PDF automatisiert testen

PDFUnit-Java

Carsten Siedentop

Inhaltsverzeichnis

Vorwort

Aktuelle Testsituation in Projekten

Telefonrechnungen, Versicherungspolicen, amtliche Bescheide, Verträge jeglicher Art werden heute als PDF-Dokumente elektronisch zugestellt. Ihre Erstellung erfolgt in vielen Programmiersprachen mit zahlreichen Bibliotheken. Je nach Komplexität der zu erstellenden Dokumente ist diese Programmierung nicht einfach und enthält wie jede Software auch Fehler, die eventuell zu fehlerhaften PDF Documenten führen. Deshalb sollte geprüft werden:

- Steht in einem bestimmten Bereich einer Seite der erwartete Text?
- Stimmt der Barcode (QR-Code) auf dem Dokument mit dem erwarteten Inhalt überein?
- Stimmt das Layout mit der Vorgabe überein?
- Stimmen die Werte der eingebetteten ZUGFeRD-Daten mit den erwarteten Daten überein?
- Stimmen die Werte der eingebetteten ZUGFeRD-Daten mit den sichtbaren Daten überein?
- Entspricht ein Dokument den Regeln von DIN 5008?
- Ist das PDF signiert? Wann und von wem?

Es sollte Entwickler, Projekt- und Unternehmensverantwortliche erschrecken, dass es bisher kaum Möglichkeiten gibt, PDF-Dokumente **automatisiert** zu testen. Und selbst diese Möglichkeiten werden im Projektalltag nicht genutzt. Manuelles Testen ist leider weit verbreitet. Das ist teuer und fehleranfällig.

Egal, ob PDF-Dokumente mit einem mächtigen Design-Werkzeug, mit MS-Word/LibreOffice oder eigenen Programmen erstellt werden oder ob sie aus einem XSL-FO Workflow herausfallen, jedes PDF-Dokument kann mit PDFUnit getestet werden.

Intuitive Schnittstelle

Die Schnittstelle von PDFUnit folgt dem Prinzip des "Fluent Builder". Alle Namen von Klassen und Methoden lehnen sich eng an die Umgangssprache an und unterstützen damit gewohnte Denkstrukturen. Dadurch entsteht Java-Code, der auch langfristig noch leicht zu verstehen ist.

Wie einfach die Schnittstelle konzipiert ist, zeigt das folgende Beispiel:

```
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.")
;
```

Ein Test-Entwickler muss weder Kenntnisse über die Struktur von PDF haben, noch etwas über die fachliche Enstehungsgeschichte des PDF-Dokumentes wissen, um erfolgreiche Tests zu schreiben.

Zeit, anzufangen

> Spielen Sie nicht weiter Lotto bei der Erstellung Ihrer PDF-Dokumente. Überprüfen Sie das Ergebnis Ihrer PDF-Erstellung durch automatisierte Tests.

Kapitel 1. Über diese Dokumentation

Wer sollte sie lesen

Die vorliegende Dokumentation richtet sich an alle Personen, die mit der Entwicklung von PDF-Dokumenten betraut sind. Das sind einerseits Mitarbeiter aus den Bereichen Softwareentwicklung und Qualitätssicherung, andererseits aber auch Projekleiter und Budgetverantwortliche.

Code-Beispiele

Die in den nachfolgenden Kapiteln abgebildeten Code-Beispiele sind analog zum Eclipse-Editor eingefärbt, um das Lesen zu erleichtern. Die Beispiele verwenden JUnit als Testrahmen. Die gleichen Tests können aber auch mit TestNG geschrieben werden. Ein Demo-Projekt mit vielen Beispielen gibt es hier: http://www.pdfunit.com/de/download/index.html.

Javadoc

Die Javadoc-Dokumentation der API ist online verfügbar: http://www.pdfunit.com/api/java-doc/index.html.

Andere Programmiersprachen

PDFUnit gibt es sowohl für Java, als auch für .NET, Perl und als XML-Implementierung. Für jede Sprache existiert eine eigene Dokumentation, nachfolgend wird die Java-API geschrieben.

Wenn es Probleme gibt

Haben Sie Schwierigkeiten, ein PDF zu testen? Recherchieren Sie zuerst im Internet, vielleicht ist dort ein ähnliches Problem schon beschrieben, eventuell mit einer Lösung. Sie können die Problembeschreibung auch per Mail an info[at]pdfunit.com schicken.

Neue Testfunktionen gewünscht?

Hätten Sie gerne neue Testfunktionen, wenden Sie sich per Mail an info[at]pdfunit.com. Das Produkt befindet sich permanent in der Weiterentwicklung, die Sie durch Ihre Wünsche gerne beeinflussen dürfen.

Verantwortlichkeit

Manche Code-Beispiele in diesem Buch verwenden PDF-Dokumente aus dem Internet. Aus rechtlichen Gründen stelle ich klar, dass ich mich von den Inhalten distanziere, zumal ich sie z.B. für die chinesischen Dokumente gar nicht beurteilen kann. Aufgrund ihrer Eigenschaften unterstützen diese Dokumente Tests, für die ich keine eigenen Testdokumente erstellen konnte - z.B. für chinesische Texte.

Danksagung

Axel Miesen hat die Perl-Schnittstelle für PDFUnit entwickelt und in dieser Zeit viele Fragen zur Java-Version gestellt, die sich auf die Entwicklung von PDFUnit-Java vorteilhaft auswirkten. Herzlichen Dank, Axel.

Bei meinem Kollegen John Boyd-Rainey möchte ich mich für die kritischen Fragen zur Dokumentation bedanken. Seine Anmerkungen haben mich dazu bewogen, manchen Sachverhalt anders zu formulieren. John hat außerdem die englische Fassung dieser Dokumentation Korrektur gelesen. Die Men-

ge der aufgedeckten Komma- und anderer Fehler muss eine Tortur für ihn gewesen sein. Herzlichen Dank, John, für Deine Ausdauer und Gründlichkeit. Die Verantwortung für noch vorhandene Fehler liegt natürlich ausschließlich bei mir.

Herstellung dieser Dokumentation

Die vorliegende Dokumentation wurde mit DocBook-XML erstellt. Die PDF- und die HTML-Version stammen aus einer einzigen Textquelle und sind somit inhaltlich identisch. In beiden Zielformaten ist das Layout noch verbesserungswürdig, wie beispielsweise die Seitenumbrüche im PDF-Format. Die Verbesserung des Layouts steht schon auf der Aufgabenliste, jedoch gibt es noch andere Aufgaben mit höherer Priorität.

Feedback

Jegliche Art von Feedback ist willkommen, schreiben Sie einfach an info[at]pdfunit.com.

Kapitel 2. Quickstart

Quickstart

Angenommen, Sie haben ein Projekt, das PDF-Dokumente erzeugt und möchten sicherstellen, dass die beteiligten Programme das tun, was sie sollen. Weiter angenommen, ein Test-Dokument soll genau eine Seite umfassen sowie die Grußformel „Vielen Dank für die Nutzung unserer Serviceleistungen“ und eine Rechnungssumme von „30,34 Euro“ enthalten. Dann könnten Sie diese Anforderungen folgendermaßen testen:

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

@Test
public void hasExpectedRegards() throws Exception {
  String filename = "quickstart/quickstartDemo_de.pdf";
  String expectedRegards = "Vielen Dank für die Nutzung unserer Serviceleistungen";
  AssertThat.document(filename)
            .restrictedTo(LAST_PAGE)
            .hasText()
            .containing(expectedRegards)
  ;
}

@Test
public void hasExpectedCharge() throws Exception {
  String filename = "quickstart/quickstartDemo_de.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 :-)
  ;
}
```

Der typische JUnit-Report zeigt entweder den Erfolg oder eine aussagekräftige Fehlermeldung:

Class com.pdfunit.test.quickstart.QuickstartTests_de

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
QuickstartTests_de	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_de.pdf' not containing the expected text: '29,89 Euro'. Found: '30,34 Euro'. com.pdfunit.errors.PDFUnitValidationException: Page(s) [1] of 'C:\...\quickstart\quickstartDemo_de.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_de.hasExpectedCharge(QuickstartTests_de.java:71) at java.util.concurrent.FutureTask.run(FutureTask.java:266) at java.lang.Thread.run(Thread.java:745)	0.009
hasExpectedRegards	Success		0.008

So einfach geht's. Die folgenden Kapitel beschreiben den Funktionsumfang, typische Testfälle und Probleme beim Umgang mit PDF-Dokumenten.

Kapitel 3. Funktionsumfang

3.1. Überblick

Syntaktischer Einstieg

Tests auf ein **einzelnes PDF-Dokument** beginnen mit der Benennung der zu testenden Datei über die Methode `AssertThat.document(..)`. Von dort aus verzweigen die Tests in unterschiedliche Testbereiche, wie z.B. Inhalt, Schriften, Layout etc.:

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

AssertThat.document(filename)          13.1: „Instantiierung der PDF-Dokumente“ (S. 160)
          ...
// Switch to one of many test scopes.

// Compare one PDF with a reference:
AssertThat.document(filename)          ❶
          .and(..)                     4.1: „Überblick“ (S. 90)
          ...
```

❶ Das PDF-Dokument kann als Datentyp `String`, `File`, `InputStream`, `URL` oder `byte[]` an die Funktion übergeben werden.

Wenn **mehrere PDF-Dokumente** in einen Test einfließen, beginnt er mit der Methode `AssertThat.eachDocument(..)`:

```
// Instantiation of PDFUnit for multiple documents:
...
File[] files = {file1, file2, file3};
AssertThat.eachDocument(filename)    ❷ 5: „Mehrere Dokumente und Verzeichnisse“ (S. 104)
          .hasText(..)
          .containing(..)
;
```

❷ Die PDF-Dokumente können als `String[]`, `File[]`, `InputStream[]`, oder `URL[]` an die Funktion übergeben werden.

Tests können sich auch auf **alle PDF-Dokumente in einem Verzeichnis** beziehen. Solche Tests beginnen mit der Methode `AssertThat.eachDocument().inFolder(..)`:

```
// Instantiation of PDFUnit for a folder:
...
File folderToCheck = new File(..);
AssertThat.eachDocument()
          .inFolder(folderToCheck) ❸ 5.3: „Verzeichnis testen“ (S. 105)
          .hasText(..)
          .containing(..)
;
```

❸ Alle PDF-Dokumente in diesem Verzeichnis werden validiert. PDF-Dokumente in Unterverzeichnisse werden nicht geprüft.

Exception für erwartete Fehler

Tests, die einen Fehler erwarten, müssen die `PDFUnitValidationException` abfangen.

Hier ein Beispiel für einen Test, der einen Fehler erwartet:

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

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

Testbereiche

Die folgende Liste gibt einen vollständigen Überblick über die Testgebiete von PDFUnit. Der jeweilige Link hinter einer Methode verweist auf das Kapitel, das das Testgebiet ausführlich beschreibt.

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

.asRenderedPage()                 3.18: „Layout - gerenderte volle Seiten“ (S. 45)

.containsImage(..)                3.7: „Bilder in Dokumenten“ (S. 21)
.containsOneImageOf(..)           3.7: „Bilder in Dokumenten“ (S. 21)

.hasAuthor()                      3.10: „Dokumenteneigenschaften“ (S. 27)
.hasBookmark()                    3.20: „Lesezeichen/Bookmarks und Sprungziele“ (S. 48)
.hasBookmarks()                   3.20: „Lesezeichen/Bookmarks und Sprungziele“ (S. 48)
.hasCreationDate()                3.8: „Datum“ (S. 24)
.hasCreator()                     3.8: „Datum“ (S. 24)
.hasEmbeddedFile(..)              3.3: „Anhänge (Attachments)“ (S. 14)
.hasEncryptionLength(..)          3.21: „Passwort“ (S. 50)
.hasField(..)                     3.14: „Formularfelder“ (S. 33)
.hasFields(..)                    3.14: „Formularfelder“ (S. 33)
.hasFont()                        3.24: „Schriften“ (S. 55)
.hasFonts()                       3.24: „Schriften“ (S. 55)
.hasFormat(..)                    3.13: „Format“ (S. 32)
.hasImage(..)                     3.7: „Bilder in Dokumenten“ (S. 21)
.hasJavaScript()                  3.16: „JavaScript“ (S. 41)
.hasJavaScriptAction()            3.2: „Aktionen (Actions)“ (S. 12)
.hasKeywords()                    3.10: „Dokumenteneigenschaften“ (S. 27)
.hasLanguageInfo(..)              3.27: „Sprachinformation (Language)“ (S. 62)
.hasLayer()                       3.17: „Layer“ (S. 43)
.hasLayers()                      3.17: „Layer“ (S. 43)
.hasLessPagesThan()               3.25: „Seitenzahlen als Testziel“ (S. 58)
.hasLocalGotoAction()             3.2: „Aktionen (Actions)“ (S. 12)
.hasModificationDate()            3.8: „Datum“ (S. 24)
.hasMorePagesThan()               3.25: „Seitenzahlen als Testziel“ (S. 58)
.hasNamedDestination()            3.20: „Lesezeichen/Bookmarks und Sprungziele“ (S. 48)

.hasNoAuthor()                    3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoCreationDate()              3.8: „Datum“ (S. 24)
.hasNoCreator()                   3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoImage()                     3.7: „Bilder in Dokumenten“ (S. 21)
.hasNoKeywords()                  3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoLanguageInfo()              3.27: „Sprachinformation (Language)“ (S. 62)
.hasNoModificationDate()          3.8: „Datum“ (S. 24)
.hasNoProducer()                  3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoProperty(..)                3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoSubject()                   3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoText()                      3.28: „Texte“ (S. 63)
.hasNoTitle()                     3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoXFAData()                   3.36: „XFA Daten“ (S. 78)
.hasNoXMPData()                   3.37: „XMP-Daten“ (S. 81)

.hasNumberOf...()                 3.4: „Anzahl verschiedener PDF-Bestandteile“ (S. 16)

.hasOCG()                         3.17: „Layer“ (S. 43)
.hasOCGs()                        3.17: „Layer“ (S. 43)
.hasOwnerPassword(..)             3.21: „Passwort“ (S. 50)
.hasPermission()                  3.6: „Berechtigungen“ (S. 20)
.hasProducer()                    3.10: „Dokumenteneigenschaften“ (S. 27)
.hasProperty(..)                  3.10: „Dokumenteneigenschaften“ (S. 27)
.hasSignatureField(..)            3.26: „Signaturen - Unterschriebenes PDF“ (S. 59)
.hasSignatureFields()             3.26: „Signaturen - Unterschriebenes PDF“ (S. 59)
.hasSubject()                     3.10: „Dokumenteneigenschaften“ (S. 27)
.hasText()                        3.28: „Texte“ (S. 63)
.hasTitle()                       3.10: „Dokumenteneigenschaften“ (S. 27)
.hasUserPassword(..)              3.21: „Passwort“ (S. 50)
.hasVersion()                     3.35: „Version“ (S. 77)
.hasXFAData()                     3.36: „XFA Daten“ (S. 78)
.hasXMPData()                     3.37: „XMP-Daten“ (S. 81)
.hasZugferdData()                 3.39: „ZUGFeRD“ (S. 84)

.haveSame...()                    4.1: „Überblick“ (S. 90)

.isCertified()                    3.38: „Zertifiziertes PDF“ (S. 83)
.isCertifiedFor(..)               3.38: „Zertifiziertes PDF“ (S. 83)
.isLinearizedForFastWebView()     3.12: „Fast Web View“ (S. 31)
.isSigned()                       3.26: „Signaturen - Unterschriebenes PDF“ (S. 59)
.isSignedBy(..)                   3.26: „Signaturen - Unterschriebenes PDF“ (S. 59)
.isTagged()                       3.34: „Tagging“ (S. 75)

.restrictedTo(..)                 13.2: „Seitenauswahl“ (S. 160)
```

Manche Testbereiche erreicht man erst nach einem weiteren Methodenaufruf:

```
// Validation of bar code and QR code:
.hasImage().withBarcode()                 3.5: „Barcode“ (S. 17)
.hasImage().withQRcode()                  3.23: „QR-Code“ (S. 53)

// Validation based on Excel files:
.compliesWith().constraints(excelRules)   3.11: „Excel-Dateien für Validierungsregeln“ (S. 30)

// Validation of DIN5008 constraints:
.compliesWith().din5008FormA()            3.9: „DIN 5008“ (S. 26)
.compliesWith().din5008FormB()            3.9: „DIN 5008“ (S. 26)
.compliesWith().pdfStandard()             3.22: „PDF/A“ (S. 52)

// Validation around ZUGFeRD:
.compliesWith().zugferdSpecification(..) 3.39: „ZUGFeRD“ (S. 84)
```

PDFUnit wird ständig weiterentwickelt und die Dokumentation aktuell gehalten. Sollten Sie Tests vermissen, schicken Sie Ihre Wünsche und Vorschläge an info[at]pdfunit.com.

3.2. Aktionen (Actions)

Überblick

PDF-Dokumente werden durch Aktionen interaktiv aber auch komplizierter. Und „kompliziert“ bedeutet, dass sie getestet werden müssen, zumal wenn die interaktiven Teile eines Dokumentes ein wichtiger Teil einer Prozesskette sind.

Technisch betrachtet ist eine „Aktion“ ein Dictionary-Objekt mit den Elementen `/S` und `/Type`. Das Element `/Type` hat immer immer den Wert „Action“ und das Element `/S` (Subtype) hat typabhängige Werte:

```
// 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
```

Für einige dieser Aktionen stellt PDFUnit Testmethoden zur Verfügung. In zukünftigen Releases werden weitere Testmethoden zur Verfügung gestellt.

```
// Simple tests:

.hasNumberOfActions(..)

// Action specific tests:

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

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

Die folgenden Abschnitte zeigen Beispiele für diese Testmethoden.

JavaScript-Actions

JavaScript-Text ist gewöhnlich etwas umfangreich, deshalb macht es Sinn, den Vergleichstext für einen JavaScript-Action-Test aus einer Datei zu lesen:

```
@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)
  ;
}
```

Die Methoden `containingSource(..)` und `equalsToSource(..)` nehmen Parameter vom Typ `java.io.Reader`, `java.io.InputStream` oder `java.lang.String` entgegen.

In dem vorhergehenden Beispiel wird der vollständige Inhalt der JavaScript-Datei mit dem Inhalt der JavaScript-Action verglichen. Whitespaces werden dabei ignoriert.

Es können aber auch einfache Zeichenkette innerhalb des JavaScript-Codes geprüft werden:

```
@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")
  ;
}
```

Das Kapitel 13.5: „Behandlung von Whitespaces“ (S. 164) geht ausführlich auf den flexiblen Umgang mit Whitespaces ein.

Goto-Actions

Goto-Actions benötigen ein Sprungziel in derselben Datei:

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

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

Dieser Test ist erfolgreich, wenn das aktuelle Test-PDF eine GoTo-Action enthält die auf das Sprungziel „destination2.1“ zeigt. Sprungziele können im Zusammenhang mit Lesezeichen (Bookmarks) getestet werden. Darauf geht Kapitel 4.12: „"Named Destinations" vergleichen“ (S. 99) ein.

Es kann auch geprüft werden, ob sich ein Sprungziel auf einer bestimmten Seite befindet:

```
@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)
  ;
}
```

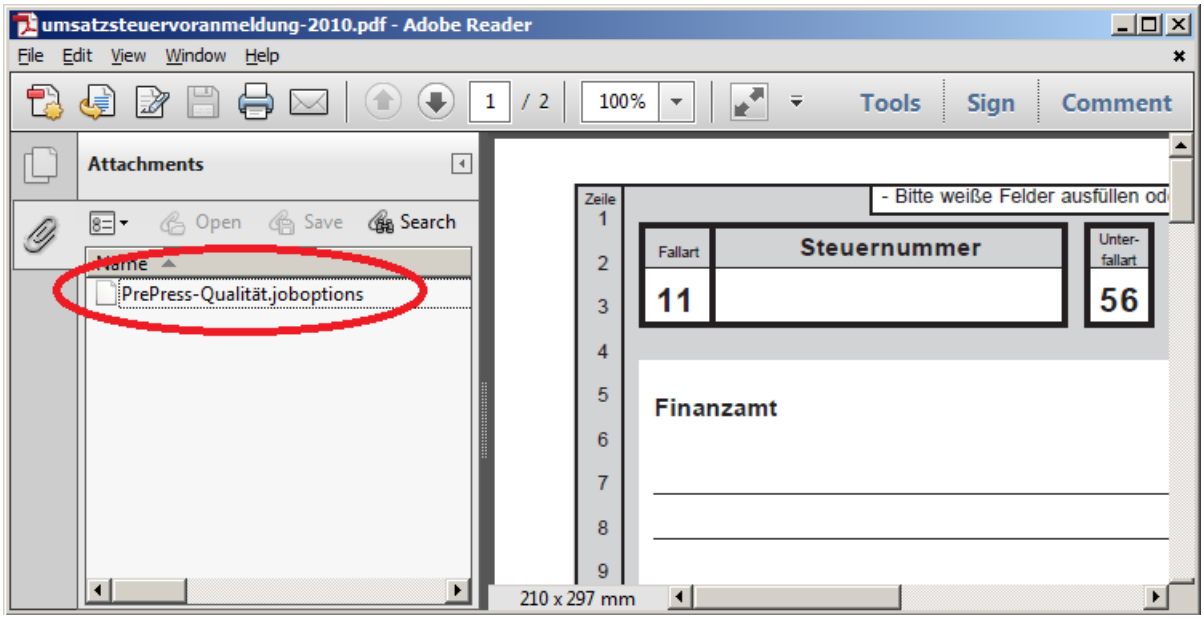
3.3. Anhänge (Attachments)

Überblick

Dateien, die als Attachments in PDF-Dokumenten enthalten sind, spielen in nachverarbeitenden Prozessen meist eine wichtige Rolle. Deshalb stellt PDFUnit Testmethoden für Attachments (eingebettete Dateien) bereit:

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

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

Die folgenden Tests beziehen sich auf das PDF-Formular für die deutsche Umsatzsteuervoranmeldung 2010, „umsatzsteuervoranmeldung-2010.pdf“. Es enthält eine Datei mit dem Namen „PrePress-Qualität.joboptions“.



Existenz

Der einfachste Test ist, zu prüfen, ob es überhaupt eingebettete Dateien gibt:

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

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

Anzahl

Etwas weiter geht die Prüfung der Anzahl der eingebetteten Dateien:

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

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

Dateiname

Danach kommen die Namen der Dateien:

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

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

Inhalt

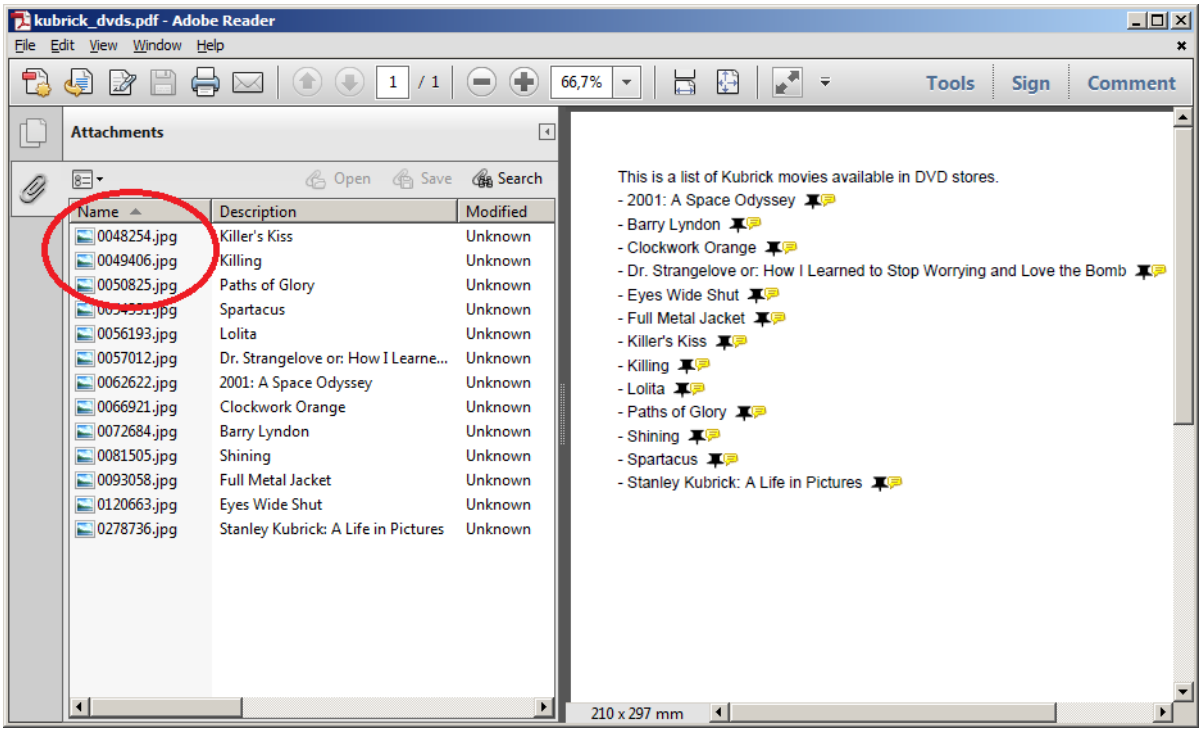
Und schließlich kann der Inhalt der in PDF eingebetteten Datei mit einer externen Datei verglichen werden:

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

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

Der Vergleich erfolgt Byte-weise. Als Parameter für die Vergleichsdatei kann entweder ein Dateiname oder eine Instanz von `java.util.File` verwendet werden.

Mehrfachaufrufe

Das nächste Beispiel bezieht sich auf die Datei „kubrick_dvds.pdf“, eine Beispieldatei von iText. Der Adobe Reader® zeigt die eingebetteten Dateien an:



Mehrere Dateien können mit einem Test überprüft werden. Wählen Sie aber einen passenderen Namen für die Testmethode, als in dem nächsten Beispiel:

```
@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)
  ;
}
```

Wenn die eingebetteten Dateien nicht als eigene Datei vorliegen, können sie mit dem Hilfsprogramm `ExtractEmbeddedFiles` aus einem bestehenden PDF-Dokument extrahiert werden. Das Programm wird in Kapitel 9.2: „Anhänge extrahieren“ (S. 124) genauer beschrieben.

3.4. Anzahl verschiedener PDF-Bestandteile

Überblick

Nicht nur die Anzahl von Seiten können Testziel sein, auch andere zählbare Teile eines PDF-Dokumentes, wie Formularfelder, Lesezeichen etc. Die folgende Liste zeigt auf, welche Dinge gezählt und damit getestet werden können:

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

❶❸ Prüfungen auf die Anzahl von Bildern werden in Kapitel 3.7: „Bilder in Dokumenten“ (S. 21) beschrieben.

❷ Prüfungen auf die Anzahl von Seiten eines PDF-Dokumentes werden in Kapitel 3.25: „Seitenzahlen als Testziel“ (S. 58) beschrieben.

Beispiele

Die Überprüfung der Anzahl von PDF-Teilen ist von der Art der Teile unabhängig. Deshalb werden hier nur zwei Beispiels gezeigt:

```
@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)
  ;
}
```

Alle Prüfungen können verkettet werden:

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

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

Erfreulicherweise dauert dieser Test mit einem Dokument von 1370 Seiten auf einem zeitgemäß ausgerüsteten Entwickler-Notebook nur ca. 2,5 Sekunden.

3.5. Barcode

Überblick

PDFUnit bietet die Möglichkeit, Barcode innerhalb von PDF-Dokumenten zu validieren. Dazu wird ZXing als Parser verwendet. Detailierte Informationen zu ZXing liefert die Homepage des Projektes.

Die vom Parser ausgelesenen Texte können dann mit verschiedenen Methoden verglichen werden. Insgesamt stehen folgende Methoden zur Verfügung:

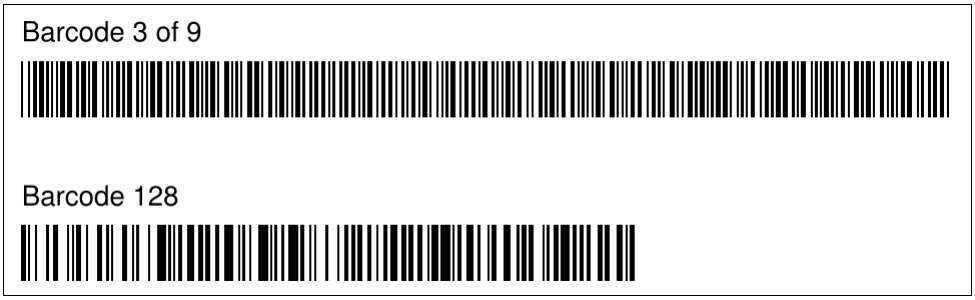
```
// Entry to all bar code validations:
.hasImage().withBarcode()

// 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(..)
```

Die folgenden Barcode Formate werden von ZXing automatisch erkannt und können in PDFUnit-Tests verwendet werden:

```
// 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
```

Beispiel - Barcode gegen Text vergleichen

In den folgenden Beispielen werden diese zwei Barcode-Muster verwendet:



Der erste Code enthält den Text 'TESTING BARCODES WITH PDFUNIT' und der zweite Code den 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")
  ;
}
```

Wenn in der ausgewählten Region mehrere Bilder existieren, muss jedes Bild den erwarteten Text enhalten. Whitespaces werden vor einem Vergleich normalisiert. Die Behandlung der Whitespaces kann aber durch einen Parameter von außen vorgegeben werden.

Der intern verwendete Barcode-Parser ZXing kennt viele Barcode-Typen, dennoch nicht alle. Deshalb stellt PDFUnit noch eine externe Schnittstelle zur Verfügung, über die kundenspezifische Barcode-Parser eingebunden werden können. Die Verwendung dieser Schnittstelle wird separat dokumentiert. Schicken Sie ein kurzes Mail an info[at]pdfunit.com, um Informationen über die Einbindung eines individuellen Barcode-Parser zu erhalten.

Beispiel - Barcode gegen Image vergleichen

Ein Barcode innerhalb eines PDF-Dokumentes kann auch gegen ein anderes Barcode-Image verglichen werden:

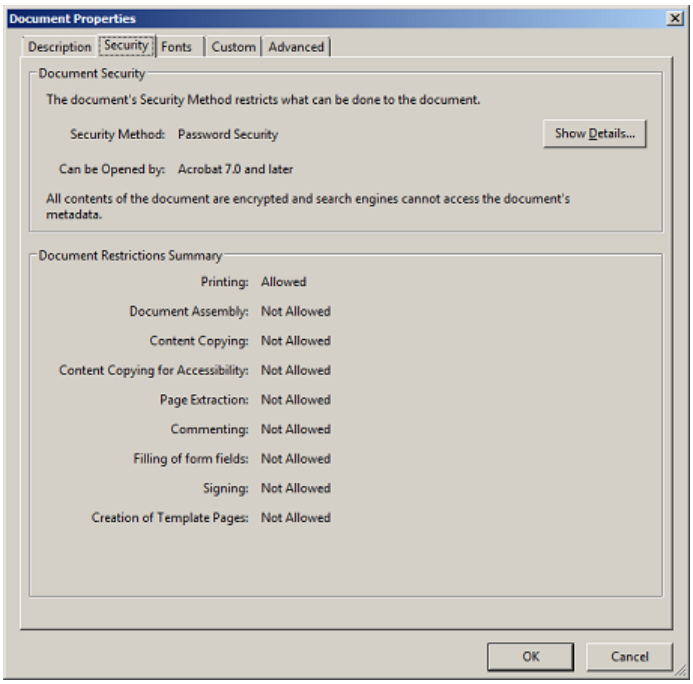
```
@Test
public void hasBarCodeMatchingImage() throws Exception {
  String filename = "documentUnderTest.pdf";

  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);
  ;
}
```

Wichtig: Der Dateityp des Bildes, beispielsweise PNG oder TIFF, muss zum Typ des Barcode Bildes im PDF-Dokument passen, damit ein Bildvergleich funktioniert.

3.6. Berechtigungen

Überblick

Wenn Sie erwarten, dass Ihr Workflow die PDF-Dokumente kopiergeschützt erstellt, sollten Sie das auch testen. Manuell können Sie die Berechtigungen im Adobe Reader® in den Dokumenteneigenschaften überprüfen:



Automatisch lassen sich Berechtigung mit passenden Testmethoden überprüfen. Alle Methoden werden mit einem Erwartungswert aufgerufen, der die Werte `true` oder `false` haben kann:

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

Beispiel

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

Die Zugriffsberechtigungen eines passwortgeschützten Dokumentes unterscheiden sich, je nachdem, ob das Dokument mit einem Owner-Password geöffnet wird oder mit einem User-Password.

3.7. Bilder in Dokumenten

Überblick

Bilder in einem PDF-Dokument sind selten optisches Beiwerk geringer Bedeutung. Sie enthalten Informationen, die in manchen Fällen vertragsrelevanten Charakter haben können. Typische Fehler im Zusammenhang mit Bildern sind:

- Befindet sich ein Bild auf der erwarteten Seite?
- Fehlt ein Bild, weil es im Erstellungsprozess nicht gefunden werden konnte?
- Enthält ein Brief wirklich das neue Unternehmens-Logo, und nicht das alte?
- Entspricht ein Text in Bildern dem erwarteten Text, beispielsweise einer Anschrift?
- Ist der Inhalt eines Barcodes oder QR-Codes der erwartete?

Alle Fehler können mit passenden Testmethoden erkannt werden:

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

.hasImage().withBarcode()          3.5: „Barcode“ (S. 17)
.hasImage().withBarcodeInRegion()  3.5: „Barcode“ (S. 17)
.hasImage().withText().xxx(..)     3.29: „Texte - in Bildern (OCR)“ (S. 68)
.hasImage().withTextInRegion()     3.29: „Texte - in Bildern (OCR)“ (S. 68)
.hasImage().withQRCode()           3.23: „QR-Code“ (S. 53)
.hasImage().withQRCodeInRegion()   3.23: „QR-Code“ (S. 53)
.hasImage().withHeightInPixel()
.hasImage().withWidthInPixel()

.hasNoImage()

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

Anzahl unterschiedlicher Bilder im Dokument

Eine wichtige Bemerkung vorweg: Die Anzahl der sichtbaren Bilder entspricht überlicherweise nicht der Anzahl der im PDF-Dokument selbst gespeicherten Bilder. Ein Logo, das auf 10 Seiten sichtbar ist, wird intern nur einmal gespeichert. Deshalb gibt es zwei Testmethoden. Die Methode `hasNumberOfDifferentImages(..)` überprüft die Anzahl der **internen** Bilder während die Methode `hasNumberOfVisibleImages(..)` die Anzahl der **sichtbaren** Bilder überprüft.

Das erste Beispiel zeigt die Validierung der Anzahl der **intern** gespeicherten Bilder:

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

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

Wie kommt man in diesem Beispiel auf die „2“? Wie können Sie für ein bestimmtes PDF wissen, welche Bilder tatsächlich intern gespeichert sind? Zur Beantwortung dieser Fragen extrahieren Sie alle Bilder Ihres PDF-Dokumentes mit dem von PDFUnit mitgelieferten Hilfsprogramm `ExtractImages`. Eine Beschreibung dieses Programms steht in Kapitel 9.3: „Bilder aus PDF extrahieren“ (S. 126).

Anzahl sichtbarer Bilder im Dokument

Das nächste Beispiel validiert die Anzahl der **sichtbaren** Bilder:

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

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

Das Beispieldokument hat 8 Bilder auf 6 Seiten, davon 2 Bilder auf Seite 3, kein Bild auf Seite 4 und 3 Bilder auf Seite 6.

Der Test auf die sichtbaren Bilder kann auf spezifizierte Seiten beschränkt werden. So werden im nächsten Beispiel nur die Bilder auf Seite 6 in einem bestimmten Bereich überprüft:

```
@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)
  ;
}
```

Bilder, die mehrfach auf einer Seite zu sehen sind, werden auch mehrfach gezählt.

Die verschiedenen Möglichkeiten, Tests auf bestimmte Seiten zu beschränken, werden in Kapitel 13.2: „Seitenauswahl“ (S. 160) ausführlich beschrieben.

Bestimmte Bilder vergleichen

Nach dem Zählen der Bilder folgt die Prüfung auf das Aussehen der Bilder. Im nachfolgenden Beispiel sucht PDFUnit innerhalb des PDF-Dokumentes ein Bild, das exakt so aussieht, wie das der angegebenen Datei:

```
@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)
  ;
}
```

Das Ergebnis eines Vergleiches zweier Bilddateien hängt von den Dateiformaten ab. PDFUnit kann alle Dateiformate verarbeiten, die von Java in ein `java.awt.image.BufferedImage` umgewandelt werden können: `JPEG`, `PNG`, `GIF`, `BMP` und `WBMP`. Die Bilder werden von PDFUnit Byte-weise verglichen. Deshalb werden die BMP- und PNG-Versionen eines Bildes nicht als gleich erkannt.

Das Bild kann in unterschiedlichen Formaten an die Testmethode gegeben werden:

```
// Types for images:

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

Beim Erzeugen von PDF kann es zu einer Formatumwandlung zwischen den Bildvorlagen als Datei und den intern gespeicherten Bildern kommen. Dadurch wird es unmöglich, die internen Bilder mit den Vorlagen zu vergleichen. Sollte es dieses Problem geben, extrahieren Sie die Bilder aus dem PDF-Dokument und verwenden für die nachfolgenden Tests ein extrahiertes Bild. Zuvor sollten sie es auf seine Korrektheit überprüft haben.

Die in einem PDF enthaltenen Bilder können auch gegen die Bilder in einem Referenz-Dokument verglichen werden. Eine genaue Beschreibung dazu enthält das Kapitel 4.4: „Bilder vergleichen“ (S. 92).

Eine Menge von Bildern als Vergleichsvorlage

Es kann die Situation geben, dass ein PDF-Dokument eines von drei möglichen Unterschriften enthält. Für diese Situation gibt es die Testmethode `matchingOneOf(..)`:

```
@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)
  ;
}
```

Der Test kann sich nicht nur auf eine Seite beziehen, wie hier die letzte Seite, sondern auch auf mehrere, wie der folgende Abschnitt zeigt.

Bilder auf unterschiedlichen Seiten

Die Suche nach Bildern kann grundsätzlich auf einzelne, mehrere individuelle oder mehrere zusammenhängende Seiten eingeschränkt werden. Es stehen dafür die Möglichkeiten zur Verfügung, die in Kapitel 13.2: „Seitenauswahl“ (S. 160) beschrieben.

