
PDF automatisiert testen

PDFUnit-XML

Carsten Siedentop

Inhaltsverzeichnis

Vorwort	5
1. Über diese Dokumentation	6
2. Quickstart	8
3. Funktionsumfang	9
3.1. Überblick	9
3.2. Aktionen (Actions)	11
3.3. Anhänge (Attachments)	17
3.4. Anzahl verschiedener PDF-Bestandteile	19
3.5. Berechtigungen	20
3.6. Bilder in Dokumenten	21
3.7. Datum	24
3.8. Dokumenteneigenschaften	26
3.9. Fast Web View	29
3.10. Format	30
3.11. Formularfelder	32
3.12. Formularfelder, Textüberlauf	38
3.13. JavaScript	40
3.14. Layer	42
3.15. Layout - gerenderte volle Seiten	44
3.16. Layout - gerenderte Seitenausschnitte	45
3.17. Lesezeichen (Bookmarks) und Sprungziele	46
3.18. Passwort	49
3.19. Schriften	50
3.20. Seitenzahlen als Testziel	54
3.21. Signaturen und Zertifikate	54
3.22. Sprachinformation (Language)	59
3.23. Texte	60
3.24. Texte - in Ausschnitten einer Seite	65
3.25. Texte - senkrecht, schräg und überkopf	65
3.26. Tagging	66
3.27. Trapping-Info	67
3.28. Version	68
3.29. XFA Daten	69
3.30. XMP-Daten	72
3.31. Zertifiziertes PDF	74
4. Vergleiche gegen ein Master-PDF	76
4.1. Überblick	76
4.2. Aktionen vergleichen	77
4.3. Anhänge (Attachments) vergleichen	78
4.4. Berechtigungen vergleichen	79
4.5. Bilder vergleichen	80
4.6. Datumswerte vergleichen	81
4.7. Dokumenteneigenschaften vergleichen	81
4.8. Formate vergleichen	82
4.9. Formularfelder vergleichen	82
4.10. JavaScript vergleichen	83
4.11. Layout vergleichen (gerenderte Seiten)	84
4.12. Lesezeichen (Bookmarks) vergleichen	85
4.13. "Named Destinations" vergleichen	86
4.14. PDF-Bestandteile vergleichen	87
4.15. Schriften vergleichen	87
4.16. Signaturnamen vergleichen	88
4.17. Text vergleichen	88
4.18. XFA-Daten vergleichen	89

4.19. XMP-Daten vergleichen	90
4.20. Sonstige Vergleiche	91
5. Tests mit mehreren Dokumenten	93
6. PDFUnit-Monitor	95
7. Unicode	99
8. XPath-Einsatz	103
9. Hilfsprogramme zur Testunterstützung	105
9.1. Allgemeine Hinweise für alle Hilfsprogramme	105
9.2. Anhänge extrahieren	105
9.3. Bilder aus PDF extrahieren	107
9.4. Feldeigenschaften nach XML extrahieren	108
9.5. JavaScript extrahieren	109
9.6. Lesezeichen nach XML extrahieren	110
9.7. PDF-Dokument seitenweise in PNG umwandeln	111
9.8. PDF-Seite ausschnittsweise in PNG umwandeln	112
9.9. Schrifteigenschaften nach XML extrahieren	114
9.10. Signaturdaten nach XML extrahieren	116
9.11. Sprungziele nach XML extrahieren	117
9.12. Unicode-Texte in Hex-Code umwandeln	118
9.13. XFA-Daten nach XML extrahieren	119
9.14. XMP-Daten nach XML extrahieren	119
10. Praxisbeispiele	121
10.1. Passt ein Text in vorgefertigte Formularfelder	121
10.2. Neues Logo auf jeder Seite	121
10.3. Unterschrift des neuen Vorstandes	122
10.4. Name des alten Vorstandes	122
10.5. Schachtelungstiefe von Bookmarks	123
11. Installation, Konfiguration, Update	124
11.1. Technische Voraussetzungen	124
11.2. Installation	124
11.3. Starten von PDFUnit-XML	127
11.4. Einstellungen in der config.properties	129
11.5. Überprüfung der Konfiguration	130
11.6. Update von PDFUnit-XML	132
11.7. Update von PDFUnit-Java	133
11.8. Deinstallation	133
12. PDFUnit für Nicht-XML Systeme	134
12.1. Kurzer Blick auf PDFUnit-Java	134
12.2. Kurzer Blick auf PDFUnit-Perl	134
12.3. Kurzer Blick auf PDFUnit-NET	135
13. Anhang	136
13.1. Instantiierung der PDF-Dokumente	136
13.2. Seitenauswahl	136
13.3. Textvergleich	138
13.4. Behandlung von Whitespaces	138
13.5. Anführungszeichen in Suchbegriffen	139
13.6. Seitenausschnitt definieren	142
13.7. Maßeinheiten - Points, Millimeter,	143
13.8. Fehlermeldungen	144
13.9. Datumsauflösung	145
13.10. Default-Namensraum in XML	145
13.11. Konfiguration überprüfen	146
13.12. Versionshistorie	146
13.13. Nicht Implementiertes, Bekannte Fehler	147
Stichwortverzeichnis	149

Vorwort

Aktuelle Testsituation in Projekten

Telefonrechnungen, Versicherungspolicen, amtliche Bescheide, Verträge jeglicher Art werden heute häufig als PDF-Dokument elektronisch zugestellt. Die Erstellung erfolgt in vielen Programmiersprachen mit zahlreichen Bibliotheken. Je nach Komplexität der zu erstellenden Dokumente ist diese Programmierung nicht einfach. In jedem Prozessschritt auf dem Weg zum PDF können Fehler entstehen:

- Steht auf Seite 2 der richtige Text?
- Erscheint das neue Logo auf allen Dokumenten?
- Sind die Schriften eingebettet, wie beabsichtigt?
- Stimmt das Layout mit der Vorgabe überein?
- Enthält das Dokument die richtige Barcode-Graphik?
- Ist das PDF signiert?

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. Es ist teuer und selber fehleranfällig.

Es war längst überfällig, ein einfach zu nutzendes Testsystem zu entwickeln.

Egal, ob PDF-Dokumente mit einem mächtigen Design-Werkzeug erstellt, aus MS-Word oder Libre-Office exportiert, über eine API erstellt wurden oder aus einem XSL-FO Workflow herausgefallen sind, mit PDFUnit kann jedes PDF-Dokument getestet werden.

Intuitive Schnittstelle

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

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

```
<testcase name="compareText_OnEveryPage">
  <assertThat testDocument="master/documentUnderTest.pdf"
              masterDocument="master/masterDocument.pdf"
  >
    <haveSameText on="EVERY_PAGE" />
  </assertThat>
</testcase>
```

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

Zeit, anzufangen

Spiele Sie nicht weiter Lotto mit den Daten und Prozessen Ihrer Dokumentenerstellung. Überprüfen Sie das Ergebnis des Workflows durch automatisierte Tests.

Kapitel 1. Über diese Dokumentation

Wer sollte sie lesen

Die vorliegende Dokumentation richtet sich in erster Linie an Mitarbeiter der Qualitätssicherung, deren Aufgabe es ist, dafür zu sorgen, dass technisch erzeugte PDF-Dokumente "richtig" sind.

Es wird davon ausgegangen, dass Sie Grundkenntnisse in XML besitzen. Ebenfalls ist ein Grundverständnis über Testautomatisierung hilfreich, aber keine Voraussetzung.

Code-Beispiele

Ein Demo-Projekt mit vielen Beispielen gibt es hier: <http://www.pdfunit.com/de/download/index.html>.

XML-Struktur

Die XML-Syntax von PDFUnit ist mit XML Schema abgesichert. Die Dokumentation des Schemas ist online verfügbar: <http://www.pdfunit.com/de/api/xml/index.html>.

Andere Programmiersprachen

PDFUnit gibt es nicht nur für XML, sondern auch für Java und Perl. Eine Implementierung in C# ist in Arbeit. Für jede Sprache existiert eine eigene Dokumentation.

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 [problem\[at\]pdfunit.com](mailto:problem[at]pdfunit.com) schicken.

Neue Testfunktionen gewünscht?

Hätten Sie gerne neue Testfunktionen, wenden Sie sich per Mail an [request\[at\]pdfunit.com](mailto:request[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 distanzieren, 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 noch laufende 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 Menge 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.

Bruno Lowagie, der Gründer von iText, hat diese Dokumentation gelesen und mir wertvolle kritische Anmerkungen zu speziellen Kapiteln geschickt. Seine tiefe Kenntnis über PDF war eine große Hilfe für mich.

Herstellung dieser Dokumentation

Die vorliegende Dokumentation wurde mit DocBook-XML erstellt. Sowohl PDF als auch HTML werden aus einer einzigen Textquelle erstellt. 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 Aufgaben mit höherer Priorität.

Feedback

Jegliche Art von Feedback ist willkommen, schreiben Sie einfach an [feedback\[at\]pdfunit.com](mailto:feedback[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 mit PDFUnit testen:

```
<testcase name="hasOnePage_de">
  <assertThat testDocument="quickstart/quickstartDemo_de.pdf">
    <hasNumberOfPages>1</hasNumberOfPages>
  </assertThat>
</testcase>

<testcase name="hasGreeting_de">
  <assertThat testDocument="quickstart/quickstartDemo_de.pdf">
    <hasText on="LAST_PAGE">
      <containing>Vielen Dank für die Nutzung unserer Serviceleistungen</containing>
    </hasText>
  </assertThat>
</testcase>

<testcase name="hasExpectedCharge_de">
  <assertThat testDocument="quickstart/quickstartDemo_de.pdf">
    <hasText on="FIRST_PAGE">
      <inClippingArea upperLeftX="172" upperLeftY="178" width="20" height="9" >
        <containing>29,89 Euro</containing> <!-- This value is intentionally false. -->
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

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

Unit Test Results.

Designed for use with [JUnit](#) and [Ant](#).

Class org.pdfunit.xml.QuickstartTests_de

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
QuickstartTests_de	3	0	1	0	0.050	2013-10-25T16:29:14	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
hasGreeting_de	Success		0.018
hasExpectedCharge_de	Failure	Page 1 of 'C:\daten\p...df\used-for-tests\quickstart\quickstartDemo_de.pdf' does not contain the expected sequence '29,89 Euro'. junit.framework.AssertionFailedError: Page 1 of 'C:\daten\p...df\used-for-tests\quickstart\quickstartDemo_de.pdf' does not contain the expected sequence '29,89 Euro'. at com.pdfunit.validators.ContentValidator.a(SourceFile:521) at com.pdfunit.validators.ContentValidator.containing(SourceFile:185) at com.pdfunit.validators.ContentValidator.containing(SourceFile:139) at org.pdfunit.xml.QuickstartTests_de.hasExpectedCharge_de(QuickstartTests_de.java:53)	0.027
hasOnePage_de	Success		0.002

[Properties »](#)

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

Jeder Test beginnt mit dem Tag `<testcase name=".." />`, das als Attribut den Namen des Tests angibt. Darin enthalten ist immer das Tag `<assertThat testDocument=".." />` mit dem Namen der zu testenden PDF-Datei.

Innerhalb von `<assertThat>` folgen dann weitere Tags, die jeweils unterschiedliche Testbereiche, wie z.B. Inhalt, Schriften, Layout etc. abdecken.

Die nachfolgenden Beispiele zeigen verschiedene Einstiege in einen Test:

```
<!-- Instantiating a test document for specific tests: -->

<testcase name="test111">
  <assertThat testDocument="documentUnderTest.pdf">
    <hasXXX> <!-- Switch to one of many test scopes. -->
      ... <!-- Use test scope specific tags here. -->
    </hasXXX>
  </assertThat>
</testcase>
```

```
<!-- Comparing a test document with a master document: -->

<testcase name="test222">
  <assertThat testDocument="documentUnderTest.pdf"
             masterDocument="masterDocument.pdf"
  >
    <haveSameXXX /> <!-- Comparing many parts of a PDF. -->
  </assertThat>
</testcase>
```

```
<!-- Using encrypted PDF documents: -->

<testcase name="test333">
  <assertThat testDocument="documentUnderTest.pdf"
             testPassword="test-password"
  >
    ...
  </assertThat>
</testcase>
```

```
<!-- Set a test to 'ignore': -->

<testcase name="test555"
          ignore="No suitable document available"
>
  <assertThat testDocument="documentUnderTest.pdf"
  >
    ...
  </assertThat>
</testcase>
```

Es können auch mehrere PDF-Dokumente in einen Test einfließen. Solche Tests beginnen mit dem Tag `<assertThatEachDocument>`:

```
<testcase name="textInMultipleDocuments">
  <assertThatEachDocument>
    <pdf name="%pdfdir%/multipleDocuments/document_en.pdf" />
    <pdf name="%pdfdir%/multipleDocuments/document_es.pdf" />
    <pdf name="%pdfdir%/multipleDocuments/document_de.pdf" />
    <hasText on="FIRST_PAGE" >
      <containing>28.09.2014</containing>
      <containing>XX-123</containing>
    </hasText>
  </assertThatEachDocument>
</testcase>
```

Das Kapitel 5: „Tests mit mehreren Dokumenten“ (S. 93) gibt ausführliche Informationen über Tests mit mehreren Dokumenten.

Syntax für erwartete Fehler

Tests, die einen Fehler erwarten, müssen dies über das Attribut `errorExpected="YES"` deklarieren:

```
<testcase name="hasSignature_DocumentNotSigned"
  errorExpected="YES"
>
  <assertThat testDocument="signed/notSigned.pdf">
    <hasSignature name="Signature2" />
  </assertThat>
</testcase>
```

Testbereiche

Die folgende Liste gibt einen vollständigen Überblick über die Testgebiete von PDFUnit. Der jeweilige Link hinter einem Tag verweist auf das Kapitel, das jedes Testgebiet ausführlich beschreibt. Die Kapitel sind alphabetisch sortiert.

```
<!-- Each of the following tags opens a new test scope: -->

<areBothForFastWebView />      4.20: „Sonstige Vergleiche“ (S. 91)
<asRenderedPage />            3.15: „Layout - gerenderte volle Seiten“ (S. 44)

<containsImage />              3.6: „Bilder in Dokumenten“ (S. 21)
<containsOneOfTheseImages />  3.6: „Bilder in Dokumenten“ (S. 21)

<hasAnyAction />               3.2: „Aktionen (Actions)“ (S. 11)
<hasAuthor />                  3.8: „Dokumenteneigenschaften“ (S. 26)
<hasBookmark />                3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 46)
<hasBookmarks />              3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 46)
<hasChainedAction />          3.2: „Aktionen (Actions)“ (S. 11)
<hasCloseAction />            3.2: „Aktionen (Actions)“ (S. 11)
<hasCreationDate />           3.7: „Datum“ (S. 24)
<hasCreationDateAfter />      3.7: „Datum“ (S. 24)
<hasCreationDateBefore />    3.7: „Datum“ (S. 24)
<hasCreator />                3.7: „Datum“ (S. 24)
<hasEmbeddedFile />           3.3: „Anhänge (Attachments)“ (S. 17)
<hasEmbeddedFileContent />    3.3: „Anhänge (Attachments)“ (S. 17)
<hasEncryptionLength />      3.18: „Passwort“ (S. 49)
<hasField />                  3.11: „Formularfelder“ (S. 32)
<hasFields />                 3.11: „Formularfelder“ (S. 32)
<hasFont />                   3.19: „Schriften“ (S. 50)
<hasFonts />                  3.19: „Schriften“ (S. 50)
<hasFormat />                 3.10: „Format“ (S. 30)
<hasImportDataAction />      3.2: „Aktionen (Actions)“ (S. 11)
<hasJavaScript />             3.13: „JavaScript“ (S. 40)
<hasJavaScriptAction />      3.2: „Aktionen (Actions)“ (S. 11)
<hasKeywords />              3.8: „Dokumenteneigenschaften“ (S. 26)
<hasLaunchAction />          3.2: „Aktionen (Actions)“ (S. 11)
<hasLayer />                  3.14: „Layer“ (S. 42)
<hasLayers />                 3.14: „Layer“ (S. 42)
<hasLessPages />             3.20: „Seitenzahlen als Testziel“ (S. 54)
<hasLocale />                 3.22: „Sprachinformation (Language)“ (S. 59)
<hasLocalGotoAction />       3.2: „Aktionen (Actions)“ (S. 11)
<hasModificationDate />      3.7: „Datum“ (S. 24)
<hasModificationDateAfter /> 3.7: „Datum“ (S. 24)
<hasModificationDateBefore /> 3.7: „Datum“ (S. 24)
<hasMorePages />             3.20: „Seitenzahlen als Testziel“ (S. 54)
<hasNamedAction />           3.2: „Aktionen (Actions)“ (S. 11)
<hasNamedDestination />      3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 46)

... continued
```

```

... continuation:

<hasNoAuthor />          3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoCreationDate />    3.7: „Datum“ (S. 24)
<hasNoCreator />         3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoKeywords />       3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoLocale />         3.22: „Sprachinformation (Language)“ (S. 59)
<hasNoModificationDate /> 3.7: „Datum“ (S. 24)
<hasNoProducer />       3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoProperty />       3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoSubject />        3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoText />           3.23: „Texte“ (S. 60)
<hasNoTitle />          3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoXFADData />       3.29: „XFA Daten“ (S. 69)
<hasNoXMPData />        3.30: „XMP-Daten“ (S. 72)

<hasNumberOfXXX />      3.4: „Anzahl verschiedener PDF-Bestandteile“ (S. 19)

<hasOCG />              3.14: „Layer“ (S. 42)
<hasOCGs />             3.14: „Layer“ (S. 42)
<hasOpenAction />       3.2: „Aktionen (Actions)“ (S. 11)
<hasOwnerPassword />    3.18: „Passwort“ (S. 49)
<hasPermission />       3.5: „Berechtigungen“ (S. 20)
<hasPrintAction />      3.2: „Aktionen (Actions)“ (S. 11)
<hasProducer />         3.8: „Dokumenteneigenschaften“ (S. 26)
<hasProperty />         3.8: „Dokumenteneigenschaften“ (S. 26)
<hasRemoteGotoActionTo /> 3.2: „Aktionen (Actions)“ (S. 11)
<hasResetFormAction />  3.2: „Aktionen (Actions)“ (S. 11)
<hasSaveAction />       3.2: „Aktionen (Actions)“ (S. 11)
<hasSignature />        3.21: „Signaturen und Zertifikate“ (S. 54)
<hasSignatures />       3.21: „Signaturen und Zertifikate“ (S. 54)
<hasSignedSignatureFields /> 3.21: „Signaturen und Zertifikate“ (S. 54)
<hasSubject />          3.8: „Dokumenteneigenschaften“ (S. 26)
<hasSubmitFormAction /> 3.2: „Aktionen (Actions)“ (S. 11)
<hasText />             3.23: „Texte“ (S. 60)
<hasTitle />            3.8: „Dokumenteneigenschaften“ (S. 26)
<hasTrappingInfo />     3.27: „Trapping-Info“ (S. 67)
<hasUnsignedSignatureFields /> 3.21: „Signaturen und Zertifikate“ (S. 54)
<hasURIAction />        3.2: „Aktionen (Actions)“ (S. 11)
<hasUserPassword />     3.18: „Passwort“ (S. 49)
<hasVersion />          3.28: „Version“ (S. 68)
<hasXFADData />         3.29: „XFA Daten“ (S. 69)
<hasXMPData />          3.30: „XMP-Daten“ (S. 72)

