field. Since this is expressed in the unit points, PDFUnit takes the values for the methods `withWidth(..)` and `withHeight(..)` by default in points. To prevent rounding errors when converting points into millimeters, the third parameter of those methods determines the allowable difference between the expected size and the actual size.

Example - Page Format

```
@Test
public void hasHugeFormat() throws Exception {
  String filename = "documentUnderTest.pdf";
  int heightMM = 1117;
  int widthMM = 863;
  DocumentFormat formatMM = new DocumentFormatMillis(widthMM, heightMM);

  AssertThat.document(filename)
            .hasFormat(formatMM)
  ;
}
```

When page formats are created, the constants `MILLIMETERS` and `POINTS` cannot be used. Use the classes `DocumentFormatMillis` and `DocumentFormatPoints` instead.

Example - Error messages

Error messages print both the original unit and millimeters. For example, if the width in the last example had been set to `111 POINTS`, PDFUnit would have shown the following error message:

```
Wrong page format in 'physical-map-of-the-world-1999_1117x863mm.pdf' on page 1.
Expected: 'height=1117.60, width=39.16 (as 'mm', converted from unit 'points')',
but was:  'height=1117.60, width=863.60 (as 'mm')'.
```

13.9. Error Messages, Error Numbers

Error messages of PDFUnit provide detailed information to support bug fixing. And they are as clear and expressive as possible. A message for an incorrect page size demonstrates this intention:

```
Wrong page format in 'multiple-formats-on-individual-pages.pdf' on page 1.
Expected: 'height=297.00, width=210.00 (as 'mm')',
but was:  'height=209.90, width=297.04 (as 'mm')'.
```

The position of an error is marked by the double brackets <[and]>. To keep error messages readable, long values are shortened. The number of dropped characters is indicated as a number in the text surrounded by '...', e.g.:

```
The expected content does not match the JavaScript in 'javaScriptClock.pdf'.
Expected: '//<[Thisfileco...41...dbyPDFUnit]>',
but was:  '//<[Constantsu...4969...);break;}}]>'.
```

PDFUnit comes with error message in German and English language. Write you wish to info[at]pdfunit.com if you need error messages in another language.

Error Numbers

Error messages come from the files `messages.properties` or `messages_LANGUAGE.properties`. Each message in these files has the format `key = value`. PDFUnit uses a classes as keys. So the term 'error number' should be replaced by the term 'error key'.

When the validation exception is caught at runtime, the method `getErrorKey()` can be used to return the key from the current error message:

```
@Test
public void testErrorKey_hasText() throws Exception {
  try {
    String filename = PATH + "content/empty-pages.pdf";
    AssertThat.document(filename)
              .restrictedTo(EVERY_PAGE)
              .hasText()
    ;
  } catch (PDFUnitValidationException e) {
    String expectedKey = "com.pdfunit.messages.TextAvailableMessage";
    String actualKey = e.getErrorKey();
    assertEquals(expectedKey, actualKey);
  }
}
```

You can also pass the error key to a logging method.

13.10. Set Language for Error Messages

PDFUnit contains error messages in English and German. The operating system's language is detected at runtime and used for error messages. If a different language is required for PDFUnit, the language has to be set for the Java process. It's not necessary to reconfigure the operating system, just set some JVM options when starting PDFUnit: PDFUnit comes with error messages in English and in German. During runtime the language of the operation system is detected and used in error messages. If a different language is desired for PDFUnit, the language has to be set for the Java process. It's not necessary to re-configure the operating system to do this: just set the JVM language options when starting PDFUnit:

```
// JVM start options:

-Duser.language=de -Duser.country=DE
-Duser.language=es -Duser.country=ES
```

Alternatively, set the required options in the environment variable `_JAVA_OPTIONS`.

```
// Environment setting for Windows:
set _JAVA_OPTIONS=-Duser.language=de -Duser.country=DE

// Environment setting for Unix:
export _JAVA_OPTIONS=-Duser.language=de -Duser.country=DE
```

Setting this environment variable affects all subsequent Java applications until it is reset. When the options are set in a script, global settings stay unchanged.

13.11. Using XPath

General Comments about XPath in PDFUnit

Using XPath to evaluate parts of a PDF document opens a wider range of testing capabilities than an API alone can provide.