Hier ein Beispiel:

```
@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)
  ;
}
```

Die Abwesenheit von Bilder prüfen

Manche Seiten oder Seitenausschnitte sollen keine Bilder enthalten. Solche Fälle sollte natürlich auch getestet werden können. Das nächste Beispiel überprüft, dass der zentrale Seitenbereich - ohne Header und Footer - der letzten Seite leer ist.

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

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

3.8. Datum

Überblick

Warum auch immer Sie das Erstellungsdatum eines Dokumentes überprüfen wollen - wegen der verschiedenen Formate, die ein Datum haben kann, ist es nicht ganz einfach. PDFUnit versucht, diese Komplexität zu kapseln, und gleichzeitig recht vielfältige Testszenarien anzubieten:

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

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

Die folgenden Abschnitte zeigen lediglich Tests für das Erstellungsdatum. Tests für das Änderungsdatum funktionieren exakt gleich:

Existenz eines Datums

Am Anfang soll getestet werden, ob ein PDF-Dokument überhaupt ein Erstellungsdatum enthält:

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

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

Wenn Ihr Dokument bewusst **kein** Erstellungsdatum enthalten soll, können Sie auch das testen:

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

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

Im nächsten Abschnitt wird ein vorhandenes Datum gegen einen Erwartungswert getestet.

Datumsauflösung

Ein erwarteter Datumswert muss eine Instanz vom Typ `java.util.Calendar` sein. Zusätzlich muss eine Datumsauflösung mitgegeben werden, die angibt, welche Datumsbestandteile für einen Test relevant sind.

Mit der Enumeration `DateResolution.DATE` werden Tag, Monat und Jahr verglichen und mit der Enumeration `DateResolution.DATETIME` zusätzlich noch Stunde, Minute und Sekunde. Für beide Werte gibt es Konstanten:

```
// Constants for date resolution:

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

Ein Test sieht dann so aus:

```
@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 Die Hilfsklasse `com.pdfunit.util.DateHelper` erleichtert es, für den erwarteten Datumswert eine Instanz von `java.util.Calender` zu erzeugen.

❷ Die Konstanten sind in der Enumeration `com.pdfunit.DateResolution` definiert.

Durch die Verwendung der Klasse `java.util.Calendar` wird das erwartete Datum formatunabhängig gemacht. Das PDF-interne Datum liegt formatiert vor und wird durch PDFUnit in eine Instanz der Klasse Calendar umgewandelt. Sollte es dabei zu Problemen kommen, besteht immer noch die Möglichkeit, das Datum auf die folgende Weise zu testen:

```
@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'")
  ;
}
```

Das Format eines Datums finden Sie, wenn Sie das Dokument mit einem Texteditor öffnen und nach der Zeichenkette "CreationDate" suchen.

Datumstest mit Ober- und Untergrenze

Sie können mit PDFUnit prüfen, ob das Erstellungsdatum eines PDF-Dokumentes nach oder vor einem gegebenen Datum liegt:

```
@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 Die jeweilige Unter- bzw. Obergrenze gehört nicht zum Gültigkeitszeitraum.

Datum einer Signatur

Das Datum einer Unterschrift eines signierten PDF-Dokumentes kann ebenfalls geprüft werden. Test dazu sind in Kapitel 3.26: „Signaturen - Unterschriebenes PDF“ (S. 59) beschrieben.

3.9. DIN 5008

Überblick

DIN 5008 definiert Regeln für das Format von Briefen und teilweise auch für deren Inhalte. PDFUnit bietet Möglichkeiten, diese Regeln zu überprüfen. Da der Standard viele Regeln als Empfehlung definiert, müssen Unternehmen die für sie relevanten Regeln auswählen. Technisch sieht das dann so aus, dass die relevanten Regeln in einer Excel-Datei abgelegt werden und alle Regeln der Excel-Datei auf ein PDF-Dokument oder mehrere Dokumente angewendet werden.

Die folgenden Methoden stehen für DIN 5008 prüfungen zur Verfügung:

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

Im Internet sind Information zu DIN 5008 unter https://de.wikipedia.org/wiki/DIN_5008 zu finden. Kapitel 3.11: „Excel-Dateien für Validierungsregeln“ (S. 30) beschreibt ausführlich, wie Regeln in Excel erfasst werden.

Beispiel - Einzeldokument validieren

Im folgenden Beispiel wird nur das PDF-Dokument dem Test übergeben. Die Excel-Datei mit den DIN 5008 Regeln wird nicht an den Test übergeben, sondern in der Konfigurationsdatei `pdfunit.config` festgelegt:

```
#
################################################################################
#
# 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
```

Das Verzeichnis kann als relativer oder absoluter Pfad angegeben werden. Ein Test sieht dann so aus:

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

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

Falls es verschiedene Excel-Datei gibt, reicht der eine Eintrag in der `pdfunit.config` nicht aus. Deshalb kann auch der Name der Excel-Datei mit den Validierungsregeln an einen Test übergeben werden:

```
@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)
  ;
}
```

Beispiel - Alle Dokumente in einem Verzeichnis validieren

Beide vorhergehenden Tests können auch mit einem Verzeichnis statt mit einer Datei ausgeführt werden. In dem Falle wird jedes PDF-Dokument in dem Verzeichnis gegen die Excel-Datei validiert.

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

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

3.10. Dokumenteneigenschaften

Überblick

PDF-Dokumente enthalten Informationen über Titel, Autor, Stichworte/Keywords und weitere Eigenschaften. Diese vom PDF-Standard vorgegebenen Informationen können durch individuelle Key-Value-Paare erweitert werden. Im Zeitalter von Suchmaschinen und Archivsystemen spielen sie eine zunehmend große Rolle. Umso wichtiger ist es, die Metadaten mit ordentlichen Werten zu füllen.

Ein Beispiel für schlechte Dokumenteneigenschaften aus der Praxis ist ein PDF-Dokument mit dem Titel „jfqd231.tmp“ (das ist tatsächlich der Titel des Dokumentes). Mit diesem Titel wird es nie gesucht und gefunden werden. Bei diesem Dokument einer amerikanischen Behörde handelt es sich um ein eingescanntes Schriftstück aus der Schreibmaschinenzeit. Es wurde 1993 eingescannt. Da aber auch der Dateiname eine semantikfreie Zahlenfolge ist, ist der Nutzen dieses Dokumentes nur marginal größer, als wenn es gar nicht existierte.

Folgende Methoden stehen zur Validierung der Metadaten zur Verfügung:

```
// 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 auf das Erstellungs- und Änderungsdatum werden in einem eigenen Kapitel beschrieben, weil sich die Art der Tests von denen der anderen Metadaten unterscheidet. Siehe Kapitel 3.8: „Datum“ (S. 24).

Metadaten eines Test-Dokumentes können auch mit den Metadaten eines anderen PDF-Dokumentes verglichen werden. Solche Vergleiche sind in Kapitel 4.6: „Dokumenteneigenschaften vergleichen“ (S. 94) beschrieben.

Autor validieren ...

Sie können den Autor eines Dokumentes manuell in einem PDF-Reader überprüfen. Einfacher geht es aber mit automatisierten Tests.

Wenn das Dokument einen **beliebigen** Wert für den Autor enthalten soll, können Sie das so testen:

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

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

Um explizit zu prüfen, dass die Dokumenteneigenschaft Autor **nicht vorhanden** ist, muss die Methode `hasNoAuthor()` verwendet werden:

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

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

Der nächste Test überprüft den Wert der Eigenschaft „Autor“:

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

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

Verschiedene Vergleichsmethoden stehen zur Verfügung. Die Methodennamen sind selbsterklärend:

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

In allen Vergleichsmethoden werden Leerzeichen **nicht** verändert. Bei so kurzen Feldern obliegt die Verantwortung über die Leerzeichen dem Testentwickler.

Alle Vergleichsmethoden arbeiten case-sensitiv.

Die Funktion `matchingRegex()` folgt den Regeln von `java.util.regex.Pattern`.

... und Creator, Keywords, Producer, Subject und Title

Die Tests auf Inhalte von Creator, Keywords, Producer, Subject und Title funktionieren genauso wie zuvor für „ Autor“ beschrieben.

Verketten von Methoden

Mehrere Testmethoden können in einem Test verkettet werden:

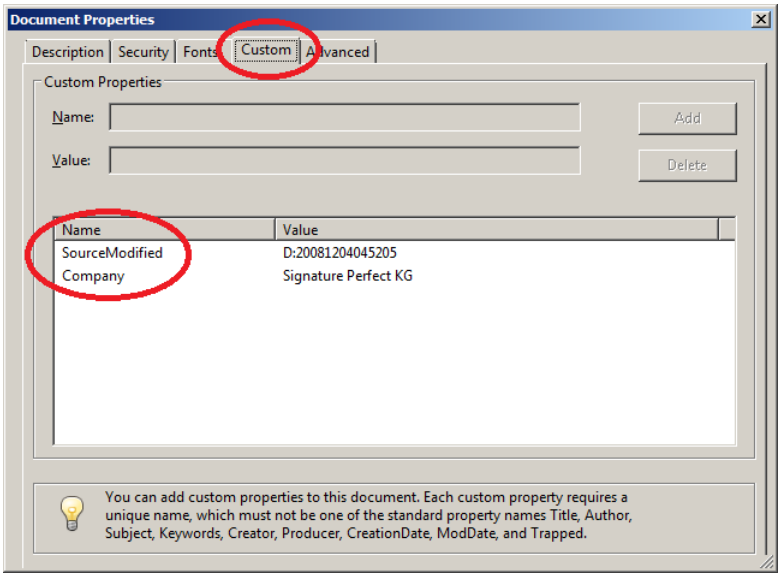
```
@Test
public void hasKeywords_allTextComparingMethods() throws Exception {
  String filename = "documentUnderTest.pdf";

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

Allgemeine Prüfung als Key-Value-Paar

Die in den vorhergehenden Abschnitten gezeigten Prüfungen auf Standardeigenschaften können auch mit der allgemeinen Methode `hasProperty(..)` ausgeführt werden. Sie prüft eine beliebige Dokumenteneigenschaft als Key/Value-Paar:

```
@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'")
  ;
}
```

Das im folgenden Beispiel benutzte Dokument besitzt zwei **individuelle** Key/Value-Werte, wie der Adobe Reader® zeigt:



Der Test auf die Existenz dieser individuellen Eigenschaften mit konkreten Werten sieht so aus:

```
@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)
  ;
}
```

Um sicherzustellen, dass eine bestimmte 'Custom-Property' **nicht** im PDF-Dokument auftaucht, muss der Test so aussehen:

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

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

Ab PDF-1.4 existiert die Möglichkeit, Metadaten intern als XML zu speichern (Extensible Metadata Platform, XMP). Das Kapitel 3.37: „XMP-Daten“ (S. 81) geht darauf ausführlich ein.

3.11. Excel-Dateien für Validierungsregeln

Überblick

Es mag ungewöhnlich erscheinen, Validierungsregeln in Excel abzulegen und diese in automatisierten Test zu verwenden. Der Grund, es dennoch zu tun, ist aber, auch Nicht-Programmierer in die Lage zu versetzen, PDF-Dokumente zu testen oder zumindestens die Tests zu erfassen, die dann mit Hilfe von Entwicklern ausgeführt werden.

Die Excel-Dateien können übrigens unverändert auch innerhalb des PDFUnit-Monitors verwendet werden, der ebenfalls für die Zielgruppe der Nicht-Programmierer gebaut wurde. Für den Monitor existiert eine separate Dokumentation unter . zur Verfügung. Eine kurze Beschreibung liefert aber auch Kapitel 8: „PDFUnit-Monitor“ (S. 120).

Eine detailierte Beschreibung des Aufbaus einer Excel-Datei und über die über diesen Weg verfügbare Funktionalität, liefert Kapitel 10: „Validierungsregeln in Excel-Dateien“ (S. 140). Lesen Sie das Kapitel ruhig später. Für das Verständnis, wie solche Excel-Dateien in programmierten Tests eingebunden werden, sind keine Kenntnisse über den Aufbau der Excel-Datei notwendig.

Es gibt nur eine Methode, mit der Excel-Dateien in Tests eingebunden werden:

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

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

Diese Methode kann auf ein PDF-Dokument, mehrere Dokumente oder alle Dokumente in einem Verzeichnis angewendet werden:

```
// 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)
```

Beispiel

Im folgenden Beispiel wird eine PDF-Datei gegen die Regeln von DIN 5008 geprüft, deren unternehmensspezifische Umsetzung in der Excel-Datei 'din5008-formA_letter-portrait.xls' liegen:

```
@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)
  ;
}
```

In einer Fehlermeldung erscheinen alle Verstöße gegen die definierten Regeln. Ein Test bricht also nicht mit dem ersten Regelverstoß ab.

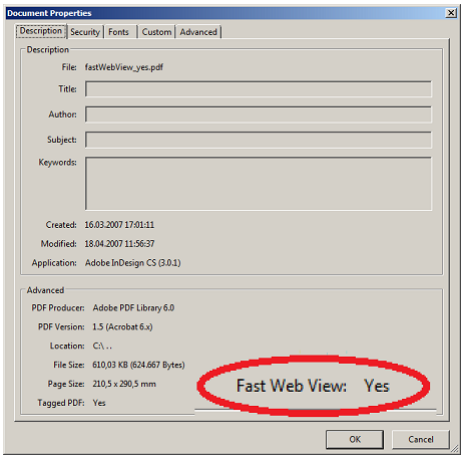
Der Aufbau der Excel-Datei wird in Kapitel 10: „Validierungsregeln in Excel-Dateien“ (S. 140) beschrieben.

3.12. Fast Web View

Überblick

Der Begriff „Fast Web View“ bedeutet, dass ein Server ein PDF-Dokument seitenweise an einen Client ausliefern kann. Die Fähigkeit, es zu tun, liegt beim jeweiligen Server. Das PDF-Dokument selber muss dieses Verhalten aber unterstützen. Dazu müssen Objekte, die zum Rendern der ersten PDF-Seite benötigt werden, am Anfang der Datei stehen. Und außerdem muss das Inhaltsverzeichnis über die Objekte auch am Anfang stehen.

Im Adobe Reader® wird „Fast Web View“ über den Eigenschaften-Dialog angezeigt:



Es gibt eine Testmethode, um linearisierte Dokumente zu testen:

```
.isLinearizedForFastWebView()
```

PDFUnit prüft im vorliegenden Release nicht alle genannten Merkmale eines linearisierten Dokumentes ab. Sollte es unerwartete Testergebnisse geben, wenden Sie sich an info[at]pdfunit.com.

Beispiel

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

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

3.13. Format

Überblick

Welches Format benötigen Sie: DIN-A4 quer, Letter hochkant oder vielleicht ein ganz individuelles Format für ein Poster? Tests auf so etwas einfaches, wie Papierformate sind scheinbar überflüssig. Aber haben Sie schon einmal eine Datei im Format „LETTER“ auf einem „DIN-A4“-Drucker gedruckt. Das geht - aber das Schriftbild sieht nicht immer gut aus. Deshalb gibt es folgende Testmethode:

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

Dokumente mit einheitlichem Seitenformat

Übliche Seitenformate können mit Konstanten geprüft werden:

```
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 Für gängige Formate existieren Konstanten in der Klasse `com.pdfunit.Constants`

Auch individuelle Papierformate können geprüft werden:

```
@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)
  ;
}
```

Es wird empfohlen, für alle Größenangaben die Einheit Millimeter zu verwenden. Zwar gibt es auch die Klasse DocumentFormatPoints, aber die Einheit "Point" ist von der Auflösung abhängig. Zwei Dokumente mit gleicher Anzahl an "Punkten" sind unterschiedlich groß, wenn ein Dokument die Auflösung 72 DPI (72 Punkt pro Inch) und das andere 150 DPI hat.

Alle Werte sind Ganzzahlen. Falls Sie ein Format testen wollen, das Werte in der Größenordnung von Zehntelmillimetern aufweist, runden Sie die Werte auf oder ab. Der Längenvergleich mit PDFUnit berücksichtigt sowieso die durch die DIN 476 erlaubte Toleranz, sodass Zehntelmillimeter keine entscheidende Rolle spielen.

Informationen zur DIN-Norm für Papierformate finden sie bei Wikipedia (http://de.wikipedia.org/wiki/Papierformat). Es muss betont werden, dass beim Vergleich aller Formate die geringere Toleranz der DIN-Norm 476 verwendet wird, auch wenn die Norm ISO 216 eine größere Toleranz erlaubt.

Formate auf einzelnen Seiten

Auch ein Dokument mit unterschiedlichen Formaten kann überprüft werden. Im folgenden Beispiel wird nur das Format auf Seite 3 überprüft:

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

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

Formattests können auf beliebige einzelne Seiten und Seitenbereiche eingeschränkt werden, wie es in Kapitel 13.2: „Seitenauswahl“ (S. 160) beschrieben ist:

3.14. Formularfelder

Überblick

Wenn Inhalte eines PDF-Dokumentes weiterverarbeitet werden sollen, spielen Formularfelder eine entscheidende Rolle. Diese sollten korrekt erstellt sein. Dafür sind vor allem korrekte und eindeutige Feldnamen wichtig, aber auch manche Eigenschaft eines Feldes.

Mit dem Hilfsprogramm `ExtractFieldInfo` können Informationen zu Formularfeldern in eine XML-Datei extrahiert. Dadurch werden Feldeigenschaften sichtbar, die ansonsten nur schwer zu ermitteln sind. Alle in der XML-Datei dargestellten Eigenschaften können in Tests überprüft werden.

In den folgenden Abschnitten werden viele Tests auf Feldeigenschaften, Größe und natürlich auch die Inhalte von Feldern beschrieben. Je nach Anwendungkontext kann der eine oder andere Tests für Sie wichtig sein:

```
// 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 Tests werden in Kapitel 3.26: „Signaturen - Unterschriebenes PDF“ (S. 59) beschrieben.

❷ Tests werden in Kapitel 3.15: „Formularfelder, Textüberlauf“ (S. 40) beschrieben.

Existenz

Mit dem folgenden Test können Sie prüfen, ob es überhaupt Felder gibt:

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

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

Feldnamen

Da bei der Verarbeitung von PDF-Dokumenten über die Feldnamen auf deren Inhalte zugegriffen wird, muss sichergestellt sein, dass es die erwarteten Felder auch gibt:

```
@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)
  ;
}
```

Das gleiche Ziel kann auch mit einer anderen Syntax erreicht werden, indem ein Array mit Feldnamen an die Methode `hasFields(..)` übergeben wird:

```
@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)
  ;
}
```

Doppelte Feldnamen sind zwar nach der PDF-Spezifikation erlaubt, bereiten bei der Weiterverarbeitung von PDF-Dokumenten höchstwahrscheinlich aber Überraschungen. PDFUnit stellt deshalb eine Methode zur Verfügung, um die Abwesenheit doppelter Namen zu prüfen:

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

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

Anzahl

Wenn es lediglich wichtig ist, wieviele Formularfelder ein PDF-Dokument enthält, nutzen Sie die Funktion `hasNumberOfFields(..)`:

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

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

Möglicherweise ist es auch interessant, sicherzustellen, dass ein PDF-Dokument **keine Felder** (mehr) besitzt:

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

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

Inhalte von Feldern

Am einfachsten ist ein Test, der prüft, ob ein bestimmtes Feld überhaupt Daten enthält:

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

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

Zur Überprüfung der Inhalte von Feldern stehen ähnliche Vergleichsmethoden zur Verfügung, wie für die Überprüfung der Inhalte von Dokumenteneigenschaften:

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

Die nachfolgenden Beispiele sollen Ihnen ein paar Anregungen für die Verwendung dieser Methoden geben:

```
@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 werden vor dem Vergleich des tatsächlichen mit dem erwarteten Text normalisiert.

Feldtypen

Formularfelder haben einen bestimmten Typ. Auch wenn die Bedeutung des Typs wohl nicht so groß ist, wie die des Namens, gibt es trotzdem eine Testmethode für Typen:

```
@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)
  ;
}
```

Das vorhergehende Code-Listing enthält bis auf das Signaturfeld alle Feldtypen, die überprüft werden können. Mit dem nächsten Beispiel wird auf Signaturfelder geprüft:

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

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

Ausführliche Tests zu Signaturen werden in Kapitel 3.26: „Signaturen - Unterschriebenes PDF“ (S. 59) beschrieben.

Die möglichen Feldtypen sind als Konstanten in `com.pdfunit.Constants` definiert. Die Namen der Konstanten entsprechen den gängigen, sichtbaren Elementen einer graphischen Anwendung. Innerhalb von PDF gibt es aber andere Typen. Weil deren Namen in einer Fehlermeldungen auftauchen können, gibt die folgende Liste die Zuordnung wider:

```
// 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"
```

Größe von Feldern

Falls die Größe von Formularfeldern wichtig ist, stehen für die Überprüfung von Länge und Breite zwei Methoden zur Verfügung:

```
@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)
  ;
}
```

Beide Methoden können mit verschiedenen Maßeinheiten aufgerufen werden, die als Konstanten vorgegeben sind. Und da es beim Umgang mit den Größen zu Rundungsfehlern kommen kann, muss als dritter Parameter noch eine erlaubte Abweichung des tatsächlichen vom erwarteten Wert mitgegeben werden. Millimeter und Points können aus Einheit verwendet werden. Points und eine Abweichung von '0' gelten als Default.

```
@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
  ;
}
```

Sie werden beim Erstellen eines Testes wahrscheinlich nicht die Maße eines Feldes kennen. Kein Problem, nehmen Sie eine beliebige Zahl für die Höhe und Breite und starten den Test. Die dann auftretende Fehlermeldung enthält die richtigen Werte in Millimetern.

Ob ein Text tatsächlich in ein Formularfeld passt, lässt sich durch die Größenbestimmung alleine nicht sicherstellen. Neben der Schriftgröße bestimmen auch die Worte am Zeilenende zusammen mit der Silbentrennung die Anzahl der benötigten Zeilen und damit die benötigte Höhe. Das Kapitel 3.15: „Formularfelder, Textüberlauf“ (S. 40) beschäftigt sich ausführlich mit diesem Thema.

Weitere Eigenschaften von Feldern

Formularfelder haben neben ihrer Größe noch weitere Eigenschaften, wie z.B. `editable` und `required`. Viele dieser Eigenschaften können manuell gar nicht getestet werden. Deshalb gehören passende Tests in jedes PDF-Testwerkzeug. Das folgende Beispiele stellt das Prinzip dar:

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

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

Insgesamt stehen folgende Methoden zur Überprüfung von Feldeigenschaften zur Verfügung:

```
// 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 Aktionen zur Validierung von Feldern

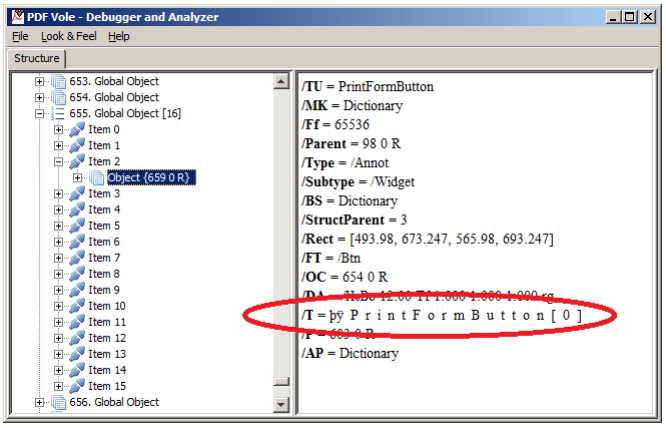
Wenn PDF-Dokumente Teil eines Workflows sind, unterliegen Formularfelder normalerweise bestimmten Plausibilitäten. Diese Plausibilitäten werden häufig durch eingebettetes JavaScript umgesetzt, um die Prüfungen schon zum Zeitpunkt der Eingabe auszuführen.

Mit PDFUnit kann geprüft werden, ob einem Formularfeld JavaScript zugeordnet ist. Erwartete Inhalte werden mit 'containing()' gegen die tatsächlichen Inhalte geprüft. Dabei werden Whitespace normalisiert:

```
@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-Feldnamen

Wenn PDF-erstellende Werkzeuge Unicode-Sequenzen **nicht** richtig verarbeiten, wird es schwierig, diese Sequenzen in PDFUnit-Tests zu verwenden. Schwierig heißt aber nicht unmöglich. Das folgende Bild zeigt, dass der Name eines Feldes PDF-intern unglücklicherweise als UTF-16BE mit Byte-Order-Mark (BOM) gespeichert wird:



Auch wenn es schwierig ist, dieser Feldname kann als Java-Unicode-Sequenz getestet werden:

```
@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)
  ;
}
```

Mehr Informationen zu Unicode und Byte-Order-Mark liefern gute Artikel auf Wikipedia.

3.15. Formularfelder, Textüberlauf

Überblick

Eine Möglichkeit, PDF-Dokumente zu erstellen, besteht darin, Platzhalter (Form Fields) einer Vorlage mit Textbausteinen zu füllen. Falls der Text aber größer ist, als der für die Anzeige zur Verfügung stehende Platz, wird der Text abgeschnitten. Überschüssiger Text (außerhalb des Fensters) wird nicht angezeigt. Als Gegenmaßnahme könnte die Schriftgröße verringert werden. Das schlägt aber auf das abschließende Erscheinungsbild durch und ist daher selten akzeptabel.

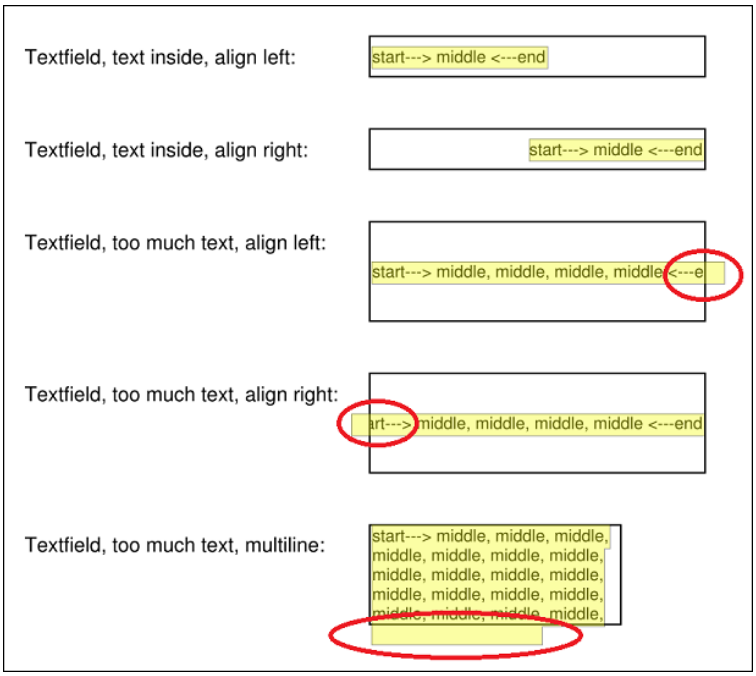
Die Prüfung, ob Text in ein Feld passt, ist nicht ganz einfach, schließlich ist das Ergebnis nicht nur von der Schriftgröße abhängig, sondern auch von der Länge der Wörter am jeweiligen Zeilenende und einer eventuell verwendeten Silbentrennung. Nach mehrfacher Nachfrage seitens verschiedener Interessenten, bietet PDFUnit nun diese beiden Testfunktionen:

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

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

Es werden nur Textfelder überprüft! Buttons, Auswahllisten, Combo-Boxen, Signaturfelder und Passwort-Felder werden nicht geprüft.

Passende Größe eines Feldes

Der Screenshot zeigt die im nachfolgenden Beispiel verwendeten Felder mit ihren Texten. Es ist gut zu erkennen, dass der Text in den letzten drei Feldern über den Rand des jeweiligen Feldes hinausgeht:



Und so funktioniert der Test auf das letzte Feld des abgebildeten Dokumentes:

```
@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()
  ;
}
```

Wenn die Exception nicht abgefangen wird, lautet sie: `Content of field 'Textfield, too much text, multiline:' of 'C:\...\fieldSizeAndText.pdf' does not fit in the available space`.

Wird die Methode `hasField(..)` für andere Felder als Textfelder aufgerufen, wird keine Exception geworfen.

Passende Größe aller Felder

Wenn ein PDF-Dokument viele Felder enthält, wäre es unnötig aufwendig, für jedes Feld einen eigenen Test zu schreiben. Deshalb können alle Felder gleichzeitig auf Textüberlauf überprüft werden:

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

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

Auch bei dieser Testmethode werden nur Textfelder überprüft, keine Buttons, Auswahllisten, Combo-Boxen, Signaturfelder und Passwort-Felder.

3.16. JavaScript

Überblick

Wenn JavaScript in Ihren PDF-Dokumenten überhaupt existiert, wird es wohl wichtig sein. Nicht selten übernimmt JavaScript eine aktive Rolle innerhalb eines Dokumenten-Workflows.

Zwar ersetzt PDFUnit kein separates JavaScript-Unittest-Werkzeug wie beispielsweise „Google JS Test“ , aber besser wenig getestet, als überhaupt nicht. Die folgenden Testfunktionen stehen zur Verfügung:

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

Existenz von JavaScript

Mit der folgenden Funktion lässt sich feststellen, ob das Dokument überhaupt JavaScript enthält:

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

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

Vergleich gegen eine Vorlage

Das erwartete JavaScript kann aus einer Datei eingelesen und mit dem des PDF-Dokumentes verglichen werden. Für die Extraktion steht das Hilfsprogramm `ExtractJavaScript` zur Verfügung. Die so erzeugte Datei kann in einem Test verwendet werden:

```
@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 Neben `java.io.File` sind auch `java.io.Reader`, `java.io.InputStream` und der Dateiname als `java.lang.String` möglich.

Das JavaScript, das mit dem JavaScript des PDF-Dokumentes verglichen wird, muss aber nicht aus einer Datei gelesen werden. Es kann auch direkt als String übergeben werden:

```
@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)
  ;
}
```

Teilstrings vergleichen

In den bisherigen Tests wurde das JavaScript eines PDF-Dokumentes immer gegen eine komplette Datei verglichen. Es kann aber auch auf Teil-Strings getestet werden, wie die folgenden Beispiele zeigen:

```
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 werden bei allen Vergleichen ignoriert, sowohl bei dem JavaScript aus dem PDF-Dokument, als auch bei dem aus Dateien oder String-Parametern.

3.17. Layer

Überblick

Die sichtbaren Inhalte eines PDF-Dokumentes können sich in mehreren Ebenen befinden und jede Ebene kann sichtbar oder unsichtbar geschaltet werden. In der PDF-Spezifikation „PDF 32000-1:2008“ heißt es dazu in Abschnitt 8.11.2.1. „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.“

Die Begriffe „Layer“ und „Optional Content Group, OCG“ bezeichnen das Gleiche. Während die Spezifikation den Begriff „OCG“ benutzt, verwendet der Adobe Reader® den Begriff „Layer“.

PDFUnit bietet folgende Testmethoden rund um das Thema Layer an:

```
// 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()
```

Eine Testmethode `matchingRegex()` wird nicht angeboten. Sie ist nicht nötigt, weil Layernamen üblicherweise kurz und klar sind.

Anzahl

Der erste Test zielt auf die Anzahl der Layer (OCG):

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

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

❶❷ „Layer“ und „Optional Content Group“ sind funktional identisch. Aus sprachlichen Gründen werden beide Begriffe als gleichwertige Methoden angeboten.

Layernamen

Der nächste Test zielt auf die Namen der Layer:

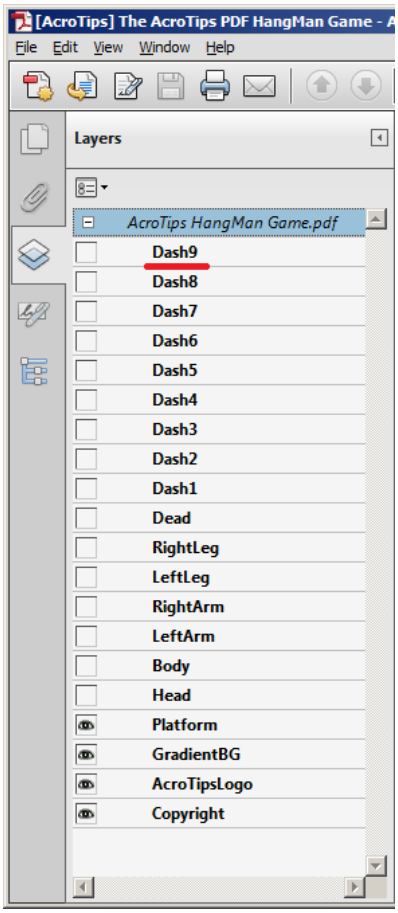
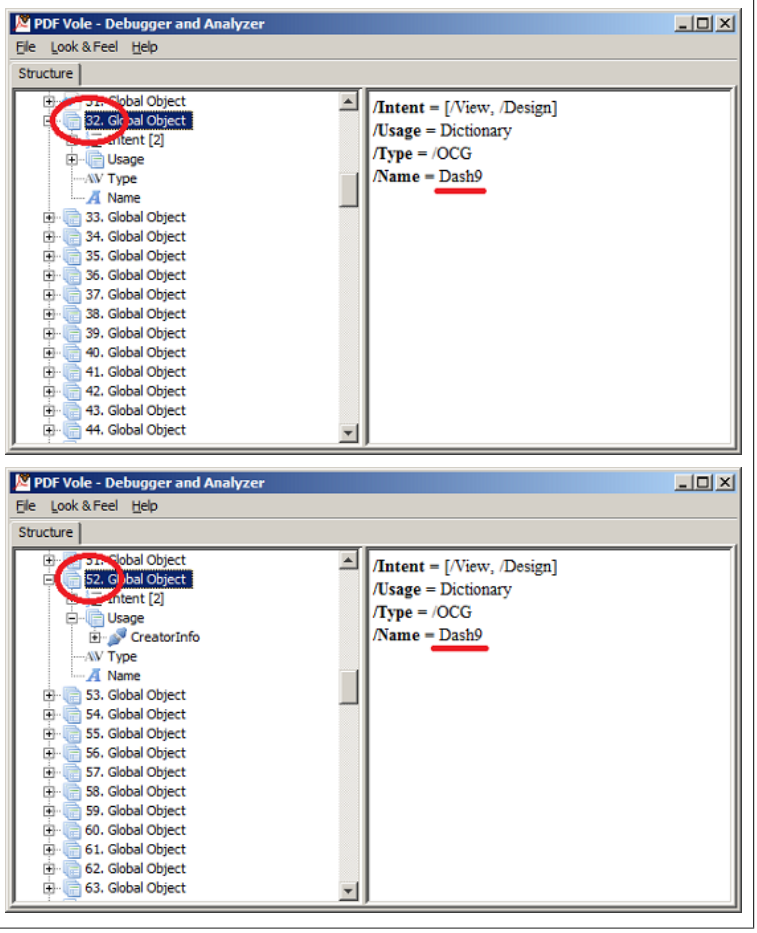
```
@Test
public void hasLayer_WithName() throws Exception {
  String filename = "documentUnderTest.pdf";

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

Für den Namen gibt es folgende Vergleichsmethoden:

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

Doppelte Layername

Die Namen von Layern innerhalb eines PDF-Dokumentes müssen laut PDF-Standard nicht eindeutig sein. Das Dokument im folgenden Beispiel enthält doppelte Layernamen. Sie werden vom Adobe Reader® nicht angezeigt, wohl aber vom Analysewerkzeug „PDFVole“, wie die folgenden Bilder zeigen:

Layer-Namen im Adobe Reader®	Layer Objekte in PDFVole
	

Anhand der Bilder ist zu sehen, dass die Layer-Objekte mit den Nummern 32 und 52 den gleichen Namen „Dash9“ haben.

Wenn ein PDF-Dokument **keine gleichnamigen** Layer haben soll, können Sie das mit einer passenden Testmethode überprüfen:

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

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

PDFUnit bietet im aktuellen Release 2016.05 noch keine Tests, um auf die Inhalte von Layern zu testen.

3.18. Layout - gerenderte volle Seiten

Überblick

Der Text eines PDF-Dokumentes hat Eigenschaften wie Schriftgröße, Schriftfarbe, Linien, die richtig sein müssen, bevor der Kunde es in den Händen hält. In die gleiche Kategorie fallen auch Absätze, Ausrichtung von Text sowie Bilder und Bildbezeichnungen. PDFUnit testet diese Layout-Aspekte, indem es das Test-Dokument seitenweise rendert und dann jede Seite:

- ... mit einer Bild-Datei vergleicht. PDFUnit liefert das Hilfsprogramm `RenderPdfToImages` mit, um eine oder mehrere PDF-Seiten zu rendern. Es ist in Kapitel 9.7: „PDF-Dokument seitenweise in PNG umwandeln“ (S. 130) beschrieben.
- ... mit der ebenfalls gerenderten Seite eines Referenz-Dokumentes vergleicht, die zuvor als richtig bewertet wurde. Das Kapitel 4.10: „Layout vergleichen (gerenderte Seiten)“ (S. 97) beschreibt diesen Vergleich.

Es gibt folgende Testmethoden:

```
// 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: 3.19: „Layout - gerenderte Seitenausschnitte“ (S. 46)
.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)
```

Für den Vergleich können beliebige Seiten ausgewählt werden. Das Kapitel 13.2: „Seitenauswahl“ (S. 160) geht näher darauf ein. Es können aber auch Teile einer Seite mit Bildern verglichen werden. Die Syntax dazu wird in Kapitel 3.19: „Layout - gerenderte Seitenausschnitte“ (S. 46) beschrieben.

Beispiel - Ausgewählte Seiten als Bild vergleichen

Die Seiten 1, 3 und 4 sollen genauso aussehen, wie die referenzierten Bilddateien:

```
@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)
  ;
}
```

Die Bilder können in allen Formaten vorliegen, die von `java.awt.image.BufferedImage` unterstützt werden. Das sind laut Javadoc `GIF`, `PNG`, `JPEG`, `BMP` und `WBMP`.

3.19. Layout - gerenderte Seitenausschnitte

Überblick

Vergleiche kompletter Seiten als gerenderte Bilder bereiten dann Schwierigkeiten, wenn sich auf einer Seite wechselnde Inhalte befinden. Der typische Vertreter für wechselnde Inhalte ist ein Tagesdatum.

Die Syntax für den Vergleich eines gerenderten Seitenausschnitts mit einer Bildvorlage ähnelt der Syntax für den Vergleich einer vollständigen Seite. Sie ist lediglich um eine Positionsangabe für die linke obere Ecke erweitert, mit der das Bild auf der Seite positioniert wird. Der Vergleich selber findet nur auf der Fläche statt, die der Größe des Bildes entspricht.