<haveSame... />        4.1: „Überblick“ (S. 76)

<isCertified />         3.31: „Zertifiziertes PDF“ (S. 74)
<isLinearizedForFastWebView /> 3.9: „Fast Web View“ (S. 29)
<isSigned />            3.21: „Signaturen und Zertifikate“ (S. 54)
<isTagged />            3.26: „Tagging“ (S. 66)

... (end of list)

```

PDFUnit wird ständig weiterentwickelt und die Dokumentation aktuell gehalten. Sollten Sie Tests vermissen, schicken Sie Ihre Wünsche und Vorschläge an [request\[at\]pdfunit.com](mailto:request[at]pdfunit.com).

3.2. Aktionen (Actions)

Überblick

PDF-Dokumente werden durch Aktionen lebendig, interaktiv aber auch komplizierter. Und „kompliziert“ bedeutet, dass sie getestet werden müssen, zumal wenn interaktive Dokumente Teil einer Prozesskette sind, in der Aktionen die Teile sind, die richtig funktionieren müssen.

Eine „Aktion“ ist ein Dictionary-Objekt mit u.a. 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 Tags zur Verfügung:

```
<!-- Tags to test actions: -->
```

```
<hasNumberOfActions />
<hasNumberOfJavaScriptActions />

<hasAnyAction>
  <containing />           (all nested tags ...
  <matchingComplete />    ...
  <matchingRegex />       ... are optional)
</hasAnyAction>

... continued
```

```
... continuation:
```

```
<hasChainedAction>
  <containing />           (all nested tags ...
  <matchingComplete />    ...
  <matchingRegex />       ... are optional)
</hasChainedAction>

<hasCloseAction>
  <containing />           (all nested tags ...
  <matchingComplete />    ...
  <matchingRegex />       ... are optional)
</hasCloseAction>

<hasImportDataAction>
  <matchingComplete filename=".." /> (tag optional, attribute required)
</hasImportDataAction>

<hasJavaScriptAction>
  <containing />           (all nested tags ...
  <matchingComplete />    ...
  <matchingRegex />       ... are optional)
</hasJavaScriptAction>

<hasLaunchAction>
  <toLaunch />             (optional)
</hasLaunchAction>

... continue
```

```

... continuation

<hasLocalGotoAction>
  <toDestination /> (optional)
</hasLocalGotoAction>

<hasNamedAction>
  <withName /> (optional)
</hasNamedAction>

<hasOpenAction>
  <containing /> (all nested tags ...
  <matchingComplete /> ...
  <matchingRegex /> ...
  <withDestinationTo /> ... are optional)
</hasOpenAction>

<hasPrintAction>
  <containing /> (all nested tags ...
  <matchingComplete /> ...
  <matchingRegex /> ... are optional)
</hasPrintAction>

... continued

```

```

... continuation:

<hasRemoteGotoActionTo file=".." (required)
                      destination=".." (optional)
                      page=".." (optional)
/>

<hasResetFormAction /> (no attributes and no nested tags!)

<hasSaveAction>
  <containing /> (all nested tags ...
  <matchingComplete /> ...
  <matchingRegex /> ... are optional)
</hasSaveAction>

<hasSubmitFormAction>
  <withDestination /> (optional)
</hasSubmitFormAction>

<hasURIAction>
  <containing /> (all nested tags ...
  <matchingComplete /> ...
  <matchingRegex /> ... are optional)
</hasURIAction>

... (end of list)

```

Die folgenden Abschnitte zeigen Beispiele für verschiedene Aktionen.

Close-Actions

Close-Actions werden beim Schließen eines PDF-Dokumentes ausgeführt:

```

<!-- The fundamental way to compare actions with expected values: -->

<testcase name="Principle_ComparingActionValues">
  <assertThat testDocument="actions/documentCloseAction.pdf">
    <hasCloseAction>
      <containing>app.alert('A sample for a DOCUMENT_CLOSE-action');</containing>
    </hasCloseAction>
  </assertThat>
</testcase>

```

Der Inhalt einer Close-Action kann auch mit dem Inhalt einer Datei verglichen werden:

```

<testcase name="hasCloseAction_MatchingComplete_ContentFromFile">
  <assertThat testDocument="actions/documentCloseAction.pdf">
    <hasCloseAction>
      <matchingComplete filename="actions/documentCloseAction.js"
                        whitespaces="IGNORE"
      />
    </hasCloseAction>
  </assertThat>
</testcase>

```

Das Attribut `whitespaces=".."` ist optional. Die Standard-Whitespace-Behandlung ist `NORMALIZE`.

ImportData-Actions

ImportData-Actions importieren Daten aus einer Datei. Sie benötigen den Dateinamen als Zusatzinformation:

```
<testcase name="hasImportDataAction_MatchingFilename">
  <assertThat testDocument="actions/chainedActions.pdf">
    <hasImportDataAction>
      <matchingComplete filename="build.xml" />
    </hasImportDataAction>
  </assertThat>
</testcase>
```

Es wird nur geprüft, ob die Aktion den erwarteten Dateinamen enthält. Ob die Datei selber existiert, wird nicht überprüft.

JavaScript-Actions

Da JavaScript-Text gewöhnlich etwas umfangreicher ist, macht es Sinn, den Vergleichstext für eine JavaScript-Action aus einer Datei zu lesen:

```
<testcase name="hasJavaScriptAction_MatchingComplete_ContentFromFile">
  <assertThat testDocument="javascript/bookmarkWithJavaScriptAction_OneSimpleAlert.pdf">
    <hasJavaScriptAction>
      <matchingComplete filename="javascript/javascriptAction_OneSimpleAlert.js" />
    </hasJavaScriptAction>
  </assertThat>
</testcase>
```

Der vollständige Inhalt der JavaScript-Datei wird mit dem Inhalt der JavaScript-Action verglichen. Whitespaces werden dabei normalisiert.

Launch-Actions

Launch-Actions starten Anwendungen oder Skripte. Das kann getestet werden:

```
<testcase name="hasLaunchAction_Notepad_Print">
  <assertThat testDocument="actions/launchActionToFile.pdf">
    <hasLaunchAction>
      <toLaunch application="c:/windows/notepad.exe" operation="print" />
    </hasLaunchAction>
  </assertThat>
</testcase>
```

Während des Tests werden lediglich die Inhalte der Attribute `application=".."` und `operation=".."` mit denen der Launch-Action des PDF-Dokumentes verglichen. Es wird nicht geprüft, ob die Applikation gestartet werden kann.

Named-Actions

Named-Actions sollten auf ihren Operationsnamen hin überprüft werden:

```
<testcase name="hasNamedAction_WithName_NextPage">
  <assertThat testDocument="actions/namedActionsNextPages.pdf">
    <hasNamedAction>
      <withName>
        <matchingComplete>/NextPage</matchingComplete>
      </withName>
    </hasNamedAction>
  </assertThat>
</testcase>
```

Goto-Actions

Goto-Actions benötigen ein Sprungziel in derselben Datei:

```
<testcase name="hasGotoAction_ToNamedDestination">
  <assertThat testDocument="actions/bookmarksWithPdfOutline.pdf">
    <hasLocalGotoAction>
      <toDestination name="destination2.1" />
    </hasLocalGotoAction>
  </assertThat>
</testcase>
```

Der Test ist erfolgreich, wenn das aktuelle Test-PDF das erwartete Sprungziel „destination2.1“ enthält.

GotoRemote-Actions

GotoRemote-Actions benötigen ein Sprungziel in einer anderen Datei:

```
<testcase name="hasGotoRemoteActionTo_NamedDestination">
  <assertThat testDocument="actions/gotoRemotePageAction.pdf">
    <hasRemoteGotoActionTo file="destination.pdf"
                          destination="destination-3"
    />
  </assertThat>
</testcase>

<testcase name="hasGotoRemoteAction_ToPage">
  <assertThat testDocument="actions/gotoRemotePageAction.pdf">
    <hasRemoteGotoActionTo file="destination.pdf"
                          page="4"
    />
  </assertThat>
</testcase>
```

Es wird lediglich überprüft, ob das Test-PDF-Dokument eine Aktion mit diesem Sprungziel besitzt. Es wird nicht geprüft, ob die Zieldatei bzw. das Sprungziel in der Zieldatei existiert.

Open-Actions

Open-Actions werden beim Laden eines PDF-Dokumentes ausgeführt. Häufig sind es JavaScript- oder Goto-Actions.

```
<testcase name="hasOpenAction_MultipleInvocation">
  <assertThat testDocument="actions/documentOpenAction_Print.pdf">
    <hasOpenAction>
      <matchingRegex>( ?ms). *print(. *)</matchingRegex>
      <matchingComplete>this.print(true);</matchingComplete>
    </hasOpenAction>
  </assertThat>
</testcase>
```

Innerhalb des Tags `<hasOpenAction />` können Texte mit den üblichen Textvergleichs-Tags verglichen werden. Darüber hinaus gibt es für Open-Actions noch das Tag `<withDestinationTo />`, mit dem das Ziel der Open-Action geprüft wird:

```
<testcase name="hasOpenAction_GotoPage2">
  <assertThat testDocument="actions/documentOpenAction_Goto.pdf">
    <hasOpenAction>
      <withDestinationTo page="2" />
    </hasOpenAction>
  </assertThat>
</testcase>
```

Print-Actions

Print-Actions sind JavaScript-Actions, die direkt vor oder direkt nach dem Drucken ausgeführt werden. Sie sind innerhalb von PDF mit den Events `WILL_PRINT` oder `DID_PRINT` verbunden.

```
<testcase name="hasPrintAction_WillPrint">
  <assertThat testDocument="actions/documentPrintActions.pdf">
    <hasPrintAction>
      <matchingComplete>app.alert('A sample for a WILL_PRINT-action');</matchingComplete>
    </hasPrintAction>
  </assertThat>
</testcase>
```

Analog zu JavaScript-Actions wird der erwartete Inhalt der Print-Action nach einer Normalisierung der Whitespaces mit der erwarteten Zeichenkette verglichen.

ResetForm-Actions

ResetForm-Actions sind parameterlos, ein Textvergleich findet nicht statt. Es wird nur geprüft, ob eine solche Action existiert:

```
<testcase name="hasResetFormAction">
  <assertThat testDocument="acrofields/javaScriptForFields.pdf">
    <hasResetFormAction />
  </assertThat>
</testcase>
```

Save-Actions

Save-Actions sind JavaScript-Actions, die direkt vor oder direkt nach dem Speichern ausgeführt werden. Sie sind mit den PDF-Events WILL_SAVE bzw. DID_SAVE verknüpft.

```
<testcase name="hasSaveAction_MultipleInvocation">
  <assertThat testDocument="actions/documentSaveActions.pdf">
    <hasSaveAction>
      <matchingComplete>app.alert('A sample for a DID_SAVE-action');</matchingComplete>
    </hasSaveAction>
  </assertThat>
</testcase>
```

Auch hier wird der Vergleich erst nach einer Normalisierung der Whitespaces durchgeführt. Alle üblichen Tags zum Vergleichen von Texten stehen zur Verfügung.

SubmitForm-Actions

SubmitForm-Actions benötigen ein Ziel, an das das Formular geschickt werden soll:

```
<testcase name="hasSubmitFormAction_ToUri">
  <assertThat testDocument="acrofields/javaScriptForFields.pdf">
    <hasSubmitFormAction>
      <withDestination toURI="http://www.geek-tutorials.com/java/itext/submit.php" />
    </hasSubmitFormAction>
  </assertThat>
</testcase>
```

PDFUnit prüft nicht die Existenz des Ziels, sondern nur die Gleichheit des Sprungziels mit der erwarteten Zeichenkette.

URI-Actions

URI-Actions benötigen eine Ziel-URI:

```
<testcase name="hasURIAction">
  <assertThat testDocument="actions/noBookmarks-manyActions.pdf">
    <hasURIAction>
      <matchingComplete>http://www.imdb.com/</matchingComplete>
    </hasURIAction>
  </assertThat>
</testcase>
```

Es findet kein Zugriff auf das Web statt, somit überprüft PDFUnit auch nicht, ob die URI existiert. Es wird nur geprüft, ob das Test-PDF eine URI-Aktion mit diesem Namen enthält.

Any Action - beliebige Aktionen

Das folgende Beispiel zeigt mehrere Testfunktionen, angewendet auf ein Test-PDF, das die verschiedenen Aktionen auch enthält:


```
<testcase name="hasAnyAction_DifferentKindOfActions">
  <assertThat testDocument="actions/chainedActions.pdf">
    <hasAnyAction>
      <matchingComplete>app.alert('Demo: the first action of five.');
```

Whitespace-Behandlung in Vergleichen

Bei Textvergleichen kann die Behandlung der Whitespaces durch den Test gesteuert werden. Im folgenden Beispiel werden Zeilenumbrüche und Leerzeilen ignoriert:

```
<testcase name="hasCloseAction_Containing_ContentFromReader">
  <assertThat testDocument="actions/documentCloseAction.pdf">
    <hasCloseAction>
      <containing filename="actions/documentCloseAction.js" whitespaces="IGNORE" />
    </hasCloseAction>
  </assertThat>
</testcase>
```

Das Kapitel 13.4: „Behandlung von Whitespaces“ (S. 138) geht ausführlich auf den flexiblen Umgang mit Whitespaces ein.

3.3. Anhänge (Attachments)

Überblick

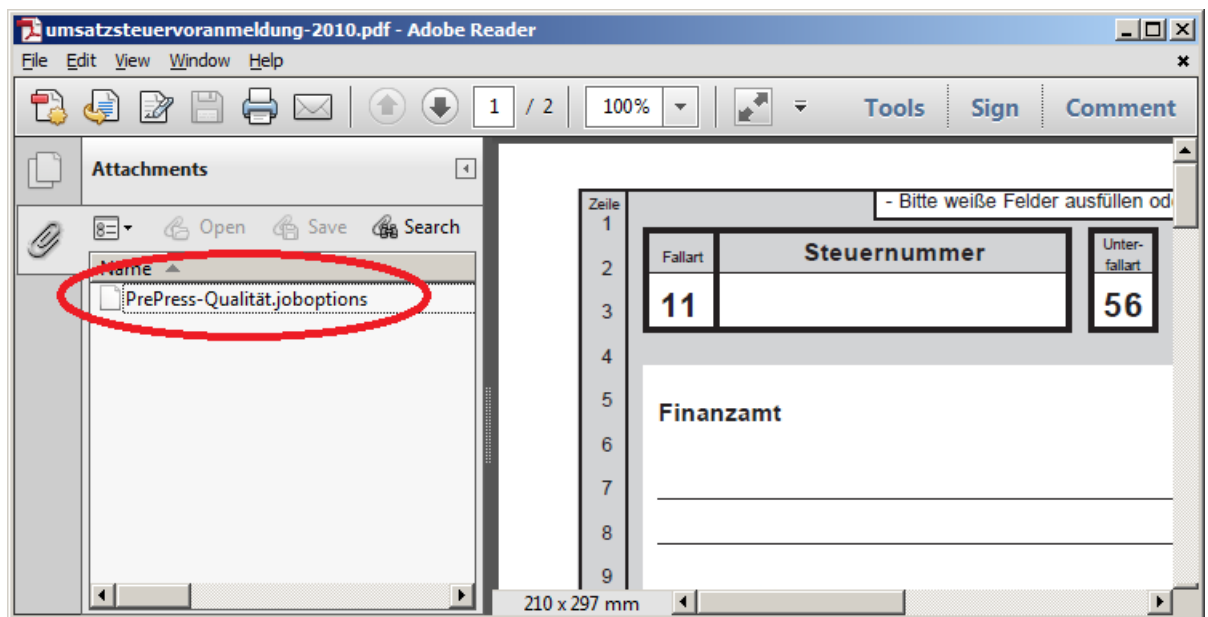
Dateien, die als Attachments in PDF-Dokumenten enthalten sind, spielen in nachverarbeitenden Prozessen meist eine wichtige Rolle. Deshalb stellt PDFUnit Tests für Attachments (eingebettete Dateien) bereit:

```
<!-- Tags to test embedded files: -->

<hasNumberOfEmbeddedFiles />
<hasEmbeddedFile name=".." (one of the two
                      content=".." attributes is required)
/>
<hasEmbeddedFileContent />
```

Existenz

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



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

```
<testcase name="hasEmbeddedFile">
  <assertThat testDocument="embeddedfiles/umsatzsteuervoranmeldung-2010.pdf">
    <hasEmbeddedFile />
  </assertThat>
</testcase>
```

Anzahl

Etwas weiter zielt die Prüfung der Anzahl der eingebetteten Dateien:

```
<testcase name="hasNumberOfEmbeddedFiles">
  <assertThat testDocument="embeddedfiles/umsatzsteuervoranmeldung-2010.pdf">
    <hasNumberOfEmbeddedFiles>1</hasNumberOfEmbeddedFiles>
  </assertThat>
</testcase>
```

Dateiname

Danach kommen die Namen der Dateien:

```
<testcase name="hasEmbeddedFile_WithName">
  <assertThat testDocument="embeddedfiles/umsatzsteuervoranmeldung-2010.pdf">
    <hasEmbeddedFile name="PrePress-Qualität.joboptions" />
  </assertThat>
</testcase>
```

Inhalt

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