Several chapters in this manual describe XPath tests. The current chapter gives you an overview with references to the special chapters.

```
// Validating a single PDF using XPath:
.hasXFAData().matchingXPath(..)        3.37: “XFA Data” (p. 76)
.hasXMPData().matchingXPath(..)        3.38: “XMP Data” (p. 79)
.hasZugferdData().matchingXPath(..)    3.39: “ZUGFeRD” (p. 82)

// Comparing two documents using XPath:
.haveXFAData().matchingXPath(..)       4.15: “Comparing XFA Data” (p. 98)
.haveXMPData().matchingXPath(..)       4.16: “Comparing XMP Data” (p. 99)
```

Extract Data as XML

PDFUnit provides utility programs for all parts of a PDF document which can be tested using XML/ XPath. They extract the information into XML files:

```
// Utilities to extract XML from PDF:

com.pdfunit.tools.ExtractBookmarks
com.pdfunit.tools.ExtractFieldInfo
com.pdfunit.tools.ExtractFontInfo
com.pdfunit.tools.ExtractNamedDestinations
com.pdfunit.tools.ExtractSignatureInfo
com.pdfunit.tools.ExtractXFAData
com.pdfunit.tools.ExtractXMPData
com.pdfunit.tools.ExtractZugferdData
```

The utilities are described in chapter 9.1: “General Remarks for all Utilities” (p. 121):

Namespaces with Prefix

A namespace with an existing prefix will be detected automatically by PDFUnit.

Default Namespace

The default namespace is not detected automatically because the XML standard allows the definition of namespaces multiple times in an XML document. A default namespace has to be declared and you have to use a prefix:

```
/**
 * The default namespace has to be declared,
 * but any alias can be used for it.
 */
@Test
public void hasXFAData_UsingDefaultNamespace() throws Exception {
  String filename = "documentUnderTest.pdf";
  DefaultNamespace defaultNS = new DefaultNamespace("http://www.xfa.org/schema/xci/2.6/");
  XMLNode aliasFoo = new XMLNode("foo:log/foo:to", "memory", defaultNS);

  AssertThat.document(filename)
            .hasXFAData()
            .withNode(aliasFoo)
  ;
}
```

It may seem strange to use an arbitrary prefix, but the Java standard requires a user-defined one. It cannot be omitted. Please use one with a better name in real projects.

The next example shows the usage of a default namespace for an `XPathExpression`:

```
@Test
public void hasXMPData_MatchingXPath_WithDefaultNamespace() throws Exception {
  String filename = "documentUnderTest.pdf";

  String xpathAsString = "//default:format = 'application/pdf'";
  String stringDefaultNS = "http://purl.org/dc/elements/1.1/";
  DefaultNamespace defaultNS = new DefaultNamespace(stringDefaultNS);
  XPathExpression expression = new XPathExpression(xpathAsString, defaultNS);

  AssertThat.document(filename)
            .hasXMPData()
            .matchingXPath(expression)
  ;
}
```

XPath Compatibility

XPath expressions can use all syntax elements and functions of XPath. However, the number of available features of the XPath engine is version dependent. PDFUnit uses the XPath engine of the JDK. So, the JDK version determines the compatibility to the XPath standard.

Chapter 13.12: “JAXP-Configuration” (p. 167) describes the JAXP configuration of a JRE/JDK and how to use an external XML-parser or XSLT-processor.

13.12. JAXP-Configuration

The standard configuration of JAXP can be changed in several ways. Because they are not all well known, they will be explained in the following sections using Xerces and Xalan.

The Java runtime reads the following environment variables and tries to load their values as classes:

```
"javax.xml.parsers.DocumentBuilderFactory"
"javax.xml.parsers.SAXParserFactory"
"javax.xml.transform.TransformerFactory"
```

These properties can be set in different ways, which are shown by the following numbered sections. If a configuration option can override another, the one with the higher is described first.

1. The JAXP properties can be set directly in a program:

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
       "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
System.setProperty("javax.xml.parsers.SAXParserFactory",
       "org.apache.xerces.jaxp.SAXParserFactoryImpl");
System.setProperty("javax.xml.transform.TransformerFactory",
       "org.apache.xalan.processor.TransformerFactoryImpl");
```

2. A configuration parameter for the Java process to be started can be set using the flag for system properties `-D` in combination with the special environment variable `_JAVA_OPTIONS`. The next example shows one of the three JAXP properties, but of course all three are possible:

```
set _JAVA_OPTIONS=-Djavax.xml.transform.TransformerFactory=
                  org.apache.xalan.processor.TransformerFactoryImpl
```

3. The configuration with the start option `-D` can also be placed in the special environment variable `JAVA_TOOL_OPTIONS`. This variable is evaluated by some JDK implementations. The next example shows this option again with only one property:

```
set JAVA_TOOL_OPTIONS=-Djavax.xml.transform.TransformerFactory=
                      org.apache.xalan.processor.TransformerFactoryImpl
```

4. The JAXP configuration can also be placed in the file `jaxp.properties` in the folder `JAVA_HOME/jre/lib`:

```
#
# Sample configuration, file %JAVA_HOME%\jre\lib\jaxp.properties.
#
# Defaults in Java 1.7.0, Windows:
#
#javax.xml.parsers.DocumentBuilderFactory = \
           com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl
#javax.xml.parsers.SAXParserFactory       = \
           com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl
#javax.xml.transform.TransformerFactory   = \
           com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl

#
# Values for Xerces and Xalan:
#
javax.xml.parsers.DocumentBuilderFactory = \
          org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
javax.xml.parsers.SAXParserFactory       = \
          org.apache.xerces.jaxp.SAXParserFactoryImpl
javax.xml.transform.TransformerFactory   = \
          org.apache.xalan.processor.TransformerFactoryImpl
```

5. And a special technique for ANT is to use the environment variable `ANT_OPTS`:

```
set ANT_OPTS=-Djavax.xml.transform.TransformerFactory=
               org.apache.xalan.processor.TransformerFactoryImpl
set ANT_OPTS=-Djavax.xml.parsers.DocumentBuilderFactory=
               org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
set ANT_OPTS=-Djavax.xml.parsers.SAXParserFactory=
               org.apache.xerces.jaxp.SAXParserFactoryImpl
```

The declared XML and XSLT classes have to be found in the classpath or in the folder `JAVA_HOME\jre\lib\ext`. This last option is not recommended because only a few developers know about it. In case you are debugging a strange problem, you would not think to look at the version of the JAR's in that folder.

Note that Eclipse reads all existing environment variables only at startup and does not change anything later.

13.13. Running PDFUnit with TestNG

PDFUnit also runs with TestNG.

If you only use the annotation `@Test` no difference can be seen. Only when you expect exceptions you can see it's TestNG:

```
@Test(expectedExceptions=PDFUnitValidationException.class)
public void hasAuthor_NoAuthorInPDF() throws Exception {
  String filename = "documentUnderTest.pdf";

  AssertThat.document(filename)
            .hasAuthor()
  ;
}
```

13.14. Version History

2010

PDFUnit began with a customer request in August 2010, to which it was not possible to create a solution using the available tool 'jPdfUnit 1.1' (http://jpdfunit.sourceforge.net). Nor did the Apache project 'PDFBox' (http://pdfbox.apache.org) provide enough functionality. Only iText (http://itextpdf.com) provided the functions which were needed to implement the customer request.

2011

Throughout 2011, the knowledge about PDF and iText grew until implementation of PDFUnit based on iText was begun at the end of 2011.

Release 2012.07

Though the testing tool PDFUnit reached its goal, work on other projects slowed further development of the initial version.

Release 2013.01

In early 2013, new functions were added, existing bugs were fixed and the user manuals were created.

Release 2014.06

Many small improvements led to a new version of PDFUnit-Java.

Release 2015.10

The main new feature is a graphical user interface called PDFUnit-Monitor which allows non-programmers to use PDFUnit. The monitor reads the test information from one or more Excel files.

Release 2016.05

The implementation was refactored, iText was replaced by PDFBox 2.0. New testing features were created to validate QR codes, bar codes and ZUGFeRD data. PDFUnit now supports analysis of text with the direction right-to-left, as well as tests of all PDF documents in a given folder.

13.15. Unimplemented Features, Known Bugs

Extraction of Field Information

Some properties of form fields are not extracted, for example 'background color', 'border color' und 'border styles'.

Extraction of Signature Information

Currently not all signature data is exported into XML. It is possible that the XML structure will change in future releases.

Color

In the current release 2016.05 colors cannot be tested directly. If colors have to be tested, the test has to be carried out using rendered pages. The sections 3.20: “Layout - Entire PDF Pages” (p. 48) and 3.21: “Layout - in Page Regions” (p. 49) describe such tests.

Content of Layers

Tests related to text and images are not restricted to layers.

Complete XMP data

In the current release only the document-level XMP data are extracted and evaluated.

Index

E

F

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z