```
// 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)
```

Beispiel - Linker Rand auf jeder Seite

Wenn Sie prüfen wollen, ob der linke Rand jeder Seite mindestens 2 cm breit unbedruckt ist, können Sie das folgendermaßen testen:

```
@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)
  ;
}
```

Die Bilddatei, 2 cm breit und genauso hoch, wie eine ganze Seite, ist leer. Genauer gesagt, enhält sie die Hintergrundfarbe einer Seite. Das Beispiel prüft also, ob der Rand jeder Seite des Dokumentes ebenfalls „leer“ ist.

Jeder Ausschnitt benötigt eine x/y-Position auf der Seite des PDF-Dokumentes. Die Werte 0/0 entsprechen der linken oberen Ecke einer Seite.

Es wird davon ausgegangen, dass jede Seite das gleiche Format hat. Wenn Sie den Seitenrand eines Dokumentes mit unterschiedlichen Seitengrößen testen wollen, müssen Sie für jede Seitengröße einen eigenen Test schreiben.

Beispiel - Logo auf Seite 1 und 2

Sie wollen überprüfen, dass sich das Firmenlogo auf den ausgewählten Seiten an der erwarteten Position befindet:

```
@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 X/Y-Koordinaten der linken oberen Ecke des Seitenausschnitts definieren

❷ Seiten auswählen, siehe Kapitel 13.2: „Seitenauswahl“ (S. 160)

❸ Vergleichsmethode aufrufen

Mehrfache Vergleiche

In einem Test können mehrere Bildvergleiche auf mehreren Seiten gleichzeitig durchgeführt werden:

```
@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)
  ;
}
```

Sie sollten sich aber überlegen, ob es nicht besser ist, hierfür zwei Tests zu schreiben. Das entscheidende Argument für getrennte Tests ist, dass Sie unterschiedliche Namen für die Tests wählen können. Der hier gewählte Name ist für den Projektalltag nicht gut genug.

3.20. Lesezeichen/Bookmarks und Sprungziele

Überblick

Lesezeichen/Bookmarks dienen der schnellen Navigation innerhalb eines PDF-Dokumentes oder auch nach außen. Der Gebrauchswert eines Buches sinkt erheblich, wenn die einzelnen Kapitel nicht über ein Lesezeichen erreichbar sind. Mit den folgenden Tests sollen eventuelle Probleme frühzeitig erkannt werden:

```
// 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()
```

Betrachtet man Lesezeichen als **Absprungmarken**, dann sind „Named Destinations“ die **Sprungziele**. Sprungziele können von Lesezeichen genutzt werden, dienen aber auch als Ziel für HTML-Links. So kann aus einer Webseite direkt an eine bestimmte Stelle innerhalb eines PDF-Dokumentes gesprungen werden.

Für Sprungziele (Named Destinations) gibt es die Testmethoden:

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

Sprunziele, Named Destinations

Namen von Sprungzielen können einfach getestet werden:

```
@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")
  ;
}
```

Da die Namen auch über externe Links funktionieren müssen, dürfen sie keine Leerzeichen enthalten. Wird beispielsweise innerhalb von LibreOffice eine Sprungmarke mit Leerzeichen `"Export to PDF"` erstellt, erzeugt LibreOffice daraus beim Export nach PDF die Sprungmarke `"First2520Bookmark"`. Der Test muss dann diese Zeichenkette benutzen:

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

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

❶ `'2520'` steht für `'%20'`, was wiederum einem Leerzeichen entspricht.

Existenz von Lesezeichen/Bookmarks

Am einfachsten kann die Existenz von Lesezeichen überprüft werden:

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

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

Anzahl

Nach dem Test, ob ein PDF-Dokument überhaupt Lesezeichen hat, ist die Anzahl der Lesezeichen prüfenswert:

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

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

Text eines Lesezeichens (Label)

Eine wichtige Eigenschaft eines Lesezeichens ist das, was der Leser sieht: das Label. Deshalb sollten Sie testen, ob ein bestimmtes Lesezeichen genauso heißt, wie Sie es erwarten:

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

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

Sprungziele von Lesezeichen

Lesezeichen können sehr unterschiedliche Sprungziele haben. Deshalb gibt es für jedes Ziel geeignete Testmethoden.

Zielt **ein bestimmtes Lesezeichen** auf die gewünschte Seitenzahl:

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

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

Gibt es **irgendein Lesezeichen** zu einer gewünschten Seitenzahl:

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

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

Gibt es ein Lesezeichen, das zu einer bestimmten Sprungmarke zeigt:

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

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

Gibt es ein Lesezeichen, dessen Ziel eine bestimmte URI ist:

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

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

PDFUnit greift während der Tests nicht auf Webseiten zu. Es wird nicht geprüft, ob eine Webseite tatsächlich existiert, sondern nur, ob das Lesezeichen ein Link ist.

Und als Letztes soll überprüft werden, dass jedes Lesezeichen auf ein irgendein Ziel verweist:

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

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

3.21. Passwort

Überblick

Generell gilt die Aussage, dass Sie alle Tests sowohl mit Dokumenten ohne Passwortschutz durchführen können, als auch mit passwortgeschützten Dokumenten. Der Syntaxunterschied zeigt sich beim ersten Zugriff auf die Dokumente. Die Syntax sieht folgendermaßen aus:

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

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

❶ Wenn das Dokument nicht passwortgeschützt ist, wird nur der 1. Parameter benutzt.
❷ Wird der 2. Parameter benutzt, gilt das PDF-Dokument als passwortgeschützt.

Diese Syntax gilt unabhängig davon, ob das Dokument mit einem „User-Passwort“ oder einem „Owner-Passwort“ verschlüsselt wurde.

Tests auf das Password selber

Es gibt Tests, die sich direkt auf ein Passwort beziehen. Wenn Sie ein PDF-Dokument mit **einem** Passwort öffnen (User- und Owner-Passwort), können Sie das **andere** Passwort auf seine Richtigkeit prüfen:

```
// 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 Öffnen der Datei mit einem Passwort
❷❹ Überprüfen des anderen Passwortes

Passwörter sollten im Source-Code lediglich für Testdokumente hart kodiert werden, wobei auch diese Aussage aus Sicherheitsgründen bedenklich ist.

Test auf die Länge der Passwortverschlüsselung

Mit welcher Verschlüsselungslänge wurde verschlüsselt:

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

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

3.22. PDF/A

Überblick

Für die Prüfung, ob ein PDF-Dokument einen der PDF-Standards (PDF/A, PDF/X, etc.) einhält, gibt es viele Tools und API's. Aber nur einige von ihnen lassen sich für automatisierte Tests nutzen. PDFUnit verwendet den 'Preflight'-Parser von PDFBox.

Informationen zum 'Preflight'-Parser stehen auf der Projekt-Site https://pdfbox.apache.org/1.8/cookbook/pdfavalidation.html zur Verfügung.

Eine PDF/A-Validierung wird mit dieser Methode gestartet:

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

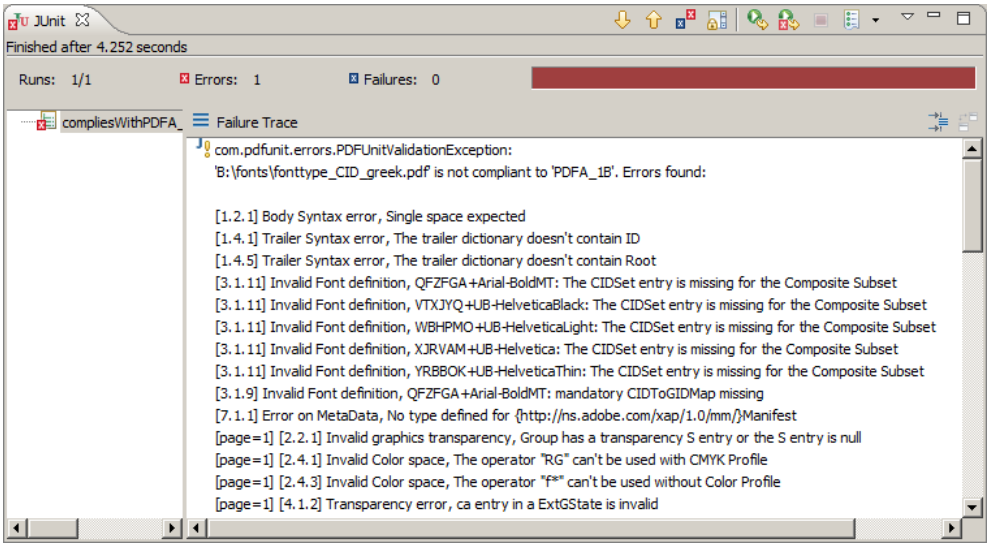
Beispiel

Das folgende Beispiel überprüft die Einhaltung des Standards 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 Es können Dateien verarbeitet werden, nicht aber Byte-Arrays und Streams.
❷ Mit den Konstanten PDFA_1A und PDFA_1B wird die PDF/A-Validierung gesteuert.

Die vom Preflight-Parser gelieferten Fehler werden von PDFUnit durchgereicht und sehen in Eclipse und als HTML beispielsweise so aus:



```
[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
```

Eine PDF/A-Validierung kann auch auf Verzeichnisse angewendet werden:

```
@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)
  ;
}
```

3.23. QR-Code

Überblick

Ein QR-Code ist ein 2-dimensionaler Code und kann wie der 1-dimensionale Barcode getestet werden. Dazu verwendet PDFUnit ebenfalls ZXing als Parser. Informationen zu ZXing liefert die Homepage des Projektes: https://github.com/zxing/zxing.

Für die Validierung von QR-Code stehen folgende Methoden zur Verfügung:

```
// 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(..)
```

Die folgenden QR-Code Formate werden von ZXing automatisch erkannt und können in PDFUnit-Tests verwendet werden:

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

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

Beispiel - Text aus QR Code prüfen

Dieser QR-Code enthält Text aus dem Roman 'Moby-Dick' von Herman Melville, unter anderem den, der im folgenden Test mit `containing` abgefragt wird:

```
@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)
  ;
}
```

Wenn in der ausgewählten Region mehrere Bilder existieren, muss jedes Bild den erwarteten Text enhalten. Whitespaces werden beim Vergleich normalisiert. Die Behandlung der Whitespaces kann aber durch einen Parameter von außen vorgegeben werden.

Der intern verwendete QR-Parser ZXing kennt viele QR-Code-Typen, dennoch nicht alle. Deshalb stellt PDFUnit noch eine externe Schnittstelle zur Verfügung, über die kundenspezifische QR-Parser eingebunden werden können. Die Verwendung dieser Schnittstelle wird separat dokumentiert. Schicken Sie ein kurzes Mail an info[at]pdfunit.com, um Informationen über die Einbindung eines individuellen QR-Parser zu erhalten.

Beispiel - QR Code in Teilen eines Bildes

Das folgende Beispiel verwendet ein Image, das zwei QR Codes enthält. Damit der Test nur auf einen QR-Code zielt, muss ein Rechteck definiert werden, dessen Referenzpunkt die linke obere Ecke des Images ist. Wichtig: Werte für Bildausschnitte sind in Points angegeben, Werte für Seitenausschnitte in Millimetern.

```
@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)
  ;
}
```

Beispiel - QR Code mit einem Image vergleichen

Und als Letztes soll noch ein Test gezeigt werden, indem ein QR-Code Bild aus einem PDF-Dokument mit einem extern vorliegenden QR-Code verglichen wird:

```
@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)
  ;
}
```

Wichtig: Der Dateityp des Bildes, beispielsweise PNG oder TIFF, muss zum Typ des QR-Code Bildes im PDF-Dokument passen, damit ein Bildvergleich funktioniert.

3.24. Schriften

Überblick

Schriften in PDF-Dokumenten sind keine einfach Sache, spielen aber spätestens dann eine Rolle, wenn eine verwendete Schriftarten nicht mehr zu den durch den PDF-Standard definierten 14 Schriften gehört. Auch für die Archivierung von PDF-Dokumenten spielen Schriften eine besondere Rolle. PDFUnit bietet unterschiedliche Testmethoden für Schriften, um verschiedene Bedürfnisse abzudecken:

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

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

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

Anzahl von Schriften

Was ist eine Schrift? Soll ein Subset einer Schrift als eigene Schrift gezählt werden? Für Softwareentwickler sind diese Fragen selten relevant, für ein Testwerkzeug schon.

In PDFUnit gelten zwei Schrift als 'gleich', wenn die für einen Test relevanten Vergleichskriterien gleiche Werte haben. Die unterstützten Vergleichskriterien werden durch Konstanten angegeben:

```
// 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
```

Die folgende Liste erläutert die Vergleichskriterien für Schriften:

Konstante	Beschreibung
ALLPROPERTIES	Alle Eigenschaften eines Fonts gelten als identifizierend. Von zwei verwendeten Schriften, die in allen Eigenschaften gleichwertig sind, wird nur eine gezählt.
BASENAME	Es werden nur die unterschiedlichen Basisschriften gezählt.
EMBEDDED	Mit diesem Filter werden sämtliche Schriften erfasst, die eingebettet sind.
NAME	Es werden Schriften mit unterschiedlichem Name gezählt.
NAME_TYPE	Die Kombination von Name und Typ einer Schrift gelten als identifizierender Teil.
TYPE	Es werden nur Schriften gezählt, die einen unterschiedlichen Typ haben.

Hier ein Beispiel, das alle möglichen Vergleichskriterien benutzt:

```
@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)
  ;
}
```

Schriftnamen

Tests, die auf die Namen von Schriften zielen, sind einfach:

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

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

Schriftnamen innerhalb eines PDF-Dokumentes enthalten gelegentlich noch ein Präfix, z.B. `FGNN-PL+ArialMT`. Weil dieses Präfix für Tests uninteressant ist, prüft PDFUnit lediglich, ob der gesuchte Schriftname in den Schriftnamen des PDF-Dokumentes als **Teilstring** enthalten ist.

Die Testmethoden können verkettet werden:

```
@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")
  ;
}
```

Weil es gelegentlich interessant ist, zu wissen, dass eine bestimmte Schriftart in einem Dokument **nicht** enthalten ist, gibt es auch hierfür eine Testmethode:

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

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

Schrifttypen

Sie können prüfen, ob **alle** in einem PDF-Dokument verwendeten Schrifttypen einem bestimmten Typ entsprechen:

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

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

Die prüfbaren Schrifttypen sind als Konstanten deklariert:

```
// 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
```

3.25. Seitenzahlen als Testziel

Überblick

Es ist manchmal sinnvoll, zu prüfen, ob ein erzeugtes PDF-Dokument genau eine Seite hat. Oder Sie müssen sicherstellen, dass das Dokument weniger als 6 Seiten umfasst, weil sonst ein höheres Briefporto anfällt. PDFUnit bietet deshalb Testmethoden an, die sich auf die Anzahl von Seiten beziehen:

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

Beispiele

Eine konkrete Seitenanzahl wird folgendermaßen überprüft:

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

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

Es sind aber auch Tests mit minimaler oder maximaler Seitenzahl möglich:

```
@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 Die Werte für Ober- und Untergrenze gelten exklusiv.

Die Methoden können verkettet werden:

```
@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)
  ;
}
```

Verzichten Sie nicht auf Tests mit Seitenzahlen weil Sie denken, sie seien zu einfach. Erfahrungsgemäß finden Sie im Umfeld eines einfachen Tests überraschende Dinge, die Sie ohne den Test nicht gefunden hätten.

3.26. Signaturen - Unterschriebenes PDF

Überblick

Wenn im Zeitalter der elektronischen Kommunikation vertraglich relevante Informationen in Form von PDF-Dokumenten ausgetauscht werden, muss irgendwie sichergestellt werden, dass die Daten auch wirklich von demjenigen stammen, von dem sie vorgeben, zu sein. Für diesen Zweck gibt es Zertifikate. Sie bestätigen - unabhängig von PDF-Dokumenten - die Echtheit von Personen- oder Unternehmensdaten. Mit einem Zertifikat kann der Inhalt von Dokumenten unterschrieben (signiert) werden. Dafür bietet PDF ein spezielles Signaturfeld an.

PDFUnit stellt für Signaturen zahlreiche Testmethoden zur Verfügung:

```
// 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()
```

Ein „signiertes PDF“ darf nicht mit einem „zertifizierten PDF“ verwechselt werden. Ein „zertifiziertes PDF“ garantiert die Einhaltung bestimmter Eigenschaften, die für eine Verarbeitung benötigt werden. Tests für zertifizierte PDF-Dokumente sind im Kapitel 3.38: „Zertifiziertes PDF“ (S. 83) beschrieben.

Existenz

Der einfachste Test ist, zu prüfen, ob ein Testdokument überhaupt signiert ist:

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

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

Wenn es mehrere Unterschriftsfelder gibt, kann auch geprüft werden, ob alle Felder signiert sind:

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

Die Methode `.allUnsigned()` prüft genau das Gegenteil.

Ein spezielles Unterschriftsfeld kann auf folgende Weise validiert werden:

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

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

Weiterhin kann die Existenz von Unterschriftsfeldern geprüft werden:

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

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

Und führt man die Intention der vorhergehende Tests weiter fort, muss die Frage gestellt werden, ob bestimmte Unterschriftsfelder unterschrieben sind. Die Antwort liefert der folgende Test:

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

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

Der umgekehrte Fall, dass ein Unterschriftsfeld keine Unterschrift enthält, kann mit der Funktion `.hasSignatureField("name").withoutSignature()` überprüft werden.

Anzahl der Unterschriften

Da ein PDF-Dokument mehrere Unterschriften enthalten kann, gibt es auch einen Test, der lediglich die Anzahl der Signaturen überprüft:

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

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

Unterschriftsdatum

Es ist manchmal interessant, festzustellen, wann ein PDF-Dokument unterschrieben wurde:

```
@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 Der Vergleich findet immer auf der Basis von Jahr-Monat-Tag statt.

Grund einer Unterschrift

Möglicherweise ist es für Tests nicht so interessant, den Grund einer Unterschrift zu überprüfen. Falls ein solcher Test aber notwendig sein sollte, dann sieht er folgendermaßen aus:

```
@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")
  ;
}
```

Vor dem Vergleich werden die Whitespaces normalisiert.

Name des Unterzeichner

Auch der Name dessen, der ein PDF-Dokument unterschrieben hat, ist für Testzwecke weniger interessant, als für die produktive Verarbeitung von PDF-Dokumenten. Dennoch gibt es dafür eine Testmethode:

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

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

Mit dem folgenden Test kann eine erwartete Unterschrift auch unabhängig von einem bestimmten Feld getestet werden:

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

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

Umfang der Unterschrift

Eine Unterschrift kann sich laut PDF Standard auch auf Teile eines Dokumentes beziehen. Deshalb ist es möglich, zu testen, ob eine Unterschrift das komplette Dokument abdeckt:

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

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

Zusammenhängend testen

Mehrere Tests, die sich auf ein Unterschriftsfeld beziehen, können verkettet werden:

```
@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()
  ;
}
```

Überlegen Sie sich aber einen besseren Namen für diesen Test!

3.27. Sprachinformation (Language)

Überblick

PDF-Dokumente können für Sehbehinderte durch Screenreader-Programme vorgelesen werden. Dazu ist es hilfreich, wenn das Dokument Auskunft über seine Sprache geben kann.

Diese Methoden stehen für Tests der Sprache zur Verfügung:

```
// Tests for PDF language:

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

Beispiele

Das folgende Beispiel testet, ob das Dokument die Kennung für die englische Sprache in Großbritannien enthält:

```
@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 PDF-typische Schreibweise

❷ Java-typische Schreibweise

Die Zeichenkette für die Sprachbezeichnung kann in beliebiger Groß-/Kleinschreibung angegeben werden. Unterstrich und Bindestrich werden ebenfalls gleichbehandelt.

Es kann auch `java.util.Locale` direkt verwendet werden:

```
@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)
  ;
}
```

Ein PDF-Dokument mit der internen Länderkennung `"en_GB"` liefert einen grünen Test, wenn es auf `Locale.en` getestet wird. Dagegen gilt ein Test als fehlerhaft, wenn ein Dokument mit der internen Länderkennung `"en"` auf `Locale.UK` getestet wird.

Ein PDF-Dokument, das **keine** Länderkennung haben soll, kann ebenfalls daraufhin getestet werden:

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

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

3.28. Texte

Überblick

Der häufigste Testfall für PDF-Dokumente ist vermutlich, die Existenz erwarteter Texte zu überprüfen. Dafür stehen vielfältige Methoden zur Verfügung:

```
// 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 Das Kapitel 13.5: „Behandlung von Whitespaces“ (S. 164) beschreibt die unterschiedlichen Möglichkeiten, mit Whitespaces umzugehen.

❷ Das Kapitel 3.39: „ZUGFeRD“ (S. 84) beschreibt, wie der sichtbare Inhalt von PDF-Dokumenten mit den Inhalten der unsichtbaren ZUGFeRD-Daten verglichen werden kann.

Text auf bestimmten Seiten

Wenn Sie einen bestimmten Text auf der ersten Seite eines Anschreibens suchen, sieht ein Test folgendermaßen aus:

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

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

Ein Text auf der letzten Seite wird folgendermaßen überprüft:

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

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

Auch Tests mit beliebigen individuellen Seiten sind möglich:

```
@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 Mit der Methode `getPages(Integer[])` können beliebige Seitenkombinationen definiert werden. Für eine einzelne Seite kann auch der Singular `PagesToUse.getPage(int)` benutzt werden.

Für typische Seiten stehen Konstanten zur Verfügung, u.a. `FIRST_PAGE`, `LAST_PAGE`, `EVEN_PAGES` und `ODD_PAGES`. Das Kapitel 13.2: „Seitenauswahl“ (S. 160) beschreibt die Seitenauswahl ausführlich.

Text auf allen Seiten

Für Prüfungen, die sich auf alle Seiten beziehen, stehen drei weitere Konstanten zur Verfügung: `ANY_PAGE`, `EACH_PAGE` und `EVERY_PAGE`. Die letzten beiden sind funktional identisch und existieren nur aus sprachlichen Gründen doppelt.

```
@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")
  ;
}
```

Die Konstanten `EVERY_PAGE` und `EACH_PAGE` fordern, dass der zu suchende Text wirklich auf **jeder** Seite existiert. Mit der Konstanten `ANY_PAGE` reicht es, wenn der erwartete Text auf **irgendeiner** Seite des Dokumentes vorkommt.

Text in Seitenausschnitten

Text kann aber nicht nur auf vollständigen Seite gesucht werden, sondern auch in Seitenausschnitten. Das Kapitel 3.30: „Texte - in Seitenausschnitten“ (S. 71) beschreibt diesen Aspekt ausführlich.

Fließende Seitenangaben mit Unter- und Obergrenze

Es kann den Wunsch geben, Texte auf jeder Seite zu überprüfen, aber nicht auf der ersten Seite. Ein solcher Test sieht folgendermaßen aus:

```
@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 Der Wert '3' ist eine exclusive Untergrenze. Die Validierung beginnt mit Seite 4.

Die Zählung der Seitenzahlen beginnt mit „1“.

Ungültige Seitenobergrenzen sind nicht unbedingt ein Fehler. Im folgenden Beispiel wird Text auf irgendeiner Seite zwischen 1 und 99 gesucht. Obwohl das Dokument nur 4 Seiten hat, endet der Test erfolgreich, weil die gesuchte Zeichenkette auf Seite 1 gefunden wird:

```
/**
 * 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")
  ;
}
```

Seitenübergreifende Textvergleiche

Im folgenden Beispiel wird ein Text gesucht, der sich über 2 Seiten erstreckt. Diese 2 Seiten müssen nicht volle Seiten sein, sondern können - wie im Beispiel - auch Seitenausschnitte sein. Auf dieses Weise kann Fließtext ohne Header und Footer analysiert werden:

```
@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)
  ;
}
```

Nach der Methode `.hasText()` stehen alle oben genannte Vergleichsmethoden zur Verfügung.

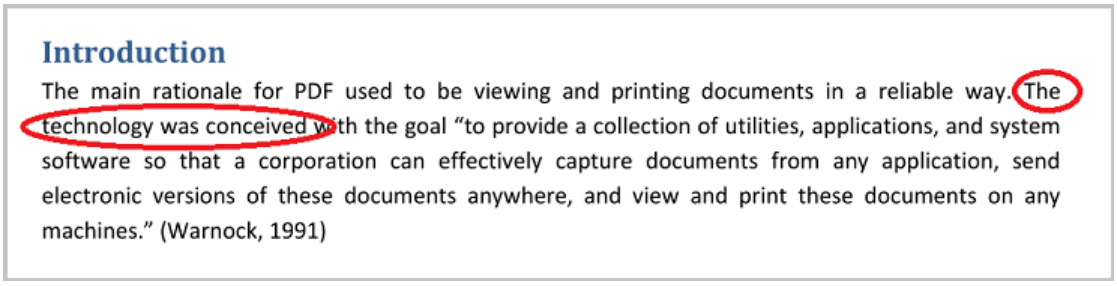
Verneinte Suche

Auch die Abwesenheit von Text kann ein wichtiges Testziel sein, vor allem wenn es auf Teile einer Seite beschränkt wird. Die Tests dazu entsprechen der allgemeinen Umgangssprache:

```
@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()
  ;
}
```

Zeilenumbrüche im Text

Zeilenumbrüche im Text werden beim Vergleich normalisiert, sowohl Zeilenumbrüche im Text der PDF-Seite, als auch die im Suchstring. Im folgenden Beispiel stammt der zu suchende Text aus dem Dokument „Digital Signatures for PDF Documents“ von Bruno Lowagie (iText Software). Der erste Absatz sieht optisch so aus:



Introduction

The main rationale for PDF used to be viewing and printing documents in a reliable way. The technology was conceived with the goal “to provide a collection of utilities, applications, and system software so that a corporation can effectively capture documents from any application, send electronic versions of these documents anywhere, and view and print these documents on any machines.” (Warnock, 1991)

Tests auf den markierten Text ohne Berücksichtigung auf Zeilenumbrüche sehen folgendermaßen aus. Beide laufen erfolgreich durch, weil Whitespaces normalisiert werden.

```
/**
 * 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)
  ;
}
```

Sollte eine Normalisierung der Whitespaces nicht erwünscht sein, so kann bei den meisten Textvergleichsmethoden noch ein zweiter Parameter übergeben werden, der die Art der Whitespace-Behandlung steuert. Folgende Möglichkeiten gibt es:

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

Keine leeren Seiten

Sie können auch überprüfen, dass Ihr PDF-Dokument keine leere Seiten enthält:

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

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

Mehrere Suchbegriffe gleichzeitig

Wenn auf einer Seite mehrere Texte gesucht werden, ist es lästig, für jeden Suchbegriff einen eigenen Funktionsaufruf zu schreiben. Deshalb können die Funktionen `containing(..)` und `notContaining(..)` mit mehreren Suchbegriffen aufgerufen werden:

```
@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")
  ;
}
```

Die Tests sind erfolgreich, wenn im ersten Beispiel **alle** Suchbegriffe gefunden werden, oder im zweiten Beispiel eben **alle nicht**.

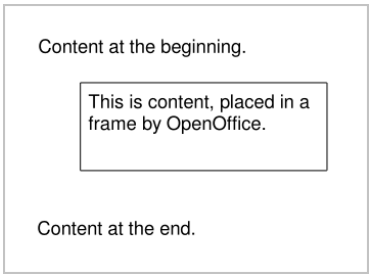
Verkettung von Textvergleichen

Textvergleiche können verkettet werden, sie beziehen sich dann jeweils auf die zuvor spezifizierten Seiten:

```
@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")
  ;
}
```

Potentielles Problem mit „Fließtext“

Die sichtbare Reihenfolge des Textes einer PDF-Seite entspricht nicht zwingend der Textreihenfolge innerhalb des PDF-Dokumentes. Im folgenden Screenshot ist der umrahmte Text ein eigenes Textobjekt, das nicht zum 'normalen' Fließtext der Seite gehört. Deshalb funktioniert auch der nachfolgende Test.




```
@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)

  ;
}
```

Wenn Sie sich den Rahmen wegdenken, könnte der Eindruck entstehen, dass der Text „im Rahmen“ Teil des Fließtextes wäre. Ein Test mit einem Erwartungswert, der diesem vermeintlichen Fließtext entspricht, würde fehlschlagen.

3.29. Texte - in Bildern (OCR)

Überblick

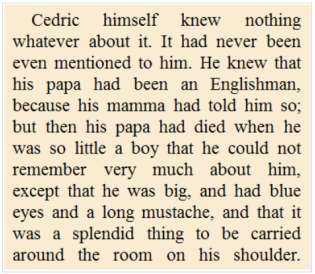
PDFUnit kann Text aus Bildern extrahieren und diesen Text auf die gleiche Weise wie Fließtext validieren. Wie in anderen PDFUnit-Tests entspricht die Syntax dieser OCR-Tests weitestgehend der Umgangssprache: jeder OCR-Test beginnt mit den Methoden `hasImage().withText()` bzw. `hasImage().withTextInRegion()`. Mit folgenden Methoden können Texte in Bildern validiert werden:

```
// 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(..)
```

Für die Texterkennung benutzt PDFUnit den OCR-Processor Tesseract.

Beispiel - Text aus Bildern prüfen

Das folgende Beispiel basiert auf einer PDF-Datei, die ein Image enthält, das den Text des Romans "Der kleine Lord" enthält. Das Image hat einen farbigen Hintergrund.


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)
  ;
}
```

Normalisierung des OCR-Textes

Wenn man den erwarteten Text des vorhergehenden Tests mit dem Bild vergleicht, fällt der Zeilenumbruch hinter dem Wort 'nothing' auf. Trotz dieses Zeilenumbruchs ist der Test erfolgreich, weil alle Leerzeichen im Zuge einer OCR-Normalisierung vor dem Vergleich eliminiert werden.

Schritte der Normalisierung der OCR-Text:

- Zeichen werden in Kleinbuchstaben umgewandelt
- Alle Leerzeichen (Whitespaces) werden entfernt
- 12 verschiedene Trennzeichen (Hyphens) werden entfernt
- 10 verschiedene Unterstriche (Underscore) werden entfernt
- Satzzeichen (Punctuation) werden entfernt

Die Ergebnisse der Texterkennung können verbessert werden, wenn der Processor 'trainiert' wurde. Sprachspezifische Trainingsdaten können von https://github.com/tesseract-ocr/tessdata heruntergeladen werden.

Beispiel - Text in Bildausschnitten

Es könnte sein, dass ein zu validierender Text in einem bestimmten Bereich des Bildes erwartet wird. Um eine solche Situation abzudecken, kann ein Bildausschnitt definiert und übergeben werden:

```
@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)
  ;
}
```

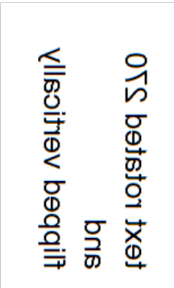
Die Einheit für die Definition von Bildausschnitten ist immer 'Pixel'. 'Millimeter' als Einheit wären schwierig in der Handhabung, weil Bilder in PDF skaliert sein können. Dadurch sind sie physisch größer oder kleiner, als sie aussehen. Um die richtigen Maße für einen Bildausschnitt zu bekommen, extrahieren Sie die Bilder aus einem PDF und vermessen den Ausschnitt anschließend mit einem gängigen Bildbearbeitungsprogramm. Für die Extraktion stellt PDFUnit das Tool `ExtractImages` zur Verfügung. Es ist in Kapitel 9.3: „Bilder aus PDF extrahieren“ (S. 126) beschrieben.

Beispiel - Gespiegelter und gedrehter Text in Bildern

Wasserzeichen oder andere Texte in Bildern sind möglicherweise absichtlich gedreht oder gespiegelt. Auch solche Texte können validiert werden. Dafür gibt es folgende Methoden:

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

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

Hier ein Bild, dessen Inhalt mit dem nachfolgenden Test validiert wird.



Der Text in dem Bild ist um 270 Grad gedreht und zusätzlich vertikal umgeklappt (flipped). Wenn ein Test diese Werte berücksichtigt, kann der tatsächliche Text gegen einen Erwartungswert geprüft werden:

```
@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)
  ;
}
```

Die erlaubten Werte für eine Drehung oder Spiegelung sind:

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

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

3.30. Texte - in Seitenausschnitten

Überblick

Es gibt die Situation, dass sich ein bestimmter Text mehrmals auf einer Seite befindet, aber nur eine der Stellen im Test benutzt werden soll. Für diese Anforderung kann der Suchbereich auf einen Teil einer Seite beschränkt werden. Die Syntax dazu ist einfach:

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

Beispiel

Das folgende Beispiel zeigt die Definition und Benutzung eines Seitenausschnitts:

```
@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.")
  ;
}
```

Für Vergleiche von Text in Seitenausschnitten stehen alle Vergleichsmethoden zur Verfügung, die auch für ganze PDF-Seiten zur Verfügung stehen. Sie sind in Abschnitt 13.4: „Textvergleich“ (S. 163) ausführlich beschrieben.

Die Einschränkung eines Vergleiches auf einen Ausschnitt einer Seite ist sowohl für Text, als auch für Bilder möglich.

3.31. Texte - Reihenfolge mehrerer Texte

Überblick

Es gibt gelegentlich den Bedarf, sicherzustellen, dass **zuerst** ein Text im PDF-Dokument vorhanden ist und **danach** ein anderer Text. Für diese Anforderungen gibt es in PDFUnit folgende Testmethoden:

```
// 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[])
```

Beispiel - hasText() ... first() ... then()

Im folgenden Beispiel wird zuerst die Überschrift von Kapitel 5 gesucht, anschließend ein Textstück, das zu Kapitel 5 gehört und als Letztes die Überschrift von Kapitel 6. Und diese Suche wird noch auf die Seite 2 beschränkt:

```
@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)
  ;
}
```

Der Textvergleich findet intern mit der Methode `containing(..)` statt. Whitespaces werden vor dem Vergleich normalisiert, es sei denn, sie werden als Parameter mitgegeben: `first(expectedText, WhitespaceProcessing)`. Die nachfolgende Methode `then()` behandelt Whitespace auf gleiche Weise wie die Methode `first()`.

Die Funktion `then(..)` kann beliebig oft wiederholt werden.

Beispiel - hasText() inOrder()

Das nächste Beispiel erfüllt den gleichen Zweck. Die gesuchten Textstücke werden aber alle gleichzeitig übergeben:

```
@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)
  ;
}
```

Die Reihenfolge der Suchbegriffe muss der Textreihenfolge im PDF-Dokument entsprechen.

Ein interner Textvergleich verwendet die Methode `contains(..)`, Whitespaces werden vorher normalisiert.

Eine abweichende Behandlung der Whitespaces kann beim Methodenaufruf als Parameter angegeben werden: `inOrder(WhitespaceProcessing, expectedText[])`.

Die Suchbegriffe können auch als Array übergeben werden.

3.32. Texte - senkrecht, schräg und überkopf

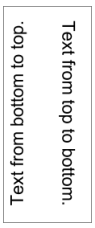
Überblick

Es gibt Dokumente, die tragen am Rand eine senkrechte Textmarke, die in weiteren Schritten der Verarbeitung zur Identifikation dient. Auch senkrechte Tabellenüberschriften treten gelegentlich in Dokumenten auf.

Um Text zu testen, der aus der normalen Position um einen beliebigen Winkel (-180 bis +180 Grad) gedreht wurde, stehen die gleichen Funktionen zur Verfügung, wie auch für „normalen“ Text.

Beispiel

Das Beispieldokument enthält die zwei senkrechten Texte:



Mit den richtigen Werten für einen Seitenausschnitt sieht der Test folgendermaßen aus:

```
// 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)
  ;
}
```

Senkrechter Text wird vom Parser mit vielen Zeilenumbrüchen gelesen. Deshalb ist es unbedingt nötig, `WhitespaceProcessing.IGNORE` zu benutzen.

Beachten Sie: Es handelt sich bei diesem kopfwärts gestellten Text immer noch um Text mit der Laufrichtung LTR (left-to-right). Texte mit der Laufrichtung RTL (right-to-left) werden von PDFUnit auch unterstützt. Beispiele dafür zeigt Kapitel 3.33: „Texte - von rechts nach links (RTL)“ (S. 74).

3.33. Texte - von rechts nach links (RTL)

Überblick

Tests mit RTL-Text unterscheiden sich nicht von Tests mit LTR-Text. Somit stehen alle Vergleichsmethoden für Texte zur Verfügung, wie für LTR-Text:

```
// 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(..)
```

Beispiel - 'hello, world' von rechts nach links

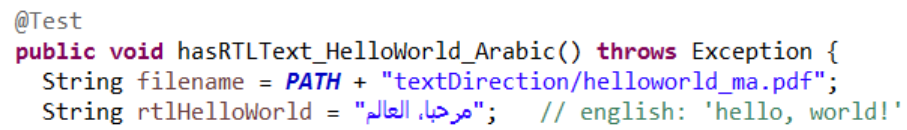
Die nachfolgenden Testbeispiele beziehen sich auf zwei PDF-Dokumente, die den Text 'hello, world' auf Arabisch und auf Hebräisch enthalten:

مرحبا، العالم	שלום, עולם

```
// 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)
  ;
}
```

Es ist durchaus nicht uninteressant, dass der Eclipse-Editor die unterschiedlichen Schriften und Schreibrichtungen problemlos umsetzt. Hier ein Screenshot mit dem Java-Code aus dem vorhergehenden Beispiel:



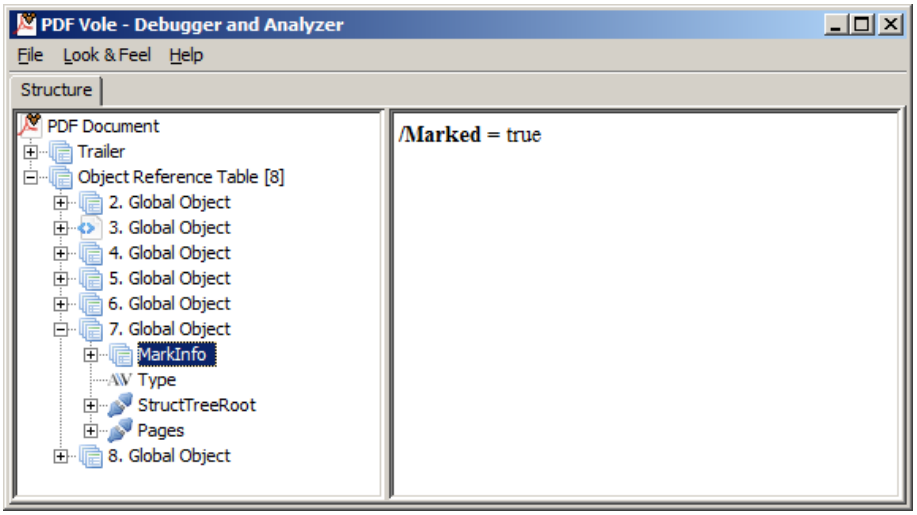
PDFUnit benutzt als PDF-Parser PDFBox. PDFBox erkennt rechts-links ausgerichteten Text und wandelt ihn in einen Java String um, ohne dass Entwickler dafür besondere Methoden aufrufen. Gratulation dafür an das Entwickler-Team.

3.34. Tagging

Überblick

Der PDF-Standard „ISO 32000-1:2008“ sagt in Kapitel 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.“ (Zitat aus: .)

Obwohl diese Formulierung nur das Wort „shall“ enthält, prüft PDFUnit, ob ein PDF-Dokument ein Dictionary mit dem Namen `/MarkInfo` enthält. Wenn darin ein Eintrag mit dem Key `/Marked` und dem Wert `true` existiert, gilt es für PDFUnit als „tagged“.



Die folgenden Methoden stehen zur Verfügung:

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

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

Beispiele

Die einfachsten Test überprüfen, ob Tagging-Informationen überhaupt vorhanden sind:

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

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

Etwas weitergehend sind Prüfungen, die auf die Existenz bestimmter Tags prüfen:

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

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

Als Letztes können Werte bestimmter Tags verifiziert werden:

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

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

3.35. Version

Überblick

Automatisch erzeugte PDF-Dokumente müssen gelegentlich einer bestimmten Version entsprechen, weil sie durch andere Werkzeuge weiterverarbeitet werden müssen. Das kann getestet werden:

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

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

Eine bestimmte Version

Für gängige PDF-Versionen gibt es Konstanten, die als Parameter verwendet werden:

```
// 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
```

Ein Beispiel für den Test auf Version „1.4“:

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

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

Versionsbereiche

Die gleichen Konstanten können als Ober- und Untergrenze genutzt werden:

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

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

❶❷ Die Ober- und Untergrenzen gelten exklusiv.

Auch zukünftige PDF-Versionen können getestet werden. Die erwartete Version muss als String im Format "%1.1f" angegeben werden.

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

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

3.36. XFA Daten

Überblick

Die „XML Forms Architecture, (XFA)“ ist eine Erweiterung der PDF-Strukturen um XML-Informationen, mit dem Ziel, PDF-Formulare in den Prozessen eines Workflow's besser verarbeiten zu können.

XFA Formulare sind nicht kompatibel zu „AcroForms“. Deshalb sind die PDFUnit-Tests für Acroforms auch nicht verwendbar. Test auf XFA-Daten basieren überwiegend auf XPath:

```
// Methods around XFA data:
.hasXFAData()
.hasXFAData().matchingXPath(..)
.hasXFAData().withNode(..)