```
<testcase name="hasEmbeddedFile_WithContent">
  <assertThat testDocument="embeddedfiles/umsatzsteuervoranmeldung-2010.pdf">
    <hasEmbeddedFile content="embeddedfiles/PrePress-Qualität.joboptions" />
  </assertThat>
</testcase>
```

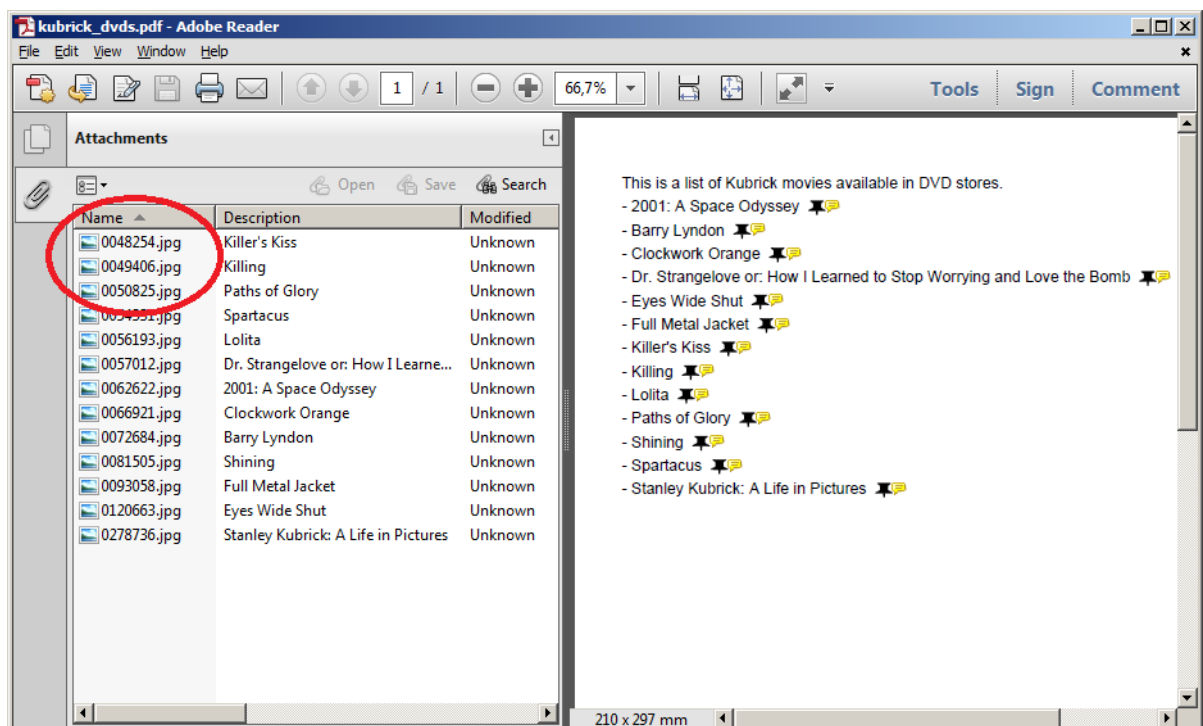
Der Vergleich erfolgt byte-weise.

Wenn die eingebetteten Dateien nicht als separate Datei vorliegen, können sie mit dem Hilfsprogramm „ExtractEmbeddedFiles“ aus einem bestehenden PDF-Dokument (Master-PDF) extrahiert werden. Das Programm wird in Kapitel 9.2: „Anhänge extrahieren“ (S. 105) genauer beschrieben.

Ein Test kann auch gleichzeitig mehrere Dateinamen überprüfen:

```
<testcase name="hasEmbeddedFile_MultipleInvocation">
  <assertThat testDocument="embeddedfiles/kubrick_dvds.pdf">
    <hasEmbeddedFile name="0048254.jpg" />
    <hasEmbeddedFile name="0049406.jpg" />
    <hasEmbeddedFile name="0050825.jpg" />
  </assertThat>
</testcase>
```

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



3.4. Anzahl verschiedener PDF-Bestandteile

Überblick

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

```
<!-- Tags to count parts of a PDF: -->
<hasNumberOfActions />
<hasNumberOfBookmarks />
<hasNumberOfDifferentImages /> ❶
<hasNumberOfEmbeddedFiles />
<hasNumberOfFields />
<hasNumberOfFonts identifiedBy=".." (required) />
<hasNumberOfJavaScriptActions />
<hasNumberOfLayers />
<hasNumberOfOCGs />
<hasNumberOfPages /> ❷
<hasNumberOfSignatures />
<hasNumberOfVisibleImages /> ❸
```

- ❸ Prüfungen auf die Anzahl von Bildern werden in Kapitel 3.6: „Bilder in Dokumenten“ (S. 21) beschrieben.
- ❷ Prüfungen auf die Anzahl von Seiten eines PDF-Dokumentes werden in Kapitel 3.20: „Seitenzahlen als Testziel“ (S. 54) beschrieben.

Beispiele

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

```
<testcase name="hasNumberOfFields">
  <assertThat testDocument="acrofields/simpleRegistrationForm.pdf">
    <hasNumberOfFields>4</hasNumberOfFields>
  </assertThat>
</testcase>
```

```
<testcase name="hasNumberOfBookmarks">
  <assertThat testDocument="bookmarks/manyBookmarks.pdf">
    <hasNumberOfBookmarks>19</hasNumberOfBookmarks>
  </assertThat>
</testcase>
```

Alle Prüfungen können kombiniert werden:

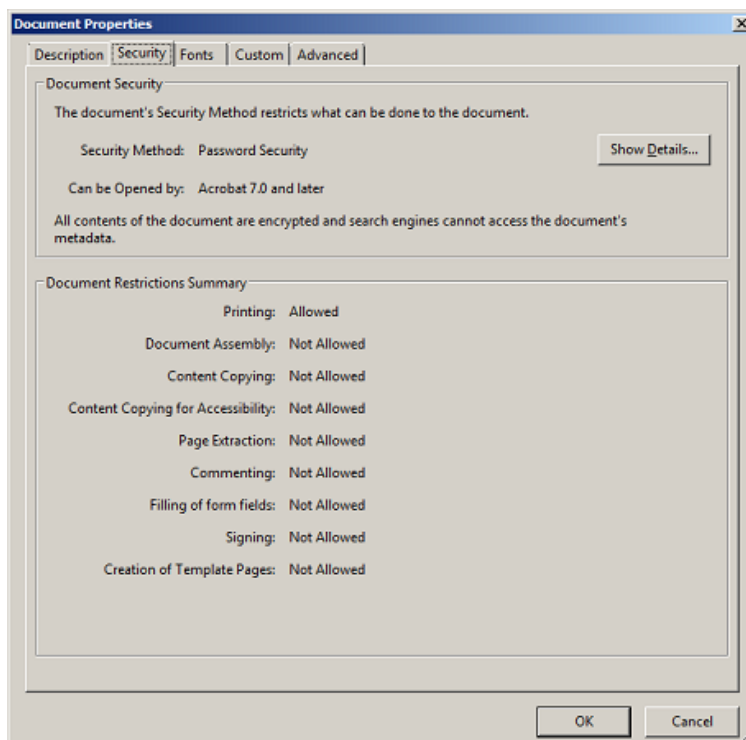
```
<testcase name="testHugeDocument_MultipleInvocation">
  <assertThat testDocument="performance/groovy_wiki-snapshot_1370.pdf">
    <hasNumberOfPages>1370</hasNumberOfPages>
    <hasNumberOfBookmarks>565</hasNumberOfBookmarks>
    <hasNumberOfActions>1896</hasNumberOfActions>
    <hasNumberOfEmbeddedFiles>0</hasNumberOfEmbeddedFiles>
  </assertThat>
</testcase>
```

Vorsicht, dieser Test mit einem Dokument von 1370 Seiten dauert ca. 10 Sekunden auf einem zeitgemäß ausgerüsteten Entwickler-Notebook. Trennen Sie die langsam laufenden Tests von den schnellen und starten Sie sie über 2 Skripte oder 2 ANT-Targets.

3.5. 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 Berechtigungen mit dem Tag `<hasPermission />` überprüfen. Das Tag hat für jede Berechtigung ein passendes Attribut:

```
<!-- Tag to test permissions: -->

<hasPermission toAllowScreenReaders=".." (one of this attributes ...
               toAssembleDocument=".." ...
               toCopyContent=".." ...
               toExtractContent=".." ...
               toFillInFields=".." ...
               toModifyAnnotations=".." ...
               toModifyContent=".." ...
               toPrint=".." ...
               toPrintInDegradedQuality=".." ... has to be used)
/>

<!-- The attributes can have the values 'YES' and 'NO' -->
```

Die Berechtigungen „copyContent“ und „extractContent“ sind funktional identisch. Sie stehen redundant zur Verfügung, um unterschiedliche sprachliche Vorlieben von Entwicklern zu bedienen.

Beispiel

```
<testcase name="hasPermission_MultiplePermissions">
  <assertThat testDocument="permissions/itext-permission_default-without-password.pdf">
    <hasPermission toPrint="YES"
                  toExtractContent="YES"
                  toModifyContent="NO"
    />
  </assertThat>
</testcase>
```

3.6. Bilder in Dokumenten

Überblick

Ein veraltetes Bild in einem Dokument kommt beim Empfänger etwa so gut an, wie eine veraltete Neujahrsansprache (Rede des deutschen Bundeskanzlers in der ARD 1986) - auf jeden Fall anders,

als sich die Designer des Dokumentes das gedacht haben. Deshalb sollten Sie überprüfen, ob das **neue** Unternehmens-Logo auch wirklich dargestellt wird.

Eine weitere Fehlerquelle im Zusammenhang mit Bildern ist, dass ein Bild während des PDF-Erstellungsprozesses nicht gefunden wird und im PDF einfach fehlt. Sichern Sie auch diese Situation durch einen passenden Test ab.

Und als Letztes sei noch der Fehler genannt, dass Bilder auf falschen Seiten auftauchen, weil der Seitenumbruch anders als erwartet funktioniert.

Allen Fehlern kann mit diesen Tags begegnet werden:

```
<!-- Tags for image tests: -->
<hasNumberOfDifferentImages />
<hasNumberOfVisibleImages />
<containsImage file=".." (required)

    on=".." (one of the page selection attributes)
    onPage=".." ...
    onEveryPageAfter=".." ...
    onEveryPageBefore=".." ...
    onAnyPageAfter=".." ...
    onAnyPageBefore=".." ... is required)
/>
```

Eine wichtige Bemerkung vorweg: Die Anzahl der sichtbaren Bilder entspricht üblicherweise 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 Tags. Das Tag `<hasNumberOfDifferentImages />` überprüft die Anzahl der **internen** Bilder, während das Tag `<hasNumberOfVisibleImages />` die Anzahl der **sichtbaren** Bilder überprüft.

Anzahl unterschiedlicher Bilder im Dokument

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

```
<testcase name="hasNumberOfDifferentImages">
  <assertThat testDocument="images/imageDemo.pdf">
    <hasNumberOfDifferentImages>2</hasNumberOfDifferentImages>
  </assertThat>
</testcase>
```

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. 107).

Anzahl sichtbarer Bilder im Dokument

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

```
<testcase name="hasNumberOfVisibleImages">
  <assertThat testDocument="images/imageDemo.pdf">
    <hasNumberOfVisibleImages>6</hasNumberOfVisibleImages>
  </assertThat>
</testcase>
```

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

Der Test auf die sichtbaren Bilder kann auf spezifizierte Seiten beschränkt werden. So werden im nächsten Beispiel nur die Bilder auf Seite 3 überprüft:

```
<testcase name="hasNumberOfVisibleImages_OnPage3">
  <assertThat testDocument="images/imageDemo.pdf">
    <hasNumberOfVisibleImages onPage="3">2</hasNumberOfVisibleImages>
  </assertThat>
</testcase>
```

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

Die Möglichkeiten, Tests auf bestimmte Seiten zu beschränken, werden in Kapitel 13.2: „Seitenauswahl“ (S. 136) 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:

```
<testcase name="containsImage">
  <assertThat testDocument="images/imageDemo.pdf">
    <containsImage file="images/apache-software-foundation-logo.png"
                  on="ANY_PAGE"
    />
  </assertThat>
</testcase>
```

Das Ergebnis eines Vergleiches zweier Bilddateien hängt von den Dateiformaten ab. PDFUnit kann JPEG, PNG, GIF, BMP und WBMP verarbeiten. Die Bilder werden von PDFUnit byte-weise verglichen. Deshalb werden die BMP- und PNG-Versionen eines Bildes nicht als gleich erkannt.

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 diese Problem geben, extrahieren Sie das gewünschte Bild als PNG. Gehen Sie dafür folgendermaßen vor:

- Extrahieren Sie alle Bilder mit dem Hilfsprogramm `ExtractImages`. Alle Bilder werden als PNG abgespeichert.
- Überprüfen Sie das gewünschte Bild auf seine Korrektheit.
- Benutzen Sie es im Test, wie im obigen Listing dargestellt.

Eine Menge von Bildern als Vergleichsvorlage

Es kann die Situation geben, dass ein PDF-Dokument eines von drei möglichen Logos enthält. Oder die Unterschrift ist eine aus einer Menge von fünf erlaubten. Für diese Situation gibt es das Tag `<containsOneOfTheseImages />`:

```
<testcase name="containsOneOfManyImages_alex">
  <assertThat testDocument="images/letter-signed-by-alex.pdf">
    <containsOneOfTheseImages on="LAST_PAGE">
      <image file="images/signature-alex.png" />
      <image file="images/signature-bob.png" />
    </containsOneOfTheseImages>
  </assertThat>
</testcase>
```

Dieser 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 gleichen Möglichkeiten zur Verfügung, wie in Kapitel 13.2: „Seitenauswahl“ (S. 136) beschrieben.

Hier ein paar Beispiele:

```
<testcase name="containsImage_OnEveryPageAfter4">
  <assertThat testDocument="images/imageDemo.pdf">
    <containsImage file="images/apache-software-foundation-logo.png"
      onEveryPageAfter="4"
    />
  </assertThat>
</testcase>
```

```
<testcase name="containsImage_OnMultipleSelectedPages">
  <assertThat testDocument="images/imageDemo.pdf">
    <containsImage file="images/apache-software-foundation-logo.png"
      onPage="1, 5" />
  </assertThat>
</testcase>
```

In einem Test kann die Prüfung auf verschiedene Bilder ausgeführt werden. Überlegen Sie aber, ob es nicht sinnvoller ist, für das folgende Beispiel zwei getrennte Tests zu schreiben:

```
<testcase name="containsImage_MultipleInvocation">
  <assertThat testDocument="images/imageDemo.pdf">
    <containsImage file="images/apache-software-foundation-logo.png"
      onEveryPageAfter="4"
    />
    <containsImage file="images/apache-ant-logo.png"
      onPage="3"
    />
  </assertThat>
</testcase>
```

Die in einem PDF enthaltenen Bilder können auch gegen die Bilder in einem Master-PDF getestet werden. Eine genaue Beschreibung dazu enthält das Kapitel 4.5: „Bilder vergleichen“ (S. 80).

3.7. 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 Testsszenarien anzubieten.

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

```
<!-- Tags to test date values: -->

<hasCreationDate />
<hasCreationDateAfter />
<hasCreationDateBefore />

<hasModificationDate />
<hasModificationDateAfter />
<hasModificationDateBefore />

<hasNoCreationDate />
<hasNoModificationDate />
```

Existenz eines Datums

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

```
<testcase name="hasCreationDate">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasCreationDate />
  </assertThat>
</testcase>
```

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


```
<testcase name="hasCreationDate_NoDateInPDF">
  <assertThat testDocument="documentInfo/documentInfo_noDateFields.pdf">
    <hasNoCreationDate />
  </assertThat>
</testcase>
```

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

Datumsauflösung

Sie müssen festlegen, welche Datumsbestandteile für den Test relevant sind. Dafür bietet das Attribut `resolution` die zwei Konstanten `resolution="DATE"` und `resolution="DATETIME"` an. Mit der Konstanten `DATE` werden Tag, Monat und Jahr verglichen und mit der Konstanten `DATETIME` zusätzlich noch Stunde, Minute und Sekunde.

Tests sehen dann so aus:

```
<testcase name="hasCreationDate_DateResolution">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasCreationDate withDate="2013-05-05" resolution="DATE" />
  </assertThat>
</testcase>
```

```
<testcase name="hasCreationDate_DateTimeResolution">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasCreationDate withDate="2013-05-05T09:33:47" resolution="DATETIME" />
  </assertThat>
</testcase>
```

Das Format des erwarteten Datums ist durch die XML Schema Definition festgelegt (`xs:date`). Da aber das PDF-interne Datum unterschiedlich formatiert sein kann, muss dessen Format-String durch einen Eintrag in der Konfigurationsdatei „`config.properties`“ definiert werden.

Wenn Sie das Attribut `resolution=".."` weglassen, wird `resolution="DATE"` angenommen.

Konfiguration des Datumsformates in der `config.properties`

Das Datumsformat innerhalb von PDF-Dokumenten variiert stark, abhängig vom jeweiligen PDF-Erstellungswerkzeug. Wenn Sie innerhalb eines Projektes nur ein Werkzeug zur PDF-Erstellung nutzen, sehen Sie diese Varianz nicht. Da PDFUnit das Datumsformat Ihres Werkzeuges aber nicht vorausahnen kann, müssen Sie es in der Datei `config.properties` definieren. Dabei gelten die Regeln der Klasse `java.util.SimpleDateFormat`.