.hasNoXFAData()
```

Existenz und Abwesenheit von XFA

Der erste Test zielt auf die reine Existenz von XFA-Daten:

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

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

Es ist auch möglich, explizit zu testen, dass ein PDF-Dokument **keine** XFA-Daten enthält:

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

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

Einzelne XML-Tags validieren

Das im nächsten Beispiel verwendete PDF-Dokument enthält folgende XFA-Daten (Ausschnitt):

```
<?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>
```

Um auf einen bestimmten Knoten zu testen, muss eine Instanz von `com.pdfunit.XMLNode` mit dem XPath-Ausdruck für den Knoten erzeugt werden. Als zweiter Parameter wird der erwartete Wert übergeben:

```
@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 analysiert die XFA-Daten des aktuellen PDF-Dokumentes und ermittelt die Namensräume selbständig, lediglich der Default-Namespace muss angegeben werden.

Für die internen Verarbeitung ergänzt PDFUnit vor dem Knoten den Pfad-Bestandteil `"//"`. Aus diesem Grund darf der Knoten im Test kein Pfad sein, der die Wurzel `"/"` enthält.

Sollte der XPath-Ausdruck für den Knoten zu mehreren Treffern führen, wird der erste Treffer verwendet.

Wenn das erwartete Ergebnis für den XPath-Ausdruck bei der Erzeugung der XMLNode-Instanz angegeben wird, dann wird dieses auch mit dem tatsächlichen Ergebnis verglichen. Andernfalls wird nur die Existenz des Knotens festgestellt.

Prüfungen auf Attribut-Knoten sind selbstverständlich auch möglich:

```
@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-basierte XFA-Tests

In PDFUnit gibt es noch die Methode `matchingXPath(..)`, um das ganze Potential von XPath nutzen zu können. Die nächsten beiden Beispiele zeigen, was damit machbar ist:

```
@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)
  ;
}
```

Eine kleine Einschränkung muss genannt werden. Die XPath-Ausdrücke können nur mit den Möglichkeiten ausgewertet werden, die die verwendete XPath-Implementierung bietet. PDFUnit nutzt normalerweise die JAXP-Implementierung des verwendeten JDK. Damit ist die XPath-Kompatibilität aber vom JDK/JRE abhängig und unterliegt dem Wandel der Zeit.

Das Kapitel 13.12: „JAXP-Konfiguration“ (S. 172) erläutert am Beispiel von Xerces, wie ein beliebiger XML-Parser genutzt werden kann.

Default-Namensraum in XPath

XML-Namensräume werden automatisch ermittelt. Wenn aber ein Default-Namensraum verwendet wird, muss dieser als Instanz von `DefaultNamespace` angegeben werden. Für diese Instanz muss in Java ein Prefix verwendet werden. Der Wert für das Prefix kann beliebig sein:

```
@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)
  ;
}
```

Auch bei der Verwendung der Klasse `XMLNode` muss der Default-Namensraum mitgegeben werden:

```
/**
 * 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)
  ;
}
```

3.37. XMP-Daten

Überblick

XMP steht für „Extensible Metadata Platform“ und ist ein von Adobe initiierter offener Standard, Metadaten in beliebige Dateitypen einzubetten. Nicht nur PDF-Dokumente, auch Bilder können mittels XMP Informationen über Ort, Format und andere Daten einbinden.

Die Metadaten in PDF sind für die Weiterverarbeitung durch andere Programme wichtig und sollten daher richtig sein. Zum Testen bietet PDFUnit die gleichen Methoden an, wie für XFA-Daten:

```
// Methods to test XMP data:
.hasXMPData()
.hasXMPData().matchingXPath(..)
.hasXMPData().withNode(..)

.hasNoXMPData()
```

Existenz und Abwesenheit von XMP

Die nächsten Beispiele zeigen die Prüfung der An- bzw. Abwesenheit von XMP-Daten:

```
@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()
  ;
}
```

Einzelne XML-Tags validieren

Die Existenz und auch die Werte einzelner XML-Knoten können überprüft werden. Das nächsten Beispiele basiert auf dem folgenden XML-Ausschnitt:

```
<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>
```

Diese XMP-Daten wurden mit dem Hilfsprogramm `ExtractXMPData` aus einem PDF-Dokumente exportiert. Kapitel 9.14: „XMP-Daten nach XML extrahieren“ (S. 138) beschreibt die Verwendung dieses Hilfprogramms.

Im folgenden Beispiel wird die Existenz zweier XML-Knoten geprüft. Für jeden zu testenden Knoten muss eine Instanz von `com.pdfunit.XMLNode` erzeugt werden:

```
@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)
  ;
}
```

Soll auch der Wert eines Knotens überprüft werden, muss der erwartete Wert als zweiter Parameter an den Konstruktor von `XMLNode` übergeben werden:

```
@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)
  ;
}
```

Der XPath-Ausdruck für den Knoten darf die Wurzel (document root) nicht enthalten, weil PDFUnit intern `//` ergänzt.

Existiert ein gesuchter Knoten mehrfach innerhalb der XMP-Daten, wird der erste Treffer verwendet.

Der Knoten darf selbstverständlich auch ein Attribut-Knoten sein.

XPath-basierte XMP-Tests

In PDFUnit gibt es noch die Methode `matchingXPath(..)`, um das ganze Potential von XPath nutzen zu können. Die nächsten beiden Beispiele zeigen, was damit machbar ist:

```
@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)
  ;
}
```

Der Funktionsumfang der Verarbeitung der XPath-Ausdrücke hängt vom verwendeten XML-Parser bzw. der XPath-Engine ab. PDFUnit verwendet die des JDK/JRE.

Im Kapitel 13.12: „JAXP-Konfiguration“ (S. 172) wird erläutert, wie ein beliebiger XML-Parser genutzt werden kann.

Default-Namensraum in XPath

XML-Namensräume werden automatisch ermittelt. Wenn aber ein Default-Namensraum verwendet wird, muss dieser als Instanz von `DefaultNamespace` angegeben werden. Und es muss für den Default-Namensraum ein Prefix verwendet werden. Der Wert für das Prefix kann beliebig sein.

```
@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)
  ;
}
```

Auch bei der Verwendung der Klasse `XMLNode` muss der Default-Namensraum mitgegeben werden:

```
@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)
  ;
}
```

3.38. Zertifiziertes PDF

Überblick

Ein „zertifiziertes PDF“ ist ein normales PDF mit Zusatzinformationen. Es enthält Informationen daüber, wie das Dokument verändert werden darf.

Für zertifizierte PDF-Dokumente bietet PDFUnit folgende Testmethoden:

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

Beispiele

Am Anfang steht der einfache Test, ob ein Dokument überhaupt zertifiziert ist:

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

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

Als Nächstes kann der Grad der Zertifizierung überprüft werden:

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

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

Zertifizierungsgrade

PDFUnit stellt die Zertifizierungs-Level als Konstanten zur Verfügung:

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

3.39. ZUGFeRD

Überblick

Ein Zusammenschluss von Verbänden und Unternehmen der Wirtschaft und des Öffentlichen Dienstes, das „Forum elektronische Rechnung Deutschland“ (FeRD), hat am 25.06.2014 die Version 1.0 eines XML-Formates für den Austausch elektronischer Rechnungen beschlossen. Die Spezifikation selber wird ZUGFeRD (Zentraler User Guide des Forums elektronische Rechnung Deutschland) genannt. Weitreichende Informationen findet man im Internet bei Wikipedia (ZUGFeRD) , bei 'FeRD' und bei der PDF-Association im 'Leitfaden zu PDF-A3 und ZUGFeRD'.

Viele Validierungstools überprüfen zwar die Übereinstimmung der XML-Daten mit der XML-Schema Spezifikation, nicht aber die Übereinstimmung der XML-Daten (unsichtbar) mit den Daten des gedruckten PDF-Dokumentes (sichtbar). Das ist mit PDFUnit einfach, sofern Sie einen Seitenbereich definieren können, in dem der zu prüfende Wert vorkommen muss.

Für Tests mit ZUGFeRD-Daten stehen diese Methoden zur Verfügung:

```
// Methods to test ZUGFeRD data:

.hasZugferdData().matchingXPath(xpathExpression)
.hasZugferdData().withNode(xmlNode)

.compliesWith().zugferdSpecification()
```

Die folgenden Beispiele beziehen sich auf das vom ZUGFeRD-Standard 1.0 mitgelieferte Beispiel-Dokument 'ZUGFeRD_1p0_BASIC_Einfach.pdf'. In jedem Beispiel wird zuerst der Test mit PDFUnit gezeigt, dann der dazugehörende Teil des PDF-Dokumentes und der XML-Daten.

Wenn Sie die ZUGFeRD-Daten eines Dokumentes sehen wollen, exportieren Sie sie am einfachsten, indem Sie das Dokument mit dem Adobe Reader® öffnen und von dort aus die angezeigte Datei 'ZUGFeRD-invoice.xml' speichern.

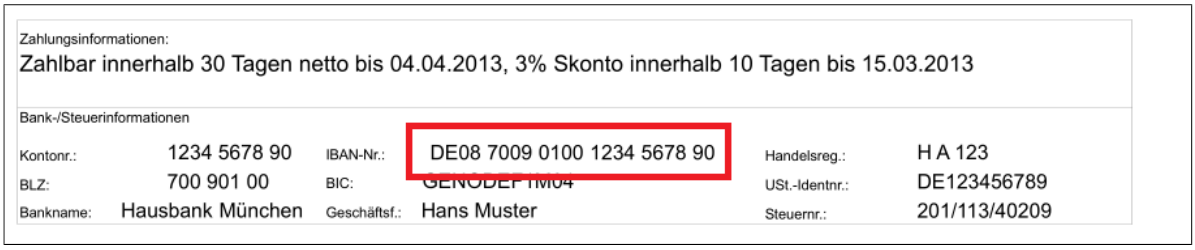
PDF-Inhalte mit ZUGFeRD-Inhalten vergleichen - IBAN

In diesem Beispiel wird ein Wert für die IBAN sowohl in den XML-Daten, als auch im PDF-Text gesucht. Das geschieht mit zwei `AssertThat`-Anweisungen. Für die Suche auf der PDF-Seite muss der pas-

sende Ausschnitt (`regionIBAN`) und für die Suche in den XML-Daten der Name des XML-Knotens angegeben werden (`nodeIBAN`).

```
@Test
public void validateIBAN() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";
  String expectedIBAN = "DE08700901001234567890";

  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)
  ;
}
```

Ausschnitt der PDF-Seite:



Ausschnitt der ZUGFeRD Daten:

```
<ram:SpecifiedTradeSettlementPaymentMeans>
    <ram:PayeePartyCreditorFinancialAccount>
        <ram:IBANID>DE08700901001234567890</ram:IBANID>
    </ram:PayeePartyCreditorFinancialAccount>
    <ram:PayeeSpecifiedCreditorFinancialInstitution>
        <ram:BICID>GENODEF1M04</ram:BICID>
    </ram:PayeeSpecifiedCreditorFinancialInstitution>
</ram:SpecifiedTradeSettlementPaymentMeans>
```

PDF-Inhalte mit ZUGFeRD-Inhalten vergleichen - noch einfacher

Es geht aber noch einfacher, nämlich mit `hasText().containingZugferdData(xmlNode)`. Die Methode extrahiert zuerst den Text aus dem angegebenen ZUGFeRD-Knoten und vergleicht diesen Text dann mit dem sichtbaren Text des angegebenen Seitenausschnitts. Der Vergleich findet intern mit der Funktion `containing()` statt, d.h. der XML-Text muss irgendwo im Seitenausschnitt gefunden werden. Zusätzlicher Text im Seitenausschnitt ist erlaubt.

```
@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)
  ;
}
```

Wichtig zu wissen: Bei dem Vergleich werden alle Leerzeichen ignoriert. Das ist notwendig, denn Zeilenumbrüche und formatierende Leerzeichen haben im sichtbaren Text auf der PDF-Seite und innerhalb des unsichtbaren XML-Textes eine ganz verschiedene Bedeutung und sind demzufolge selten gleich.

PDF-Inhalte mit ZUGFeRD-Inhalten vergleichen - Rechnungsanschrift

Der vereinfachte Funktionsaufruf des vorigen Beispiels funktioniert aber nur, wenn die XML-Daten als Ganzes im vorgegebenen Seitenausschnitt enthalten sind. Im folgenden Beispiel steht die Postleitzahl in den XML-Daten getrennt vom Städtename getrennt. Deshalb muss für die Überprüfung der Rechnungsanschrift wieder mit zwei `AssertThat`-Aufrufen gearbeitet werden.

Weil der zu prüfende Text nur ein Teil des XML-Knotens ist, enthält der XPath-Ausdruck die Funktion `contains()`. Der Knoten `ram:PostalTradeAddress` enthält zusätzlich noch den Wert 'DE', der aber nicht im sichtbaren Text vorkommt.

```
@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)
  ;
}
```

In diesem Test unterscheiden sich die Whitespaces im PDF-Dokument von denen in den XML-Daten. Deshalb muss der XPath-Ausdruck die Funktion `normalize-space()` enthalten.

Ausschnitt der PDF-Seite:

Ausschnitt der ZUGFeRD Daten:

```
<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>
```

PDF-Inhalte mit ZUGFeRD-Inhalten vergleichen - Artikel

Das Neue am folgenden Beispiel ist die Tatsache, dass sich die Reihenfolge der Textteile des zu prüfenden Textes im PDF und in den XML-Daten unterscheidet. Deshalb kann nicht auf die zusammenhängende Artikelbezeichnung 'Trennblätter A4 GTIN: 4012345001235' geprüft werden. Der Test validiert lediglich 'GTIN: 4012345001235'. Dafür wird in XPath die Funktion `contains()` und in PDFUnit die Funktion `hasText().containing()` benutzt.

```
@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)
  ;
}
```

Ausschnitt der PDF-Seite:

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**	

Ausschnitt der ZUGFeRD Daten:

```
<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>
```

Komplizierte Prüfungen auf ZUGFeRD-Daten

Es gibt gelegentlich den Wunsch, anspruchsvolle Prüfungen auf die XML-Daten anzuwenden. Diesen Wunsch unterstützt PDFUnit, indem es die Methode `matchingXPath(..)` zur Verfügung stellt, mit der das volle Potential von XPath auf die ZUGFeRD-Daten losgelassen werden kann.

Das folgende Beispiel überprüft, dass die Anzahl der fakturierten Artikel (TradeLineItems) genau '1' ist.

```
@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)
  ;
}
```

Ausschnitt der ZUGFeRD Daten:

```
<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>
```

Noch anspruchsvoller ist es, mit Hilfe von XPath zu prüfen, dass die Summe der Preise aller bestellten Artikel der Gesamtsumme (netto) entspricht, die an anderer Stelle im Dokument gespeichert ist. Ob eine solche Prüfung Gegenstand automatisierter Tests ist oder eher eine Prüfung in der Produktionsumgebung, darf gefragt werden. Aber sie zeigt, wie einfach es ist, komplizierte Prüfungen auf ZUGFeRD-Daten durchzuführen:

```
@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)
  ;
}
```

Um einen solchen XPath-Ausdruck zu entwickeln, müssen Ihnen die XML-Daten vorliegen. Sie können die ZUGFeRD entweder aus dem Adobe Reader® speichern (rechte Maustaste) oder Sie extrahieren sie mit dem Hilfsprogramm `ExtractZugferdData` von PDFUnit. Es ist in Kapitel 9.15: „ZUGFeRD-Daten extrahieren“ (S. 139) beschrieben.

ZUGFeRD-Daten gegen die Spezifikation prüfen

Als Letztes muss noch erwähnt werden, dass PDFUnit auch die Einhaltung der ZUGFeRD-Spezifikation überprüfen kann:

```
@Test
public void compliesWithZugferdSpecification() throws Exception {
  String filename = "ZUGFeRD_1p0_BASIC_Einfach.pdf";
  AssertThat.document(filename)
            .compliesWith()
            .zugferdSpecification(ZugferdVersion.VERSION10)
  ;
}
```

Kapitel 4. Vergleiche gegen ein Referenz-PDF

4.1. Überblick

Viele Tests folgen dem Prinzip, ein einmal getestetes PDF-Dokument als Referenz für neu erstellte Dokumente zu benutzen. Solche Tests sind sinnvoll, wenn Prozesse, die das PDF erstellen, geändert werden, das Ergebnis aber unverändert bleiben soll.

Initialisierung

Die Instantiierung eines Referenz-Dokumentes erfolgt über die Methode `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 Das Test-Dokument ist verschlüsselt und wird mit dem Passwort geöffnet.

❷ Das Referenz-Dokument ist hier nicht verschlüsselt. Falls es verschlüsselt wäre, muss das Passwort der Methode `and()` als zweiter Parameter übergeben werden.

Passwörter dienen nur zum Öffnen der Dokumente, die Tests werden von dem Passwort nicht beeinflusst.

Überblick

Die folgende Liste gibt einen vollständigen Überblick über die vergleichenden Tests von PDFUnit. Links führen zu Kapiteln, die den jeweiligen Test ausführlich beschreiben.

```
// Methods to compare two PDF documents:

.haveSameAccessPermission()          4.3: „Berechtigungen vergleichen“ (S. 91)
.haveSameAccessPermission(..)        4.3: „Berechtigungen vergleichen“ (S. 91)
.haveSameAppearance()                4.10: „Layout vergleichen (gerenderte Seiten)“ (S. 97)
.haveSameAuthor()                    4.6: „Dokumenteneigenschaften vergleichen“ (S. 94)
.haveSameBookmarks()                 4.11: „Lesezeichen (Bookmarks) vergleichen“ (S. 98)
.haveSameCreationDate()              4.5: „Datumswerte vergleichen“ (S. 93)
.haveSameCreator()                   4.6: „Dokumenteneigenschaften vergleichen“ (S. 94)
.haveSameEmbeddedFiles(..)           4.2: „Anhänge (Attachments) vergleichen“ (S. 91)
.haveSameFieldsByName()              4.8: „Formularfelder vergleichen“ (S. 95)
.haveSameFieldsByValue()             4.8: „Formularfelder vergleichen“ (S. 95)
.haveSameFormat()                    4.7: „Formate vergleichen“ (S. 95)
.haveSameImages()                    4.4: „Bilder vergleichen“ (S. 92)
.haveSameJavaScript()                4.17: „Sonstige Vergleiche“ (S. 103)
.haveSameKeywords()                  4.17: „Sonstige Vergleiche“ (S. 103)
.haveSameLanguageInfo()              4.17: „Sonstige Vergleiche“ (S. 103)
.haveSameLayerNames()                4.17: „Sonstige Vergleiche“ (S. 103)
.haveSameModificationDate()          4.5: „Datumswerte vergleichen“ (S. 93)
.haveSameNamedDestinations()         4.12: „"Named Destinations" vergleichen“ (S. 99)
.haveSameNumberOfBookmarks()         4.11: „Lesezeichen (Bookmarks) vergleichen“ (S. 98)
.haveSameNumberOfEmbeddedFiles()     4.2: „Anhänge (Attachments) vergleichen“ (S. 91)
.haveSameNumberOfFields()            4.8: „Formularfelder vergleichen“ (S. 95)
.haveSameNumberOfImages()            4.4: „Bilder vergleichen“ (S. 92)
.haveSameNumberOfLayers()            4.13: „PDF-Bestandteile vergleichen“ (S. 100)
.haveSameNumberOfNamedDestinations() 4.12: „"Named Destinations" vergleichen“ (S. 99)
.haveSameNumberOfPages()             4.13: „PDF-Bestandteile vergleichen“ (S. 100)

...  continued
```

```
... continuation:

.haveSameProducer()                 4.6: „Dokumenteneigenschaften vergleichen“ (S. 94)
.haveSameProperties()               4.6: „Dokumenteneigenschaften vergleichen“ (S. 94)
.haveSameProperty(..)               4.6: „Dokumenteneigenschaften vergleichen“ (S. 94)
.haveSameSubject()                  4.6: „Dokumenteneigenschaften vergleichen“ (S. 94)
.haveSameTaggingInfo()              4.17: „Sonstige Vergleiche“ (S. 103)
.haveSameText()                     4.14: „Text vergleichen“ (S. 100)
.haveSameTitle()                    4.6: „Dokumenteneigenschaften vergleichen“ (S. 94)
.haveSameXFAData()                  4.15: „XFA-Daten vergleichen“ (S. 101)
.haveSameXMPData()                  4.16: „XMP-Daten vergleichen“ (S. 102)
```

4.2. Anhänge (Attachments) vergleichen

Anzahl

Wenn es um die Anzahl der eingebetteten Dateien geht, sieht ein Vergleich so aus:

```
@Test
public void haveSameNumberOfEmbeddedFiles() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfEmbeddedFiles()
  ;
}
```

Namen und Inhalte

Für einen Vergleich der eingebetteten Dateien nach Name oder Inhalt gibt es eine parametrierte Testmethode:

```
@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 Die Dateien werden Byte-weise verglichen, sodass Dateien jeglicher Art verglichen werden können.

Die beiden Konstanten sind in der allgemeinen Klasse `com.pdfunit.Constants` definiert:

```
// Constants defining the kind comparing embedded files:

com.pdfunit.Constants.COMPARED_BY_CONTENT
com.pdfunit.Constants.COMPARED_BY_NAME
```

Eingebettete Dateien können mit dem Hilfsprogramm `ExtractEmbeddedFiles` extrahiert werden. Siehe Kapitel 9.2: „Anhänge extrahieren“ (S. 124).

4.3. Berechtigungen vergleichen

Mit PDFUnit können zwei Dokumente hinsichtlich ihrer Berechtigungen verglichen werden. Das folgende Beispiel vergleicht **alle Berechtigungen**:

```
@Test
public void compareAllPermission() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAccessPermission()
  ;
}
```

Sollen nur **einzelne Rechte** identisch sein, können diese durch typisierte Konstanten eingeschränkt werden:

```
@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)
  ;
}
```

Folgende Konstanten stehen zur Verfügung:

```
// 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
```

4.4. Bilder vergleichen

Anzahl

Der erste Test vergleicht die Anzahl der Bilder eines PDF-Dokumentes miteinander:

```
@Test
public void haveSameNumberOfImages() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfImages()
  ;
}
```

Der Vergleich der Anzahl der Bilder kann auf ausgewählte Seiten eingeschränkt werden:

```
@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()
  ;
}
```

Die Möglichkeiten der Seitenauswahl sind in Kapitel 13.2: „Seitenauswahl“ (S. 160) beschrieben.

Bildinhalte

Die in einem Dokument enthaltenen Bilder können mit denen eines Referenz-Dokumentes verglichen werden. Bilder zweier Dokumente gelten als gleich, wenn sie Byte-weise identisch sind:

```
/**
 * 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()
  ;
}
```

Bei diesem Vergleich bleibt unberücksichtigt, auf welchen Seiten die Bilder vorkommen, und auch, wie häufig ein Bild im Dokument verwendet wird.

Wenn aber Bilder auf bestimmten Seiten gleich sein sollen, müssen die Seiten zuvor eingeschränkt werden:

```
@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 Die Reihenfolge der Bilder spielt für den Vergleich keine Rolle.

Bei etwaigen Unklarheiten über die im PDF enthaltenen Bilder können alle Bilder eines PDF-Dokumentes mit dem Hilfsprogramm `ExtractImages` extrahiert werden. Siehe Kapitel 9.3: „Bilder aus PDF extrahieren“ (S. 126).

4.5. Datumswerte vergleichen

Es macht selten Sinn, Datumswerte zweier PDF-Dokumente miteinander zu vergleichen, aber für eventuelle Anwendungsfälle stehen doch Vergleichsmethoden zur Verfügung:

```
// Methods comparing dates:

.haveSameCreationDate()
.haveSameModificationDate()
```

Im folgenden Beispiel wird das Änderungsdatum verglichen:

```
@Test
public void haveSameModificationDate() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameModificationDate()
  ;
}
```

Die Vergleiche zweier Datumswerte finden immer in der Auflösung `DateResolution.DATE` statt.

4.6. Dokumenteneigenschaften vergleichen

Es kann interessant sein, sicherzustellen, dass zwei Dokumente den gleichen Titel oder gleiche Schlüsselwörter haben. Insgesamt stehen folgende Vergleichsmethoden für Dokumenteneigenschaften zur Verfügung:

```
// Comparing document properties:

.haveSameAuthor()
.haveSameCreationDate()
.haveSameCreator()
.haveSameKeywords()
.haveSameLanguageInfo()
.haveSameModificationDate()
.haveSameProducer()
.haveSameProperties()
.haveSameProperty(String)
.haveSameSubject()
.haveSameTitle()
```

Als Beispiel für den Vergleich aller Eigenschaften soll hier stellvertretend der Vergleich der Autoren stehen:

```
@Test
public void haveSameAuthor() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAuthor()
  ;
}
```

Der Vergleich von Custom-Eigenschaften ist mit der Methode `haveSameProperty(..)` möglich:

```
@Test
public void haveSameCustomProperty() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameProperty("Company")
            .haveSameProperty("SourceModified")
  ;
}
```

Mit dieser Methode können natürlich auch die Standardeigenschaften verglichen werden.

Um alle Eigenschaften zweier PDF-Dokumente miteinander zu vergleichen, gibt es noch die allgemeine Methode `haveSameProperties()`:

```
@Test
public void haveSameProperties_AllProperties() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameProperties()
  ;
}
```

4.7. Formate vergleichen

Seitenformate zweier Dokumente sind gleich, wenn Breite und Höhe gleiche Werte haben. Beim folgenden Vergleich wird eine durch die DIN 476 definierte Toleranz der Seitenlängen berücksichtigt.

```
@Test
public void haveSameFormat() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameFormat()
  ;
}
```

Der Vergleich der Seitenformate zweier Dokumente kann auf bestimmte Seiten beschränkt werden:

```
@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)
  ;
}
```

Die Möglichkeiten, Seiten zu selektieren, sind in Kapitel 13.2: „Seitenauswahl“ (S. 160) ausführlich beschrieben.

4.8. Formularfelder vergleichen

Anzahl

Der einfachste vergleichende Test für Felder ist der, festzustellen, dass beide Dokumente gleich viele Felder enthalten:

```
@Test
public void haveSameNumberOfFields() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfFields()
  ;
}
```

Feldnamen

Beim nächst größeren Test müssen neben der Anzahl auch die Namen der Felder in jedem PDF-Dokument gleich sein. Diese Aussage wird folgendermaßen getestet:

```
@Test
public void haveSameFields_ByName() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameFieldsByName()
  ;
}
```

Feldinhalte

Wenn über die Namen von Feldern hinaus auch die Inhalte der Felder verglichen werden sollen, muss die Methode `haveSameFieldsByValue()` verwendet werden. Gleichnamige Felder müssen gleiche Inhalte haben, sonst schlägt der Test fehl:

```
@Test
public void haveSameFields_ByValue() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameFieldsByValue() ❶
  ;
}
```

❶ Whitespaces werden für den Vergleich „normalisiert“, siehe 13.5: „Behandlung von Whitespaces“ (S. 164).

Kombination mehrerer Tests

Unterschiedliche Vergleiche können in einem Testfall verkettet werden:

```
@Test
public void compareManyItems() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAppearance()
            .haveSameText
  ;
}
```

4.9. JavaScript vergleichen

Zwei PDF-Dokumente können „gleiches“ JavaScript enthalten. Ihr Vergleich erfolgt zeichenweise unter Vernachlässigung der Whitespaces mit der Methode `haveSameJavaScript()`:

```
@Test
public void haveSameJavaScript() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameJavaScript()
  ;
}
```

Wenn Sie den JavaScript Code sehen wollen, können Sie ihn mit dem Hilfsprogramm `ExtractJavaScript` extrahieren. Kapitel 9.5: „JavaScript extrahieren“ (S. 128) beschreibt die nötigen Arbeitsschritte.

4.10. Layout vergleichen (gerenderte Seiten)

PDF-Dokumente können seitenweise als gerenderte Bilder (PNG) verglichen werden:

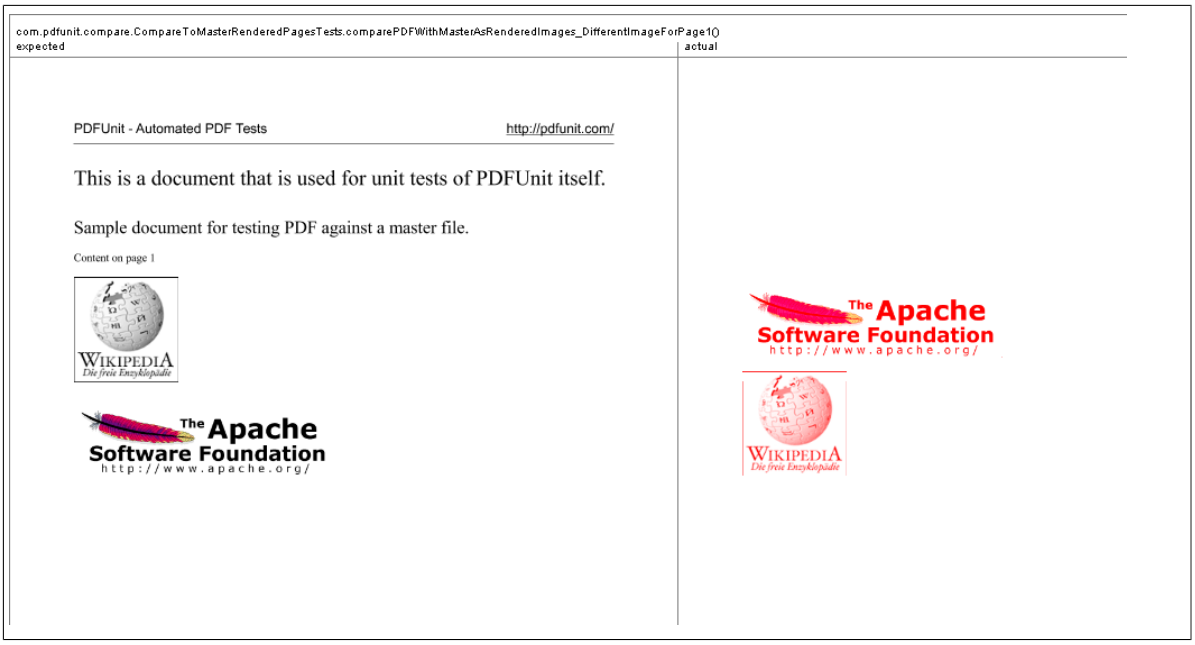
```
@Test
public void haveSameAppearanceCompleteDocument() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .restrictedTo(EVERY_PAGE)
            .haveSameAppearance()
  ;
}
```

Ein Test kann auf vielfältige Weise auf individuelle Seiten eingeschränkt werden. Alle Möglichkeiten werden in Kapitel 13.2: „Seitenauswahl“ (S. 160) beschrieben.

Der Vergleich zweier Seiten kann auch auf einen Ausschnitt der Seite beschränkt werden:

```
@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()
  ;
}
```

Wird bei einem Test mit gerenderten Seiten ein Fehler erkannt, erstellt PDFUnit einen Fehlerreport als **Diff-Image**. Hier ein Beispiel:



Die Überschrift dieses Diff-Bildes enthält den Namen des Tests und in der Fehlermeldung wird der Name des Diff-Bildes genannt. So ist eine Querverbindung zwischen Test und Fehlerbild gegeben.

```
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'.
```

Der Dateiname der Diff-Image-Datei enthält:

- den vollständigen Name der Testdatei,
- ein formatiertes Datum im Format 'yyyyMMdd-HHmmssSSS'
- und als Suffix die Zeichenkette '.out.png'.

Der Ort für die Ablage der Diff-Images wird in der Konfigurationsdatei `pdfunit.config` über den Schlüssel `output.path.diffimages` bestimmt.

Um die Dokumente eines fehlgeschlagenen Tests weiter zu analysieren, hat sich das Programm `DiffPDF` bewährt. Informationen zu dieser Open-Source-Anwendung von Mark Summerfield gibt es auf dessen Projekt-Site http://soft.rubypdf.com/software/diffpdf. Unter Linux kann das Programm beispielsweise mit `apt-get install diffpdf` installiert werden. Für Windows gibt es eine 'Portable Application unter http://portableapps.com/apps/utilities/diffpdf_portable.

4.11. Lesezeichen (Bookmarks) vergleichen

Anzahl

Am einfachsten können Sie die Anzahl der Lesezeichen zweier Dokumente vergleichen:

```
@Test
public void haveSameNumberOfBookmarks() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfBookmarks()
  ;
}
```

Lesezeichen mit Eigenschaften

Darüber hinaus können Lesezeichen als Ganzes verglichen werden. Die Lesezeichen zweier PDF-Dokumente gelten als 'gleich', wenn die Werte folgender Attribute gleich sind:

- 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()
  ;
}
```

Wenn es Unklarheit über die Lesezeichen gibt, können alle Informationen über Lesezeichen mit dem Hilfsprogramm `ExtractBookmarks` in eine XML-Datei exportiert und dort analysiert werden. Siehe Kapitel 9.6: „Lesezeichen nach XML extrahieren“ (S. 129).

4.12. "Named Destinations" vergleichen

„Named Destinations“ sind sicher selten ein Testziel, was auch daran liegt, dass es bisher keine Testwerkzeuge dafür gab. Mit PDFUnit kann aber überprüft werden, ob zwei Dokumente die gleichen „Named Destinations“ haben.

Anzahl

Am einfachsten ist es, die Anzahl von „Named Destinations“ zweier Dokumente zu vergleichen:

```
@Test
public void compareNumberOfNamedDestinations() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNumberOfNamedDestinations()
  ;
}
```

Namen und interne Position

Wenn die Namen von 'Named Destinations' für zwei Dokumente gleich sein sollen, kann das auf die folgende Weise getestet werden:

```
@Test
public void compareNamedDestinations() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameNamedDestinations()
  ;
}
```

4.13. PDF-Bestandteile vergleichen

Die Anzahl verschiedener Dokumentenbestandteile eines Test-Dokumentes kann mit der Anzahl in einem Referenz-Dokument verglichen werden.

Auch, wenn einige solcher Tests schon in den anderen Kapiteln beschrieben sind, soll an dieser Stelle ein Überblick über alle zählbaren Bestandteile gegeben werden, für die es Vergleichsmethoden gibt:

```
// Overview of counting the number of parts of a PDF document:

.haveSameNumberOfBookmarks()
.haveSameNumberOfEmbeddedFiles()
.haveSameNumberOfFields()
.haveSameNumberOfImages()
.haveSameNumberOfLayers()
.haveSameNumberOfNamedDestinations()
.haveSameNumberOfPages()
.haveSameNumberOfTaggingInfo()
```

Nachfolgend werden Beispiele gezeigt, die in keinem anderen Kapitel dargestellt werden:

```
@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. Text vergleichen

Sie können Texte auf beliebigen Seiten zweier PDF-Dokumente vergleichen. Das folgende Beispiel überprüft, dass der Text auf der ersten bzw. letzten Seite zweier Test-Dokumente übereinstimmt. Whitespaces werden dabei normalisiert:

```
@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()
  ;
}
```

Ein Vergleich kann auf individuelle Seiten beschränkt werden. Alle Möglichkeiten, Seiten auszuwählen, werden in Kapitel 13.2: „Seitenauswahl“ (S. 160) beschrieben:

Zusätzlich kann der Textvergleich noch auf Seitenausschnitte beschränkt werden:

```
@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()
  ;
}
```

Und wie bei anderen Tests auch, kann bei einem Textvergleich die Behandlung von Whitespaces gesteuert werden:

```
@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. XFA-Daten vergleichen

XFA-Daten von zwei PDF-Dokumenten miteinander komplett zu vergleichen, ist insofern nicht sinnvoll, weil sie meistens ein Erstellungs- und Änderungsdatum enthalten. Deshalb bietet PDFUnit den Vergleich von XFA-Daten auf der Basis von XPath-Ausdrücken an.