```
#####
# Declaring the default format for dates in PDF documents.
# Use the format strings according to java.util.SimpleDateFormat.
#####
# Using date only:
#dateformat = 'D:'yyyyMMdd
# Using date and time:
dateformat = 'D:'yyyyMMddHHmmss
```

Vorsicht: Wenn Sie hier ein Format `dateformat = 'D:'yyyy` angeben, werden Monat und Tag mit dem Wert `0101` ergänzt. Das dürfte in den seltensten Fällen so gewollt sein.

Sie können nur **ein** Format in der Konfigurationsdatei festlegen. Wenn Sie in einem Projekt PDF-Dokumente mit abweichendem Datumsformat testen möchten, können Sie den Weg über Properties nutzen und das formatierte Datum als Wert abfragen:

```
<testcase name="hasProperty_CreationDate">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasProperty name="CreationDate">
      <matchingComplete>D:20131027172417+01'00'</matchingComplete>
    </hasProperty>
    <hasProperty name="CreationDate">
      <startingWith>D:20131027</startingWith>
    </hasProperty>
  </assertThat>
</testcase>
```

Datumstest mit Ober- und Untergrenze

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

```
<testcase name="hasCreationDate_Before">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasCreationDateBefore withDate="2099-01-01" resolution="DATE" />
  </assertThat>
</testcase>
```

```
<testcase name="hasCreationDate_After">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasCreationDateAfter withDate="1999-01-01" resolution="DATE" />
  </assertThat>
</testcase>
```

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

Ausstellungsdatum eines benutzten Zertifikates

PDF-Dokumente enthalten neben dem Erstellungs- und Änderungsdatum noch das Ausstellungsdatum von Zertifikaten. Test dazu sind in Kapitel 3.21: „Signaturen und Zertifikate“ (S. 54) beschrieben.

3.8. 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 Realität 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 semantikkfreie Zahlenfolge ist, ist der Nutzen dieses Dokumentes nur marginal größer, als wenn es nicht existierte.

Folgende Tags stehen zur Validierung der Metadaten zur Verfügung:

```
<!-- Tags to test document properties: -->

<hasAuthor />
<hasCreator />
<hasKeywords />
<hasProducer />
<hasProperty />
<hasSubject />
<hasTitle />

<hasNoAuthor />
<hasNoCreator />
<hasNoKeywords />
<hasNoProducer />
<hasNoProperty />
<hasNoSubject />
<hasNoTitle />

<hasCreationDate />
<hasCreationDateAfter />
<hasCreationDateBefore />
<hasModificationDate />
<hasModificationDateAfter />
<hasModificationDateBefore />
<hasNoCreationDate />
<hasNoModificationDate />
```

Metadaten eines Test-Dokumentes können auch mit den Metadaten eines anderen PDF-Dokumentes verglichen werden. Solche Vergleiche sind in Kapitel 4.7: „Dokumenteneigenschaften vergleichen“ (S. 81) beschrieben.

Autor validieren ...

Sie können den Autor eines Dokumentes manuell im 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:

```
<testcase name="hasAuthor">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasAuthor />
  </assertThat>
</testcase>
```

Um explizit zu prüfen, dass die Dokumenteneigenschaft Autor **nicht vorhanden**, muss das Tag `<hasNoAuthor />` verwendet werden:

```
<testcase name="hasNoAuthor">
  <assertThat testDocument="documentInfo_noAuthorTitleSubjectKeywordsApplication.pdf">
    <hasNoAuthor />
  </assertThat>
</testcase>
```

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

```
<testcase name="hasAuthor_matchingComplete">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasAuthor>
      <matchingComplete>PDFUnit.com</matchingComplete>
    </hasAuthor>
  </assertThat>
</testcase>
```

Verschiedene Tags zum Vergleichen von Texten stehen zur Verfügung. Deren Namen sind selbsterklärend:

```
<!-- Comparing text for author, creator, keywords, producer, subject, title: -->
<containing      />
<endsWith        />
<matchingComplete />
<matchingRegex   />
<notContaining    />
<notMatchingRegex />
<startingWith     />
```

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

Alle Vergleiche arbeiten case-sensitiv.

Die Umsetzung des Tags `<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.

Für jede Dokumenteneigenschaft gibt es die Tags `<hasXXX />` und `<hasNoXXX />`.

Die Tags zum inhaltlichen Vergleich einer Dokumenteneigenschaft können auch gleichzeitig verwendet werden:

```
<!-- Multiple string comparisons are possible -->
<testcase name="hasKeywords_allTextComparingTags">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasKeywords>
      <notContaining>--</notContaining>
    </hasKeywords>
    <hasKeywords>
      <matchingRegex>.*key.*</matchingRegex>
    </hasKeywords>
    <hasKeywords>
      <startingWith>PDFUnit</startingWith>
    </hasKeywords>
  </assertThat>
</testcase>
```

Diese Art Test ist aber nicht empfehlenswert, weil der Name des Tests nicht spezifisch genug ausgedrückt werden kann.

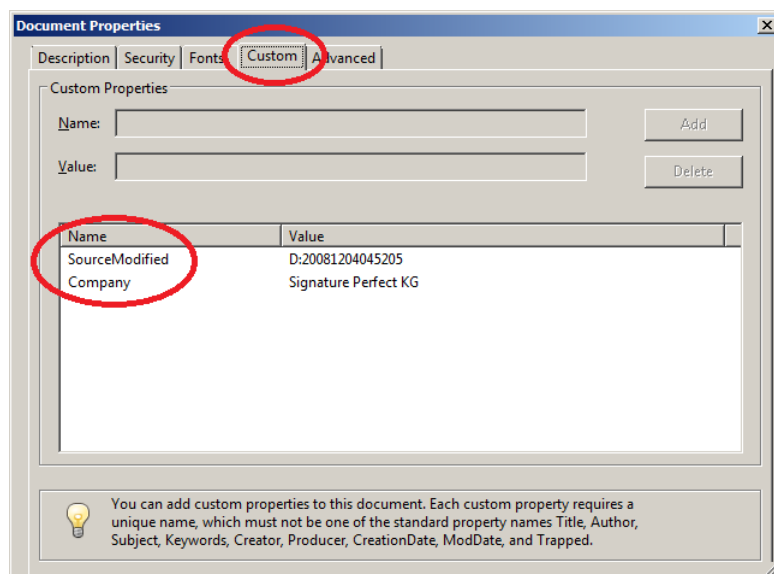
Allgemeine Prüfung als Key-Value-Paar

Die in den vorhergehenden Abschnitten gezeigten Prüfungen auf Standardeigenschaften können alle auch mit dem allgemeinen Tag `<hasProperty />` ausgeführt werden:

```
<testcase name="hasProperty_StandardProperties">
  <assertThat testDocument="customproperties/Leitfaden_Elektronische_Signatur.pdf">
    <hasProperty name="Title">
      <matchingComplete>PDFUnit sample - Demo for Document Infos</matchingComplete>
    </hasProperty>
    <hasProperty name="Subject">
      <matchingComplete>Demo for Document Infos</matchingComplete>
    </hasProperty>
    <hasProperty name="CreationDate">
      <matchingComplete>D:20131027172417+01'00'</matchingComplete>
    </hasProperty>
    <hasProperty name="ModDate">
      <matchingComplete>D:20131027172417+01'00'</matchingComplete>
    </hasProperty>
  </assertThat>
</testcase>
```

`<hasProperty />` prüft eine beliebige Dokumenteneigenschaft als Key/Value-Paar.

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 dann so aus:

```
<testcase name="hasProperty_CustomProperties">
  <assertThat testDocument="customproperties/Leitfaden_Elektronische_Signatur.pdf">
    <hasProperty name="Company">
      <matchingComplete>Signature Perfect KG</matchingComplete>
    </hasProperty>
    <hasProperty name="SourceModified">
      <matchingComplete>D:20081204045205</matchingComplete>
    </hasProperty>
  </assertThat>
</testcase>
```

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

```
<testcase name="hasNoProperty">
  <assertThat testDocument="customproperties/Leitfaden_Elektronische_Signatur.pdf">
    <hasNoProperty name="OldProperty_ShouldNotExist" />
  </assertThat>
</testcase>
```

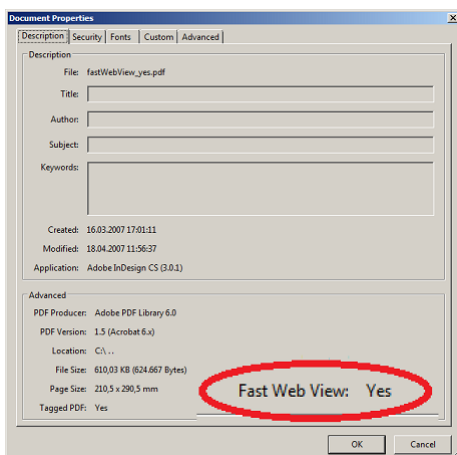
Ab PDF-1.4 existiert die Möglichkeit, Metadaten intern als XML zu speichern (Extensible Metadata Platform, XMP). Das Kapitel 3.30: „XMP-Daten“ (S. 72) geht darauf ausführlich ein.

3.9. 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.

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



PDFUnit prüft, ob ein PDF-Objekt (Dictionary) mit dem Schlüssel `/Linearized` mit dem Wert `1` existiert. Wenn zusätzlich die in diesem Dictionary gespeicherte Dateilänge mit der tatsächlichen Dateilänge übereinstimmt, liefert der Test einen „grünen Balken“:

```
<!-- Tag to verify fast web view: -->
<isLinearizedForFastWebView />
```

Beispiel

```
<testcase name="isLinearizedForFastWebView">
  <assertThat testDocument="fastWebView/fastWebView_yes.pdf">
    <isLinearizedForFastWebView />
  </assertThat>
</testcase>
```

3.10. 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. Es geht zwar, aber das Schriftbild sieht nicht mehr so gut aus, wie erwartet.

Deshalb steht das Tag `<hasFormat />` für Formattests zur Verfügung:

```
<!-- Tag to check the format: -->

<hasFormat format=".."          (either 'format' or
  width=".."                  'width' and 'height' has to be used)
  height=".."
  unit=".."                   (optional)

  on=".."                     (one of the page selection attributes
  onPage=".."                  ...
  onEveryPageAfter=".."       ...
  onEveryPageBefore=".."     ...
  onAnyPageAfter=".."         ...
  onAnyPageBefore=".."       ... can be used)

/>
```

Dokumente mit einheitlichem Seitenformat

Seitenformate können mit vordefinierten Konstanten geprüft werden:

```
<testcase name="hasFormat_A4Landscape">
  <assertThat testDocument="format/format_A4-Landscape.pdf">
    <hasFormat format="A4_LANDSCAPE" /> ❶
  </assertThat>
</testcase>
```

```
<testcase name="hasFormat_LetterPortrait">
  <assertThat testDocument="format/format_Letter-Portrait.pdf">
    <hasFormat format="LETTER_PORTRAIT" /> ❷
  </assertThat>
</testcase>
```

❶❷ Konstanten existieren für gängige Formate.

Auch individuelle Papierformate können geprüft werden:

```
<testcase name="hasFormat_FreeFormat_1117x836_mm">
  <assertThat testDocument="format/physical-map-of-the-world-1999_1117x863mm.pdf">
    <hasFormat width="863.8" height="1117.6" unit="MILLIMETER" /> ❸
  </assertThat>
</testcase>
```

```
<testcase name="hasFormat_FreeFormat_10x15_cm">
  <assertThat testDocument="format/format_Individual-10x15-cm.pdf">
    <hasFormat width="10.0" height="15.0" unit="CENTIMETER" /> ❹
  </assertThat>
</testcase>
```

❸❹ Erlaubte Einheiten sind POINTS, MILLIMETER, CENTIMETER, INCH und DPI72. DPI72 und POINTS sind gleichwertig. Die Einheiten werden in dem Attribut `unit=".."` angegeben.

Das Thema der verschiedenen Papierformate und deren Maße in Points, Millimeter und Inches wird bei www.prepressure.com gut dargestellt.

Die durch die DIN-Norm für Papierformate DIN 476 (<http://de.wikipedia.org/wiki/Papierformat>) erlaubte Toleranz der Seitenlängen wird beim Vergleich zwischen erwartetem und tatsächlichem Format berücksichtigt. 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.

Dokumente mit mehreren Formaten

Ein Dokument mit unterschiedlich großen Seiten kann ebenfalls auf seine Formate überprüft werden:

```
<testcase name="hasFormat_DifferentFormatsOnDifferentPages">
  <assertThat testDocument="format/format_multiple-formats-on-individual-pages.pdf">
    <hasFormat format="A4_LANDSCAPE" on="FIRST_PAGE" />
    <hasFormat format="A5_PORTRAIT" onPage="3" />
  </assertThat>
</testcase>
```

Formattests können auf beliebige einzelne Seiten und Seitenbereiche eingeschränkt werden, wie es in Kapitel 13.2: „Seitenauswahl“ (S. 136) beschrieben ist:

```
<testcase name="hasFormat_OnAnyPageBefore">
  <assertThat testDocument="format/format_multiple-formats-on-individual-pages.pdf">
    <hasFormat format="A4_LANDSCAPE" onAnyPageBefore="3" />
  </assertThat>
</testcase>
```

```
<testcase name="hasFormat_OnAllPagesAfter">
  <assertThat testDocument="format/format_multiple-formats-on-individual-pages.pdf">
    <hasFormat format="A5_PORTRAIT" onEveryPageAfter="2" />
  </assertThat>
</testcase>
```

3.11. Formularfelder

Überblick

Wenn Inhalte eines PDF-Dokumentes weiterverarbeitet werden sollen, spielen Formularfelder eine entscheidende Rolle. Diese sollten schon bei der Erstellung des PDF-Dokumentes korrekt erstellt sein. Dafür sind vor allem korrekte und eindeutige Feldnamen wichtig.

Mit dem Hilfsprogramm `ExtractFieldsInfo` können alle Informationen zu Formularfeldern in eine XML-Datei extrahiert und für XML- und XPath-basierte Tests genutzt werden.

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

```
<!-- Tags for tests on fields: -->
```

```
<hasField    withName                (required)

    width=".."                        (optional, ...
    height=".."                      ... but used together)
    unit=".."                        (optional, default = MILLIMETER)

    hasMultipleLines".."            (optional)
    hasSingleLine".."              (optional)
    isEditable".."                  (optional)
    isExportable".."                (optional)
    isHidden".."                    (optional)
    isMultiSelectable".."           (optional)
    isPasswordProtected".."         (optional)
    isReadOnly".."                  (optional)
    isPrintable".."                 (optional)
    isRequired".."                  (optional)
    isSigned".."                    (optional)
    isVisible".."                   (optional)

    withType".."                    (optional)

/>

... continued
```

```
... continuation
```

```
<!-- Nested tags of <hasField /> are described later in this chapter -->
<!-- The constants for the attribute 'withType' and described later in this chapter -->

<hasFields                                />
<hasNumberOfFields                        />
<hasSignedSignatureFields                  />
<hasUnsignedSignatureFields                />

<!-- Nested tags of <hasFields /> are: -->

<allithoutDuplicateNames />
<allWithoutTextOverflow /> ❶
<matchingXPath />
<matchingXML />
```

- ❶ Dieser Test wird in Kapitel 3.12: „Formularfelder, Textüberlauf“ (S. 38) separat beschrieben.

Existenz

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


```
<testcase name="hasFields_NoFieldsAvailable"
  errorExpected="YES"
>
  <assertThat testDocument="acrofields/noAcrofieldDemo.pdf">
    <hasFields />
  </assertThat>
</testcase>
```

Anzahl

Wenn es lediglich wichtig ist, wieviele Formularfelder ein PDF-Dokument enthält, nutzen Sie das Tag `<hasNumberOfFields />`:

```
<testcase name="hasNumberOfFields">
  <assertThat testDocument="acrofields/simpleRegistrationForm.pdf">
    <hasNumberOfFields>4</hasNumberOfFields>
  </assertThat>
</testcase>
```

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

```
<testcase name="hasNumberOfFields_NoFieldsAvailable">
  <assertThat testDocument="acrofields/noAcrofieldDemo.pdf">
    <hasNumberOfFields>0</hasNumberOfFields>
  </assertThat>
</testcase>
```

Feldnamen

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

```
<testcase name="hasField_MultipleInvocation">
  <assertThat testDocument="acrofields/simpleRegistrationForm.pdf">
    <hasField withName="name" />
    <hasField withName="address" />
    <hasField withName="postal_code" />
    <hasField withName="email" />
  </assertThat>
</testcase>
```

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

```
<testcase name="hasFields_AllWithoutDuplicateNames">
  <assertThat testDocument="acrofields/javascriptForFields.pdf">
    <hasFields>
      <allWithoutDuplicateNames />
    </hasFields>
  </assertThat>
</testcase>
```

Inhalte von Feldern

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

```
<testcase name="hasField_WithAnyValue">
  <assertThat testDocument="acrofields/javascriptForFields.pdf">
    <hasField withName="ageField">
      <withAnyValue />
    </hasField>
  </assertThat>
</testcase>
```

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

```

<!-- Tags to check content in fields -->

<containing           />
<endsWith             />
<havingJavaScriptAction />
<matchingComplete     />
<matchingRegex        />
<notContaining        />
<notMatchingRegex     /> (useful, because regular expressions are
                           not designed to find 'Not-Matches')

<startingWith         />
<withAnyValue         />
<withoutTextOverflow  />

```

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

```

<testcase name="hasField_MatchingComplete">
  <assertThat testDocument="acrofields/plugin-pdf_form_maker.pdf">
    <hasField withName="Text 1">
      <matchingComplete>
        Single Line Text
      </matchingComplete>
    </hasField>
  </assertThat>
</testcase>

```

```

<!-- This is a small test to protect fields against SQL injection. -->

<testcase name="hasField_NotContaining_SQLComment">
  <assertThat testDocument="acrofields/plugin-pdf_form_maker.pdf">
    <hasField withName="Text 1">
      <notContaining>--</notContaining>
      <notContaining>/></notContaining>
    </hasField>
  </assertThat>
</testcase>

```

Feldtypen

Formularfelder haben einen bestimmten Typ. Auch wenn die Bedeutung des Typs wohl nicht so groß ist, wie die des Namens, so gibt es trotzdem das Attribut `withType=".."`, um den Typ eines Feldes zu überprüfen:

```

<testcase name="hasFieldWithType_MultipleInvocation">
  <assertThat testDocument="acrofields/plugin-pdf_form_maker.pdf">
    <hasField withName="Text 25"      withType="TEXT"      />
    <hasField withName="Check Box 7"  withType="CHECKBOX"   />
    <hasField withName="Radio Button 4" withType="RADIOBUTTON" />
    <hasField withName="Button 19"     withType="PUSHBUTTON" />
    <hasField withName="List Box 1"    withType="LIST"      />
    <hasField withName="List Box 1"    withType="CHOICE"     />
    <hasField withName="Combo Box 5"   withType="CHOICE"     />
    <hasField withName="Combo Box 5"   withType="COMBO"      />
  </assertThat>
</testcase>

```

Die möglichen Feldtypen sind als Konstanten für das Attribut `withType` definiert. Die Namen der Konstanten entsprechen den gängigen, sichtbaren Elementen einer graphischen Anwendung. Innerhalb von PDF gibt es aber andere Typen. Weil die 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              -> "choice"
COMBO               -> "choice"
LIST               -> "choice"
CHECKBOX           -> "button"
PUSHBUTTON         -> "button"
RADIOBUTTON        -> "button"
SIGNATURE          -> "sig"
TEXT               -> "text"

```

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

```
<testcase name="hasField_WithType_Signature">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasField withName="Signature2" isSigned="YES" />
  </assertThat>
</testcase>
```

Ausführliche Tests zu Signaturen und Zertifikaten werden in Kapitel 3.21: „Signaturen und Zertifikate“ (S. 54) beschrieben.

Größe von Feldern

Falls die Größe von Formularfeldern wichtig ist, stehen für die Überprüfung von Länge und Breite die Attribute `width=".."` und `height=".."` zur Verfügung:

```
<testcase name="hasField_WidthAndHeight">
  <assertThat testDocument="acrofields/notExportableAcrofield.pdf">
    <hasField withName="Title of 'someField'"
              width="159"      (default is MILLIMETER)
              height="11"     (default is MILLIMETER)
    />
  </assertThat>
</testcase>
```

```
<!--
  When @unit is omitted, the values of width are taken as "MILLIMETER".
-->

<testcase name="hasField_Width">
  <assertThat testDocument="acrofields/notExportableAcrofield.pdf">
    <hasField withName="Title of 'someField'" width="159" />
    <hasField withName="Title of 'someField'" width="159" unit="MILLIMETER" />
    <hasField withName="Title of 'someField'" width="15.9" unit="CENTIMETER" /> ❶
    <hasField withName="Title of 'someField'" width="450" unit="DPI72" /> ❷
    <hasField withName="Title of 'someField'" width="450" unit="POINTS" />
    <hasField withName="Title of 'someField'" width="6.26" unit="INCH" />
  </assertThat>
</testcase>
```

❶❷ Die Formate POINTS und DPI72 sind identisch.

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 in Zusammenhang mit der Silbentrennung die Anzahl der benötigten Zeilen und damit die benötigte Höhe. Das Kapitel 3.12: „Formularfelder, Textüberlauf“ (S. 38) beschäftigt sich ausführlich mit diesem Thema.

Weitere Eigenschaften von Feldern

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

```
<testcase name="hasField_Editable">
  <assertThat testDocument="acrofields/plugin-pdf_form_maker.pdf">
    <hasField withName="Combo Box 4" isEditable="YES" />
  </assertThat>
</testcase>
```

Insgesamt stehen folgende Attribute zur Überprüfung von Feldeigenschaften zur Verfügung:

```
<!-- Attributes to check field properties: -->

hasMultipleLines="YES"
hasSingleLine="YES"

isEditable="YES/NO"
isExportable="YES/NO"
isHidden="YES/NO"
isMultiSelectable="YES/NO"
isPasswordProtected="YES"
isPrintable="YES/NO"
isReadOnly="YES/NO"
isRequired="YES/NO"
isSigned="YES"
isVisible="YES/NO"
```

Bei den Vergleichen spielen Whitespaces keine Rolle:

```
<testcase name="hasField_MultiLineField_MultipleInvocations">
  <assertThat testDocument="acrofields/plugin-pdf_form_maker.pdf">
    <hasField withName="Text multi"
              hasMultipleLines="YES"
              isExportable="YES"
    >
      <matchingComplete>
        Multiple Line Support:
        First Line;
        Second Line;
      </matchingComplete>
    </hasField>
  </assertThat>
</testcase>
```

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.

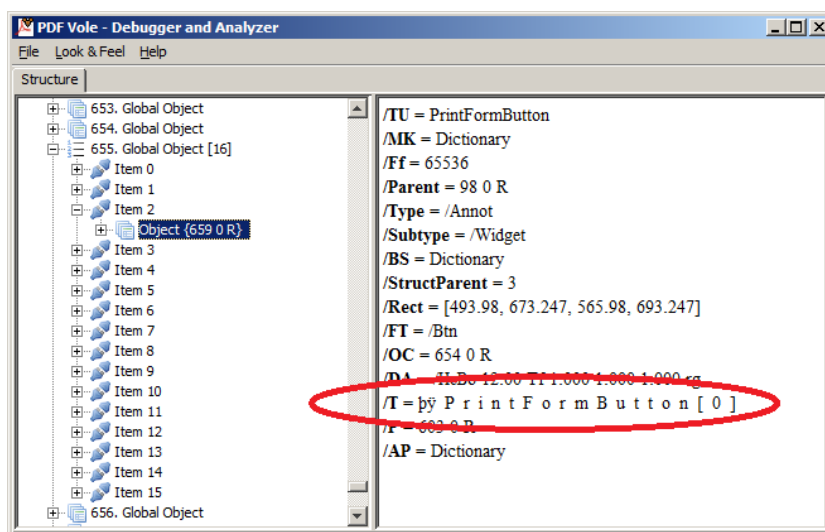
Mit PDFUnit kann geprüft werden, ob ein Formularfeld mit einer Aktion verknüpft ist:

```
<testcase name="hasField_HavingJavaScriptAction_MultipleInvocation">
  <assertThat testDocument="acrofields/javaScriptForFields.pdf">
    <hasField withName="ageField" >
      <havingJavaScriptAction>Validate</havingJavaScriptAction>
    </hasField>
    <hasField withName="nameField" >
      <havingJavaScriptAction>Keystroke</havingJavaScriptAction>
    </hasField>
    <hasField withName="commentField" >
      <havingJavaScriptAction>Keystroke</havingJavaScriptAction>
    </hasField>
  </assertThat>
</testcase>
```

Tags für das Testen des eigentlichen JavaScript-Codes sind in Kapitel 3.13: „JavaScript“ (S. 40) beschrieben.

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:

```
<!--
The name of the field consists of UTF-16BE code represented as ASCII.
Use a Unicode sequence for the field name to test it.
-->

<testcase name="hasField_NameContainingUnicode_UTF16">
  <assertThat testDocument="unicode/unicode_inFieldnames.pdf">
    <hasField withName="\u00fe\u00ff\u0000F\u0000o\u0000r\u0000m\...\u0000]" />
  </assertThat>
</testcase>

<!-- The Unicode sequence in this example is abbreviated. -->
```

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

Feldinformationen gegen XML prüfen

Das Kapitel 9.4: „Feldeigenschaften nach XML extrahieren“ (S. 108) beschreibt, wie mit dem Extraktionsprogramm `ExtractFieldsInfo` Informationen über alle Formularfelder eines PDF-Dokumentes extrahiert werden können. Die dabei entstehende XML-Datei kann in Tests als Vergleichsinstanz benutzt werden:

```
<testcase name="hasField_MatchingXML">
  <assertThat testDocument="acrofields/plugin-pdf_form_maker.pdf">
    <hasFields>
      <matchingXML file="acrofields/plugin-pdf_form_maker.xml"/>
    </hasFields>
  </assertThat>
</testcase>
```

Feldinformation mit XPath validieren

Diese extrahierten XML-Daten von Formularfeldern können auch für XPath-Abfragen genutzt werden. Das ermöglicht es, Abhängigkeiten zwischen mehreren Feldern zu testen („cross-constraints“). Die folgenden Beispiele vermitteln eine Idee von den Möglichkeiten:

```
<testcase name="hasField_MatchingXPath_NumberOfTextFields">
  <assertThat testDocument="acrofields/plugin-pdf_form_maker.pdf">
    <hasFields>
      <matchingXPath expr="count(//field[./@type='text']) = 43"/>
    </hasFields>
  </assertThat>
</testcase>
```

Das Tag `<matchingXPath />` kann mehrfach in einem Test auftauchen:

```
<testcase name="hasField_MatchingXPath_MultipleInvocation">
  <assertThat testDocument="acrofields/plugin-pdf_form_maker.pdf">
    <hasFields>
      <matchingXPath expr="count(//field[./@type='text']) = 43"/>
      <matchingXPath expr="count(//field[./@type='button']) = 54"/>
      <matchingXPath expr="count(//field[./@type='choice']) = 5"/>
      <matchingXPath expr="count(//field[./@type='signatur']) = 0"/>
    </hasFields>
  </assertThat>
</testcase>
```

Die beiden folgenden Beispiele überprüfen, ob es unsignierte Signaturfelder gibt:

```
<testcase name="hasField_MatchingXPath_HavingUnSignedSignatureFields_1">
  <assertThat testDocument="acrofields/certificateform.pdf">
    <hasUnsignedSignatureFields />
  </assertThat>
</testcase>
```

```
<testcase name="hasField_MatchingXPath_HavingUnSignedSignatureFields_2">
  <assertThat testDocument="acrofields/certificateform.pdf">
    <hasFields>
      <matchingXPath expr="count(//field[./@type='sig'][./@isSigned='false']) > 0"/>
    </hasFields>
  </assertThat>
</testcase>
```

PDFUnit verwendet den XSLT-Prozessor der aktuellen Java Runtime. Ob alle Syntaxelemente und Funktionen von XPath 2.0 unterstützt werden, entnehmen Sie daher bitte der Dokumentation Ihrer eingesetzten JRE bzw. des JDK. Einschränkungen seitens PDFUnit bestehen nicht.

3.12. 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 nur teilweise angezeigt. 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 Tests:

```
<!-- Tags to check field overflow: -->

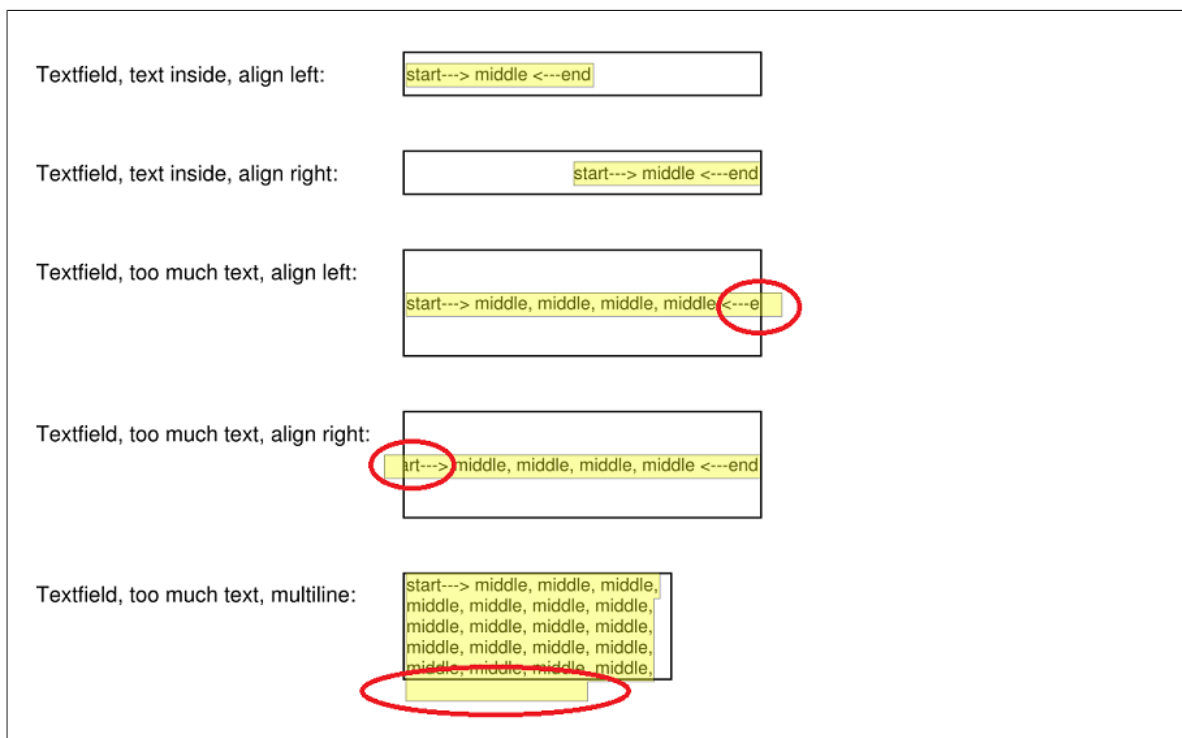
<!-- Check text overflow for one text field: -->
<hasField ... >
  <withoutTextOverflow />
</hasField ... >

<!-- Check text overflow for all text fields: -->
<hasFields ... >
  <allWithoutTextOverflow />
</hasFields ... >
```

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

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:

```
<testcase name="hasField_WithoutTextOverflow_Fieldname"
  errorExpected="YES"
>
  <assertThat testDocument="acrofields/fieldSizeAndText.pdf">
    <hasField withName="Textfield, too much text, multiline:">
      <withoutTextOverflow />
    </hasField>
  </assertThat>
</testcase>
```

Wenn die Exception nicht abgefangen würde (errorExpected="YES"), lautet sie: Content of field 'Textfield, too much text, multiline:' of 'C:\...\fieldSizeAndText.pdf' does not fit in the available space.

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:

```
<testcase name="hasFields_AllWithoutTextOverflow">
  <assertThat testDocument="acrofields/javaScriptForFields.pdf">
    <hasFields>
      <allWithoutTextOverflow />
    </hasFields>
  </assertThat>
</testcase>
```

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

Technische Randbedingung

Der Inhalt eines Feldes wird aus technischen Gründen nicht durch das Test-Tag `<containing>..</containing>` erkannt. Soll er dennoch überprüft werden, muss das PDF-Dokument erst „flach geklopft“ werden (engl. 'flatten').

Insofern schlägt der folgende Test fehl:

```
<!--
Text from AcroFields (type PdfName.ANNOTS) is not detectable.
Flatten it first.
-->
<testcase name="hasTextOnFirstPage_DocumentWithFields">
  <assertThat testDocument="&testfile;">
    <hasText on="FIRST_PAGE">
      <inClippingArea upperLeftX="0" upperLeftY="0" width="595" height="842" unit="POINTS">
        <containing whitespaces="NORMALIZE">
          middle
        </containing>
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

3.13. 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.

Das folgende Tag können Sie nutzen:

```
<!-- Tag to verify JavaScript: -->
<hasJavaScript>
  <containing /> (optional)
  <equals /> (optional)
  <matchingComplete /> (optional)
</hasJavaScript>
```

Existenz von JavaScript

Das folgende Beispiel zeigt, wie geprüft wird, ob das Dokument überhaupt JavaScript enthält:

```
<testcase name="hasJavaScript">
  <assertThat testDocument="javascript/javaScriptClock.pdf">
    <hasJavaScript />
  </assertThat>
</testcase>
```

Vergleich gegen eine Vorlage

Das erwartete JavaScript kann aus einer Datei eingelesen und mit dem des PDF-Dokumentes verglichen werden. Das Hilfsprogramm `ExtractJavaScript` kann dazu benutzt werden, den JavaScript-Code eines PDF-Dokumentes in eine Textdatei extrahiert:

```
<testcase name="hasJavaScript_ScriptFromFile">
  <assertThat testDocument="javascript/javaScriptClock.pdf">
    <hasJavaScript>
      <equals toFile="javascript/javascriptClock.js" />
    </hasJavaScript>
  </assertThat>
</testcase>
```

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:


```

<testcase name="hasJavaScript_ComparedToString">
  <assertThat testDocument="javascript/javaScriptClock.pdf">
    <hasJavaScript>
      <matchingComplete>
        <![CDATA[
          // Constants used by the time calculations
          var oneSec = 1000;
          var oneMin = 60 * oneSec;
          var oneHour = 60 * oneMin;

          var strokeNormal = this.getField("\SWStart").strokeColor;
          var strokeLight = ["RGB",.35,.35,1];
          var fillNormal = this.getField("\SWStart").fillColor;
          var fillLight = ["RGB",.35,.35,0.7];

          function SetFldEnable(oFld, bEnable)
          {
            if(oFld)
            {
              oFld.strokeColor = bEnable?strokeNormal:strokeLight;
              oFld.fillColor = bEnable?fillNormal:fillLight;
              oFld.readonly = !bEnable;
              oFld.textColor = bEnable?color.white:["G",.7];
            }
          }

          ... (code shortened for presentation)

        ]]>
      </matchingComplete>
    </hasJavaScript>
  </assertThat>
</testcase>

```

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:

```

<testcase name="hasJavaScript_ContainingText">
  <assertThat testDocument="javascript/javaScriptClock.pdf">
    <hasJavaScript>
      <containing>
        function DoTimers()
        {
          var nCurTime = (new Date()).getTime();
          ClockProc(nCurTime);
          StopwatchProc(nCurTime);
          CountdownProc(nCurTime);
          this.dirty = false;
        }
      </containing>
    </hasJavaScript>
  </assertThat>
</testcase>

```

```

<testcase name="hasJavaScript_ContainingFunction_MultipleFunctionnames">
  <assertThat testDocument="javascript/javaScriptClock.pdf">
    <hasJavaScript>
      <containing>StopWatchProc</containing>
      <containing>SetFldEnable</containing>
      <containing>DoTimers</containing>
      <containing>ClockProc</containing>
      <containing>CountDownProc</containing>
      <containing>CDEnables</containing>
      <containing>SWSetEnables</containing>
    </hasJavaScript>
  </assertThat>
</testcase>

```

Whitespaces spielen bei allen Vergleichen keine Rolle, weder bei dem JavaScript aus dem PDF-Dokument, noch beim dem aus Dateien oder Tag-Inhalten.

Da es sich bei dem extrahierten JavaScript um eine Textdatei handelt und nicht um XML, gibt es keine XML- und XPath-basierten Tests.

3.14. 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 Tags rund um das Thema „Layer“ an:

```
<!-- Tags to test layers: -->

<!--
  'Layer' means the same as 'OCG, Optional Content Group'.
  So they have the same type:
-->

<hasNumberOfLayers />, <hasNumberOfOCGs />

<!-- Nested tags: -->
<hasLayer> or <hasOCG>
  <withName>                                (required)
    <matchingComplete />                    (one of these nested tags ...
    <containing />                          ...
    <startingWith />                        ... is required)
  </withName>
</hasLayer> or </hasOCG>

<hasLayers> or <hasOCGs>
  <allWithoutDuplicateNames /> (optional)
</hasLayers> or </hasOCGs>
```

Ein Tag `<endingWith />` steht nicht zur Verfügung, weil doppelte Layernamen intern mit einem Suffix erweitert werden und das Namensende somit nicht vorhersehbar ist.

Ein Tag `matchingRegex />` wird ebenfalls nicht angeboten. Es wird nicht benötigt, denn Layernamen sind üblicherweise kurz und bekannt.

Anzahl

Die ersten Tests zielen auf die Anzahl der Layer (OCG):

```
<testcase name="hasNumberOfOCGs">
  <assertThat testDocument="layer/hang-man-game.pdf">
    <hasNumberOfOCGs>40</hasNumberOfOCGs> ❶
    <hasNumberOfLayers>40</hasNumberOfLayers> ❷
  </assertThat>
</testcase>
```

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

Layernamen

Der nächste Test zielt auf die Namen der Layer:

```
<testcase name="hasLayerWithName_MatchingComplete">
  <assertThat testDocument="layer/simpleLayerDemo.pdf">
    <hasLayer>
      <withName>
        <matchingComplete>Parent Layer</matchingComplete>
      </withName>
    </hasLayer>
  </assertThat>
</testcase>
```

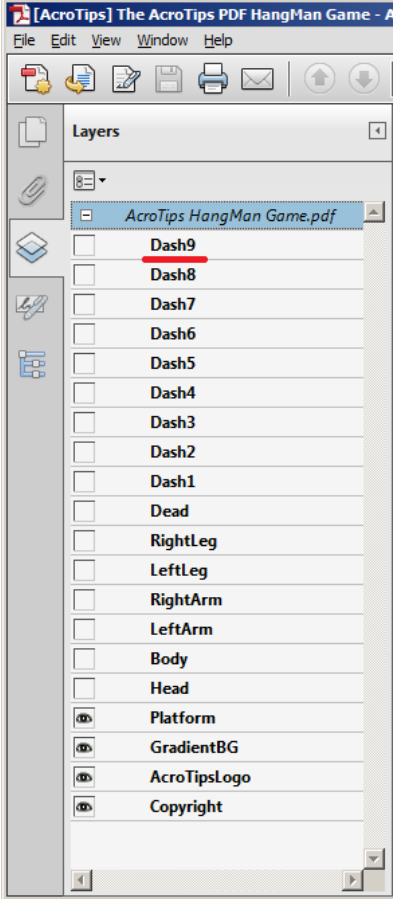
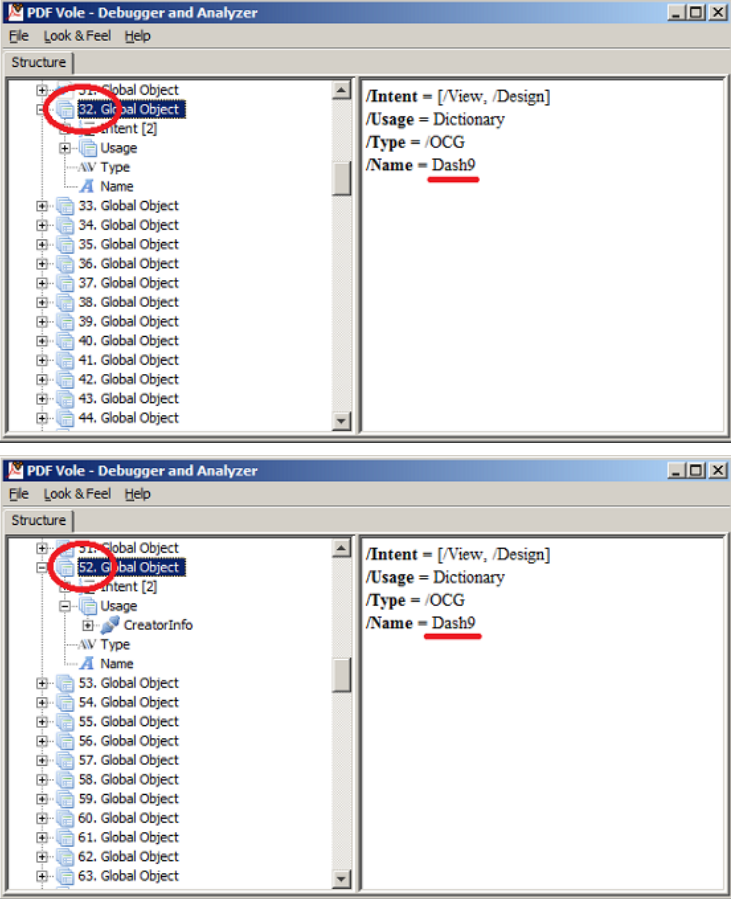
Die Tags `endingWith />` und `matchingRegex />` werden aus den am Anfang des Kapitels erläuterten Gründen nicht angeboten.

Alle Vergleiche werden case-insensitiv ausgeführt. Leerzeichen bleiben erhalten:

```
<!-- Layer names are treated case-insensitive. -->
<testcase name="hasLayerWithName_CaseInsensitive">
  <assertThat testDocument="layer/simpleLayerDemo.pdf">
    <hasLayer>
      <withName>
        <matchingComplete>parent layer</matchingComplete>
      </withName>
    </hasLayer>
    <hasLayer>
      <withName>
        <matchingComplete>Parent Layer</matchingComplete>
      </withName>
    </hasLayer>
  </assertThat>
</testcase>
```

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 einem passenden Test überprüfen:

```
<testcase name="hasLayers_AllWithoutDuplicateNames">
  <assertThat testDocument="layer/simpleLayerDemo.pdf">
    <hasLayers>
      <allWithoutDuplicateNames />
    </hasLayers>
    <hasOCGs> <!-- <hasOCGs /> is equal to <hasLayers /> -->
      <allWithoutDuplicateNames />
    </hasOCGs>
  </assertThat>
</testcase>
```

PDFUnit bietet im aktuellen Release 2015.10 noch keine Test, um auf die Inhalte von Layern zu testen.

3.15. 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. 111) beschrieben.
- ... mit den ebenfalls gerenderten Seiten eines Master-Dokumentes vergleicht, das zuvor als richtig bewertet wurde. Das Kapitel 4.11: „Layout vergleichen (gerenderte Seiten)“ (S. 84) beschreibt diesen Vergleich.

Für Tests von PDF-Seiten als gerendertes Bild steht das Tag `<asRenderedPage />` zur Verfügung.

```
<!-- Tags to compare rendered PDF pages with image files: -->

<asRenderedPage on=".." (One of these attributes ...
  onPage=".." ...
  onEveryPageAfter=".." ...
  onEveryPageBefore=".." ...
  onAnyPageAfter=".." ...
  onAnyPageBefore=".." ... is required)
/>

<!-- Nested tag of <asRenderedPage> : -->

<isEqualto image=".." (required)
  positionUpperLeftX=".." (optional)
  positionUpperLeftY=".." (optional)
  unit=".." (optional)
/>
```

Für den Vergleich können beliebige Seiten angegeben werden. Das Kapitel 13.2: „Seitenauswahl“ (S. 136) geht näher darauf ein.

Dieses Kapitel beschreibt die Vergleiche für ganze Seiten. Wenn es keinen Sinn macht, eine vollständige PDF-Seite als gerendertes Bild zu vergleichen, kann der Vergleich auf einen Ausschnitt beschränkt werden. Das nachfolgende Kapitel 3.16: „Layout - gerenderte Seitenausschnitte“ (S. 45) geht darauf näher ein.

Beispiel - Ausgewählte Seiten als Bild vergleichen

Die Seiten 1, 3 und 4 sollen genauso aussehen, wie die eines Vergleichsdokumentes:

```
<testcase name="compareAsRenderedPage_MultipleImages">
  <assertThat testDocument="master/documentUnderTest.pdf">
    <asRenderedPage onPage="1, 3, 4">
      <isEqualTo image="master/documentUnderTest_Page1.png" />
      <isEqualTo image="master/documentUnderTest_Page3.png" />
      <isEqualTo image="master/documentUnderTest_Page4.png" />
    </asRenderedPage>
  </assertThat>
</testcase>
```

Bildformate der Vergleichsbilder

Die Bilder können in den Formaten GIF, PNG, JPEG, BMP und WBMP vorliegen.

3.16. 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.

```
<!-- Testing a section of a rendered page: -->
<testcase name="..">
  <assertThat testDocument="..">
    <asRenderedPage on="..">                (page selection)
      <isEqualTo positionUpperLeftX=".."    (optional)
                  positionUpperLeftY=".."  (optional)
                  unit=".."                (optional)
                  image=".."              (required)
      />
    </asRenderedPage>
  </assertThat>
</testcase>
```

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:

```
<testcase name="compareAsRenderedPage_LeftMargin">
  <assertThat testDocument="master/documentUnderTest.pdf">
    <asRenderedPage on="EVERY_PAGE">
      <isEqualTo positionUpperLeftX="0" positionUpperLeftY="0"
                  unit="DPI72"
                  image="master/marginFullHeight2cmWidth.png"
      />
    </asRenderedPage>
  </assertThat>
</testcase>
```

Die Bilddatei, 2 cm breit und genauso hoch, wie eine ganze Seite, ist leer. Genauer gesagt, enthält 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:

```
<testcase name="compareRenderedSectionToImages_ImageAsFilename">
  <assertThat testDocument="images/documentWithLogo.pdf">
    <asRenderedPage onPage="1, 2">
      <isEqualTo positionUpperLeftX="135" positionUpperLeftY="35"
        unit="MILLIMETER"
        image="images/logo.png"
      />
    </asRenderedPage>
  </assertThat>
</testcase>
```

Mehrfache Vergleiche

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

```
<testcase name="compareAsRenderedPage_MultipleInvocation">
  <assertThat testDocument="master/documentUnderTest.pdf">
    <asRenderedPage onPage="3, 4">
      <isEqualTo positionUpperLeftX="0" positionUpperLeftY="0"
        unit="DPI72"
        image="master/marginFullHeight2cmWidth.png"
      />
      <isEqualTo positionUpperLeftX="480" positionUpperLeftY="50"
        unit="DPI72"
        image="master/subImage_page3-page4.png"
      />
    </asRenderedPage>
  </assertThat>
</testcase>
```

Sie sollten sich aber überlegen, ob es nicht besser ist, 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.

3.17. 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 das Inhaltsverzeichnis erreichbar sind. Mit den folgenden Tests sollen eventuelle Probleme frühzeitig erkannt werden:

```
<!-- Tags for tests on bookmarks: -->
<hasNumberOfBookmarks />
<hasBookmarks />

<hasBookmark withLabel=".."          (One of these attributes ...
  withLinkToName=".."              ...
  withLinkToPage=".."              ...
  withLinkToURI=".."              ...
  withoutDeadLink=".."            ... has to be used)
/>

<hasBookmark withLabel=".."          (Only these two attributes ...
  linkingToPage=".."              ... can be used together.)
/>

<!-- Nested tags of <hasBookmarks />: -->
<hasBookmarks>
  <matchingXPath />      (optional)
  <matchingXML />       (optional)
</hasBookmarks />
```

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 diese Tags:

```
<!-- Tags to check named destinations: -->

<hasNamedDestination />

<!-- Nested tags: -->
<hasNamedDestination>
  <withName />      (optional)
</hasNamedDestination>
```

Sprungziele, Named Destinations

Namen von Sprungzielen können einfach getestet werden:

```
<testcase name="hasNamedDestination_WithName">
  <assertThat testDocument="namedDestination/manyNamedDestinations.pdf">
    <hasNamedDestination>
      <withName>Seventies</withName>
      <withName>Eighties</withName>
      <withName>1999</withName>
      <withName>2000</withName>
    </hasNamedDestination>
  </assertThat>
</testcase>
```

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:

```
<!--
The conversion of the bookmarks by LibreOffice converts every
space in a bookmark label into "2520" in the named destination.
-->
<testcase name="hasNamedDestination_CreatedWithLibreOffice">
  <assertThat testDocument="namedDestination/problem_convert-bookmarks-to-pdf.pdf">
    <hasNamedDestination>
      <withName>First2520Bookmark</withName> ❶
    </hasNamedDestination>
  </assertThat>
</testcase>
```

❶ "2520" steht für "%20", das wiederum einem Leerzeichen entspricht.

Existenz von Lesezeichen (Bookmarks)

Am einfachsten kann die Existenz von Lesezeichen überprüft werden:

```
<testcase name="hasBookmarks">
  <assertThat testDocument="bookmarks/diverseContentOnMultiplePages.pdf">
    <hasBookmarks />
  </assertThat>
</testcase>
```

Anzahl

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

```
<testcase name="hasNumberOfBookmarks">
  <assertThat testDocument="bookmarks/manyBookmarks.pdf">
    <hasNumberOfBookmarks>19</hasNumberOfBookmarks>
  </assertThat>
</testcase>
```

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:

```
<testcase name="hasBookmark_withLabel">
  <assertThat testDocument="bookmarks/diverseContentOnMultiplePages.pdf">
    <hasBookmark withLabel="Content on page 3." />
  </assertThat>
</testcase>
```

Sprungziele von Lesezeichen

Lesezeichen können sehr unterschiedliche Sprungziele haben. Für jedes Sprungziel gibt es deshalb ein geeignetes Attribut innerhalb des Tags `<hasBookmark />`.

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

```
<testcase name="hasBookmark_WithLabelLinkingToPage">
  <assertThat testDocument="bookmarks/diverseContentOnMultiplePages.pdf">
    <hasBookmark withLabel="Content on first page." linkingToPage="1"/>
  </assertThat>
</testcase>
```

Das Attribut `linkingToPage=".."` kann nur zusammen mit dem Attribut `withLabel=".."` genutzt werden. In einem solchen Test muss dieses Label zu der erwarteten Seitenzahl verweisen.

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

```
<testcase name="hasBookmark_WithLinkToPage">
  <assertThat testDocument="bookmarks/diverseContentOnMultiplePages.pdf">
    <hasBookmark withLinkToPage="1" />
  </assertThat>
</testcase>
```

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

```
<testcase name="hasBookmark_WithLinkToName">
  <assertThat testDocument="bookmarks/twoBookmarkToSameDestination.pdf">
    <hasBookmark withLinkToName="Destination on Page 1" />
  </assertThat>
</testcase>
```

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

```
<testcase name="hasBookmark_WithLinkToURI">
  <assertThat testDocument="bookmarks/bookmarkWithURLAction.pdf">
    <hasBookmark withLinkToURI="http://www.wikipedia.org/" />
  </assertThat>
</testcase>
```

Und als Letztes soll überprüft werden, dass es kein Lesezeichen mit einem „toten Link“ gibt:

```
<!--
Looking for dead internal links (GOTO) of any bookmark.
A 'dead link' means that a bookmark is not pointing to a page.
-->
<testcase name="hasBookmark_WithoutDeadLink">
  <assertThat testDocument="bookmarks/diverseContentOnMultiplePages.pdf">
    <hasBookmark withoutDeadLink="YES" />
  </assertThat>
</testcase>
```

PDFUnit greift während der Tests nicht auf Webseiten zu. Insofern ist ein „toter Link“ ein Lesezeichen, das auf keine Seite oder kein Sprungziel verweist.

Lesezeichen mit XML/XPath testen

Die nachfolgenden Tests basieren auf einer XML-Struktur, die mit dem Hilfsprogramm `Extract-Bookmarks` erzeugt werden kann.

Das aktuelle Test-PDF-Dokument kann vollständig mit der exportierten XML-Datei verglichen werden:

```
<!--
  When comparing PDF parts against any XML,
  whitespaces and comments are ignored.
-->
<testcase name="hasBookmarks_MatchingXML_AsFileName">
  <assertThat testDocument="bookmarks/bookmarksWithPdfOutline.pdf">
    <hasBookmarks>
      <matchingXML file="bookmarks/bookmarksWithPdfOutline.xml" /> ❶
    </hasBookmarks>
  </assertThat>
</testcase>
```

- ❶ Beim Vergleich von PDF-Informationen mit XML spielen Leerzeichen/Whitespaces und XML-Kommentare keine Rolle.

Bookmark-Informationen des aktuellen Test-PDF-Dokumentes können mit individuellen XPath-Ausdrücken evaluiert werden:

```
<testcase name="hasBookmarks_MatchingXPath_MultipleInvocation_version1">
  <assertThat testDocument="bookmarks/bookmarksWithPdfOutline.pdf">
    <hasBookmarks>
      <matchingXPath expr="count(//Title) = 5" />
      <matchingXPath expr="count(//Title[count(ancestor::*) > 2] ) = 0" />
    </hasBookmarks>
  </assertThat>
</testcase>
```

3.18. Passwort

Überblick

Generell gilt die Aussage, dass Sie alle Tests sowohl mit nicht-verschlüsselten als auch mit passwortgeschützten PDF-Dokumenten durchführen können. Passwörter werden beim Tag `<assertThat />` angegeben. Die Syntax sieht folgendermaßen aus:

```
<!-- Tags for tests with passwords: -->

<hasEncryptionLength />
<hasOwnerPassword />
<hasUserPassword />

<!-- Access to encrypted PDF documents: -->
<assertThat testDocument=".." (required)
  testPassword=".." (required if the PDF under test is encrypted)
  masterDocument=".." (required if a master PDF is used)
  masterPassword=".." (required if the master PDF is encrypted)
/>
```

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

Tests auf das Passwort 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: -->

<testcase name="hasOwnerPassword">
  <assertThat testDocument="content/diverseContentOnMultiplePages_encrypted.pdf"
    testPassword="user-password" ❶
  >
    <hasOwnerPassword>owner-password</hasOwnerPassword> ❷
  </assertThat>
</testcase>

```

```

<!-- Verify the user-password of the document: -->

<testcase name="hasUserPassword">
  <assertThat testDocument="content/diverseContentOnMultiplePages_encrypted.pdf"
    testPassword="owner-password" ❸
  >
    <hasUserPassword>user-password</hasUserPassword> ❹
  </assertThat>
</testcase>

```

- 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. „Hart kodiert“ bedeutet aber auch, dass das Passwort nie wechselt.

Test auf Verschlüsselungslänge

Mit welcher Verschlüsselungslänge wurde verschlüsselt:

```

<testcase name="hasEncryptionLength">
  <assertThat testDocument="content/diverseContentOnMultiplePages_encrypted.pdf"
    testPassword="user-password"
  >
    <hasEncryptionLength>128</hasEncryptionLength>
  </assertThat>
</testcase>

```

3.19. Schriften

Überblick

Schriften in PDF-Dokumenten sind keine einfache Sache, spielen aber spätestens dann eine Rolle, wenn eine verwendete Schriftart nicht mehr zu den durch den PDF-Standard definierten 14 Schriftarten gehört. Auch für die Archivierung von PDF-Dokumenten spielen Schriften eine besondere Rolle. PDFUnit bietet rund um das Thema „Schriften“ unterschiedliche Tags an:

```

<!-- Tags to test fonts: -->

<hasNumberOfFonts identifiedBy=".." (Filters are explained later)
/>

<hasFonts ofTypeOnly=".."> (Either this attribute,
  <matchingXPath /> or one of the
  <matchingXML /> nested elements.)
</hasFonts>

<hasFont withNameContaining=".." (One of this two
  withNameNotContaining=".." attributes is required)
/>

```

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. Da es das Ziel eines Unit-

tests ist, beim zweiten Aufruf dasselbe Ergebnis zu liefern, wie beim ersten Aufruf, ist es eigentlich egal, wie gezählt wird. Da alle PDF-Werkzeuge die Frage entscheiden müssen, was sie zählen, liefern sie auch unterschiedliche Werte für die Anzahl von Schriften.