Übersicht

Es gibt nur eine Testmethode, die aber ist sehr mächtig:

```
// Method for tests with XMP data

.haveSameXFAData().matchingXPath(XPathExpression)
```

Falls es Zweifel über die Inhalte der XFA-Daten gibt, können diese mit dem Hilfsprogramm `ExtractXFAData` in eine XML-Datei extrahiert werden. Siehe Kapitel 9.13: „XFA-Daten nach XML extrahieren“ (S. 137).

Beispiel mit verschiedenen Knoten in XFA-Daten und Namensräumen

Im folgenden Beispiel werden zwei Knoten aus den XFA-Daten mit jeweils einem XPath-Ausdruck überprüft. Das Ergebnis der XPath-Auswertung muss für beide PDF-Dokumente identisch sein.

```
@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)
  ;
}
```

Beachten Sie den Umgang mit dem Default-Namensraum. Er muss mit der Klasse `com.pdfunit.DefaultNamespace` deklariert werden. Alle anderen Namensräume müssen nicht deklariert werden, sie werden automatisch erkannt.

Sie können mehrere XPath-Vergleiche in einem Test ausführen, sogar mit wechselnden Default-Namensräumen. Überlegen Sie sich aber, ob Sie den vorhergehenden Test nicht lieber in einzelne Tests aufteilen.

Werden zwei Dokumente verglichen, die beide keine XFA-Daten enthalten, wirft PDFUnit eine Exception. Es macht keinen Sinn, etwas zu vergleichen, das es nicht gibt. Eine ausführliche Beschreibung für das Arbeiten mit XPath in PDFUnit steht in Kapitel 13.11: „XPath-Einsatz“ (S. 170).

Beispiel - XPath-Ausdruck mit XPath-Funktion

Die XPath-Ausdrücke dürfen auch XPath-Funktionen enthalten:

```
@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. XMP-Daten vergleichen

Auch die XMP-Daten zweier PDF-Dokumente werden auf XPath-Basis miteinander verglichen. Die Implementierungen der XMP- und XFA-Testmethoden sind gleich und damit auch die Schnittstelle. Weil das vorhergehende Kapitel 3.36: „XFA Daten“ (S. 78) die Tests schon ausführlich beschreibt, soll hier nur noch ein Beispiel für XMP-Tests gezeigt werden.

Im Zweifelsfall können XMP-Daten mit dem Hilfsprogramm `ExtractXMPData` in eine Datei exportiert und dort analysiert werden. Siehe Kapitel 9.14: „XMP-Daten nach XML extrahieren“ (S. 138).

Übersicht

Folgende Methode steht zur Verfügung:

```
// Method for tests with XMP data:

.haveSameXMPData().matchingXPath(XPathExpression)
```

Beispiel

```
@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)
  ;
}
```

Werden zwei Dokumente verglichen, die beide keine XMP-Daten enthalten, wirft PDFUnit eine Exception. Dieses Verhalten ist sicherlich diskussionswürdig, jedoch macht es keinen Sinn, etwas auf Gleichheit zu vergleichen, das nicht existiert. Ein solcher Test kann ersatzlos gelöscht werden.

4.17. Sonstige Vergleiche

In den vorhergehenden Kapiteln wurden viele Beispiele gezeigt, um zwei PDF-Dokumente zu vergleichen, aber nicht alle. Die folgende Liste nennt die restlichen Tests, die ohne zusätzliche Erläuterungen verständlich sein sollten:

```
// Various methods, comparing PDF. Not described before:

.haveSameJavaScript()
.haveSameKeywords()
.haveSameLanguageInfo()
.haveSameLayerNames()
.haveSameTaggingInfo()
```

Verkettung von Vergleichsmethoden

Vergleichsmethoden können auch verkettet werden:

```
@Test
public void haveSameAuthorTitle() throws Exception {
  String filenameTest = "documentUnderTest.pdf";
  String filenameReference = "reference.pdf";

  AssertThat.document(filenameTest)
            .and(filenameReference)
            .haveSameAuthor()
            .haveSameTitle()
  ;
}
```

Das Beispiel ist nur eine Syntax-Demo. In der Praxis sollten sie hieraus zwei Tests machen und beiden einen guten Namen geben.

Kapitel 5. Mehrere Dokumente und Verzeichnisse

5.1. Überblick

Für die Mengen-Tests stehen fast alle Testmethoden zur Verfügung, die auch für Tests mit einzelnen PDF-Dokumenten existieren. Die folgende Liste zeigt die Methoden, die sowohl für ein ganzes Verzeichnis, als auch für ein angegebene Dokumentenmenge verwendet werden können. Ein Link hinter jeder Methode verweist auf die Beschreibung des jeweiligen Tests.

```
// Methods to validate a set of PDF documents:

.compliesWith()
   .constraints(..)             3.11: „Excel-Dateien für Validierungsregeln“ (S. 30)
   .din5008FormA()              3.9: „DIN 5008“ (S. 26)
   .din5008FormB()              3.9: „DIN 5008“ (S. 26)
   .pdfStandard()               3.22: „PDF/A“ (S. 52)
   .zugferdSpecification(..)    3.39: „ZUGFeRD“ (S. 84)

.containsOneImageOf(..)         3.7: „Bilder in Dokumenten“ (S. 21)
.hasAuthor()                    3.10: „Dokumenteneigenschaften“ (S. 27)
.hasBookmark()                  3.20: „Lesezeichen/Bookmarks und Sprungziele“ (S. 48)
.hasBookmarks()                 3.20: „Lesezeichen/Bookmarks und Sprungziele“ (S. 48)
.hasEncryptionLength(..)        3.21: „Passwort“ (S. 50)
.hasField(..)                   3.14: „Formularfelder“ (S. 33)
.hasFields()                    3.14: „Formularfelder“ (S. 33)
.hasFont()                      3.24: „Schriften“ (S. 55)
.hasFonts()                     3.24: „Schriften“ (S. 55)
.hasFormat(..)                  3.13: „Format“ (S. 32)
.hasImage()                     3.7: „Bilder in Dokumenten“ (S. 21)
   .withBarcode()               3.5: „Barcode“ (S. 17)
   .withQRcode()                3.23: „QR-Code“ (S. 53)
.hasJavaScript()                3.16: „JavaScript“ (S. 41)
.hasKeywords()                  3.10: „Dokumenteneigenschaften“ (S. 27)
.hasLanguageInfo(..)            3.27: „Sprachinformation (Language)“ (S. 62)
.hasNoAuthor()                  3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoImage()                   3.7: „Bilder in Dokumenten“ (S. 21)
.hasNoKeywords()                3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoLanguageInfo()            3.27: „Sprachinformation (Language)“ (S. 62)
.hasNoProperty()                3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoSubject()                 3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoText()                    3.28: „Texte“ (S. 63)
.hasNoTitle()                   3.10: „Dokumenteneigenschaften“ (S. 27)
.hasNoXFAData()                 3.36: „XFA Daten“ (S. 78)
.hasNoXMPData()                 3.37: „XMP-Daten“ (S. 81)

.hasNumberOf...()               3.4: „Anzahl verschiedener PDF-Bestandteile“ (S. 16)

.hasOwnerPassword(..)           3.21: „Passwort“ (S. 50)
.hasPermission()                3.6: „Berechtigungen“ (S. 20)
.hasProperty(..)                3.10: „Dokumenteneigenschaften“ (S. 27)
.hasSignatureField(..)          3.26: „Signaturen - Unterschriebenes PDF“ (S. 59)
.hasSignatureFields()           3.26: „Signaturen - Unterschriebenes PDF“ (S. 59)
.hasSubject()                   3.10: „Dokumenteneigenschaften“ (S. 27)
.hasText(..)                    3.28: „Texte“ (S. 63)
.hasTitle()                     3.10: „Dokumenteneigenschaften“ (S. 27)
.hasUserPassword(..)            3.21: „Passwort“ (S. 50)
.hasVersion()                   3.35: „Version“ (S. 77)
.hasXFAData()                   3.36: „XFA Daten“ (S. 78)
.hasXMPData()                   3.37: „XMP-Daten“ (S. 81)
.hasZugferdData()               3.39: „ZUGFeRD“ (S. 84)
.isCertified()                  3.38: „Zertifiziertes PDF“ (S. 83)
.isCertifiedFor(..)             3.38: „Zertifiziertes PDF“ (S. 83)
.isLinearizedForFastWebView()   3.12: „Fast Web View“ (S. 31)
.isSigned()                     3.26: „Signaturen - Unterschriebenes PDF“ (S. 59)
.isSignedBy(..)                 3.26: „Signaturen - Unterschriebenes PDF“ (S. 59)
.isTagged()                     3.34: „Tagging“ (S. 75)

.passedFilter(..)               5.3: „Verzeichnis testen“ (S. 105)
```

Ein Test auf mehrere Dokumente oder Verzeichnisse bricht mit dem ersten fehlerhaften Dokument ab.

Die nächsten beiden Kapitel zeigen Beispiele für Tests mit einer Dokumentenmenge und mit einem Verzeichnis.

5.2. Mehrere Dokumente testen

Dieses Beispiel prüft, ob alle drei übergebenen Dokumente zwei bestimmte Textstücke enthalten:

```
@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)
  ;
}
```

Die PDF-Dokumente werden hier als `String[]` übergeben. Weiterhin werden die Typen `File[]`, `InputStream[]` und `URL[]` unterstützt.

5.3. Verzeichnis testen

Tests mit Verzeichnissen beginnen alle mit folgenden Methoden:

```
File folder = new File(folderName);
AssertThat.eachDocument()                  ❶
          .inFolder(folder)                ❷
          .passedFilter(filter)            ❸
          ...
;
```

❶❷ Diese beiden Methoden sind der syntaktische Einstieg für Tests, die sich auf alle Dateien im angegebenen Verzeichnis beziehen, die die Dateiendung `PDF` haben. Der Vergleich auf die Dateiendung erfolgt case-insensitiv.

❸ Mit einem (optionalen) Filter kann die Menge der zu testenden Dokumente eingeschränkt werden. Es können mehrere Filter verwendet werden. Ohne Filter werden alle PDF-Dokumente des Verzeichnisses getestet.

Ein Test bricht ab, sobald ein Dokument im Verzeichnis den Test nicht erfüllt.

Falls die Verwendung von Filtern dazu führt, dass kein Dokument mehr übrig bleibt, wird eine Fehlermeldung erzeugt. Ebenso gibt es einen Fehler, wenn das Verzeichnis keine Dokumente enthält.

Beispiel - Alle Dokumente im Verzeichnis validieren

Das folgende Beispiel testet, ob alle PDF-Dokumente im angegebenen Verzeichnis die ZUGFeRD-Spezifikation erfüllen:

```
@Test
public void filesInFolderCompliesZugferdSpecification() throws Exception {
  String folderName = PATH + "zugferd10";
  File folder = new File(folderName);
  AssertThat.eachDocument()
            .inFolder(folder)
            .compliesWith()
            .zugferdSpecification(VERSION10)
  ;
}
```

Beispiel - Ausgewählte Dokumente im Verzeichnisse validieren

Das nächste Beispiel überprüft den Titel aller PDF-Dokumente eines Verzeichnisses, die 'pdfunit-perl' im Namen enthalten:

```
@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")
  ;
}
```

Es gibt zwei Arten von Filtern. Der Filter `FilenameContainingFilter` untersucht, ob ein Dateiname einen bestimmten String enthält, und der Filter `FilenameMatchingFilter` wendet einen Regulären Ausdruck auf den Dateinamen an. Wichtig: In beiden Fällen bezieht sich die Analyse des Dateinamens auf den vollständigen Namen, d.h. auch auf den Pfad. Deswegen sollte beispielsweise ein Regulärer Ausdruck immer mit '`.*`' beginnen.

Kapitel 6. Praxisbeispiele

6.1. Text im Header ab Seite 2

Ausgangssituation

Ihr Unternehmen verschickt elektronische Rechungen. Jede Seite der Rechnung nach der Titelseite soll einen Link auf die Homepage Ihres Unternehmens enthalten.

Problem

Der Header der ersten Seite unterscheidet sich von dem der anderen Seiten. Außerdem gibt es den zu testenden Link auch noch im Fließtext. Die Existenz dort darf den geplanten Test aber nicht stören.

Lösungsansatz

Für den Test müssen sowohl die relevanten Seiten spezifiziert werden, als auch der Bereich auf jeder Seite. Der folgende Code zeigt, wie einfach das ist.

Lösung

```
@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. Passt ein Text in vorgefertigte Formularfelder?

Ausgangssituation

Ein PDF-Dokument wird auf der Basis einer Dokumentenvolage (Template) erstellt. Die Platzhalter für unterschiedliche Texte sind Formularfelder, beispielsweise Textbausteine für AGB's.

Problem

Die Texte können größer sein, als der Platz in den Feldern.

Lösungsansatz

PDFUnit stellt eine Testmethode zur Verfügung, mit der ein **Text-Overflow** festgestellt werden kann.

Lösung

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

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

Der Test ist auch für einzelne Felder möglich:

```
@Test
public void noTextOverflow_OneField() throws Exception {
  String filename = "documentUnderTest.pdf";
  String fieldname = "Textfield, text inside, align left:";

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

In Kapitel 3.15: „Formularfelder, Textüberlauf“ (S. 40) ist dieses Beispiel detailiert beschrieben.

6.3. Name des alten Vorstandes

Ausgangssituation

Der Vorstand hat gewechselt.

Problem

Der Name des früheren Vorsitzenden darf nicht mehr im Header der PDF-Dokumente auftauchen.

Lösungsansatz

PDFUnit bietet die Möglichkeit, Inhalte von PDF-Dokumenten auf ihre **Nicht-Existenz** zu überprüfen.

Lösung

```
@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. Unterschrift des neuen Vorstandes

Ausgangssituation

Der Vorstand hat gewechselt.

Problem

Vertragsrelevante PDF-Dokumente, die in gedruckter Form an Kunden geschickt werden, benötigen eine gültige Unterschrift. Somit muss die Unterschrift unter den Dokumenten die des **neuen** Vorsitzenden sein.

Die neue und die alte Unterschrift liegen zwar jeweils als Bilddatei vor. Beide Dateien haben aber den gleichen Namen, damit ein Vorstandswechsel nicht zu einer Programmänderung führt.

Lösungsansatz

Die Bilddatei wird Byte-weise gegen das Bild innerhalb des PDF-Dokumentes verglichen.

Lösung

```
@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. Neues Logo auf jeder Seite

Ausgangssituation

Zwei Unternehmen fusionieren.

Problem

Für einen Teil der Dokumente ändert sich das Logo. Das neue Logo muss demnächst auf jeder Seite sichtbar sein.

Lösungsansatz

Das neue Logo liegt als Bilddatei vor und wird von PDFUnit im Test verwendet.

Lösung

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

  AssertThat.document(filename)
            .containsImage(newLogoImage)
  ;
}
```

6.6. Unternehmensregeln für die Briefgestaltung

Ausgangssituation

Ihr Unternehmen hat Regeln für das Layout und die inhaltliche Gestaltung von Geschäftsbriefen erstellt.

Problem

Eine manuelle Überprüfung der Einhaltung der Regeln ist aufwendig, auf Dauer zu teuer und selber fehleranfällig.

Lösungsansatz

Die Regeln werden in einer Excel-Datei erfasst und diese in automatisierten Test verwendet.

Lösung

```
@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. ZUGFeRD-Daten gegen sichtbaren Text validieren

Ausgangssituation

Ihr Unternehmen verschickt PDF-Dokumente mit eingebetteten ZUGFeRD-Daten.

Problem

ZUGFeRD Daten werden in automatischen Prozessen erstellt und weiterverarbeitet. Da sie unsichtbar sind, sind sie schwer auf Korrektheit zu überprüfen. Einen Unterschied zwischen den sichtbaren Daten eines PDF-Dokumentes und den unsichtbaren prozessrelevanten ZUGFeRD-Daten sollte es nicht geben. Aber wie prüfen Sie das?

Lösungsansatz

PDFUnit liest Werte aus dem ZUGFeRD-Dokument und vergleicht sie mit den Werten auf einer PDF-Seite oder einem Seitenausschnitt:

Lösung

```
@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)
  ;
}
```

Manchmal sind die ZUGFeRD-Daten und die sichtbaren Daten unterschiedlich formatiert. Das ist z.B. für die Rechnungssumme im verwendeten Beispieldokument 'ZUGFeRD_1p0_BASIC_Einfach.pdf'

der Fall. Sie wird auf der PDF-Seite mit deutschem Dezimalkomma und im ZUGFeRD-Dokument mit englischem Dezimalkomma dargestellt. In solchen Fällen müssen andere Testmethoden von PDFUnit benutzt werden. Sie sind Kapitel 3.39: „ZUGFeRD“ (S. 84) ausführlich dargestellt.

6.8. PDF-Dokumente zum Download auf Webseiten

Ausgangssituation

Sie bieten auf Ihrer Webseite benutzerspezifisch generierte PDF-Dokumente an.

Problem

Das PDF kann nur im Kontext seiner Webseite getestet werden, weil die Anwendung nunmal so konzipiert ist. Also müssen die Eingaben für den Test über die Webanwendung durchgeführt werden und am Ende des Geschäftsvorgang das generierte PDF über den Browser geladen werden.

Lösungsansatz

Selenium bietet gute Möglichkeiten, ein PDF-Dokument innerhalb einer Webseite zu selektieren. Dieses wird als Stream an PDFUnit übergeben.

Lösung

Der eigentliche Test besteht aus den folgenden Zeilen:

```
/**
 * 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")
  ;
}
```

Die restlichen Zeilen zur Vervollständigung des Beispiel sind:

```
/**
 * 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()...

}
```

Weitere Informationen zu Selenium sind auf der Projekt-Site http://seleniumhq.org/ zu finden.

6.9. HTML2PDF - Hat die dynamische PDF-Erstellung funktioniert?

Ausgangssituation

Eine Webanwendung erzeugt dynamische Webseiten und bietet außerdem die Möglichkeit, den Inhalt der aktuellen Webseite als PDF-Dokument herunterzuladen. Dazu wird die HTML-Seite dynamisch in PDF gerendert.

Problem

Wie kann sichergestellt werden, dass der Inhalt der HTML-Seite und der Inhalt der PDF-Seite übereinstimmen? Es ist ja nicht ausgeschlossen, dass das Rendering-Werkzeug „Randbedingungen“ benötigt, die unbekannt sind und damit eventuell nicht eingehalten werden.

Lösungsansatz

Die HTML-Seite wird mit Selenium angesteuert, der erwartete Text wird mit Selenium ausgelesen und in Variablen gespeichert.

Anschließend wird die PDF-Erstellung über die Webseite angestoßen und das erhaltene PDF-Dokument mit PDFUnit auf genau dieselben Texte überprüft.

Lösung

```
/**
 * 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();
  }

}
```

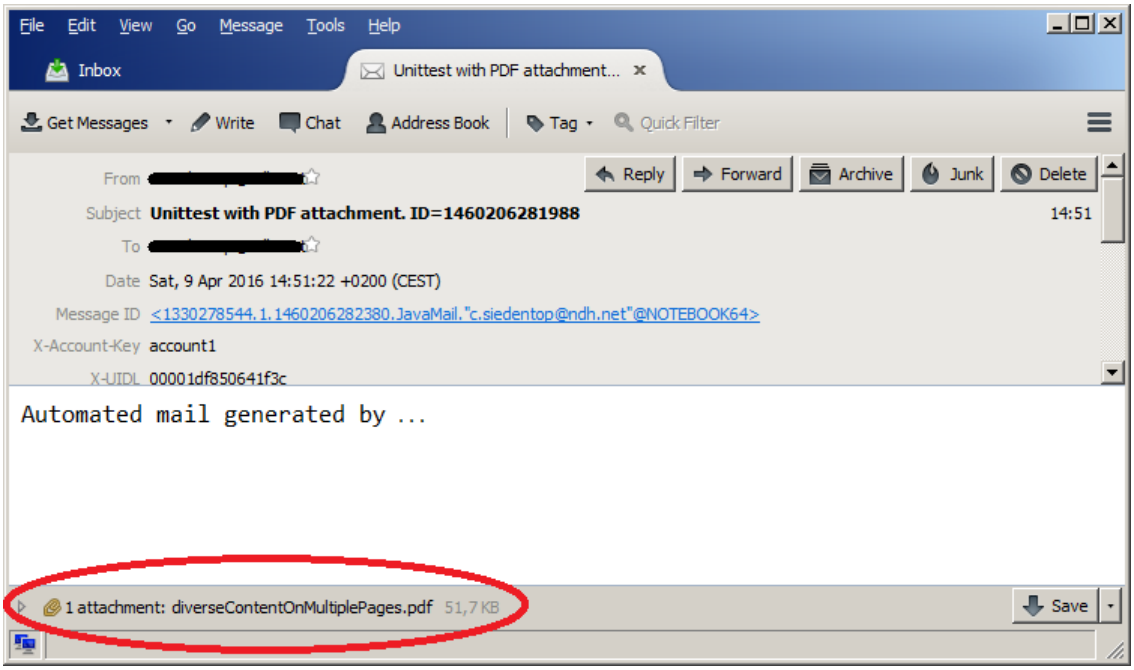
6.10. PDF als Mailanhang testen

Ausgangssituation

Ihr Unternehmen stellt seinen Kunden monatliche Abrechnungen per Mail zu.

Problem

Wie kann die PDF-Datei aus diesem Mail validiert werden?



Es gilt zwei Probleme zu lösen. Einerseits muss innerhalb des Tests ein Mail verschickt werden und unmittelbar wieder ausgewertet werden. Und andererseits ist der PDF-Anhang des erhaltenen Mails zu analysieren.

Lösungsansatz

Das erste Problem löst Dumbster, eine Java-API zum Testen von Mail-Anwendungen, und das zweite Problem löst PDFUnit. Der entscheidende Schritt ist, dass der Mail-Anhang als Byte-Array an PDFUnit übergeben wird.

Lösung

```
/**
 * 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")
  ;
}
```

Und hier die restlichen Teil des Tests:

```
/**
 * 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()...

}
```

Anstatt dieser einfachen Prüfung hätten auch Rechnungsdaten über ZUGFeRD validiert werden können. Beispielsweise stellt der folgende Test sicher, dass der Wert der IBAN innerhalb der ZUGFeRD-Daten mit dem Wert der IBAN auf der ersten Seite des Dokumentes gleich ist.

```
...
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. PDF aus einer Datenbank lesen und testen

Ausgangssituation

Eine Anwendung stellt PDF-Dokumente in eine Datenbank ein.

Problem

Wie kann sichergestellt werden, dass das PDF-Dokument in der Datenbank den Erwartungen entspricht? Und wie wird der Test so geschrieben, dass der INSERT gegen die Datenbank beliebig oft funktioniert?

Lösungsansatz

Die Aufgabe, die Datenbank vor jedem Test zu bereinigen, damit ein INSERT immer funktioniert, wird mit DBUnit umgesetzt. Nachdem die Anwendung in die Datenbank geschrieben hat, wird mit JDBC ein InputStream erstellt, über den das PDF-Dokument aus der DB gelesen wird. Über diesen Stream liest PDFUnit das Dokument aus der Datenbank.

Lösung

```
/**
 * 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();
}
```

Dieses Beispiel funktioniert mit jeder JDBC-fähigen Datenbank. Das vollständige Programm ist im Demo-Projekt enhalten. Es kann über diesen Link (PDFUnit, Download) heruntergeladen werden kann.

Weitere Informationen zu DBUnit stehen auf der Projekt-Homepage.

6.12. Caching von Testdokumenten

Ausgangssituation

Sie haben viele Tests - das ist gut.

Problem

Die Tests laufen zu langsam - das ist schlecht.

Lösungsansatz

Wenn Ihre PDF-Dokumente groß sind, lohnt es sich, diese nur einmal zu instantiieren und anschließend viele Tests darauf auszuführen.

Lösung

```
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.

}
```

Sie könnten zwar auch alle Testmethoden in einer Testmethode verketten. Aber wie würden Sie den Test dann nennen? Der Inhalt einer Testmethode sollte sich möglichst in ihrem Namen widerspiegeln, sonst werden Testreports mit mehreren hundert Tests schwer verständlich. `testAll` ist daher ein schlechter Name, er ist bedeutungsleer.

PDFUnit arbeit intern zustandslos. Da aber auch verschiedene externe Bibliotheken verwendet werden, kann die Zustandslosigkeit nicht zu 100% garantiert werden. Sollte es Probleme geben, ändern Sie die Annotation `@BeforeClass` in `@Before` und entfernen den `static` Modifier. Dann wird das PDF-Dokument wieder für jeden Test neu instantiiert:

```
@Before  // Document will be instantiated for each test. No caching:
public void loadTestDocument() throws Exception {
  String filename = "documentUnderTest.pdf";
  document = AssertThat.document(filename);
}
```

Kapitel 7. PDFUnit für Nicht-Java Systeme

7.1. Kurzer Blick auf PDFUnit-NET

Als 'PDFUnit-NET' steht PDFUnit auch für .NET-Anwendungen seit Dezember 2015 zur Verfügung.

```
[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")
  ;
}
```

Die Kompatibilität zu PDFUnit-Java wird dadurch erreicht, dass aus der Java-Version eine DLL generiert wird. Das hat allerdings zur Folge, dass die Methodennamen in C# mit Kleinbuchstaben beginnen.

Für PDFUnit-NET existiert eine eigene Dokumentation.

7.2. Kurzer Blick auf PDFUnit-Perl

Für Perl-Umgebungen gibt es 'PDFUnit-Perl'. Diese Version von PDFUnit umfasst das Perl-Modul `PDF::PDFUnit`, notwendige Skripte und Laufzeitkomponenten. In Verbindung mit verschiedenen CPAN-Moduln wie beispielsweise `TEST::More` oder `Test::Unit` sind automatisierte Tests möglich, die zur Java-API von PDFUnit 100%ig kompatibel sind.

Es wird angestrebt, das Perl-Modul in das CPAN-Archiv einzustellen.

Zwei kleine Code-Beispiele auf der Basis von `TEST::More`:

```
#
# 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."
;
```

Die Verwendung von PDFUnit-Perl ist in einem eigenen Handbuch dokumentiert.

7.3. Kurzer Blick auf PDFUnit-XML

Tester müssen keine Java-Kenntnisse besitzen, um PDF-Dokumente automatisiert zu testen. Für eine auf XML basierende Systemlandschaft gibt es unter der Bezeichnung 'PDFUnit-XML' Laufzeitkomponenten, Skripte, XML Schema und Stylesheets zum Testen von PDF-Dokumenten. Die Funktionalität ist voll kompatibel zu 'PDFUnit-Java'.

Die folgenden Beispiele geben einen Einblick in 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>
```

Die Namen der Tags und Attribute stimmen überwiegend mit der Java-API überein und folgen ebenfalls der Idee des 'Fluent Interfaces' (http://de.wikipedia.org/wiki/Fluent_Interface).

Die XML-Syntax ist mit passenden XML Schema Dateien abgesichert.

Eine genaue Beschreibung steht als eigenständige Dokumentation zur Verfügung.

Kapitel 8. PDFUnit-Monitor

Der PDFUnit-Monitor ist eine graphische Anwendung, um Tests für PDF-Dokumente anzustoßen und das Ergebnis anzeigen zu lassen. Die Zielgruppe für die Anwendung sind Nicht-Programmierer.

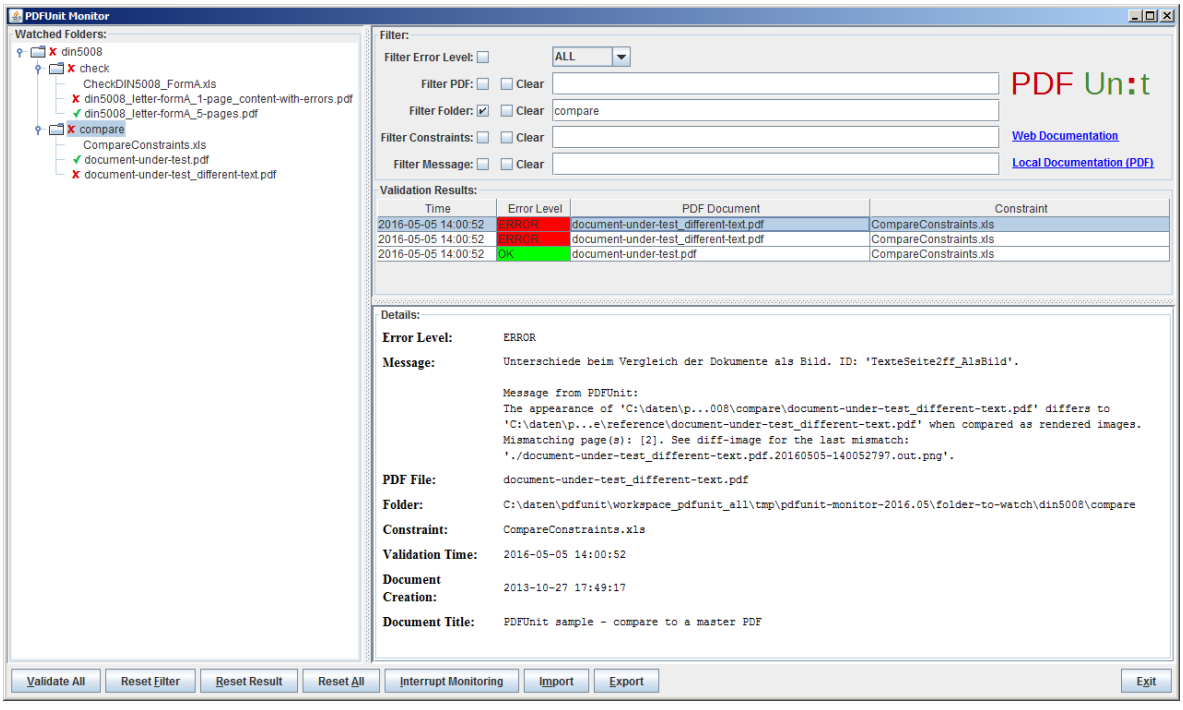
Der Funktionsumfang des PDFUnit-Monitors ist groß. Eine umfassende Beschreibung an dieser Stelle würde den Rahmen der vorliegenden Dokumentation sprengen. Deshalb existiert für ihn eine gesonderte Dokumentation und auch ein erklärendes Video. Beides kann über diesen Link (Download) von den Webseiten von PDFUnit heruntergeladen werden. Die separate Dokumentation beschreibt auch die Installation und Konfiguration des PDFUnit-Monitors. Die nachfolgenden Abschnitte beschreiben kurz die Hauptfunktionen.

Überwachte Verzeichnisse

Der PDFUnit-Monitor überwacht alle PDF-Dokumente unterhalb eines definierten Verzeichnisses und prüft die dortigen Dokumente gegen Regeln, die in Excel-Dateien hinterlegt sind, die in den überwachten Verzeichnissen liegen müssen. Erfüllt ein PDF alle Regeln, wird es in der Baumstruktur mit einem grünen Haken versehen. Verletzt ein PDF eine oder mehrere Regeln, werden alle Regelverletzungen in eine Übersichtsliste eingetragen. Zusätzlich erhält der Dateiname ein rotes Kreuz. Diese Statusanzeige geht auf die Verzeichnisnamen über. Enthält ein Verzeichnis und all seine Unterverzeichnisse ausschließlich gültige PDF-Dokumente, wird es mit einem grünen Haken dargestellt, andernfalls mit einem roten Kreuz.

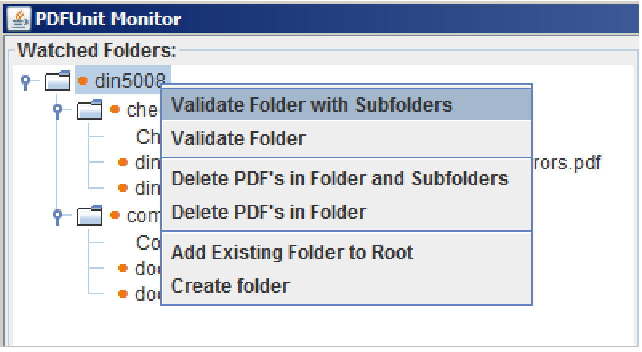
Verzeichnisse mit dem Namen 'reference' werden nicht überwacht. In ihnen müssen PDF-Dokumente abgelegt werden, die bei einer vergleichenden Prüfung als Referenz dienen.

Das folgende Bild zeigt den PDFUnit-Monitor. Die linke Seite ist die Verzeichnisstruktur mit ihren PDF- und Excel-Dokumenten. Die rechte Seite zeigt oben die Fehlerliste mit Filtermöglichkeiten und unten die Details zu einem einzelnen Fehler.



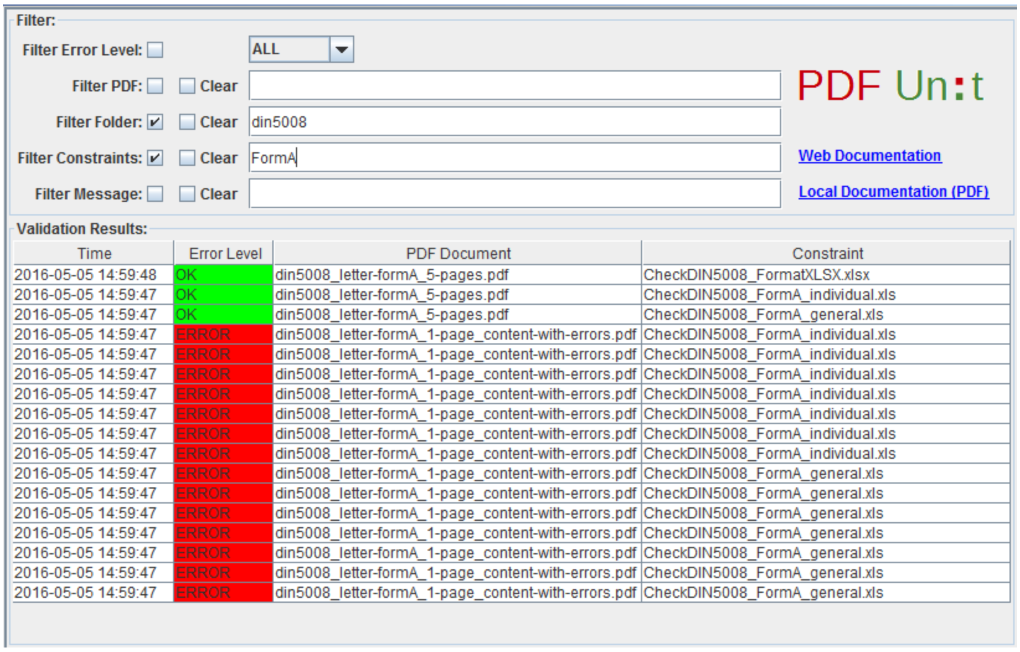
Ein Doppelklick auf ein PDF-Dokument in der Baumstruktur öffnet das Dokument mit der Standardanwendung des Betriebssystems. Gleiches gilt für einen Doppelklick auf eine Excel-Datei.

Die Verzeichnisstruktur ist nicht nur eine reine Darstellung. Wie unter Windows, Linux und MacOS üblich, können verschiedene Funktionen über die rechte Maustaste ausgelöst werden. Das folgende Bild gibt einen kleinen Einblick in das Kontextmenü:



Fehlerübersicht mit Filtermöglichkeiten

Der Monitor zeigt in der oberen Hälfte der rechten Seite die Ergebnisse der Prüfungen aller PDF-Dokumente als Liste an. Für jede Regelverletzung existiert ein eigener Eintrag. Die Details einer Regelverletzung erscheinen in der unteren Hälfte sobald eine Zeile der Liste selektiert wird.

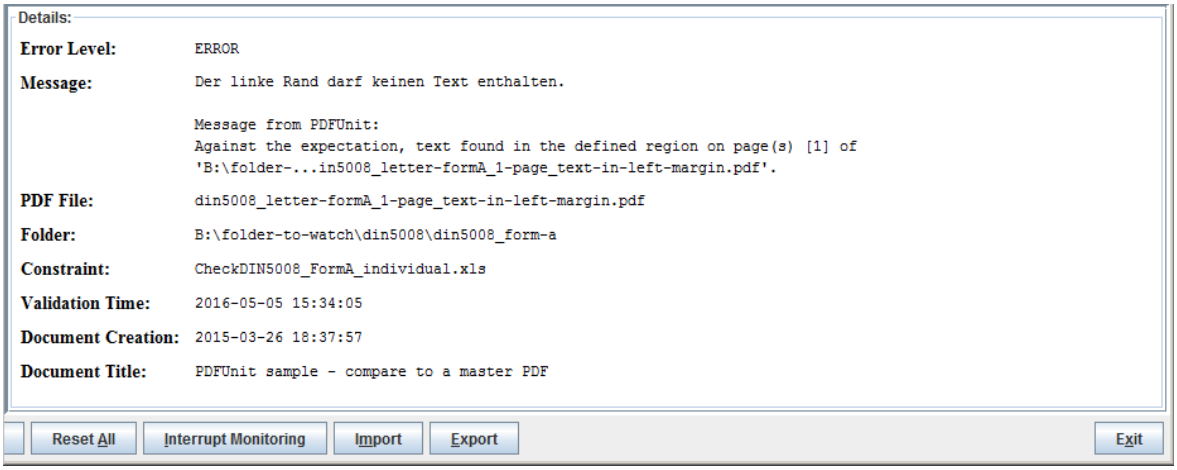


Die Liste der Fehler kann über Filter eingeschränkt werden. Die Filtermöglichkeiten stehen mit der Verzeichnisstruktur in Verbindungen. Wird eine Excel-Datei in der Struktur angeklickt, wird ein Filter mit diesem Namen aktiviert und es werden alle PDF-Dokumente, die mit dieser Excel-Datei geprüft wurden, angezeigt. Filter gibt es für PDF-Dokumente, Verzeichnisse, Excel-Dateien, und Fehlermeldungen.

Wird in der Liste ein PDF-Dokument oder eine Excel-Datei doppelt angeklickt, öffnet sich die Standardanwendung für diesen Dateityp.

Details zum Fehler

Wird ein Eintrag in der Fehlerliste selektiert, werden Details über den Fehler und über das fehlerhafte PDF in der unteren Hälfte der rechten Seite des Monitoroberfläche angezeigt.



Der erste Teil der Fehlermeldung stammt aus der Excel-Datei und wird von der Person, die die Tests erstellt, geschrieben. Weitere Teile der Meldung stammen vom Testwerkzeug PDFUnit. Neben der eigentlichen Fehlermeldung werden weitere nützliche Informationen über das PDF-Dokument, die Regeldatei und den Testzeitraum angezeigt.

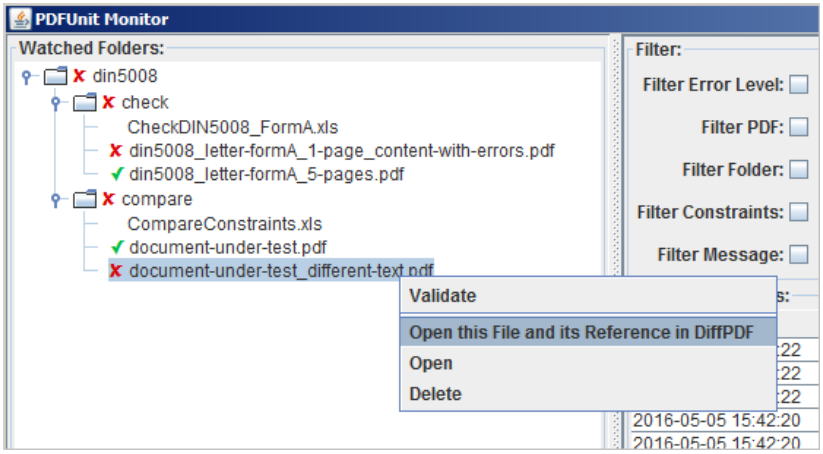
Die Fehlermeldungen von PDFUnit gibt es momentan in Deutsch und in Englisch. Weitere Sprachen können mit wenig Aufwand zur Verfügung gestellt werden.

Vergleich eines PDF-Dokumentes gegen eine Vorlage

PDF-Dokumente können auch gegen eine Vorlage verglichen werden. Die Regeln für den Vergleich werden ebenfalls in einer Excel-Datei abgelegt. Erkennt der PDFUnit-Monitor einen Unterschied zwischen dem Test-Dokument und dem Referenzdokument, wird der Name des Test-Dokuments in der Verzeichnisstruktur mit einem roten Kreuz markiert.

Ein Referenz-Dokument muss den gleichen Namen haben, wie das 'PDF-Under-Test', und wird im Unterverzeichnis 'reference' des Ordners gesucht, in dem sich das 'PDF-Under-Test' befindet.

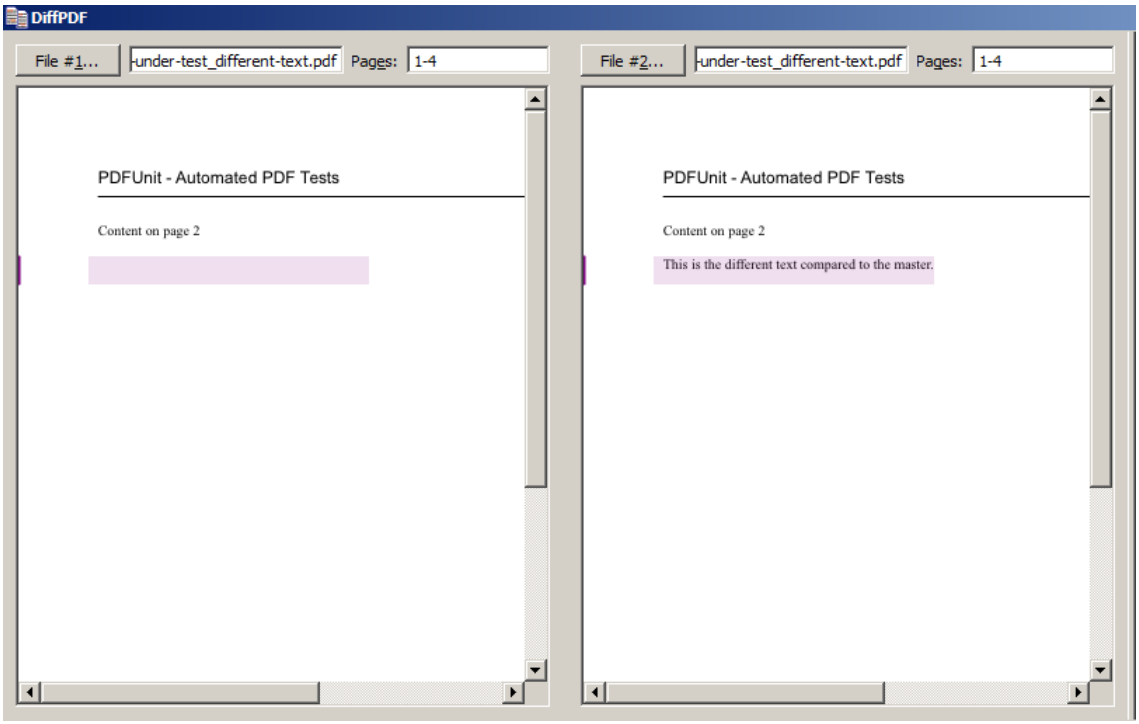
Wenn bei der Validierung ein Unterschied festgestellt wird, kann über die rechte Maustaste das Programm 'DiffPDF 1.5.1 portable' gestartet werden, das den Unterschied gut darstellt:



Das Programm stammt von Mark Summerfield und steht als 'Portable App' über diesen Link (Download) zum Download zur Verfügung. DiffPDF kann in Englisch, Deutsch, Französisch und Tschechisch benutzt werden. Herzlichen Dank an alle Beteiligten für Ihre Arbeit und das großartige Ergebnis.

Das nächste Bild zeigt die Anwendung DiffPDF unmittelbar, nachdem sie aus dem PDFUnit-Monitor heraus gestartet wurde. Auf der linken Seite wird die Vorlage dargestellt, auf der rechten das aktuelle

Testdokument. Die Anwendung positioniert sich direkt auf dem ersten Fehler, hier auf Seite 2. Die Abweichungen werden farblich hinterlegt. Das Bild zeigt nicht die Buttons, mit denen von Fehler zu Fehler gesprungen werden kann.



Export und Import der Ergebnisse

Die Testergebnisse können über den Button 'Export' als XML-Datei exportiert werden und stehen damit auch für einen dauerhaften Nachweis zur Verfügung. Mit XSLT-Stylesheets können die exportierten Dateien in HTML-Reports umgewandelt werden. Über den Button 'Import' werden sie wieder importiert.

Mehrsprachigkeit

Der PDFUnit-Monitor steht momentan für die Sprachen Deutsch und Englisch zur Verfügung. Eine Erweiterung auf andere Sprachen ist strukturell vorgesehen und kann auf Wunsch mit wenig Aufwand realisiert werden.

Kapitel 9. Hilfsprogramme zur Testunterstützung

9.1. Allgemeine Hinweise für alle Hilfsprogramme

PDFUnit stellt Hilfsprogramme zur Verfügung, die Teilinformationen von PDF-Dokumenten in Dateien extrahieren, die anschließend in Tests genutzt werden können:

```
// Utility programs belonging to PDFUnit:

ConvertUnicodeToHex          9.12: „Unicode-Texte in Hex-Code umwandeln“ (S. 136)
ExtractBookmarks             9.6: „Lesezeichen nach XML extrahieren“ (S. 129)
ExtractEmbeddedFiles         9.2: „Anhänge extrahieren“ (S. 124)
ExtractFieldInfo             9.4: „Feldeigenschaften nach XML extrahieren“ (S. 127)
ExtractFontInfo              9.9: „Schrifteigenschaften nach XML extrahieren“ (S. 133)
ExtractImages                9.3: „Bilder aus PDF extrahieren“ (S. 126)
ExtractJavaScript            9.5: „JavaScript extrahieren“ (S. 128)
ExtractNamedDestinations     9.11: „Sprungziele nach XML extrahieren“ (S. 136)
ExtractSignatureInfo         9.10: „Signaturdaten nach XML extrahieren“ (S. 135)
ExtractXFAData               9.13: „XFA-Daten nach XML extrahieren“ (S. 137)
ExtractXMPData               9.14: „XMP-Daten nach XML extrahieren“ (S. 138)
ExtractZugferdData           9.15: „ZUGFeRD-Daten extrahieren“ (S. 139)
RenderPdfPageRegionToImage   9.8: „PDF-Seitenausschnitt in PNG umwandeln“ (S. 131)
RenderPdfToImages            9.7: „PDF-Dokument seitenweise in PNG umwandeln“ (S. 130)
```

Die Hilfsprogramme erzeugen Dateien, deren Namen sich aus dem der jeweiligen Eingabedatei ableiten. Damit es keine Namenskonflikte mit eventuell bestehenden Dateien gibt, gelten diese Namenskonventionen:

- Die Namen beginnen mit einem Unterstrich.
- Die Namen besitzen zwei Suffixe. Das vorletzte lautet `.out`, das letzte ist der übliche Dateityp.

Beispielsweise wird aus der Datei `foo.pdf` die Ausgabe `_bookmarks_foo.out.xml` erzeugt. Benennen Sie sie um, wenn Sie diese Datei in Ihren Tests verwenden.

In den folgenden Kapiteln werden Batchdateien abgebildet, die zeigen, wie die Programme gestartet werden. Die Batchdateien sind Teil des Releases. Sie müssen aber einige der Inhalte, nämlich Classpath, Eingabedatei und Ausgabeverzeichnis an Ihre projektspezifischen Gegebenheiten anpassen.

Werden die Programme fehlerhaft gestartet, wird auf der Konsole ein Hilfetext mit der vollständigen Aufrufsyntax angezeigt.

Die Hilfsprogramme laufen auch in Shell-Skripten für Unix-Systeme. Entwickler im Unix-Umfeld sind sicherlich in der Lage, die hier gezeigten Vorlagen von Windows in Shell-Skripte zu übertragen. Falls Sie Hilfe benötigen, wenden Sie sich an info[at]pdfunit.com.

9.2. Anhänge extrahieren

Das Hilfsprogramm `ExtractEmbeddedFiles` erstellt für jede in einem PDF-Dokument enthaltene Datei eine separate Ausgabedatei.

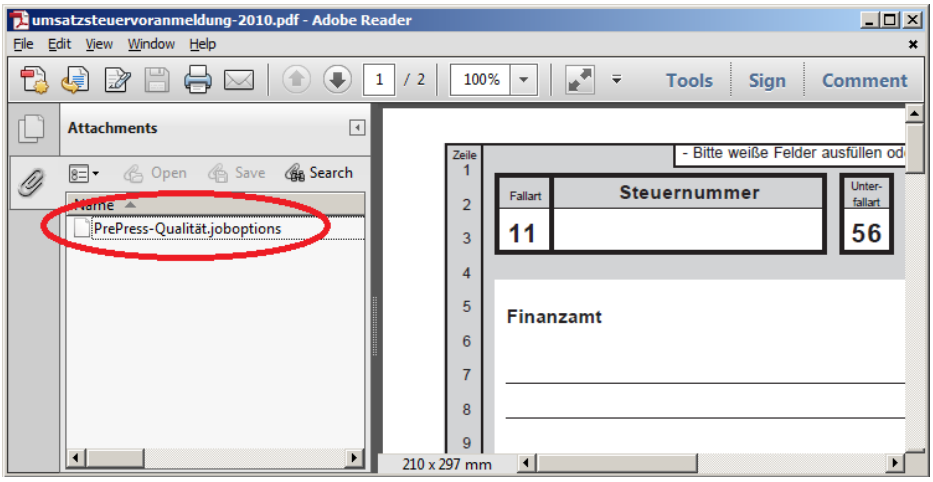
Der Export erfolgt Byte-weise, dadurch werden alle Dateiformate unterstützt.

Aufruf

```
::
:: 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
```

Eingabe

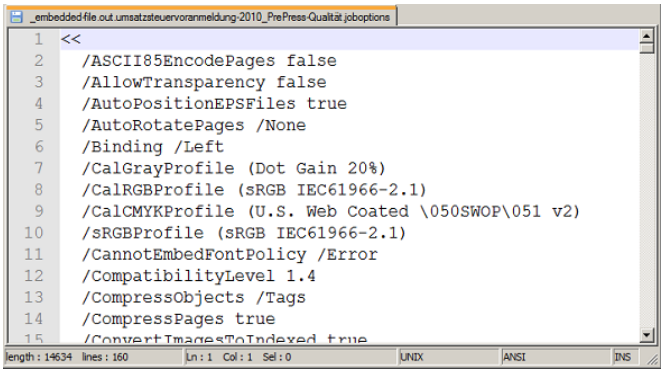
Das PDF-Dokument `umsatzsteuervoranmeldung-2010.pdf` enthält die eingebettete Datei `PrePress-Qualität.joboptions`.



Ausgabe

Der Name der erzeugten Datei enthält sowohl den Namen des PDF-Dokumentes, als auch den Namen der eingebetteten Datei. Dadurch ist eine Zuordnung zwischen Datei und PDF jederzeit möglich: `_embedded-file_umsatzsteuervoranmeldung-2010_PrePress-Qualität.joboptions.out`.

Hier ein kleiner Ausschnitt aus dem Inhalt:



9.3. Bilder aus PDF extrahieren

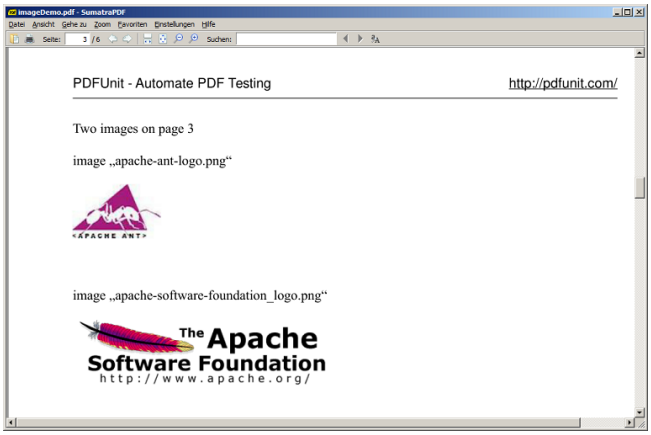
Das Programm `ExtractImages` extrahiert alle Bilder aus einem PDF-Dokument. Jedes Bild wird als eigene Datei gespeichert. Die Tests mit diesen Bildern werden in Kapitel 3.7: „Bilder in Dokumenten“ (S. 21) beschrieben.

Aufruf

```
::
:: 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
```

Eingabe

Die Eingabedatei `imageDemo.pdf` enthält zwei Bilder:



Ausgabe

Nach der Ausführung des Hilfsprogramms entstehen zwei Bilddateien:

```
# created images:

.\tmp\_exported-image_imageDemo.pdf_Im4-0.out.png   ❶
.\tmp\_exported-image_imageDemo.pdf_Im12-1.out.jpg  ❷
```

❶❷ Die Nummer im Dateinamen entspricht der Objekt-Nummer innerhalb des PDF-Dokumentes.

9.4. Feldeigenschaften nach XML extrahieren

Das Hilfsprogramm `ExtractFieldInfo` erstellt eine XML-Datei mit zahlreichen Informationen zu allen Formularfeldern. So können Sie das Format von Feldern sehen und Eigenschaften, wie beispielsweise 'readonly'. Der Inhalt eines Feldes wird nicht extrahiert!

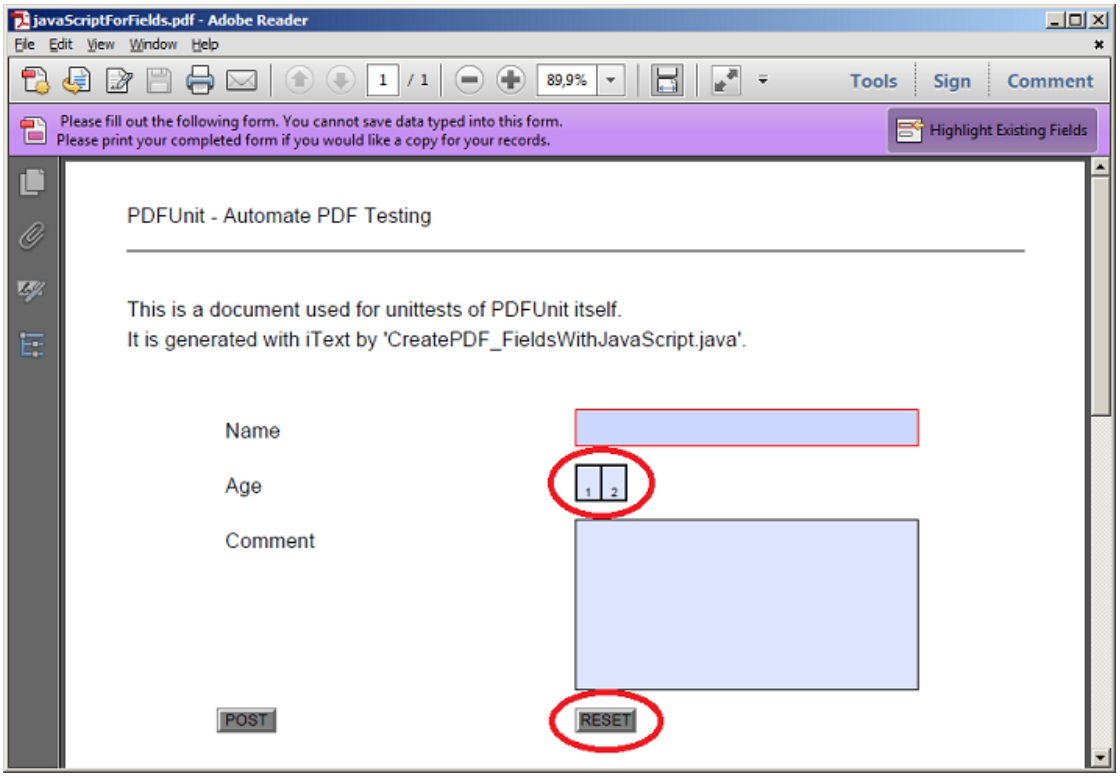
Die Tests auf Feldeigenschaften werden in Kapitel 3.14: „Formularfelder“ (S. 33) beschrieben.

Aufruf

```
::
:: 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
```

Eingabe

Die Eingabedatei `javaScriptForFields.pdf` ist ein eigenes Beispieldokument mit 3 Eingabefeldern und zwei Buttons:



Ausgabe

Die erzeugte Ausgabedatei `_fieldinfo_javaScriptForFields.out.xml` wurde zur besseren Darstellung formatiert und gekürzt:

```
<?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>
```

9.5. JavaScript extrahieren

Das Hilfsprogramm `ExtractJavaScript` extrahiert JavaScript aus einem PDF-Dokument und erstellt daraus eine Textdatei. Das Kapitel 3.16: „JavaScript“ (S. 41) beschreibt, wie die Datei in Tests verwendet werden kann.

Aufruf

```
::
:: 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
```

Eingabe

Die Datei `javaScriptForFields.pdf`, die schon im vorhergehenden Kapitel 9.4: „Feldeigenschaften nach XML extrahieren“ (S. 127) als Beispiel herhalten musste, enthält für die Felder `nameField`, `ageField` und `comment` auch JavaScript.

Innerhalb des Java-Programms, mit dem das PDF-Dokument erstellt wird, sieht der JavaScript-Code für das Feld „ageField“ so aus:

```
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;"
       + "  }"
       + "}"
;
```

Ausgabe

Die erstellte Datei heißt `_javascript_javaScriptForFields.out.txt` und enthält den folgenden JavaScript-Code:

```
var nameField = this.getField('nameField');nameField.setAction('Keystroke', ...
var ageField = ...;function checkAge() {  if(event.value < 12) {...
var commentField = this.getField('commentField');commentField.setAction(...
```

Sie können die Datei gerne neu formatieren, damit sie besser lesbar wird. Hinzugefügte Whitespaces beeinflussen einen PDFUnit-Test nicht.

Hinweis

JavaScript wird auch für die Umsetzung der Dokumenten-Aktionen `OPEN`, `CLOSE`, `PRINT` und SAVE verwendet. Das hier beschriebene Hilfsprogramm extrahiert aber kein JavaScript, das an Aktionen gebunden ist. Dafür wird es in zukünftigen Releases ein neues Hilfsprogramm geben.

9.6. Lesezeichen nach XML extrahieren

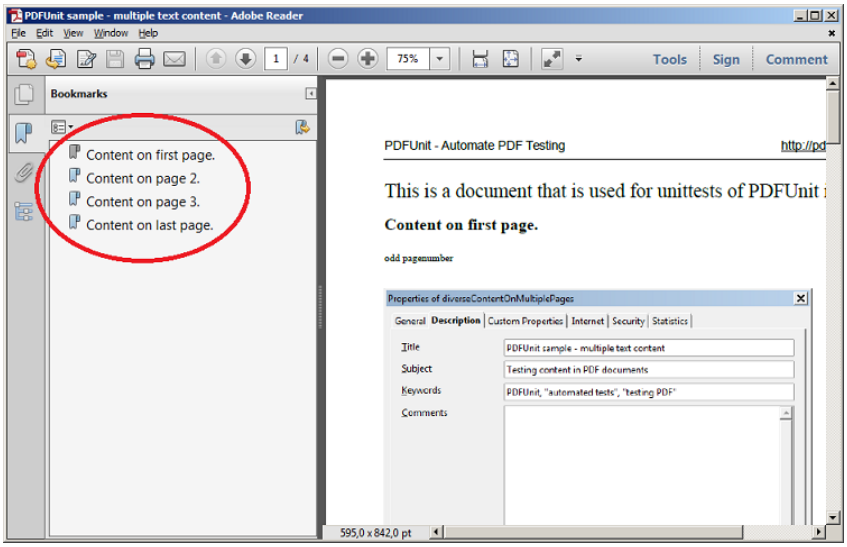
PDFUnit enthält das Hilfprogramm `ExtractBookmarks`, das Lesezeichen/Bookmarks von PDF-Dokumenten nach XML extrahiert. Das Kapitel 3.20: „Lesezeichen/Bookmarks und Sprungziele“ (S. 48) beschreibt die Verwendung der erzeugten XML-Datei für Bookmarks-Tests.

Aufruf

```
::
:: 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
```

Eingabe

Die zu bearbeitende Datei heißt `diverseContentOnMultiplePages.pdf` und ist ein Beispieldokument mit 4 Bookmarks:



Ausgabe

Die erzeugte Datei `_bookmarks_diverseContentOnMultiplePages.out.xml` dient als Grundlage für 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>
```

9.7. PDF-Dokument seitenweise in PNG umwandeln

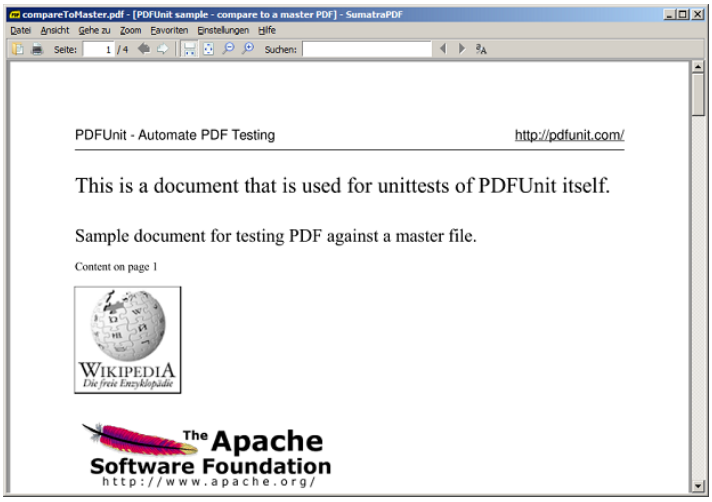
Sie können formatierten Text testen, indem Sie eine PDF-Seite in ein Bild rendern und dieses Bild anschließend gegen eine Bildvorlage vergleichen. Das Kapitel 3.18: „Layout - gerenderte volle Seiten“ (S. 45) beschreibt Layout-Tests unter Verwendung gerenderter Seiten. Mit dem Hilfsprogramm `RenderPdfToImages` erzeugen Sie seitenweise PNG-Dateien von PDF-Dokumenten.

Aufruf

```
::
:: 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
```

Eingabe

Die Eingabe-Datei `documentUnderTest.pdf` enthält 4 Seiten mit unterschiedlichen Bildern und Texten. Die erste Seite sieht im PDF-Reader „SumatraPDF“ (http://code.google.com/p/sumatrapdf) folgendermaßen aus:

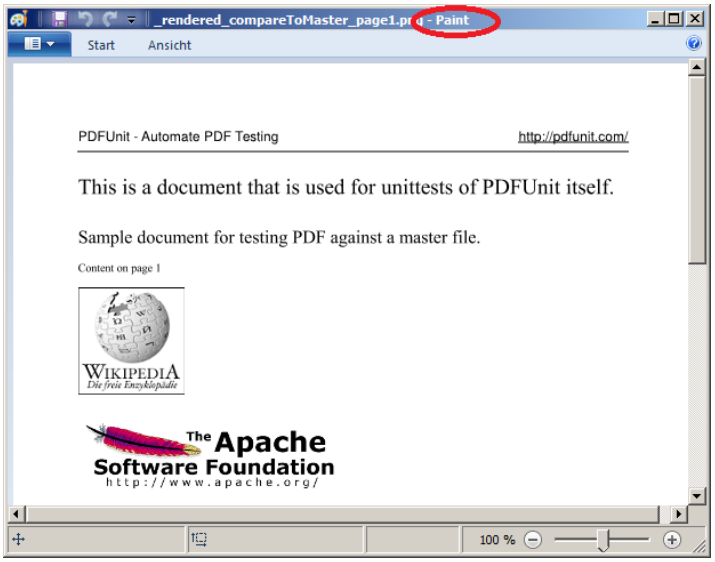


Ausgabe

Nach dem Rendern sind 4 Dateien entstanden:

```
.\tmp\_rendered_documentUnderTest_page1
.\tmp\_rendered_documentUnderTest_page2
.\tmp\_rendered_documentUnderTest_page3
.\tmp\_rendered_documentUnderTest_page4
```

Von diesen sieht die erste Datei als Bild genauso aus, wie im PDF-Reader (s.o.):



PDFUnit benutzt intern den gleichen Algorithmus zum Rendern, wie ihn auch das Extraktionsprogramm benutzt. Insofern bedeuten Abweichungen in einem Test, dass sich das PDF-Dokument seit dem Zeitpunkt des Renderns verändert hat.

9.8. PDF-Seitenausschnitt in PNG umwandeln

Die Gründe, um Tests mit gerenderten Ausschnitten einer PDF-Seite durchzuführen, sind in Kapitel 3.19: „Layout - gerenderte Seitenausschnitte“ (S. 46) beschrieben. Um den Seitenausschnitt „richtig“ zu ermitteln, stellt PDFUnit das kleine Hilfsprogramm `RenderPdfPageRegionToImage` zur Verfügung. Mit ihm wird der durch die Aufrufparameter bestimmte Ausschnitt als PNG-Datei exportiert und kann danach „per Augenschein“ auf seine Richtigkeit überprüft werden. Wenn der Ausschnitt stimmt, übernehmen Sie die Parameter in Ihren Test.

Aufruf

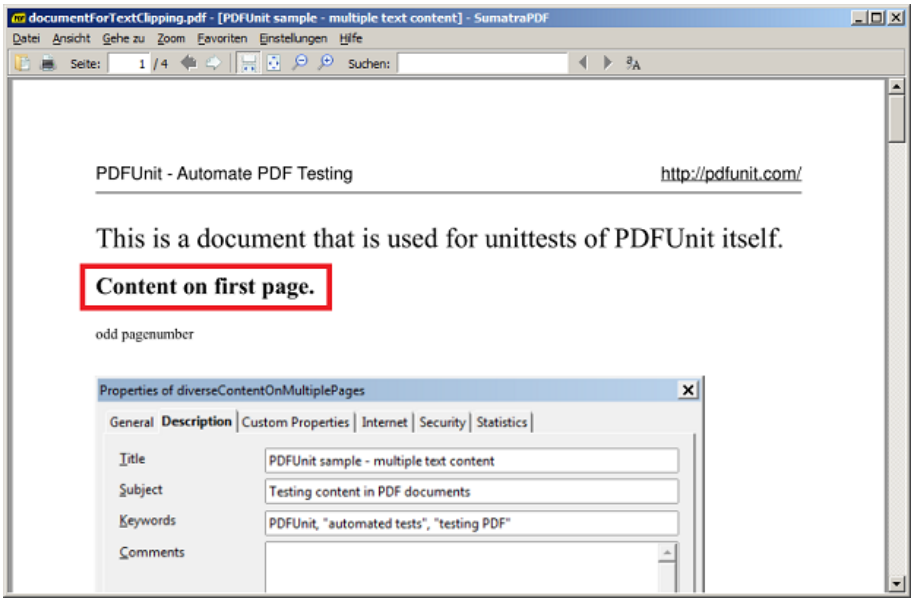
```
::
:: Render a part of a PDF page into an image file
::

@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 Zeilenumbruch nur für diese Dokumentation

Die 4 Werte, die den Ausschnitt beschreiben, müssen Millimeter (mm) sein.

Eingabe

Die Eingabedatei `documentForTextClipping.pdf` enthält im oberen Bereich den Text: „`Content on first page.`“



Ausgabe

Die erzeugte Bilddatei muss auf ihre Richtigkeit überprüft werden.

Damit Sie bei mehreren Seitenausschnitten nicht den Überblick verlieren, enthält der Dateiname die Ausschnittparameter. PDFUnit und das Hilfsprogramm `RenderPdfPageRegionToImage` nutzen den gleichen Algorithmus. Deshalb können Sie die Parameter aus dem Skript direkt in Ihren Test übernehmen oder auch nachträglich aus dem Dateinamen ableiten:

```
#
# Parameters from filename:
#

_rendered_documentForTextClipping_page-1_area-50-130-170-25.out.png
                                                 |  |   |   |
                                                 |  |   |   +- height
                                                 |  |   +- width
                                                 |  +- upperLeftY
                                                +- upperLeftX
```

9.9. Schrifteigenschaften nach XML extrahieren

Wie in Kapitel 3.24: „Schriften“ (S. 55) beschrieben, bergen Schriften eine Komplexität, die ruhig öfter getestet werden sollte. Sie können Informationen über Schriften mit dem Hilfsprogramm `ExtractFontInfo` als XML-Datei aus PDF extrahieren. Diese XML-Datei zeigt, wie die Schriften für PDFUnit aussehen.

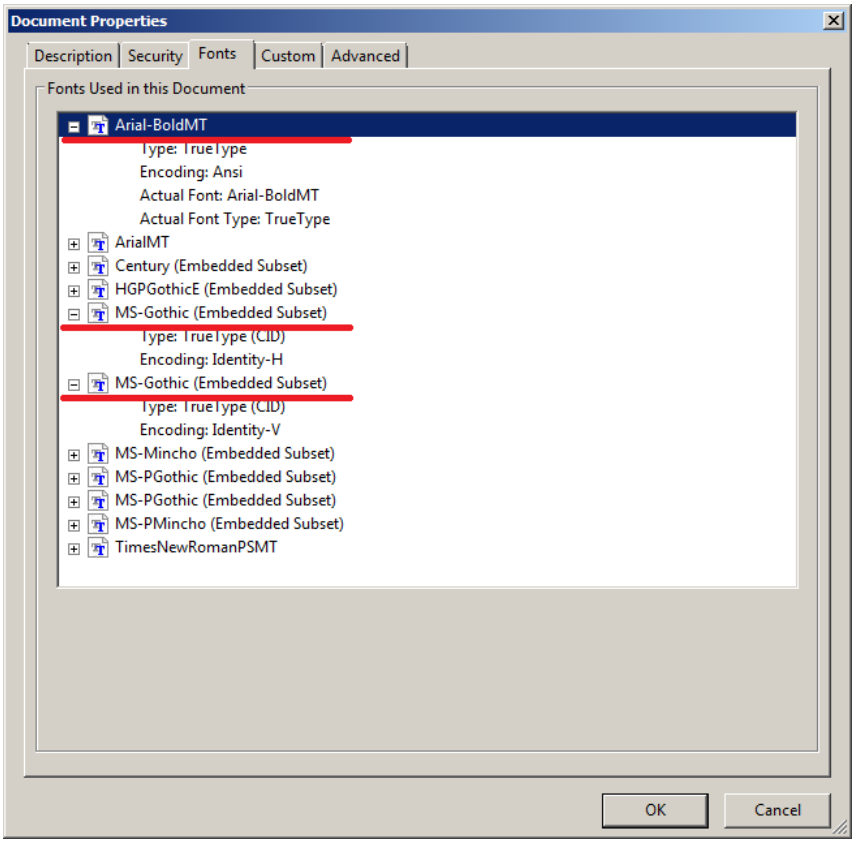
Der Algorithmus, der die XML-Datei erzeugt, ist der gleiche, der von PDFUnit für Tests verwendet wird.

Aufruf

```
::
:: 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
```

Eingabe

Für das japanische PDF-Dokument `fonts_11_japanese.pdf` zeigt der Adobe Reader® folgende Schriften:



Ausgabe

Die erzeugte Ausgabedatei `_fontinfo_fonts_11_japanese.out.xml` sieht dann so aus:

```
<?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>
```

Die XML-Datei listet jedes Subset einer Schriftart einzeln auf. Dadurch können sich Abweichungen von der Anzeige durch den Adobe Reader® ergeben.

9.10. Signaturdaten nach XML extrahieren

Signaturen enthalten eine große Zahl an Informationen, von denen einige in Tests überprüft werden können. Das Kapitel 3.26: „Signaturen - Unterschriebenes PDF“ (S. 59) beschreibt die Tests mit Signaturen.

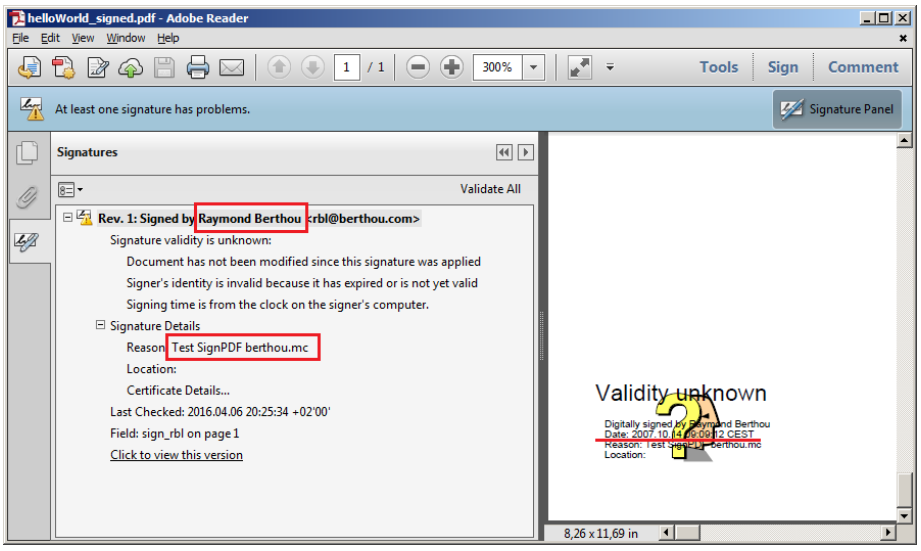
Mit dem folgenden Skript starten Sie die Extraktion:

Aufruf

```
::
:: 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
```

Eingabe

Der Adobe Reader® zeigt die Signaturdaten für die Datei `helloWorld_signed.pdf` an:



Ausgabe

Und hier der Inhalt der erzeugten Datei `_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>
```

Es wird in zukünftigen Releases weitere Funktionen geben, um Signaturen zu testen. Das kann zu Änderungen der XML-Dateien führen. Nehmen Sie bei Problemen in jedem Fall eine aktuelle Dokumentation zur Hand.

9.11. Sprungziele nach XML extrahieren

„Named Destinations“, die Sprungziele innerhalb von PDF-Dokumenten, können schlecht getestet werden, da man sie ja nicht sieht. Mit dem Hilfsprogramm `ExtractNamedDestinations` können Sie sie aber extrahieren und die sichtbar gemachten Namen anschließend in Ihren Tests verwenden. Das Kapitel 3.20: „Lesezeichen/Bookmarks und Sprungziele“ (S. 48) beschreibt diese Tests ausführlich.

Und so sieht das Extraktionsskript aus:

Aufruf

```
::
:: 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
```

Eingabe

Die im Beispiel verwendete Datei `bookmarksWithPdfOutline.pdf` enthält verschiedene Sprungziele.

Ausgabe

Es entsteht die Datei `_named-destinations_bookmarksWithPdfOutline.out.xml` mit folgendem Inhalt:

```
<?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>
```

9.12. Unicode-Texte in Hex-Code umwandeln

PDFUnit kann mit Unicode-Daten umgehen, Kapitel 11: „Unicode“ (S. 146) beschreibt das Thema ausführlich.

Die folgenden Abschnitte beschreiben ein kleines Werkzeug, das einen Unicode-String in seinen Hex-Code umwandelt, damit Sie diesen in Ihren Tests verwenden können. Für wenige 'unlesbare' Zeichen ist dieser Weg einfacher, als einen neuen Font auf Ihrem Rechner zu installieren.

Das Programm `ConvertUnicodeToHex` konvertiert eine beliebige Zeichenkette in ASCII-Code und wandelt dabei Nicht-ASCII-Zeichen in ihren jeweiligen Unicode-Hex-Code. Beispielsweise wird das Euro-Zeichen in `\u20AC` umgewandelt.

Die Eingabedatei selber kann in einem beliebigen Encoding vorliegen, es muss nur vor der Programmausführung korrekt gesetzt sein.

Aufruf

Das Javaprogramm wird mit dem Parameter `-D` gestartet:

```
::
:: 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
```

Der vorletzte Parameter ist die Eingabedatei, der letzte Parameter das Ausgabeverzeichnis.

Eingabe

Die im Skript verwendete Eingabedatei `unicode-to-hex.in.txt` enthält folgende Werte:

```
äöü € @
```

Ausgabe

Die erzeugte Datei `_unicode-to-hex.out.txt` hat folgenden Inhalt:

```
#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 @
```

Die Eingabe selber wird getrimmt. Wenn Sie für Ihren Test Leerzeichen am Anfang oder Ende benötigen, müssen Sie diese nach der Umwandlung in Unicode wieder hinzufügen.