In PDFUnit gelten zwei Schrift als „equals“, wenn die für einen Test relevanten Vergleichskriterien gleiche Werte haben. Die Vergleichskriterien werden mit dem Attribut `identifiedBy="..."` angegeben:

```
<!-- Constants to identify fonts: -->
identifiedBy="ALLPROPERTIES"
identifiedBy="BASENAME"
identifiedBy="BASENAME_ENCODING"
identifiedBy="BASENAME_ENCODING_ENCODINGDIFF"
identifiedBy="CONVERTIBLE2UNICODE"
identifiedBy="EMBEDDED"
identifiedBy="EMBEDDED_CONVERTIBLE2UNICODE"
identifiedBy="NAME"
identifiedBy="NAME_TYPE"
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.
BASENAME_ENCODING	Der Name der Basisschrift und das Encoding werden gemeinsam als Unterscheidungsmerkmal benutzt.
BASENAME_ENCODING_ENCODINGDIFF	Zusätzlich zur vorhergehenden Identifizierung müssen zwei Schriften auch noch einen unterschiedlichen Wert in der Eigenschaft Encoding-Differenz besitzen. Die „Encoding-Differenz“ ist der Wert des PDF-Objektes mit dem Namen /Differences.
CONVERTIBLE2UNICODE	Dieser Filter zählt nur solche Schriften, die in Unicode konvertiert werden können.
EMBEDDED	Mit diesem Filter werden sämtliche Schriften erfasst, die eingebettet sind.
EMBEDDED_CONVERTIBLE2UNICODE	Zusätzlich zu dem vorhergehenden Filter gilt hier noch die Eigenschaft, nach Unicode konvertierbar zu sein, als Unterscheidungsmerkmal.
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 die verschiedenen Vergleichskriterien benutzt:

```
<testcase name="hasNumberOfFonts_Japanese">
  <assertThat testDocument="fonts/fonts_11_japanese.pdf">
    <hasNumberOfFonts identifiedBy="ALLPROPERTIES">65</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="BASENAME">9</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="BASENAME_ENCODING">16</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="BASENAME_ENCODING_ENCODINGDIFF">16</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="CONVERTIBLE2UNICODE">46</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="EMBEDDED">6</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="EMBEDDED_CONVERTIBLE2UNICODE">0</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="NAME">50</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="NAME_TYPE">55</hasNumberOfFonts>
    <hasNumberOfFonts identifiedBy="TYPE">3</hasNumberOfFonts>
  </assertThat>
</testcase>
```

Schriftnamen

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

```
<testcase name="hasFont_WithNameContaining">
  <assertThat testDocument="fonts/fonts_15_openoffice.pdf">
    <hasFont withNameContaining="Arial" />
  </assertThat>
</testcase>
```

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.

In einem Test können mehrere Schriftnamen getestet werden:

```
<testcase name="hasHasFont_MultipleNames">
  <assertThat testDocument="fonts/fonts_15_openoffice.pdf">
    <hasFont withNameContaining="Arial" />
    <hasFont withNameContaining="Georgia" />
    <hasFont withNameContaining="Tahoma" />
    <hasFont withNameContaining="TimesNewRoman" />
    <hasFont withNameContaining="Verdana" />
    <hasFont withNameContaining="Verdana-BoldItalic" />
  </assertThat>
</testcase>
```

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

```
<testcase name="hasFontWithName_NotContaining">
  <assertThat testDocument="fonts/fonts_15_openoffice.pdf">
    <hasFont withNameNotContaining="ComicSansMS" />
  </assertThat>
</testcase>
```

Komplexere Tests auf Schriftnamen können mit XPath-Ausdrücken realisiert werden. Sie werden weiter unten in diesem Kapitel beschrieben.

Schrifttypen

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

```
<testcase name="hasFonts_OfThisTypeOnly_TrueType">
  <assertThat testDocument="fonts/fonts_15_openoffice.pdf">
    <hasFonts ofThisTypeOnly="TRUETYPE" />
  </assertThat>
</testcase>
```

Die prüfbaren Schrifttypen sind als Konstanten deklariert:

```
<!-- constants for font types -->
ofThisTypeOnly="CID"
ofThisTypeOnly="CID_TYPE0"
ofThisTypeOnly="CID_TYPE2"
ofThisTypeOnly="CJK"
ofThisTypeOnly="MMTYPE1"
ofThisTypeOnly="OPENTYPE"
ofThisTypeOnly="TRUETYPE"
ofThisTypeOnly="TYPE0"
ofThisTypeOnly="TYPE1"
ofThisTypeOnly="TYPE3"
```

XML-Datei als Referenz

Alle von PDFUnit intern verwendeten Schriftinformationen können mit dem Extraktionsprogramm `ExtractFontsInfo` nach XML exportiert werden. Die erzeugte XML-Datei kann für Tests verwendet werden:

Die Datei enthält diese Informationen:

```
<?xml version="1.0" encoding="UTF-8" ?>
<fontlist>
  ...
  <font name="Courier"           baseFontName="Courier"
        type="Type1"             embedded="false"
        encoding="WinAnsiEncoding" convertibleToUnicode="false"
  />
  <font name="FGNPL+ArialMT"      baseFontName="ArialMT"
        type="TrueType"          embedded="true"
        encoding="WinAnsiEncoding" convertibleToUnicode="false"
  />
  ...
</fontlist>
```

Ein Test gegen die XML-Datei sieht dann so aus:

```
<testcase name="hasFontsMatchingXML_ComparedAsFile">
  <assertThat testDocument="fonts/fonts_52_itext.pdf">
    <hasFonts>
      <matchingXML file="fonts/fonts_52_itext.xml" />
    </hasFonts>
  </assertThat>
</testcase>
```

Whitespaces spielen bei den Vergleichen mit der XML-Datei keine Rolle.

XPath-Abfrage auf Schriften

Auf der Basis der Schriftinformationen im XML-Format können anspruchsvolle Tests mit XPath-Abfragen umgesetzt werden:

```
<!--
  This XML code needs double quotes outside and single quotes inside,
  because the generated Java code also needs double quotes outside.
-->
<testcase name="hasFontsMatchingXPath_MultipleInvocation">
  <assertThat testDocument="fonts/fonts_52_itext.pdf">
    <hasFonts>
      <matchingXPath expr="count(//font[@baseFontName='ArialMT']) = 1" />
      <matchingXPath expr="count(//font[@type='Type1']) = 5" />
    </hasFonts>
  </assertThat>
</testcase>
```

Wenn Sie Problem mit XPath-Ausdrücken haben, exportieren Sie die Schriftinformationen mit dem Extraktionsprogramm `ExtractFontsInfo` und überprüfen den XPath-Ausdruck direkt an der XML-Datei. Eclipse stellt dafür die „XPath“-View zur Verfügung.

Weitere Informationen zu XPath stehen im Kapitel 8: „XPath-Einsatz“ (S. 103).

3.20. 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 Porto anfällt. PDFUnit bietet deshalb Tags an, um die Anzahl der Seiten zu überprüfen:

```
<!-- Tags to verify page numbers: -->

<hasNumberOfPages />
<hasLessPages than=".." (required) />
<hasMorePages than=".." (required) />
```

Beispiele

Eine konkrete Seitenanzahl wird folgendermaßen überprüft:

```
<testcase name="hasNumberOfPages">
  <assertThat testDocument="format/format_Letter-Portrait.pdf">
    <hasNumberOfPages>1</hasNumberOfPages>
  </assertThat>
</testcase>
```

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

```
<testcase name="hasNumberOfPagesLessThan">
  <assertThat testDocument="format/format_multiple-formats-on-individual-pages.pdf">
    <hasLessPages than="6" /> <!-- The document has 5 pages. -->
  </assertThat>
</testcase>
```

```
<testcase name="hasMorePagesThan">
  <assertThat testDocument="format/format_multiple-formats-on-individual-pages.pdf">
    <hasMorePages than="2" /> <!-- The document has 5 pages. -->
  </assertThat>
</testcase>
```

Die Werte für Ober- und Untergrenze gelten exklusiv.

Auf beide Grenzen kann gleichzeitig getestet werden:

```
<!--
Validating that a document has a number of pages in a allowed range.
-->
<testcase name="hasNumberOfPages_InRange">
  <assertThat testDocument="format/format_multiple-formats-on-individual-pages.pdf">
    <hasMorePages than="2" />
    <hasLessPages than="8" />
  </assertThat>
</testcase>
```

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

3.21. Signaturen und Zertifikate

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 zahlreiche Tags für Signaturen und Zertifikate zur Verfügung:

```
<!-- Tags to test signatures: -->
<hasNumberOfSignatures />
<isSigned />

<hasSignature name=".." (required)
>
  <coveringWholeDocument /> (optional)
  <withNumberOfRevisions /> (optional)
  <withReason /> (optional)
  <withRevision /> (optional)
  <withCertificate /> (optional)
  <withSigningDate /> (optional)
  <withSigningName /> (optional)
</hasSignature>

<hasSignature name=".." (required)
>
  <withCertificate (optional)
  >
    <validForCurrentDate /> (optional)
    <validFor /> (optional)
    <validFrom /> (optional)
    <validUntil /> (optional)
    <havingSubjectField name=".." value=".."
  </withCertificate>
</hasSignature>

<hasSignatures>
  <matchingXPath /> (optional)
  <matchingXML /> (optional)
</hasSignatures>
```

Bei den Tests, die die Eigenschaften eines Zertifikates überprüfen, greift PDFUnit nur auf Daten innerhalb des PDF-Dokumentes zu. Ein Zugriff in's Internet findet nicht statt. Somit wird nicht überprüft, ob ein Zertifikat zurückgezogen wurde.

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 und in „Profilen“ definiert sind. Tests für zertifizierte PDF-Dokumente sind im Kapitel 3.31: „Zertifiziertes PDF“ (S. 74) beschrieben.

Existenz

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

```
<testcase name="isSigned">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <isSigned />
  </assertThat>
</testcase>
```

Signaturnamen und -anzahl

Ein PDF-Dokument kann mehrere Signaturen enthalten, wenn es beispielsweise von mehreren Personen unterschrieben wurde. Insofern zielen die nächsten Tests auf die Anzahl und die Namen der Zertifikate:

```
<testcase name="hasNumberOfSignatures">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasNumberOfSignatures>1</hasNumberOfSignatures>
  </assertThat>
</testcase>
```

```
<testcase name="hasSignature">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2" />
  </assertThat>
</testcase>
```

Gültigkeitsdatum

Wenn PDF-Dokumente Signaturen nutzen, ist es gelegentlich interessant, festzustellen, ob ein Testdatum innerhalb des Gültigkeitszeitraumes des Zertifikates liegen:

```
<testcase name="hasSignatureValidForCurrentDate">
  <assertThat testDocument="signed/find_document.pdf">
    <hasSignature name="Signature2">
      <withCertificate>
        <validForCurrentDate />
      </withCertificate>
    </hasSignature>
  </assertThat>
</testcase>
```

```
<testcase name="hasSignatureWithSigningDate_DATE">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2">
      <withSigningDate date="2009-07-16" pattern="yyyy-MM-dd" /> ❶
    </hasSignature>
  </assertThat>
</testcase>
```

- ❶ Das Attribut „pattern“ bestimmt, dass der Vergleich auf der Basis von Jahr-Monat-Tag stattfindet.

Eine weitere Testmöglichkeit ist die, sicherzustellen, dass das Zertifikat innerhalb eines Zeitraumes gültig ist:

```
<testcase name="hasSignature_FirstAndLastDate">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2">
      <withCertificate>
        <validFrom date="20060822" pattern="yyyyMMdd" />
        <validUntil date="20090904" pattern="yyyyMMdd" />
      </withCertificate>
    </hasSignature>
  </assertThat>
</testcase>
```

Revision, Grund (Reason), Unterzeichner (Sign Name)

Manche Informationen einer Signatur können mit speziellen Tags getestet werden:

```
<testcase name="hasSignature_CoveringWholeDocument">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2">
      <coveringWholeDocument />
    </hasSignature>
  </assertThat>
</testcase>
```

```
<testcase name="hasSignature_WithSigningName">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2">
      <withSigningName name="John B Harris"/>
    </hasSignature>
  </assertThat>
</testcase>
```

```
<testcase name="hasSignature_WithRevision">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2">
      <withRevision nr="1" />
    </hasSignature>
  </assertThat>
</testcase>
```



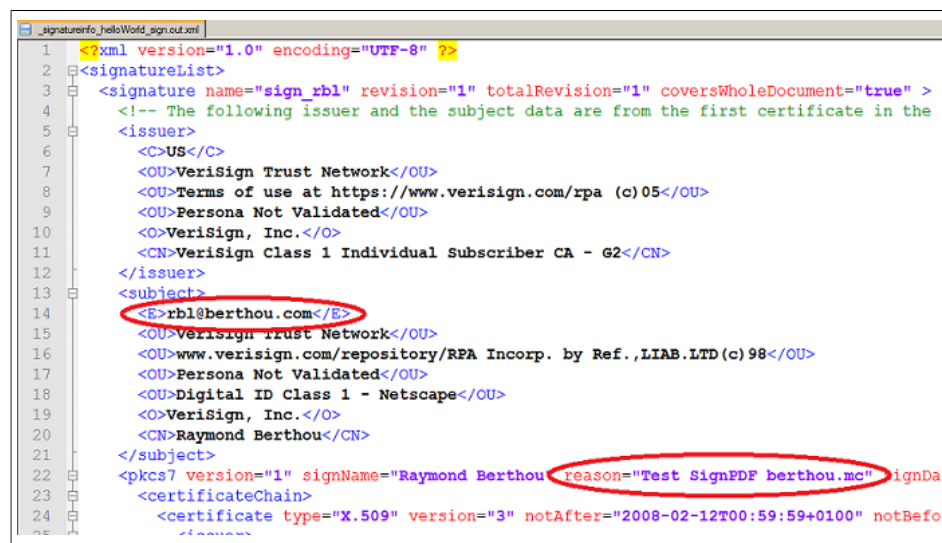
```
<testcase name="hasSignature_WithNumberOfRevisions">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2">
      <withNumberOfRevisions nr="1" />
    </hasSignature>
  </assertThat>
</testcase>
```

```
<testcase name="hasSignature_WithReason">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2">
      <withReason>I am the author of this document</withReason>
    </hasSignature>
  </assertThat>
</testcase>
```

Der Rest der Informationen ist über Tests auf Basis von XML und XPath erreichbar.

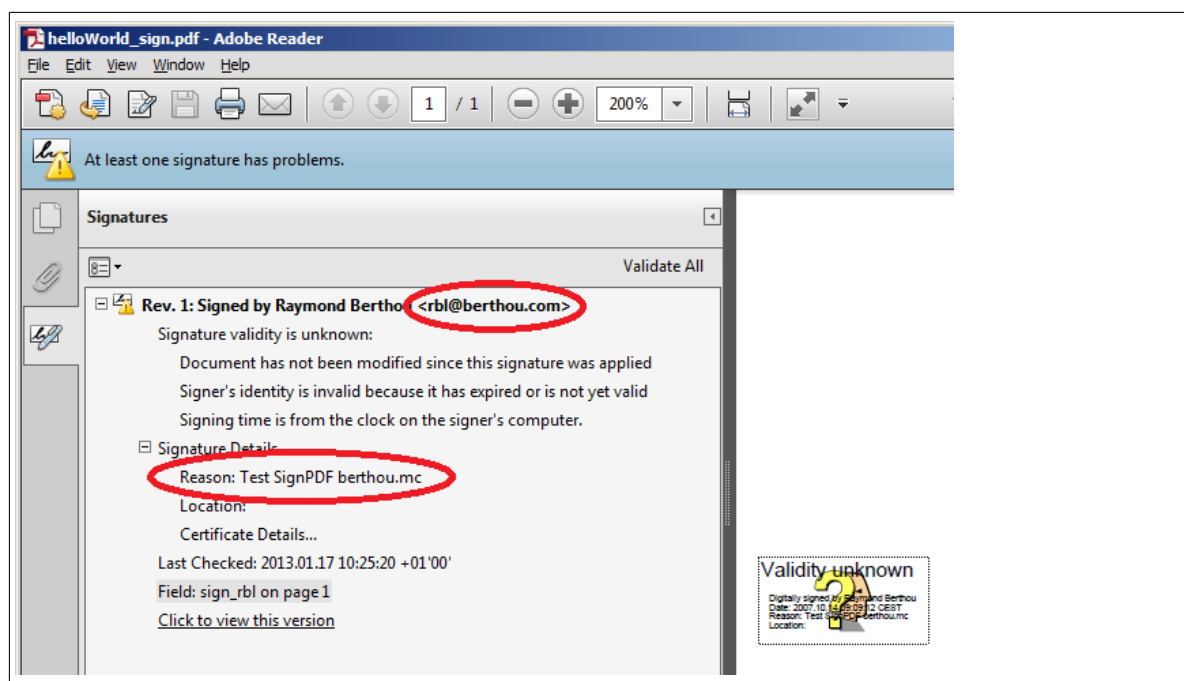
Vergleiche auf Basis von XML und XPath

Jede Information einer Signatur kann mit XPath-Ausdrücken überprüft werden. Um den XPath-Ausdruck überhaupt entwickeln zu können, müssen Sie die Signaturdaten mit dem Hilfsprogramm ExtractSignaturesInfo in eine XML-Datei überführen. Die Datei hat dieses Aussehen (Ausschnitt):



```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <signatureList>
3   <signature name="sign_rbl" revision="1" totalRevision="1" coversWholeDocument="true" >
4     <!-- The following issuer and the subject data are from the first certificate in the
5     <issuer>
6       <C>US</C>
7       <OU>VeriSign Trust Network</OU>
8       <OU>Terms of use at https://www.verisign.com/rpa (c) 05</OU>
9       <OU>Persona Not Validated</OU>
10      <O>VeriSign, Inc.</O>
11      <CN>VeriSign Class 1 Individual Subscriber CA - G2</CN>
12    </issuer>
13    <subject>
14      <E>rbl@berthou.com</E>
15      <OU>VeriSign Trust Network</OU>
16      <OU>www.verisign.com/repository/RPA Incorp. by Ref., LIAB.LTD (c) 98</OU>
17      <OU>Persona Not Validated</OU>
18      <OU>Digital ID Class 1 - Netscape</OU>
19      <O>VeriSign, Inc.</O>
20      <CN>Raymond Berthou</CN>
21    </subject>
22    <pkcs7 version="1" signName="Raymond Berthou" reason="Test SignPDF berthou.mc" ignDa
23    <certificateChain>
24      <certificate type="X.509" version="3" notAfter="2008-02-12T00:59:59+0100" notBefo
```

Das dazugehörige Bild vom Adobe Reader® zeigt die gleichen Informationen:



Die Signaturinformationen eines PDF-Dokumentes können vollständig mit der XML-Datei verglichen werden:

```
<testcase name="hasSignatures_MatchingXML">
  <assertThat testDocument="signed/helloWorld_sign.pdf">
    <hasSignatures>
      <matchingXML file="signed/helloWorld_sign.xml" />
    </hasSignatures>
  </assertThat>
</testcase>
```

Wenn nur Teile der XML-Datei testrelevant sind, muss für das Testziel ein passender XPath-Ausdruck gefunden werden. Im folgenden Beispiel wird überprüft, ob das erste Zertifikat ein OU-Tag mit einem bestimmten Wert hat:

```
<testcase name="hasSignatures_MatchingXPath_OneOfManyOU">
  <assertThat testDocument="signed/helloWorld_sign.pdf">
    <hasSignatures>
      <matchingXPath expr="//certificate[1]/subject[OU='Digital ID Class 1 - Netscape']" />
    </hasSignatures>
  </assertThat>
</testcase>
```

Hinweis: Zum Erarbeiten und Überprüfen eines XPath-Ausdrucks gibt es in Eclipse die „XPath-View“.

Zusammenhängend testen

Mehrere Tests einer Signatur können kombiniert werden:

```
<testcase name="differentAspectsAroundSignature">
  <assertThat testDocument="signed/helloWorld_sign.pdf">
    <isSigned />
    <hasSignature name="sign_rbl" >
      <withSigningName name="Raymond Berthou" />
      <withSigningDate date="2007-10-14T09:09:12+0200" pattern="yyyy-MM-dd'T'HH:mm:ssZ" />
      <withRevision nr="1" />
      <withCertificate>
        <havingSubjectField name="O" value="VeriSign, Inc." />
      </withCertificate>
    </hasSignature>
  </assertThat>
</testcase>
```

Überlegen Sie sich aber einen guten Namen für diesen Test!

3.22. Sprachinformation (Language)

Überblick

PDF-Dokumente können für Sehbehinderte durch Screenreader-Programme vorgelesen werden. Damit das funktioniert, muss das Dokument die Sprache angeben können, in der das Dokument erstellt ist.

Diese Sprachvorgabe kann auf einfache Weise überprüft werden:

```
<!-- Tag to verify locale: -->

<hasLocale name=".."          (optional)
           string=".."        (optional)
           expectedEmpty=".." (optional)
/>

<hasNoLocale />
```

Beispiele

```
<testcase name="hasLocale_CaseInsensitive_LowerCase">
  <assertThat testDocument="language/_languageInfo/localeDemo_en-GB.pdf">
    <hasLocale string="en-gb" /> ❶
  </assertThat>
</testcase>
```

```
<testcase name="hasLocale_CaseInsensitive_UpperCase">
  <assertThat testDocument="language/_languageInfo/localeDemo_en-GB.pdf">
    <hasLocale string="en_GB" /> ❷
  </assertThat>
</testcase>
```

- 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 eine Konstante von `java.util.Locale` direkt verwendet werden, achten Sie dabei aber auf die Korrektheit der Groß-/Kleinschreibung.