9.13. XFA-Daten nach XML extrahieren

Mit dem Programm `ExtractXFAData` können Sie XFA-Daten exportieren und anschließend zusammen mit XPath in Tests verwenden, wie es in Kapitel 3.36: „XFA Daten“ (S. 78) gezeigt wird.

Aufruf

```
::
:: 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
```

Eingabe

Als Eingabe für das Skript dient die Datei `xfa-enabled.pdf`, ein Beispieldokument von iText.

Ausgabe

Die erzeugte XML-Datei `_xfadata_xfa-enabled.out.xml` ist sehr groß. Deshalb wurden im folgenden Bild einige XML-Tags zusammengefaltet, um einen besseren Eindruck zu vermitteln:

9.14. XMP-Daten nach XML extrahieren

Das Hilfsprogramm `ExtractXMPData` liest die XMP-Daten eines PDF-Dokumentes der Dokumenten-Ebene (document level) aus und schreibt sie in eine XML-Datei. Diese Datei kann anschließend für solche PDFUnit-Tests verwendet werden, wie sie in Kapitel 3.37: „XMP-Daten“ (S. 81) beschrieben sind.

XMP-Daten können nicht nur auf der Dokumenten-Ebene vorkommen, sondern auch in anderen Teilen eines PDF-Dokumentes. Solche XMP-Daten werden im aktuellen Release nicht extrahiert. Für zukünftige Versionen von PDFUnit ist die Extraktion aller XMP-Daten geplant.

Aufruf

```
::
:: 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
```

Eingabe

Es werden die XMP-Daten der Datei LXX_vocab.pdf extrahiert.

Ausgabe

Die erzeugte XML-Datei _xmpdata_LXX_vocab.out.xml wird hier verkürzt dargestellt:

```
<?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>
```

9.15. ZUGFeRD-Daten extrahieren

Die unsichtbaren ZUGFeRD-Daten als abrechnungs- und vertragsrelevanter Teil eines PDF-Dokumentes im Kontext einer vollautomatischen Verarbeitung und die sichtbaren Inhalte dieses Dokumentes sollten identisch sein.

Wenigstens in der Testphase von Programmen, die solche Rechnungen erstellen, muss diese Gleichheit überprüft werden. Dazu ist es notwendig, die ZUGFeRD-Daten eines Dokumentes sichtbar zu machen. Für die Extraktion stellt PDFUnit das Hilfsprogramm ExtractZugferdData zur Verfügung.

Aufruf

Das Programm wird über ein Script gestartet:

```
::
:: 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
```

Der Name der erzeugten Datei leitet sich von der Ursprungsdatei ab. In diesem Fall wird die Datei _zugferd_ZUGFeRD_1p0_BASIC_Einfach.out.xml erzeugt. Ihr Inhalt ist die ZUGFeRD-Datei des PDF-Dokumentes ohne Kommentarzeilen.

Kapitel 10. Validierungsregeln in Excel-Dateien

Regeln für eine Validierung von PDF-Dokumenten können in Excel-Dateien abgelegt werden. Deren Struktur sowie die an der Struktur hängende Funktionalität wird in den folgenden Abschnitten beschrieben.

Aufbau einer Excel-Datei

Eine Excel-Datei wird nach Tabellenblättern (Sheets) mit festgelegten Namen durchsucht:

Excel-Sheet	Bedeutung
regions	Definition von Seitenausschnitten
check	Definition von Testfällen für einzelne PDF-Dokumente
compare	Definition von Testfällen für den Vergleich eines PDF-Dokumentes mit einer geprüften Vorlage

Alle drei Tabellenblätter haben einen festgelegten Aufbau, der in den nachfolgenden Abschnitten beschrieben wird. In allen drei Tabellenblättern bewirkt ein Stern '*' in der ersten Spalte, dass die Zeile als Kommentar gilt.

Die Reihenfolge der Spalten darf nicht geändert werden. Zusätzliche, individuelle Spalten hinter den festgelegten Spalten sind erlaubt. Zusätzliche Tabellenblätter in der Excel-Datei stören ebenfalls nicht.

Ein Tabellenblatt darf Leerzeilen enthalten. Falls ein Blatt aber zu viele leere Zeilen hat, kann es passieren, dass die letzten Zeilen mit Daten nicht mehr ausgelesen werden. Deshalb sollten leere Zeilen nicht verwendet werden. Setzen Sie in Spalte 1 einen Stern, dann gilt die Zeile als nicht-leer und bereitet keine technischen Probleme.

Definition von Seitenausschnitten

Zusätzlich zu der Angabe von Seiten, ist es fast immer notwendig, Tests lediglich auf Ausschnitte einzelner Seiten zu beschränken. So macht ein Vergleich zweier Dokumente mit dem Text der vollen Seiten häufig keinen Sinn, weil sich die Dokumente in ihrem Datum unterscheiden. Deshalb erwartet PDFUnit, dass jeder Test in Excel einen Seitenausschnitt referenziert. Alle Seitenausschnitte werden in dem Sheet 'regions' definiert.

Ein Seitenausschnitt wird durch 4 Werte beschrieben: die x/y-Koordinaten der linken oberen Ecke und die Breite und Höhe des Ausschnitts. Alle Werte sind Millimeter-Angaben. Die Werte dürfen zwar Nachkommastellen haben, jedoch werden die Nachkommastellen auf die nächste Ganzzahl gerundet. Das nächste Bild zeigt ein Beispiele für Ausschnitte:

	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

Wie zu erkennen ist, enthält das Tabellenblatt neben den 4 Werten für die Ausschnitte noch die Spalte mit der Überschrift `id`. Jeder Ausschnitt muss eine eindeutige ID besitzen. Diese ID wird von den Testfalldeklarationen in den Tabellenblättern 'check' und 'compare' referenziert.

Struktur des Excel-Sheets 'check', Definition von Testfällen

Das Tabellenblatt 'check' dient zur Erfassung aller Informationen, die für Testfälle benötigt werden, die sich auf Einzeldokumente beziehen, also nicht für Testfälle, bei denen zwei Dokumente miteinander verglichen werden. Das sind:

Name der Spalte	**Bedeutung**
id	Name (ID) des Testfalls.
pages	Definition der Seiten, auf die sich ein Testfall bezieht.
region	Referenz auf einen Seitenausschnitt, der im Tabellenblatt 'regions' definiert ist.
constraint	Art der Überprüfung. Die erlaubten Werte sind weiter unten beschrieben.
expected value	Der erwartete Wert, der in die Validierung einfließt.
whitespace	Angabe, wie mit Whitespaces (Leerzeichen, Zeilenumbrüchen, etc.) umgegangen werden soll. Die erlaubten Werte für diese Spalte werden weiter unten beschrieben.
message	Diese Spalte nimmt eine Fehlermeldung auf. Eine Fehlermeldung kann Platzhalter enthalten. Die erlaubten Platzhalter werden ebenfalls weiter unten beschrieben.

	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}.

Benennung der Seiten, auf die sich ein Test bezieht

Ein Testfall bezieht sich häufig auf bestimmte Seiten eines Dokumentes. Deshalb können die gewünschten Seiten spezifiziert werden. Die folgende Liste zeigt alle verfügbaren Beispiele:

Seiten	Syntax in Excel
eine einzelne Seite	1
mehrere einzelne Seiten	1, 3, 5
alle Seiten	all
alle Seiten ab der angegebenen (einschließlich)	2...
alle Seiten vor der angegebenen (einschließlich)	...5
alle Seiten zwischen den angegebenen (einschließlich)	2...5

Zwei Seitenzahlen müssen durch ein Leerzeichen getrennt werden. Das Komma ist optional, es dient lediglich der besseren Lesbarkeit.

Spalte 'constraint', verschiedene Arten von Textvergleichen

Die Spalte 'constraint' dient dazu, festzulegen, welche Prüfungen durchgeführt werden sollen, ob beispielsweise ein bestimmter Text in einem Bereich enthalten sein soll ('must contain') oder ob dieser Text gerade nicht in dem Bereich enthalten sein darf ('must not contain'). Die folgende Liste zeigt die mögliche Werte der Spalte 'constraint' im Tabellenblatt 'check' für Textvergleiche:

Schlüsselwort	Verhalten
'must contain'	Der Text, der in der Spalte 'expected value' steht, muss innerhalb des angegebenen Bereichs im PDF-Dokument vorkommen. Zusätzlich muss die interne Behandlung von Leerzeichen vorgegeben werden.
'must not contain'	Der Text, der in der Spalte 'expected value' steht, darf innerhalb des angegebenen Bereichs im PDF-Dokument nicht vorkommen. Zusätzlich muss die interne Behandlung von Leerzeichen vorgegeben werden.
'must be empty'	Der angegebene Bereich darf keinen Text enthalten.
'must not be empty'	Der angegebene Bereich muss Text enthalten
'must match'	Der Text, der in der Spalte 'expected value' angegeben ist, wird als regulärer Ausdruck auf den Text im angegebenen Bereich angewendet. Es muss mindestens einen Treffer geben.
'must not match'	Der reguläre Ausdruck in der Spalte 'expected value' darf für den Text innerhalb des angegebenen Bereichs keinen Treffer ergeben.

Wichtig: die Spalte 'constraint' darf nicht leer sein. In einem solchen Fall bringt das System eine Fehlermeldung.

Im Tabellenblatt 'compare' sind in der Spalte 'constraint' andere Werte erlaubt. Sie werden weiter unten erläutert.

Spalte 'constraint', Signaturen und Bilder testen

Weiterhin kann die Spalte 'constraint' im Tabellenblatt 'check' auch Schlüsselwörter für Tests auf Unterschriften und Bilder enthalten:

Schlüsselwort	Verhalten
'is signed'	Ein PDF-Dokument muss unterschrieben sein.

Schlüsselwort	Verhalten
'is signed by'	Ein PDF-Dokument muss von der Person unterschrieben sein, deren Name in der Spalte 'expected value' steht.
'has number of images'	Auf den ausgewählten Seiten und Seitenausschnitten wird die Anzahl der sichtbaren Bilder überprüft. Die erwartete Anzahl muss in der Spalte 'expected value' angegeben werden.

Spalte 'whitespace', Umgang mit Leerzeichen

Textvergleiche können an Leerzeichen und Zeilenumbrüchen scheitern. Schon bei einem Wechsel der Schriftart können andere Zeilenumbrüche entstehen. Um Tests davon unabhängig zu machen, kann PDFUnit auf dreierlei Arten mit Whitespaces umgehen:

Schlüsselwort	Verhalten
'ignore'	Text wird so komprimiert, dass er keine Whitespaces mehr enthält.
'keep'	Alle Whitespaces bleiben erhalten.
'normalize'	Whitespaces am Anfang und am Ende eines Textes werden gelöscht. Mehrfache Whitespaces innerhalb eines Textes werden auf ein Leerzeichen reduziert.

Fehlerhafte Whitespace-Werte führen zu einer Fehlermeldung. Die Spalte 'whitespace' darf aber leer gelassen werden. In dem Falle gilt 'normalize' als Voreinstellung.

Beim Vergleich zweier PDF-Dokumente spielen Whitespaces nicht immer eine Rolle, beispielsweise beim Vergleich von Lesezeichen (Bookmarks). In den Fälle wird eine vorhandene Angabe der Whitespace-Behandlung ignoriert.

Spalte 'expected value', erwarteter Text

Wenn eine Prüfung gegen einen Erwartungswert stattfindet, muss es eine Spalte geben, die diesen Wert aufnimmt. Diese Spalte heißt 'expected value'.

Der Inhalt der Spalte 'expected value' wird als Regulärer Ausdruck interpretiert, wenn die Spalte 'constraint' die Werte 'must match' oder 'must not match' enthält. Informationen über reguläre Ausdrücke gibt es im Internet unter anderem bei selfhtml.

Der Inhalt der Spalte 'expected value' wird in eine Ganzzahl umgewandelt, wenn die Spalte 'constraint' den Text 'has number of images' enthält.

Spalte 'message', Fehlermeldungen mit Platzhaltern

In den Excel-Dateien können auch individuelle Fehlermeldung hinterlegt werden, die im Fehlerfalle zusätzlich zu den Standardmeldungen von PDFUnit ausgegeben werden. Eine solche Fehlermeldung kann Platzhalter für Laufzeitinformationen enthalten. Das folgende Bild zeigt mehrere Beispiele:

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}.

Wie aus dem Bild ersichtlich ist, werden Platzhalter im Text in geschweifte Klammern eingefasst. Folgende Platzhalter können benutzt werden:

Platzhalter, Schlüsselwort	Bedeutung
{id}	Die ID des aktuellen Testfalls
{pages}	Seitenzahl der Seite, auf der ein Fehler erkannt wurde
{region}	Der Wert der Spalte 'region'
{constraint}	Der Wert der Spalte 'constraint'

Die Platzhalter können an beliebiger Stelle im Text eingebaut werden. Die Werte des aktuellen Tests für die Platzhalter werden zur Laufzeit in Hochkommata eingefasst. Insofern müssen solche Hochkommata im Meldungstext nicht berücksichtigt werden.

Struktur des Excel-Sheets 'compare', Definition von Testfällen für Vergleiche

Mit Excel-Validierungsregeln können auch vergleichende Tests durchgeführt werden. Dabei wird das aktuelle PDF-Dokument gegen eine Vorlage verglichen.

Vergleichendes Testen hat zur Folge, dass in der Excel-Datei keine Angabe mehr über einen erwarteten Text gemacht werden muss. Somit entfällt im Tabellenblatt 'compare' die Spalte 'expected value'. Es verbleiben diese:

	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}.

Die Bedeutungen der Spalten ändert sich nicht, sie sind in den vorhergehenden Abschnitten beschrieben. Lediglich die erlaubten Werte für die Spalte 'constraint', also die Angabe, wie ein Test ausgeführt werden soll, hat sich geändert. Für Vergleiche sind folgende Constraint-Typen erlaubt:

Schlüsselwort	Verhalten
'same text'	Zwei PDF-Dokumente müssen im angegebenen Bereich den gleichen Text enthalten. Zusätzlich muss noch die interne Behandlung von Leerzeichen vorgegeben werden.
'same appearance'	Zwei PDF-Dokumente müssen im angegebenen Bereich als gerendertes Bild identisch sein.
'same bookmarks'	Zwei PDF-Dokumente müssen die gleichen Lesezeichen haben. Aus technischen Gründen darf der Wert in der Spalte 'region' nicht leer sein. Da sich Lesezeichen aber nicht auf einen Seitenausschnitt beziehen, muss die Spalte 'region' den Wert 'NO_REGION' enthalten.

Ein Test, Lesezeichen eines PDF-Dokumentes mit einer Vorlage zu vergleichen, sieht folgendermaßen aus:

	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 sucht die PDF-Dokumente für einen Vergleichs im Unterverzeichnis mit dem Namen 'reference' direkt unterhalb des Verzeichnisses der gerade im Test befindlichen PDF-Datei. Dort wird eine Datei mit dem gleichen Namen, wie die aktuelle Test-PDF-Datei geladen.

Fehlermeldungen zur Laufzeit

Eine Validierung eines PDF-Dokumentes gegen einer Excel-Datei endet nicht mit dem ersten erkannten Fehler. Alle Regeln werden durchlaufen. Für jede Regelverletzung wird eine Fehlermeldung erzeugt.

Kapitel 11. Unicode

Unicode in PDF

Funktionieren die bisher beschriebenen Tests auch mit Inhalten, die nicht ISO-8859-1 sind, beispielsweise mit russischen, griechischen oder chinesischen Texten und Metadaten?

Eine schwierige Frage. Denn auch wenn bei der Entwicklung von PDFUnit viel Wert darauf gelegt wurde, generell mit Unicode zu funktionieren, kann eine pauschale Antwort nur gegeben werden, wenn die eigenen Tests für PDFUnit selber mit „allen“ Möglichkeiten durchgetestet wurde. PDFUnit hat zwar etliche Tests für griechische, russische und chinesische Dokumente, aber für hebräische, arabische und japanische PDF-Dokumente nur wenig Tests. Insofern kann die eingangs gestellte Frage nicht abschließend beantwortet werden.

Die folgenden Tipps im Umgang mit UTF-8 Dateien lösen nicht nur Probleme im Zusammenhang mit PDFUnit. Sie sind sicher auch in anderen Situationen hilfreich.

Einzelne Unicode-Zeichen

Metadaten und Schlüsselwörter können Unicode-Zeichen enthalten. Wenn Ihre Entwicklungsumgebung die fremden Fonts nicht unterstützt, können Sie ein Unicode-Zeichen in Java mit `\uXXXX` schreiben, wie hier das Copyright-Zeichen „©“ als `\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'
  ;
}
```

Längere Unicode-Texte

Es wäre nun zu mühsam, für längere Texte den Hex-Code aller Buchstaben herauszufinden. Deshalb stellt PDFUnit das kleine Programm `ConvertUnicodeToHex` zur Verfügung. Übergeben Sie den ausländischen Text als String an das Werkzeug, entnehmen Sie der daraus erzeugten Datei anschließend den Hex-Code und fügen ihn in Ihr Testprogramm ein. Eine genaue Beschreibung steht in Kapitel 9.12: „Unicode-Texte in Hex-Code umwandeln“ (S. 136). Das Test mit Unicode sieht dann so aus:

```
@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)
  ;
}
```

Unicode in XPath-Ausdrücken

In Kapitel 13.11: „XPath-Einsatz“ (S. 170) wird beschrieben, wie PDFUnit-Tests zusammen mit XPath funktionieren. Auch die XPath-Ausdrücke können Unicode enthalten. Im folgende Beispiel wird

analysiert, ob irgendein XML-Knoten unterhalb von `rsm:HeaderExchangedDocument` das Zeichen mit dem Unicode `\u20AC` enthält:

```
@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)
  ;
}
```

UTF-8 File-Encoding für Shell-Skripte

Vorsicht bei Daten, die aus dem Dateisystem gelesen werden. Deren Interpretation ist vom Encoding des jeweiligen Dateisystems abhängig. Deshalb ist jedes Java-Programm, das Dateien verarbeitet, also auch PDFUnit, von der Umgebungsvariablen `file.encoding` abhängig.

Es gibt mehrere Möglichkeiten, diese Umgebungsvariable für den jeweiligen Java-Prozess zu setzen:

```
set _JAVA_OPTIONS=-Dfile.encoding=UTF8
set _JAVA_OPTIONS=-Dfile.encoding=UTF-8

java -Dfile.encoding=UTF8
java -Dfile.encoding=UTF-8
```

UTF-8 File-Encoding für ANT

Während der Entwicklung von PDFUnit gab es zwei Tests, die unter Eclipse fehlerfrei liefen, unter ANT aber mit einem Encoding-Fehler abbrachen. Die Ursache lag in der Java-System-Property `file.encoding`, die in der DOS-Box nicht auf `UTF-8` stand.

Der folgende Befehl löste das Encoding-Problem unter ANT **nicht**:

```
// does not work for ANT:

ant -Dfile.encoding=UTF-8
```

Statt dessen wurde die Property so gesetzt, wie im vorhergehenden Abschnitt für Shell-Skripte beschrieben:

```
// Used when developing PDFUnit:

set JAVA_TOOL_OPTIONS=-Dfile.encoding=UTF-8
```

Maven - UTF-8 Konfiguration in der pom.xml

In der `pom.xml` können Sie UTF-8 an vielen Stellen konfigurieren. Die folgenden Code-Ausschnitte zeigen mehrere Beispiele, wählen Sie die passenden für Ihr Problem:

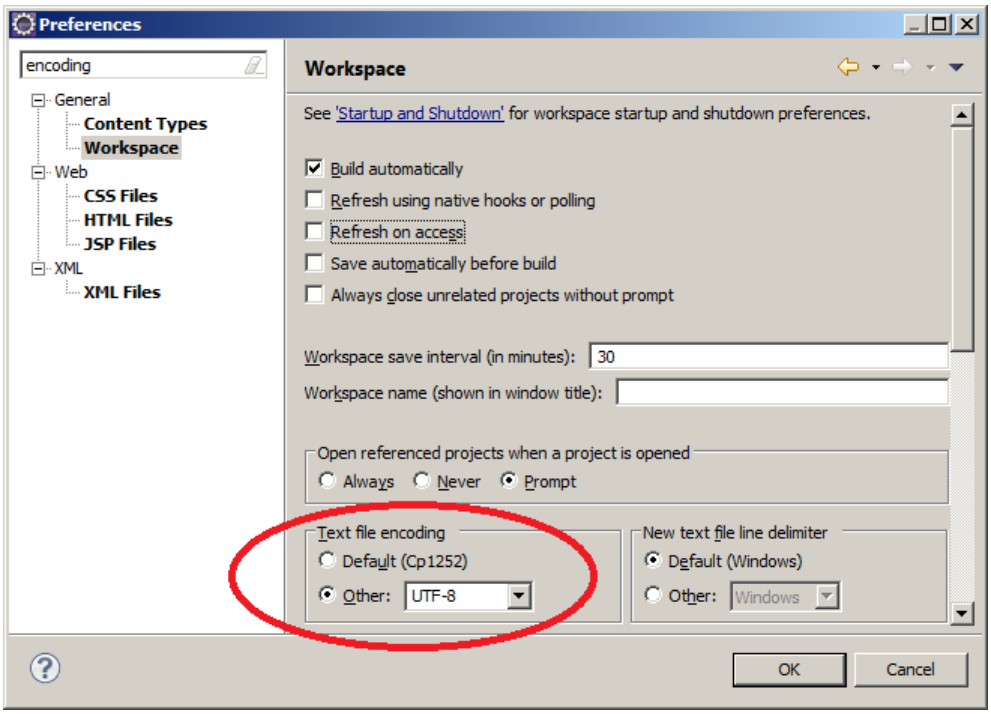
```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
```

```
<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>
```

Eclipse auf UTF-8 einstellen

Wenn Sie XML-Dateien in Eclipse erstellen, ist es nicht unbedingt nötig, Eclipse auf UTF-8 einzurichten, denn XML-Dateien sind auf UTF-8 voreingestellt. Für andere Dateitypen ist aber die Codepage des Betriebssystems voreingestellt. Sie sollten daher, wenn Sie mit Unicode-Daten arbeiten, das Default-Encoding für den gesamten Workspace auf UTF-8 einstellen:



Abweichend von dieser Standardeinstellung können einzelne Dateien in einem anderen Encoding gepeichert werden.

Fehlermeldungen und Unicode

Wenn Tests fehlschlagen, die auf Unicode-Inhalte testen, kann es sein, dass Eclipse oder ein Browser die Fehlermeldung nicht ordentlich dargestellen. Ausschlaggebend dafür ist das File-Encoding der Ausgabe, das von PDFUnit selber nicht beeinflusst werden kann. Wenn Sie in Eclipse, ANT oder Maven dafür gesorgt haben, dass „UTF-8“ als Codepage verwendet wird, sind die meisten Probleme beseitigt. Danach können noch Zeichen aus der Codepage „UTF-16“ die Darstellung der Fehlermeldung korrumpieren.

Das PDF-Dokument im nächsten Beispiel enthält einen Layer-Namen, der UTF-16BE-Zeichen enthält. Um die Wirkung der Unicode-Zeichen in der Fehlermeldung zu zeigen, wurde der erwartet Layername bewusst falsch gewählt:

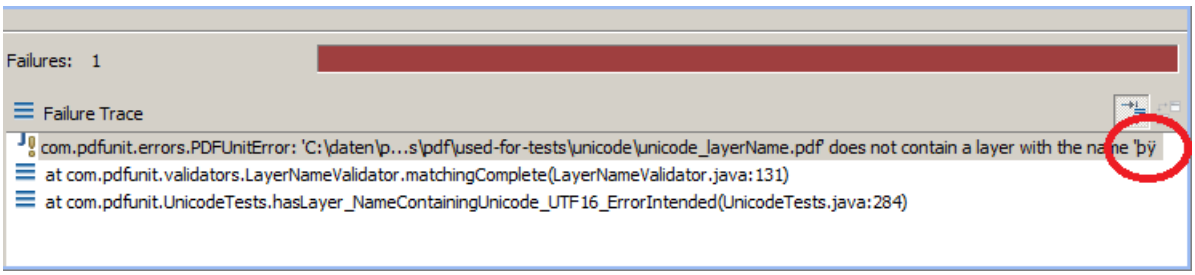
```
/**
 * The name of the layers consists of UTF-16BE and contains the
 * byte order mark (BOM). The error message is not complete.
 * It was corrupted by the internal Null-bytes.
 */
@Test
public void hasLayer_NameContainingUnicode_UTF16_ErrorIntended() throws Exception {
  String filename = "documentUnderTest.pdf";

//  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);
  ;
}
```

Wenn die Tests mit ANT ausgeführt wurden, zeigt ein Browser die von PDFUnit erzeugte Fehlermeldung fehlerfrei an, einschließlich der Zeichenkette þÿEbene _XXX am Ende:

'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)
```

Eclipse dagegen hat in der JUnit-View Probleme mit den Null-Bytes. Die Meldung `'... \unicode_layerName.pdf' does not contain a layer with the name 'þÿ'` endet nicht mit dem Text þÿEbene _XXX. Sie wird nach der internen Byte-Order-Markierung abgeschnitten:



Unicode für unsichtbare Zeichen -

Im praktischen Betrieb trat einmal ein Problem auf, bei dem ein „non-breaking space“ in den Testdaten enthalten war, das zunächst als normales Leerzeichen wahrgenommen wurde. Der String-Vergleich lieferte aber einen Fehler, der erst durch die Verwendung von Unicode beseitigt werden konnte:

```
@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)
  ;
}
```

Kapitel 12. Installation, Konfiguration, Update

12.1. Technische Voraussetzungen

PDFUnit benötigt mindestens Java-7 als Laufzeitumgebung.

Wenn Sie PDFUnit über ANT, Maven oder andere Automatisierungswerkzeuge ausführen, benötigen Sie natürlich noch die Installationen dieser Werkzeuge.

Getestete Umgebungen

Mit den folgenden Systemen wurde PDFUnit erfolgreich getestet:

Betriebssystem

- 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

Weitere Java/Betriebssystem-Kombinationen werden ständig getestet.

Sollte es Probleme mit der Installation geben, schreiben Sie an info[at]pdfunit.com.

12.2. Installation

Download und Entpacken von PDFUnit-Java

Laden Sie die Datei `pdfunit-java-VERSION.zip` aus dem Internet: . Wenn Sie eine Lizenz erworben haben, erhalten Sie eine neue ZIP-Datei per Mail.

Entpacken Sie die ZIP-Datei, z.B. in den Projektordner `PROJECT_HOME/lib/pdfunit-java-VERSION`. Dieser Ordner wird nachfolgend `PDFUNIT_HOME` genannt.

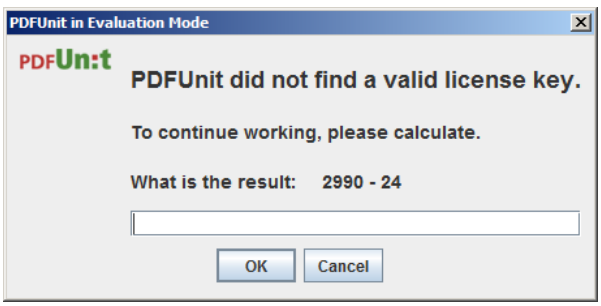
Classpath konfigurieren

Alle JAR-Dateien, die von PDFUnit mitgeliefert werden, müssen in den Classpath aufgenommen werden. Ebenso die Datei `pdfunit.config`.

Sofern Sie ein lizensiertes PDFUnit verwenden, muss auch die Lizenzschlüsseldatei `license-key_pdfunit-java.lic` im Classpath liegen.

PDFUnit ohne Lizenzschlüssel nutzen

Es ist erlaubt, PDFUnit zu Evaluationszwecken ohne Lizenz zu verwenden. Wenn Sie dann einen Test starten, erscheint ein kleines Fenster mit einer leichten Rechenaufgabe, die Sie lösen müssen. Mit der richtigen Lösung laufen die Tests durch, andernfalls nicht und Sie müssen sie neu starten.



Das Fenster mit der Rechenaufgabe ist gelegentlich durch andere Anwendungen verdeckt. Dann „hängt“ das ANT-Skript oder Maven-Skript. Sie finden das Dialogfenster, wenn Sie alle Anwendungsfenster minimieren.

Lizenzschlüssel beantragen

Wenn Sie PDFUnit im kommerziellen Umfeld einsetzen, benötigen Sie eine Lizenz. Schreiben Sie ein Mail an info[at]pdfunit.com, Sie erhalten umgehend eine Antwort.

Die Lizenzkosten werden individuell gestaltet. Ein kleines Unternehmen muss nicht genauso viel zahlen, wie ein großes Unternehmen. Und wer nur wenige PDF-Dokumente testet, zahlt selbstverständlich auch weniger. Sollten Sie in den Besitz einer kostenlosen Lizenz kommen wollen, lassen Sie sich Argumente einfallen - es ist möglich.

Lizenzschlüssel installieren

Wenn Sie eine Lizenz beantragt haben, erhalten Sie eine ZIP-Datei mit PDFUnit-Java und eine separate Datei `license-key_pdfunit-java.lic`. Installieren Sie die ZIP-Datei, wie oben beschrieben, und sorgen Sie dafür, dass auch die Lizenzdatei im Classpath aufgenommen wird.

Jede Änderung an der Lizenzdatei macht diese unbrauchbar. Nehmen Sie in einem solchen Falle mit PDFUnit.com Verbindung auf und beantragen Sie eine neue Lizenzdatei.

Überprüfung der Installation

Wenn Sie Probleme mit der Konfiguration haben, starten Sie das Skript zur Überprüfung der Installation: `verifyInstallation.bat` oder `verifyInstallation.sh`. Es ist in Kapitel 12.6: „Überprüfung der Konfiguration“ (S. 155). ausführlich beschrieben.

12.3. Classpath in Eclipse, ANT, Maven definieren

Alle Entwicklungsumgebungen benötigen die folgenden Dateien im Classpath:

- alle JAR-Dateien, die von PDFUnit ausgeliefert werden
- die Datei `pdfunit.config`
- die Datei `license-key_pdfunit-java.lic`, falls eine Lizenz verwendet wird

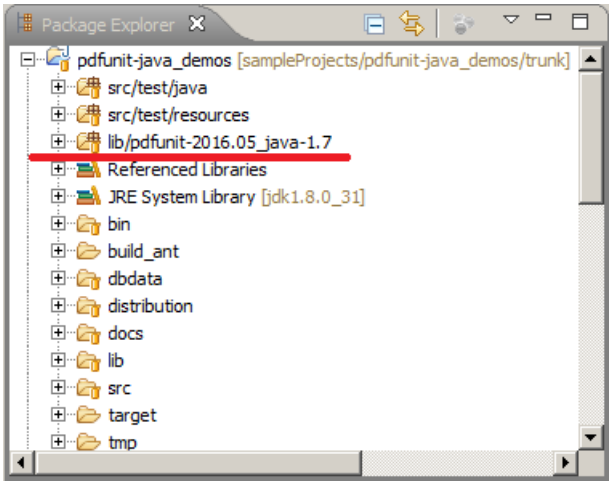
Sollten die Dateien nicht auffindbar sein, gibt es entsprechende Fehlermeldungen:

- `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.`

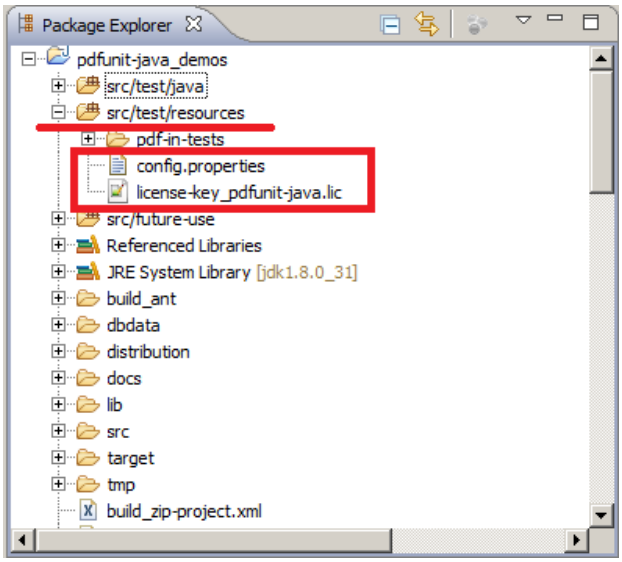
Nachfolgend werden Beispiele gezeigt, wie der Classpath in verschiedenen Umgebungen konfiguriert werden kann. Zusätzlich wird in Kapitel 12.4: „Pfade über Systemumgebungsvariablen setzen“ (S. 154) eine Alternative beschrieben, den Ort der Dateien `pdfunit.config` und `license-key_pdfunit-java.lic` über Systemumgebungsvariablen der Java-Runtime zu deklarieren.

Eclipse konfigurieren

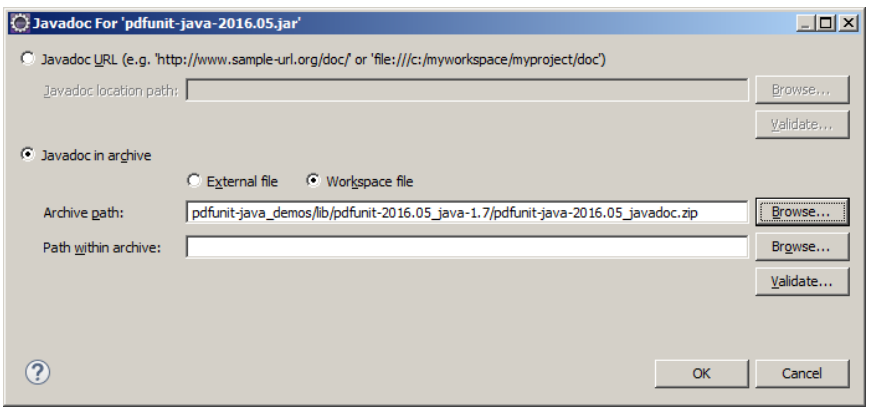
Die einfachste Konfiguration für Eclipse ist, das komplette Installationsverzeichnis `PDFUNIT_HOME` und alle JAR-Dateien einzeln in den Classpath aufzunehmen:



Eine andere Möglichkeit ist die, die Datei `pdfunit.config` in das Verzeichnis `src/test/resources` zu verschieben und das Verzeichnis dann in den Classpath aufzunehmen:



Als Letztes können Sie noch die mitgelieferte Javadoc-Datei in Eclipse registieren, damit die Javadoc-Kommentare von PDFUnit in Eclipse angezeigt werden. Im Verzeichnis `PDFUNIT_HOME` befindet sich dazu eine Datei mit dem Namensmuster `pdfunit-java-VERSION_javadoc.zip`.



ANT konfigurieren

Es gibt verschiedene Möglichkeiten, ANT für PDFUnit zu konfigurieren. In allen Varianten müssen die JAR-Dateien des Installationsverzeichnisses `PDFUNIT_HOME` und der Verzeichnisse `PDFUNIT_HOME/lib/*` in den Classpath aufgenommen werden. Ebenso muss die Datei `pdfunit.config` im Classpath liegen.

Wenn Sie keine Änderungen in der `pdfunit.config` benötigen, ist es am einfachsten, zusätzlich zu den JAR-Dateien `PDFUNIT_HOME` selbst in den Classpath aufzunehmen, wie es das folgende Listing zeigt:

```
<!--
  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>
```

Sie können die Datei `pdfunit.config` aber auch in ein beliebiges Verzeichnis legen, beispielsweise in `src/test/resources`. Diese Variante wird empfohlen, wenn Sie Änderungen an der Konfiguration vornehmen. Die Konfigurationsdatei wird in Kapitel 12.5: „Einstellungen in der pdfunit.config“ (S. 154) beschrieben. Der Classpath in ANT sieht dann folgendermaßen aus:

```
<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>
```

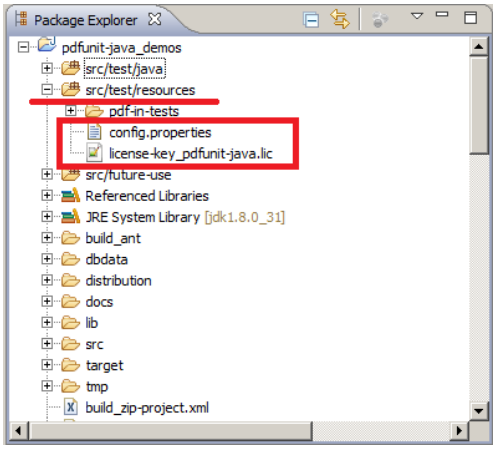
Maven konfigurieren

PDFUnit wird noch nicht über ein zentrales Repository zur Verfügung gestellt. Um es dennoch mit Maven zu nutzen, muss es selber in ein lokales oder unternehmenszentrales Repository eingestellt werden. Dazu wechseln Sie in das Verzeichnis `PDFUNIT_HOME\lib` und führen dort den folgenden Maven-Befehl aus:

```
mvn install:install-file -Dfile=<PATH_TO>pdfunit-java-VERSION.jar -DpomFile=<PATH_TO>pom.xml
```

Anschließend kopieren Sie die Datei `pdfunit.config` in das Verzeichnis `src/test/resources`.

Das folgende Bild zeigt die Projektstruktur nach dem Kopieren:



In der `pom.xml` Ihres Projektes nehmen Sie diese Abhängigkeit auf:

```
<dependency>
  <groupId>com.pdfunit</groupId>
  <artifactId>pdfunit</artifactId>
  <version>2016.05</version>
  <scope>compile</scope>
</dependency>
```

Letzter Schritt für lizensiertes PDFUnit

Die Lizenzdatei `license-key_pdfunit-java.lic` muss immer im Classpath liegen, sonst erscheint die oben beschriebene Message-Box mit der Rechenaufgabe.

12.4. Pfade über Systemumgebungsvariablen setzen

Die Dateien `pdfunit.config` und die Lizenzdatei können auch außerhalb des Classpath's liegen, wenn deren Orte über entsprechende Java-Runtime Umgebungsvariablen deklariert werden. Die Umgebungsvariablen lauten:

- `-Dpdfunit.configfile`
- `-Dpdfunit.licensekeyfile`

Abhängig vom Testsystem (Eclipse, ANT, Maven) können diese Parameter auf vielfältige Weise gesetzt werden. Nutzen Sie die allgemeinen Informationen dieser Systeme, um zu erfahren, wie dort Java System-Properties gesetzt werden. Eine weniger bekannte Möglichkeit, die für alle Umgebungen funktioniert, ist die Betriebssystem-Umgebungsvariable `_JAVA_OPTIONS`:

```
set _JAVA_OPTIONS=-Dpdfunit.configfile=..\myfolder\pdfunit.config
```

Sollten Sie zu diesem Thema Fragen haben, schreiben Sie ein Mail an: info[at]pdfunit.com.

12.5. Einstellungen in der pdfunit.config

Normalerweise muss PDFUnit nicht konfiguriert werden, es gibt mit der Datei `pdfunit.config` aber die Möglichkeit dazu, die nachfolgend beschrieben wird.

Länderkennung der PDF-Dokumente

Für das Arbeiten mit Datumswerten und für das Umwandeln von Zeichenketten in Kleinbuchstaben benötigt Java eine Länderkennung. Diese Länderkennung wird aus der Konfigurationsdatei gelesen.

Die erlaubten Werte entsprechen denen der Klasse `java.util.Locale`. Bei der Auslieferung steht diese Länderkennung auf en (englisch).

```
################################################################################
# Locale of PDF documents, required by some tests.
################################################################################
pdf.locale = en
#pdf.locale = de_DE
#pdf.locale = en_UK
```

Der Wert für die Länderkennung kann groß oder klein geschrieben werden. Ebenso werden ein Unterstrich und ein Minus akzeptiert.

Falls der Key für die Länderkennung versehentlich verändert oder gelöscht wird, entnimmt PDFUnit die Länderkennung der Java-Runtime (`Locale.getDefault()`).

Ausgabeverzeichnis für Fehlerbilder

Wenn beim Vergleich gerenderter Seiten eines Testdokumentes und eines Vergleichsdokumentes Unterschiede erkannt werden, wird ein Fehlerbild erstellt. Das Bild enthält auf der linken Seite das vollständige Referenzdokument und auf der rechten die Differenzen des aktuellen Testdokumentes in roter Farbe. Der Name des Testes erscheint am oberen Rand des Bildes.

Das Ausgabeverzeichnis können Sie in der Konfigurationsdatei festlegen. In der Standardeinstellung werden Diff-Images, die zu existierenden Dateien gehören, in dem Verzeichnis abgelegt, in dem das Testdokument liegt. Das mag für manche Zwecke sinnvoll sein. Wenn Sie aber ein einheitliches, fest vorgegebenes Verzeichnis haben möchten, legen Sie es in der Konfigurationsdatei über die Property `diffimage.output.path.files` fest:

```
################################################################################
#
# 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. Überprüfung der Konfiguration

Überprüfung mit Skript

Die Installation von PDFUnit kann mit einem mitgelieferten Programm überprüft werden. Das Programm wird über das Skript `verifyInstallation.bat` bzw. `verifyInstallation.sh` gestartet:

```
::
:: 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
```

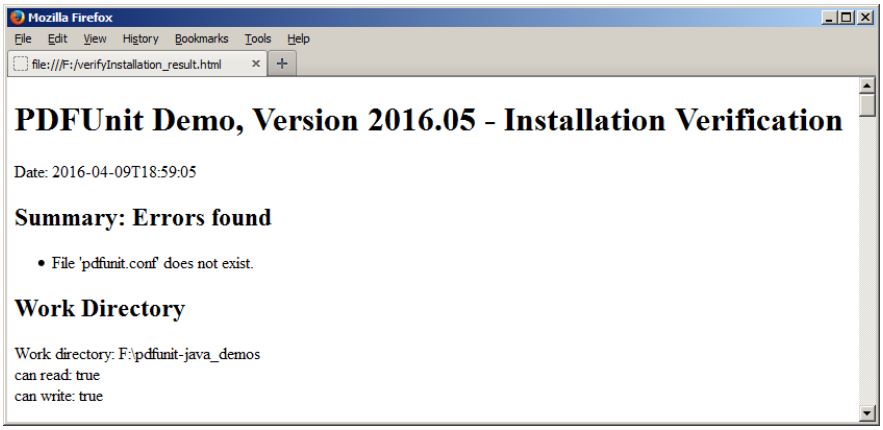
Passen Sie die Pfade an die Verhältnisse Ihrer Installation an.

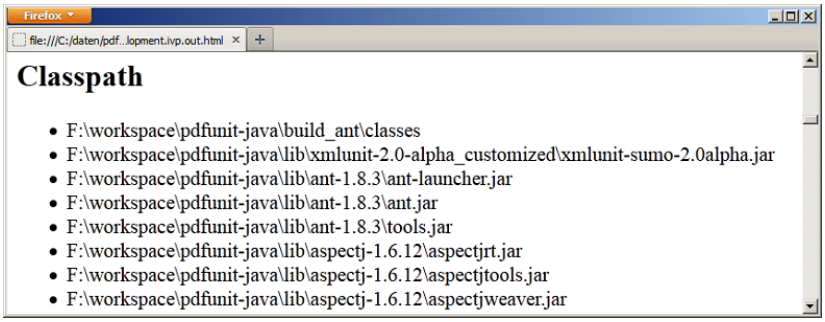
Die Stylesheet-Option kann entfallen. Sie dient vor allem dazu, die Verwendung eigener Stylesheets zu ermöglichen.

Das Skript erzeugt folgende Ausgabe auf der Konsole:

```
Checking installation ...
... finished. Report created, see 'verifyInstallation_result.html'.
```

Der Report listet einerseits eventuelle Fehler auf und andererseits protokolliert er allgemeine Laufzeitinformationen wie Classpath, Umgebungsvariablen und Dateien:





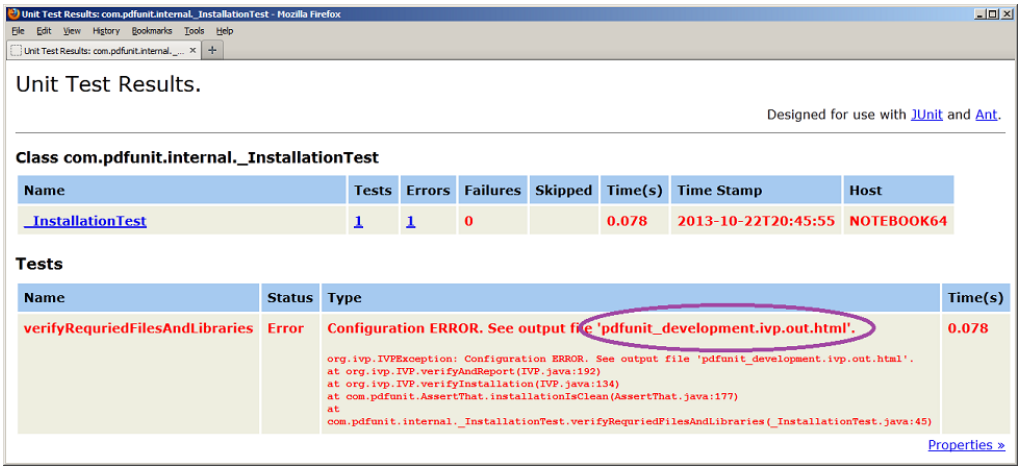
Überprüfung als Unittest

Die Überprüfung der Installation kann auch als Unittest durchgeführt werden. Dadurch ist es möglich, die Systemumgebung der laufenden Tests im Kontext von ANT, Maven oder Jenkins sichtbar zu machen.

Innerhalb eines Unittests wird dazu diese spezielle Testmethode aufgerufen:

```
/*
 * 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");
}
```

Die Methode führt die gleichen Prüfungen aus, wie das zuvor beschriebene Skript. Falls ein Konfigurationsfehler vorliegt, wird der Test „rot“ und verweist in der Fehlermeldung auf die Report-Datei:



Die Report-Datei enthält dieselben Informationen (s.o.), als wäre sie über ein Skript erzeugt worden.

12.7. Installation eines neuen Releases

Die Installation eines neuen Releases von PDFUnit-Java verläuft genauso, wie die Erstinstallation, weil Releases immer vollständig zur Verfügung gestellt werden, nie als Differenz zum vorhergehenden Release.

Beschaffung des neuen Releases

Wenn Sie PDFUnit ohne Lizenzdatei einsetzen, laden Sie sich die neue ZIP-Datei aus dem Internet: .

Wenn Sie PDFUnit mit Lizenzdatei einsetzen, erhalten Sie das neue Release per Mail mit der ZIP-Datei als Anhang sowie einer separaten Datei mit den Lizenzdaten.

Vorbereitende Schritte für alle Umgebungen

Bevor Sie mit dem Releasewechsel beginnen, führen Sie alle vorhandenen Unittests mit dem alten Release durch. Diese sollten „grün“ sein.

Sichern Sie Ihr Projekt.

Durchführung des Updates

Entpacken Sie das neue Release, jedoch nicht in das bestehende Projekt. Nachfolgend wird der Ordner mit dem neuen Release `PDFUNITJAVA_HOME_NEW` genannt. Der Ordner Ihres bestehenden Projektes mit dem alten Release wird nachfolgend `PROJECT_HOME` genannt.

Löschen Sie das Verzeichnis `PROJECT_HOME/lib/pdfunit-OLD-VERSION`.

Kopieren Sie das Verzeichnis `PDFUNITJAVA_HOME_NEW` an die Stelle des alten Releases, beispielsweise nach `PROJECT_HOME/lib/pdfunit-NEW-VERSION`.

Sollten Sie im alten Release die Datei `pdfunit.config` in einem anderem Verzeichnis, als dem Installationsverzeichnis verwendet haben, so kopieren Sie die neue Datei jetzt aus dem Verzeichnis `PROJECT_HOME/lib/pdfunit-NEW-VERSION` an Ihren gewünschten Ort.

Sollten Sie im alten Release Änderungen an der `pdfunit.config` vorgenommen haben, so übertragen Sie die Änderungen in die `pdfunit.config` des neuen Releases.

Wenn Sie ein lizensiertes PDFUnit-Java einsetzen, kopieren Sie die neue Lizenzdatei `license-key_pdfunit-java.lic` an den Ort, an dem sie beim alten Release lag.

Weitere Schritte für ANT

Für ANT sind keine weiteren Schritte notwendig, wenn Sie den Classpath so konfiguriert haben, wie weiter oben beschrieben.

Weitere Schritte für Maven

Das neue Release muss in Ihr lokales oder unternehmensweites Repository eingetragen werden. Öffnen Sie dazu eine Konsole, wechseln in das Verzeichnis `PROJECT_HOME/lib/pdfunit-NEW-VERSION` und führen dort diesen Befehl aus:

```
mvn install:install-file -Dfile=pdfunit-java-VERSION.jar -DpomFile=pom.xml
```

Weitere Schritte für Eclipse

Nehmen Sie die neuen JAR-Dateien in den Build-Path auf. Entfernen Sie die alten JAR-Dateien aus dem Build-Path, damit Eclipse keinen Build-Fehler mehr anzeigt.

Verknüpfen Sie die Javadoc-Dokumentation in Eclipse erneut so, wie in Kapitel : „Eclipse konfigurieren“ (S. 151) beschrieben.

Letzter Schritt

Führen Sie Ihre bestehenden Tests mit dem neuen Release durch. Sofern es keine dokumentierten Inkompatibilitäten zwischen dem alten und neuen PDFUnit-Release gibt, sollten Ihre Tests erfolgreich durchlaufen. Andernfalls lesen Sie die Release-Informationen.

12.8. Deinstallation

Analog zur Installation „per Copy“ wird PDFUnit durch das Löschen der Installationsverzeichnisse wieder sauber deinstalliert. Einträge in Systemverzeichnisse oder in die Registry können nicht zurückbleiben, weil solche nie erstellt wurden. Vergessen Sie nicht, in Ihren eigenen Skripten die Referenzen auf JAR-Dateien oder Verzeichnisse von PDFUnit zu entfernen.

Kapitel 13. Anhang

13.1. Instantiierung der PDF-Dokumente

Die folgende Liste zeigt alle Methoden und ihre Datentypen zum Einlesen von PDF-Dokumenten:

```
// 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 Auch ein Referenz-PDF kann als `String`, `File`, `URL`, `InputStream` oder `byte[]` eingelesen werden.

❸ PDFUnit erkennt alle PDF-Dateien im angegebenen Verzeichnis und führt Tests mit jedem Dokument aus. Wenn ein Dokument als fehlerhaft erkannt wird, bricht der Test ab.

Wenn die PDF-Dokumente passwort-geschützt sind, benötigt PDFUnit entweder das „User-Password“ oder das „Owner-Password“ als zusätzlichen Parameter.

13.2. Seitenauswahl

Vordefinierte Seiten

Für Tests, die sich auf bestimmte Seiten eines PDF-Dokumentes beziehen, existieren in der Klasse `com.pdfunit.Constants` vorgefertigte Konstanten, deren Bedeutung sich aus ihrem Namen ergibt:

```
// 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
```

Die Konstanten aus dem linken Block sind mit denen aus dem rechten Block funktional identisch. Der rechte Block wird noch unterstützt, um zu früheren Releases kompatibel zu sein.

Hier ein Beispiel mit einer der Konstanten:

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

  AssertThat.document(filename)
            .restrictedTo(EVEN_PAGES)
            .hasText()
            .containing("Content", "even pagenumber")
  ;
}
```

Individuelle Seiten

Das nächste Beispiel zeigt, wie beliebige, individuelle Seiten definiert werden können.

```
@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")
  ;
}
```

Es stehen zwei Methoden zur Verfügung, um einzelne oder mehrere Seiten auszuwählen:

```
// How to define individual pages:

PagesToUse.getPage(2);
PagesToUse.getPages(1, 2, 3);
```

Offene Seitenbereiche

Für Tests, die auf Bereiche am Ende oder am Anfang eines Dokumentes zielen, gibt es weitere Methoden:

```
// 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);
```

Die Werte für Ober und Untergrenzen gelten exklusiv.

Das nachfolgende Beispiel prüft das Format ab Seite 3 (einschließlich) bis zum Ende des Dokumentes.

```
@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()
  ;
}
```

Seitenbereiche innerhalb eines Dokumentes

Und als Letztes gibt es die Syntax `PagesToUse.spanningFrom().to()`, um Tests auf einen Bereich innerhalb eines Dokumentes zu beschränken. Das folgende Beispiel validiert Text, der zwei Seiten überspannt.

```
@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)
  ;
}
```

Wichtige Hinweise

- Seitenzahlen beginnen mit '1'.
- Die Seitenangaben in `before(int)` und `after(int)` sind jeweils exklusiv gemeint.
- Die Seitenangaben in `from(int)` und `to(int)` sind jeweils inklusiv gemeint.
- `ON_EVERY_PAGE` bedeutet, dass der gesuchte Text wirklich auf jeder Seite existieren muss.
- Dagegen reicht es, wenn bei `ON_ANY_PAGE` der gesuchte Text auf einer Seite existiert.
- Die Verwendung von `restrictedTo(PagesToUse)` und von `restrictedTo(PageRegion)` hat keinen Einfluss auf seitenunabhängige Testmethoden.

13.3. Seitenausschnitt definieren

Text- und Bildvergleiche können auf Ausschnitte einer Seite beschränkt werden. Dazu wird ein rechteckiger Ausschnitt durch vier Werte definiert: die **linke obere** Ecke mit ihren x/y-Koordinaten sowie die Breite und Höhe des Auschnittes:

```
// 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 Wenn keine Einheit mitgegeben wird, gilt die Einheit `MILLIMETERS`.

Seitenausschnitte können in den Einheiten Millimeter oder Points definiert werden. Weitere Informationen dazu liefert das Kapitel, 13.8: „Maßeinheiten - Points und Millimeter“ (S. 168) beschrieben.

Hier ein Beispiel:

```
@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")
  ;
}
```

So leicht die Benutzung eines solchen Rechtecks ist, so liegt die Schwierigkeit wahrscheinlich darin, die richtigen Werte für den Ausschnitt zu ermitteln. PDFUnit stellt deshalb das kleine Hilfsprogramm `RenderPdfPageRegionToImage` zur Verfügung. Mit diesem können Sie das Rechteck mit den notwendigen Werten auf der Basis der Einheiten `mm` als PNG-Datei extrahieren:

```
::
:: 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
```

Das so entstandene Bild müssen Sie überprüfen. Enthält es exakt den gewünschten Ausschnitt? Falls nicht, variieren Sie die Werte solange, bis der Ausschnitt passt. Anschließend übernehmen Sie die Werte in Ihren Test.

13.4. Textvergleich

Ein erwarteter Text und der tatsächliche Text einer PDF-Seite können auf folgende Art miteinander verglichen werden:

```
// 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 Bei diesen Methoden werden die Whitespaces „normalisiert“. Das heißt, Leerzeichen am Anfang und Ende werden entfernt und alle Whitespaces innerhalb eines Textes werden auf ein Leerzeichen reduziert.

❷ Die Behandlung von Whitespaces wird über den zweiten Parameter gesteuert. Es stehen die Konstanten `IGNORE`, `NORMALIZE` und `KEEP` zur Verfügung, sie sind in Kapitel 13.5: „Behandlung von Whitespaces“ (S. 164) separat beschrieben. Diese Beeinflussung der Whitespace-Behandlung gilt für alle aufgeführten Methoden mit 'WhitespaceProcessing' als zweitem Parameter.

❸❹ Die Whitespace-Behandlung der Methoden `first(..)` und `then(..)` hängen zusammen. Die Method `then(..)` behandelt Whitespace genauso, wie die zuvor aufgerufene Methode `first(..)`.

Vergleiche mit Regulären Ausdrücken folgen den Regeln und Möglichkeiten der Klasse `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.*")
  ;
}
```

Die Methoden `containing(String[])` und `notContaining(String[])` können mit mehreren Suchbegriffen aufgerufen werden. Ein Test mit `containing(String[])` gilt als erfolgreich, wenn jeder Suchbegriff auf jeder ausgewählten Seiten auftaucht. Ein Test mit `notContaining(String[])` ist erfolgreich, wenn alle Suchbegriffe auf allen ausgewählten Seite fehlen:

```
@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. Behandlung von Whitespaces

In fast allen Tests werden Texte verglichen. Viele Vergleiche würden nicht funktionieren, wenn die Whitespaces eines Textes „so, wie sie sind“ Teil des Vergleiches wären. Deshalb gibt es für Tests, bei denen eine flexible Behandlung von Whitespaces sinnvoll ist, die Möglichkeit einer benutzergesteuerten Whitespace-Behandlung. PDFUnit stellt folgenden Konstanten zur Verfügung:

```
// 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 Text wird so komprimiert, dass er keine Whitespaces mehr enthält.

❸❹ Alle Whitespaces bleiben erhalten.

❺❻ Whitespaces am Anfang und am Ende eines Textes werden gelöscht. Whitespaces innerhalb eines Textes werden auf ein Leerzeichen reduziert.

Jeweils zwei Konstanten haben die gleiche Bedeutung. Sie werden redundant angeboten, um verschiedene sprachliche Vorlieben zu bedienen.

Ein Beispiel:

```
@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)
  ;
}
```

In diesem Beispiel wird die erwartete Zeichenkette vollständig ohne Zeilenumbrüche formuliert, obwohl die PDF-Seite mehrere davon enthält. Durch die Angabe `IGNORE_WHITESPACES` funktioniert der Test trotzdem bestens.

`NORMALIZE_WHITESPACES` ist die Standardbehandlung, falls nichts anderes angegeben wird. Testmethoden, bei denen eine flexible Behandlung von Whitespaces nicht sinnvoll ist, bieten keine Möglichkeit für eine benutzergesteuerte Whitespace-Behandlung.

Testmethoden, die Reguläre Ausdrücke verarbeiten, verändern Whitespaces nicht. Bei Bedarf muss die Behandlung der Whitespaces in den regulären Ausdruck integriert werden, beispielsweise so:

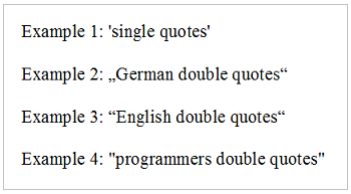
```
(?ms).*print(.*)
```

Der Teilausdruck `(?ms)` bedeutet, dass die Suche über mehrere Zeilen reicht. Zeilenumbrüche werden als 'Character' interpretiert.

13.6. Anführungszeichen in Suchbegriffen

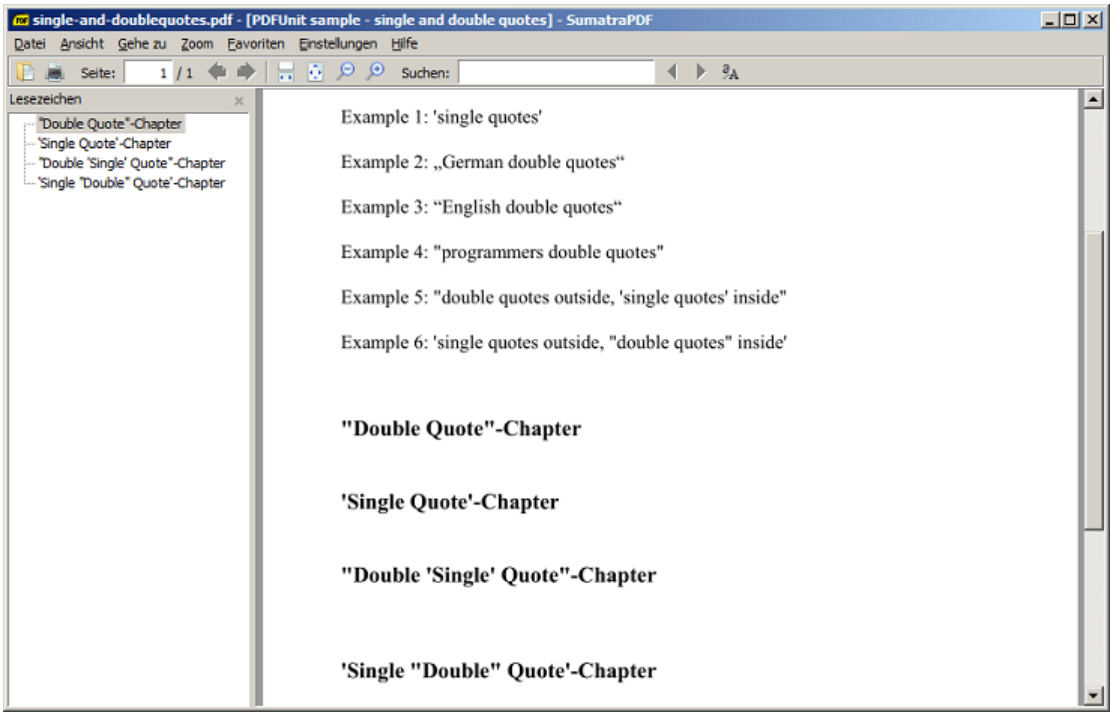
Unterschiedliche Arten von Anführungszeichen

Wichtiger Hinweis: Der Begriff „Anführungszeichen“ wird in Texten unterschiedlich umgesetzt, wie das folgende Bild zeigt:



“Englische“ und „deutsche“ Anführungszeichen stören nicht während der Ausführung von Tests. Lediglich bei ihrer Erstellung könnten Sie das Problem haben, sie in Ihren Editor zu bekommen. Tipp: kopieren Sie die gewünschten Anführungszeichen von einem Textverarbeitungsprogramm oder einem bestehenden PDF-Dokument und fügen Sie sie dann in Ihre Datei ein.

Die "programmers double quotes" benötigen eine besondere Aufmerksamkeit, weil sie in Java als Zeichenkettenbegrenzer dienen. Die nachfolgenden Absätze und Beispiele gehen detailiert darauf ein, sie basieren alle auf dem folgenden Dokument:



Gültige Beispiele

'Single-Quotes', “englische“ und „deutsche“ Anführungszeichen innerhalb von Zeichenketten bereiten alle keine Probleme, lediglich "Double-Quotes" müssen mit einem Backslash maskiert werden:

```
@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.*\".*")
  ;
}
```

Anführungszeichen innerhalb von XPath-Ausdrücken

Zeichenketten, die im Laufe der Verarbeitung mit XPath weiterverarbeitet werden, dürfen **nicht gleichzeitig** Single- und Double-Quotes enthalten. Diese Bedingung ist im folgenden Beispiel verletzt:

```
@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)
  ;
}
```

Der Fehler lässt sich nur vermeiden, wenn die XPath-Bedingung anders formuliert wird:

```
@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. Datumsauflösung

Der Vergleich eines Datums (Erstellungs- oder Änderungsdatum) mit einem Erwartungswert kann sich entweder nur auf Jahr-Monat-Tag beziehen oder zusätzlich noch auf Stunde-Minute-Sekunde. Für die Unterscheidung dieser beiden Möglichkeiten stellt PDFUnit zwei Konstanten zur Verfügung:

```
// Constants for date resolutions:

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

In den folgenden Methoden werden diese Konstanten verwendet:

```
// 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()
```

Der Vergleich von Datumswerten zweier PDF-Dokumenten findet immer in der Auflösung `DateResolution.DATE` statt.

13.8. Maßeinheiten - Points und Millimeter

Tests mit Seitenausschnitten benötigen Werte für Länge und Breite. Die Werte können in Millimeter oder Points angegeben sein. Dafür existieren die folgenden Konstanten:

```
// Predefined format units:

com.pdfunit.Constants.MILLIMETERS
com.pdfunit.Constants.POINTS
```

Bei der Umrechnung von Points in Millimeter wird mit 72 DPI (Dots per Inch) gerechnet. Nachfolgend ein paar Beispiele, in denen Maßeinheiten explizit oder implizit eine Rolle spielen:

Beispiel - Größe von Formularfeldern

```
@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)
  ;
}
```

Mit der Breite und Höhe von Formularfeldern sind die PDF-internen Feldeigenschaften gemeint. Weil diese in 'Points' gespeichert sind, ist auch der Default in PDFUnit für diese Methoden 'Points'. Und weil ein Umrechnen von Millimetern in Points zu Rundungsfehlern führen kann, muss als dritter Parameter noch die erlaubte Abweichung mitgegeben werden.

Beispiel - Größe des Seitenformats

```
@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)
  ;
}
```

Das Format für die Abmessung einer Seite wird nicht über Konstanten gesteuert, sondern über die zwei Klassen `DocumentFormatMillis` und `DocumentFormatPoints`.

Beispiel - Fehlermeldungen

In Fehlermeldungen werden sowohl Millimeter, als auch die ursprünglich verwendete Einheit ausgegeben. Würde beispielsweise die Breite im letzten Beispiel mit `111 POINTS` angegeben, würde PDFUnit folgende Fehlermeldung ausgeben:

```
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. Fehlermeldungen, Fehlernummern

Fehlermeldungen von PDFUnit gibt es in englischer und deutscher Sprache. Sie liefern detailierte Informationen, um eine Fehlerbehebung zu erleichtern. Insgesamt wird versucht, Meldungen so sprechend wie möglich zu gestalten. Diese Absicht demonstriert eine Fehlermeldung für ein falsches Seitenformat:

```
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')'.
```

Damit Fehlermeldungen lesbar bleiben, werden lange Parameterinhalte verkürzt und die Position des Fehlers durch die Zeichen `<[` und `]>` markiert. Die Anzahl der verkürzten Zeichen wird mit `'...NN...'` dargestellt:

```
The expected content does not match the JavaScript in 'javaScriptClock.pdf'.
Expected: '//<[Thisfileco...41...dbyPDFUnit]>',
but was:  '//<[Constantsu...4969...);break;}}]>'.
```

Die Bereitstellung von Fehlermeldungen in weiteren Sprachen neben Deutsch und Englisch ist technisch vorbereitet. Schreiben Sie Ihre Wünsche an info[at]pdfunit.com.

Fehlernummern

Fehlertexte stehen in den Dateien `messages.properties` bzw. `messages_SPRACHE.properties`. Jede Fehlermeldung in diesen Dateien hat das Format `key = value`. Als `key` wird im Falle von PDFUnit der Name einer Klasse verwendet, insofern muss der Begriff 'Fehlernummer (error number)' durch den Begriff 'Fehlerschlüssel (error key)' ersetzt werden.

Wenn zur Laufzeit ein Fehler abgefangen wurde, kann dieser Fehlerschlüssel abgefragt werden:

```
@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);
  }
}
```

Im catch-Zweig könnte der Error-Key auch an eine Logging-Funktion übergeben werden.

13.10. Sprache für Fehlermeldungen einstellen

PDFUnit wird mit Fehlermeldungen für Englisch und Deutsch ausgeliefert. Zur Ausführungszeit wird automatisch die Sprache gewählt, die das Betriebssystem an den Java-Prozess meldet. Wenn eine andere Sprache gewünscht wird, muss diese Sprache für Java eingestellt werden. Dazu muss man nicht das Betriebssystem verbiegen, es reicht, die gewünschte Sprache beim Start der JVM anzugeben:

```
// JVM start options:

-Duser.language=de -Duser.country=DE
-Duser.language=es -Duser.country=ES
```

Es gibt noch eine weitere Möglichkeit, für Java-Anwendungen die Sprachumgebung zu setzen, indem die zuvor gezeigen JVM-OPTIONS in die Umgebungsvariable `_JAVA_OPTIONS` geschrieben werden:

```
// 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
```

Ist die Umgebungsvariable gesetzt, wirkt sie auf alle Java-Prozesse, die anschließend gestartet werden. Wird die Umgebung in einem Skript gesetzt, wirken die Optionen nur innerhalb des Skriptes, die globalen Einstellungen der Maschine bleiben unverändert.

13.11. XPath-Einsatz

Allgemeine Erläuterungen zu XPath in PDFUnit

Die Nutzung von XPath zur Bestimmung von Teilen eines PDF-Dokumentes öffnet ein weites Feld von Testmöglichkeiten, das mit einer API alleine nicht abgedeckt werden kann.

Verschiedene Kapitel enthalten schon eine Beschreibung der XPath-Testfunktionen, sofern die Testbereiche XPath-Tests besitzen. Dieses Kapitel hier dient als Übersicht mit Verweisen zu den Spezialkapitel.

```
// Validating a single PDF using XPath:
.hasXFAData().matchingXPath(..)        3.36: „XFA Daten“ (S. 78)
.hasXMPData().matchingXPath(..)        3.37: „XMP-Daten“ (S. 81)
.hasZugferdData().matchingXPath(..)    3.39: „ZUGFeRD“ (S. 84)

// Comparing two documents using XPath:
.haveXFAData().matchingXPath(..)       4.15: „XFA-Daten vergleichen“ (S. 101)
.haveXMPData().matchingXPath(..)       4.16: „XMP-Daten vergleichen“ (S. 102)
```

Daten als XML extrahieren

Für alle Teile eines PDF-Dokumentes, für die es XPath-Tests gibt, und für nicht sichtbare Eigenschaften eines Dokumentes werden Extraktionsprogramme zur Verfügung gestellt:

```
// 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
```

Die Hilfsprogramme werden im Kapitel 9.1: „Allgemeine Hinweise für alle Hilfsprogramme“ (S. 124) genauer beschrieben.

Namensräume mit Präfix

Namensräume, für die ein Präfix definiert ist, werden von PDFUnit automatisch erkannt.

Default-Namensraum

Der Default-Namensraum kann nicht automatisch ermittelt werden, weil es in einem XML-Dokument prinzipiell mehrere Default-Namensräume geben darf. Aus diesem Grund muss der Default-Namensraum im Test angegeben werden. Er kann mit einem **beliebigen** Präfix verwendet werden:

```
/**
 * 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)
  ;
}
```

Es sieht komisch aus, das willkürliche Prefix zu `foo` zu verwenden, aber der Java-Standard verlangt ein **beliebiges** Präfix. Es darf aus Java-Sicht nicht weggelassen werden. In der Praxis wählen Sie bitte ein sprechenderes Prefix.

Das nächste Beispiel zeigt die Deklaration des Default-Namensraumes für eine `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-Kompatibilität

Für XPath-Ausdrücke stehen im Prinzip alle Syntaxelemente und Funktionen von XPath zur Verfügung. Allerdings ist die Menge der tatsächlich verfügbaren Funktionen von der Version des verwendeten XML-Parsers und XSLT-Prozessors abhängig. PDFUnit verwendet den vom JDK mitgelieferten XML-Parser bzw. XSLT-Prozessor (Standard JAXP). Insofern bestimmt die jeweils verwendete Java-Engine die Kompatibilität zum XPath-Standard.

Das Kapitel 13.12: „JAXP-Konfiguration“ (S. 172) erläutert die allgemeine JAXP-Konfiguration eines JRE/JDK, um z.B. Xerces als externen XML-Parser zu nutzen.

13.12. JAXP-Konfiguration

Die Standard-JAXP-Konfiguration des JDK kann über die allgemeinen Mechanismen der JAXP-Konfiguration verändert werden. Weil die aber nicht allgemein bekannt sind, werden sie am Beispiel von Xerces und Xalan nachfolgend erläutert.

Die Java-Runtime liest die Werte folgender JAXP-Umgebungsvariablen ein und lädt dann die dort angegebene Java-Klasse:

```
"javax.xml.parsers.DocumentBuilderFactory"
"javax.xml.parsers.SAXParserFactory"
"javax.xml.transform.TransformerFactory"
```

Diese Umgebungsvariablen können auf unterschiedliche Weise gesetzt werden, wie die nachfolgende Liste zeigt. Eine Konfigurationsmöglichkeit, die eine andere übersteuert, steht in der Aufzählung weiter oben.

1. Die JAXP Umgebungsvariablen können im Programm selber gesetzt werden:

   ```
   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. Die Konfigurationsdaten können mit der Startoption `-D` in der Umgebungsvariablen `_JAVA_OPTIONS` bereitgestellt werden. Hier wird nur ein Wert als Beispiel dargestellt, drei sind natürlich auch möglich:

   ```
   set _JAVA_OPTIONS=-Djavax.xml.transform.TransformerFactory=
                     org.apache.xalan.processor.TransformerFactoryImpl
   ```

3. Die Konfigurationsdaten können mit der Startoption `-D` in der Umgebungsvariablen `JAVA_TOOL_OPTIONS` bereitgestellt werden. Diese Umgebungsvariable wird von einigen JDK-Implementierungen ausgewertet. Auch hier wird nur ein Wert als Beispiel:

   ```
   set JAVA_TOOL_OPTIONS=-Djavax.xml.transform.TransformerFactory=
                         org.apache.xalan.processor.TransformerFactoryImpl
   ```

4. Die Konfigurationsdaten können in der Datei `jaxp.properties` im Verzeichnis `JAVA_HOME/jre/lib` zur Verfügung gestellt werden:

```
#
# 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. Die JAXP-Umgebungsvariablen können ANT über die Umgebungsvariable `ANT_OPTS` gesetzt werden:

```
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
```

Die deklarierten XML/XSLT-Klassen müssen entweder im Classpath oder im Verzeichnis `JAVA_HOME\jre\lib\ext` liegen. Letzteres ist nicht zu bevorzugen, weil kaum jemand diese Möglichkeit kennt und eine eventuelle Fehlersuche wegen dieses Nicht-Wissens unnötig lange dauern würde.

Beachten Sie, dass Eclipse seine Umgebungsvariablen beim Start einliest und anschließend nicht verändert. Insofern müssen Sie Eclipse nach der Änderung einer Umgebungsvariablen neu starten.

13.13. Einsatz mit TestNG

PDFUnit läuft auch mit TestNG.

Wenn Sie lediglich die einfache Annotation `@Test` verwenden, ist sowieso kein Unterschied erkennbar. Erst wenn z.B. Exceptions erwartet werden, ist TestNG zu erkennen:

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

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

13.14. Versionshistorie

2010

Der Ursprung von PDFUnit lag in einer Kundenanfrage im August 2010, die mit der damals verfügbaren Bibliothek 'jPdfUnit 1.1' (http://jpdfunit.sourceforge.net) nicht gelöst werden konnte. Auch das Apache Projekt 'PDFBox' (http://pdfbox.apache.org) konnte die gewünschten Funktionen nicht ausreichend abdecken, wohl aber iText (http://itextpdf.com) mit ein wenig Programmierung 'drum-herum'.

2011

Im Laufe des Jahres 2011 wuchs das Wissen über PDF und iText und Ende des Jahres 2011 begann die Entwicklung von PDFUnit auf der Basis von iText.

Release 2012.07

Das Testwerkzeug erfüllte die gestecken Ziele. Die Weiterentwicklung ging jedoch wegen anderer beruflicher Verpflichtungen nur langsam voran.

Release 2013.01

Zu Beginn des Jahres 2013 gab es wieder mehr Zeit für die Weiterentwicklung. In dieser Zeit entstand die Dokumentation. Bestehende Funktionen wurden abgerundet, fehlerbereinigt neue hinzugefügt.

Release 2014.06

Zahlreiche kleine Verbesserungen rechtfertigten eine neue Version von PDFUnit-Java.

Release 2015.10

Die wesentliche Erweiterung war der PDFUnit-Monitor. Er bedient die Zielgruppe der Nicht-Entwickler und liest Testinformationen aus einer Excel-Datei.

Release 2016.05

Die Implementierung wurde von iText auf PDFBox 2.0 umgestellt. Zusätzlich wurde der Funktionsumfang um die Validierung von QR-Code, Barcode und ZUGFeRD-Daten erweitert. Text mit der Schreibrichtung rechts-links können analysiert werden und es wurde die Möglichkeit geschaffen, ganze Verzeichnisse zu prüfen.

13.15. Nicht Implementiertes, Bekannte Fehler

Extraktion von Feldinformation

Es werden nicht alle Eigenschaften von Formularfeldern extrahiert, beispielsweise nicht 'background color', 'border color' und 'border styles'.

Extraktion von Signaturdaten

Es werden noch nicht alle verfügbaren Signaturdaten nach XML exportiert. Insofern wird es zukünftig noch Änderungen am XML-Format der Signaturdaten geben.

Farben

Im aktuellen Release 2016.05 werden keine Farben analysiert. Falls Farben dennoch getestet werden sollen, kann das über gerenderte Seiten geschehen. Die Kapitel 3.18: „Layout - gerenderte volle Seiten“ (S. 45) und 3.19: „Layout - gerenderte Seitenausschnitte“ (S. 46) beschreiben diese Art der Tests.

Ebenenbezogene (Layer) Inhalte

Der Vergleich auf Texte und Bilder bezieht sich noch nicht auf einzelne Ebenen.

Vollständige XMP-Daten

Im aktuellen Release werden nur die XMP-Daten der Dokumentenebene (document level) extrahiert und ausgewertet. Zukünfig werden alle XMP-Daten extrahiert.

Stichwortverzeichnis

G

H

I

J

K

L

U

V

W

X

Z