```
<testcase name="hasLocale_LocaleInstance_GERMANY">
  <assertThat testDocument="language/_languageInfo/localeDemo_de.pdf">
    <hasLocale name="Locale.GERMANY" />
  </assertThat>
</testcase>
```

```
<testcase name="hasLocale_LocaleInstance_GERMAN">
  <assertThat testDocument="language/_languageInfo/localeDemo_de.pdf">
    <hasLocale name="Locale.GERMAN" />
  </assertThat>
</testcase>
```

Ein PDF-Dokument mit der 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 Länderkennung "en" auf `Locale.UK` getestet wird.

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

```
<testcase name="hasLocale_LanguageEmpty">
  <assertThat testDocument="language/_languageInfo/localeDemo_null.pdf">
    <hasNoLocale />
  </assertThat>
</testcase>
```

3.23. Texte

Überblick

Der häufigste Testfall für PDF-Dokumente ist vermutlich, die Existenz erwarteter Texte zu überprüfen. Dafür steht das Tag `<hasText />` zur Verfügung, das weitere Tags und Attribute für den eigentlichen Textvergleich bereitstellt:

```
<!-- Tags to verify content: -->

<hasText />

<!-- Nested tags of <hasText />: -->

<hasText >
  <inClippingArea />      (optional)

  <!-- Comparing content: -->
  <containing />           (optional)
  <endingWith />          (optional)
  <matchingComplete />    (optional)
  <matchingRegex />       (optional)
  <startingWith />        (optional)

  <!-- Prove the absence of text: -->
  <notContaining />        (optional)
  <notEndingWith />        (optional)
  <!-- <notMatchingRegex /> is intentionally not provided -->
  <notMatchingRegex />    (optional)
  <notStartingWith />     (optional)
</hasText />

<!-- Attributes of <hasText /> to select pages. -->
<!-- One of these attributes has to be used: -->

<hasText on=".." />
<hasText onPage=".." />
<hasText onEveryPageAfter=".." />
<hasText onEveryPageBefore=".." />
<hasText onAnyPageAfter=".." />
<hasText onAnyPageBefore=".." />

<!--
Whitespace processing, see: 13.4: „Behandlung von Whitespaces“ (S. 138)
-->
```

Text auf bestimmten Seiten

Wenn Sie einen bestimmten Text auf der ersten Seite eines Anschreibens suchen, vielleicht die Adresse in der Anschrift, sieht ein Test folgendermaßen aus:

```
<testcase name="hasText_OnFirstPage_Containing">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Content on first page.</containing>
    </hasText>
  </assertThat>
</testcase>
```

Die Selektion bestimmter Seiten geschieht über das Attribut `on=".."`, für das mehrere Konstanten zur Verfügung stehen, beispielsweise `on="FIRST_PAGE"`, `on="EVERY_PAGE"`, `on="ODD_PAGES"` etc. Das Kapitel 13.2: „Seitenauswahl“ (S. 136) beschreibt die Seitenauswahl ausführlich.

Ein Text auf der letzten Seite wird folgendermaßen überprüft:

```
<testcase name="hasText_OnLastPage">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="LAST_PAGE">
      <containing>Content on last page.</containing>
    </hasText>
  </assertThat>
</testcase>
```

Auch Tests mit beliebigen individuellen Seiten sind möglich, dazu wird das Attribut `onPage=" . . "` zu Verfügung gestellt:

```
<testcase name="hasText_OnIndividualPages">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onPage="2, 3">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

Die Seitenzahlen im Attribut `onPage=" . . "` müssen durch Kommata getrennt werden.

Das Kapitel 13.2: „Seitenauswahl“ (S. 136) beschreibt die Seitenauswahl ausführlich.

Text auf allen Seiten

Um Text auf allen Seiten zu suchen, stehen im Attribut `on=" . . "` drei Konstanten zur Verfügung: `on="EACH_PAGE"`, `on="EVERY_PAGE"` und `on="ANY_PAGE"`.

```
<testcase name="hasText_OnEveryPage">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="EVERY_PAGE">
      <startingWith>PDFUnit</startingWith>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="hasText_OnAnyPage">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="ANY_PAGE">
      <containing>Page # 3</containing>
    </hasText>
  </assertThat>
</testcase>
```

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

Verneinte Suche

Die Logik der beiden vorhergehenden Beispiele ist klar. Unklar wird die Logik aber bei der Verneinung beider Aussagen. In der Umgangssprache ist der Unterschied zwischen „Jede Seite enthält den Suchbegriff nicht“ und „Irgendeine Seite enthält den Suchbegriff nicht“ nicht klar.

Um Fehler zu vermeiden, erlaubt PDFUnit keine verneinten Testmethoden in Kombination mit der Konstanten `ON_ANY_PAGE`. Der folgende Test ist daher **nicht** zulässig:

```
<testcase name="hasText_NotMatchingRegex">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="ANY_PAGE">
      <notEndingWith>wrongValueIntended</notEndingWith>
    </hasText>
  </assertThat>
</testcase>
```

Die Fehlermeldung lautet:

```
Searching text 'ON_ANY_PAGE' in combination with negated methods is not supported.
```

Anstatt zu testen, ob „irgendeine Seite einen Suchbegriff NICHT enthält“ prüfen Sie lieber, dass „Jede Seite den Suchbegriff enthält“ und fangen dann die Exception ab.

Zeilenumbrüche im Text

Zeilenumbrüche im Text werden beim Vergleich ignoriert, 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:

```
<!--
The PDF document has a (visible) line break after the word "The".
The search string does not contain a line break.
-->
```

```
<testcase name="hasText_ContainingLineBreaks_LineBreakInPDF">
  <assertThat testDocument="digitalsignatures20121017.pdf">
    <hasText on="FIRST_PAGE">
      <containing>The technology was conceived</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<!--
The expected search string intentionally contains other line breaks.
-->
```

```
<testcase name="hasText_ContainingLineBreaks_LineBreakInExpectedString">
  <assertThat testDocument="digitalsignatures20121017.pdf">
    <hasText on="FIRST_PAGE">
      <containing>
        The
        technology
        was
        conceived
      </containing>
    </hasText>
  </assertThat>
</testcase>
```

Text in Seitenausschnitten

Text kann aber nicht nur auf vollständigen Seite gesucht werden, sondern auch auf Seitenausschnitten. Das Kapitel 13.6: „Seitenausschnitt definieren“ (S. 142) beschreibt diesen Aspekt ausführlich.

Seiten ohne Text

Sie können auch testen, ob Ihr PDF-Dokument leere Seiten enthält:

```
<testcase name="hasText_AnyPageEmpty">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="EVERY_PAGE" />
  </assertThat>
</testcase>
```

Wenn Sie explizit prüfen wollen, ob eine Seite oder ein Ausschnitt einer Seite leer ist, geht das mit dem Tag `hasNoText`:

```
<testcase name="hasNoTextInClippingArea" >
  <assertThat testDocument="&pdfdir;/emptyPages/pagesPartiallyEmpty.pdf">
    <hasNoText on="FIRST_PAGE" >
      <inClippingArea upperLeftX="70" upperLeftY="80" width="90" height="60" />
    </hasNoText>
  </assertThat>
</testcase>
```

Mehrere Suchbegriffe gleichzeitig

Wenn auf einer Seite mehrere Texte gesucht werden, ist es lästig, für jeden Suchbegriff einen eigenen Test zu schreiben. Deshalb können manche Tags mehrfach benutzt werden:

```
<testcase name="hasText_Containing_MultipleTokens">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="ODD_PAGES">
      <containing>on</containing>
      <containing>page</containing>
      <containing>odd pagenumber</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="hasText_NotContaining_MultipleTokens">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="FIRST_PAGE">
      <notContaining>even pagenumber</notContaining>
      <notContaining>Page #2</notContaining>
    </hasText>
  </assertThat>
</testcase>
```

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

Die Tags `<startingWith />` und `<endingWith />` dürfen nicht mehrfach auftreten.

Mehrfach verwendete Textvergleiche beziehen sich alle auf die spezifizierten Seiten des umschließenden Tags `<hasText />`:

```
<testcase name="hasText_MultipleInvocation">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="ANY_PAGE">
      <startingWith>PDFUnit</startingWith>
      <containing>Content on last page.</containing>
      <matchingRegex>.*[Cc]ontent.*</matchingRegex>
      <endingWith>of 4</endingWith>
    </hasText>
  </assertThat>
</testcase>
```

Für Textvergleiche, die sich auf unterschiedliche Seiten beziehen, muss das Tag `<hasText />` wiederholt werden:

```
<!--
Different pages and different comparisons in one concatenated statement.
This test works, but it is not recommended.
When the test fails, the error analysis is more complicated than
if you had 3 individual tests.
-->
<testcase name="hasText_ComplexSearchOverDifferentPages">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="ANY_PAGE">
      <startingWith>PDFUnit - Automated PDF Tests</startingWith>
    </hasText>
    <hasText on="EVEN_PAGES">
      <containing>Content</containing>
      <containing>even pagenumber</containing>
    </hasText>
    <hasText on="ODD_PAGES">
      <containing>odd pagenumber</containing>
    </hasText>
  </assertThat>
</testcase>
```

Eine solcher Test ist aber nicht sinnvoll, weil der Name des Tests nicht klar genug gewählt werden kann.

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:

```
<testcase name="hasText_OnAnyPageAfter">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onAnyPageAfter="1">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

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.
-->
<testcase name="hasText_OnAllPagesBefore_WrongPageNumber">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onAnyPageBefore="99">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

Potentielles Problem mit „Fließtext“

Die sichtbare Reihenfolge des Textes einer PDF-Seite entspricht nicht zwingend der Textreihenfolge innerhalb des PDF-Dokumentes. Das kann zur Folge haben, dass PDFUnit scheinbaren „Fließtext“ nicht als solchen erkennt, aber PDFUnit nutzt intern die Fähigkeiten von iText, Textobjekte über ihre Positionen auf der Seite zusammenzubauen.

Obwohl der Text im Rahmen ein eigenes Textobjekt ist, funktioniert ein Test über den fortlaufenden Text "the beginning. This is content":

Content at the beginning.

This is content, placed in a
frame by OpenOffice.

Content at the end.

```
<!--
  The PDF document does not store the text in the visible order.
-->
<testcase name="hasText_TextNotInVisibleOrder">
  <assertThat testDocument="content/contentNotInVisibleOrder.pdf">
    <hasText on="FIRST_PAGE">
      <containing>
        Content at the beginning.
        This is content, placed in a frame by OpenOffice.
        Content at the end.
      </containing>
    </hasText>
  </assertThat>
</testcase>
```


3.24. Texte - in Ausschnitten einer Seite

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:

```
<!-- Compare text inside a clipping area: -->
<testcase name="..">
  <assertThat testDocument="..">
    <hasText on=".." >
      <inClippingArea upperLeftX=".." upperLeftY=".." width=".." height=".." >
        ... <!-- compare text here -->
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

Beispiel

Das folgende Beispiel zeigt die Definition und Benutzung eines Seitenausschnitts:

```
<testcase name="hasTextOnFirstPage_MultipleValidations">
  <assertThat testDocument="content/documentForTextClipping.pdf">
    <hasText on="FIRST_PAGE" >
      <inClippingArea upperLeftX="50" upperLeftY="130" width="170" height="25" > ❶
        <startingWith>Content</startingWith>
        <containing>on first</containing>
        <endingWith>page.</endingWith>
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

- ❶ Hier wird der Seitenausschnitt definiert. Genaue Informationen dazu liefert das Kapitel 13.6: „Seitenausschnitt definieren“ (S. 142). Die Möglichkeiten, Maß-Einheiten wie MILLIMETER für die Definition zu benutzen, beschreibt Kapitel 13.7: „Maßeinheiten - Points, Millimeter, ...“ (S. 143).

Für Vergleiche von Text in Seitenausschnitten stehen alle Tags zur Verfügung, die auch für ganze PDF-Seiten zur Verfügung stehen. Sie sind in Kapitel 13.3: „Textvergleich“ (S. 138) ausführlich beschrieben.

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

3.25. 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 Tags zur Verfügung, wie auch für „normalen“ Text.

Beispiel

Das Beispieldokument enthält zwei senkrechte Texte:

Text from bottom to top.
Text from top to bottom.

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

```
<testcase name="hasText_RotatedText_InClippingArea">
  <assertThat testDocument="writeDirection/verticalText.pdf">
    <hasText on="FIRST_PAGE">
      <inClippingArea upperLeftX="110" upperLeftY="130"
        width="130" height="300" unit="POINTS"
      >
        <containing>Text from bottom to top.</containing>
        <containing>Text from top to bottom.</containing>
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

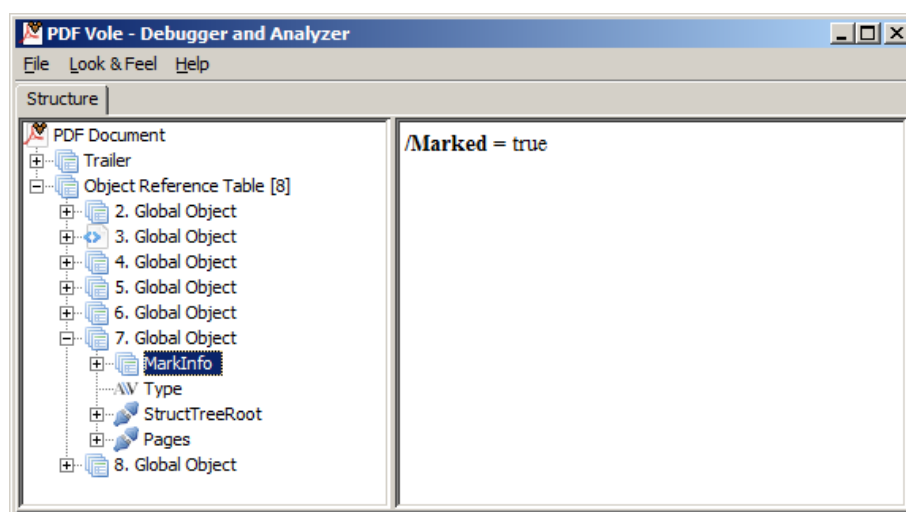
Beachten Sie: Es handelt sich bei diesem kopfwärts gestellten Text immer noch um Text mit der Laufrichtung LTR (left-to-right). Texte mit der Laufrichtung RTL (right-to-left) werden von PDFUnit in zukünftigen Releases unterstützt.

3.26. 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: http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf.)

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 Tags stehen für Tests zur Verfügung:

```
<!-- Tag to verify tagging information: -->
<isTagged />
<!-- Inner tag of <isTagged />: -->
<with key=".." (required)
      andValue=".." (optional)
/>
```

Beispiele

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

```
<testcase name="isTagged">
  <assertThat testDocument="tagged/itext-created_tagged.pdf">
    <isTagged />
  </assertThat>
</testcase>
```

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

```
<testcase name="isTagged_WithKey">
  <assertThat testDocument="tagged/xdp_2.0.pdf">
    <isTagged>
      <with key="LetterspaceFlags" />
    </isTagged>
  </assertThat>
</testcase>
```

Als Letztes können Werte bestimmter Tags verifiziert werden:

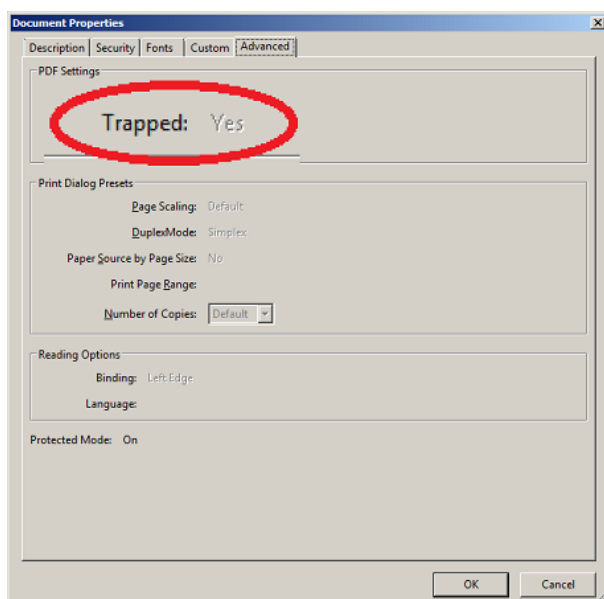
```
<testcase name="isTaggedWithKeyValue_MultipleInvocations">
  <assertThat testDocument="tagged/xdp_2.0.pdf">
    <isTagged>
      <with key="Marked" andValue="true" />
      <with key="LetterspaceFlags" andValue="0" />
    </isTagged>
  </assertThat>
</testcase>
```

3.27. Trapping-Info

Überblick

Der Begriff Trapping ist in Wikipedia (<http://de.wikipedia.org/wiki/Trapping>) gut beschrieben. Es geht darum, zu erreichen, dass bei einem Mehrfarbendruck, bei dem jede Farbe separat gedruckt wird, kein weißes Papier zwischen den Farbwechseln sichtbar wird. Der Wikipedia-Artikel enthält weiterführende Links zu Erklärungen von Adobe.

Ein PDF-Dokument gibt Auskunft darüber, ob es schon "getrappt" ist, oder nicht. Im Adobe Reader® wird die „Trapped“-Eigenschaft über den Eigenschaften-Dialog angezeigt:



„Trapped“ kann die Werte Yes, No und Unknown haben. Die dazu gehörenden Tests sind überschaubar:

```
<!-- Tag to verify trapping information: -->
<hasTrappingInfo value=".." (required)
/>

<!--
  Only these constants are allowed for the attribute 'value':
-->

value="YES"
value="NO"
value="UNKNOWN"
```

Beispiele

```
<testcase name="hasTrappingInfo_Yes">
  <assertThat testDocument="trapping/trapping-info_yes.pdf">
    <hasTrappingInfo value="YES" />
  </assertThat>
</testcase>
```

```
<testcase name="hasTrappingInfo_No">
  <assertThat testDocument="trapping/trapping-info_no.pdf">
    <hasTrappingInfo value="NO" />
  </assertThat>
</testcase>
```

```
<testcase name="hasTrappingInfo_Unknown">
  <assertThat testDocument="trapping/trapping-info_unknown.pdf">
    <hasTrappingInfo value="UNKNOWN" />
  </assertThat>
</testcase>
```

3.28. Version

Überblick

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

```
<!-- Tag to verify version: -->

<hasVersion matching=".."      (One of the three)
           greaterThan=".."    attributes has
           lessThan=".."      is required.)
/>
```

Eine bestimmte Version

In PDFUnit ist das Format für die PDF-Versionen durch XML Schema auf das Format „Zahl Punkt Zahl“ festgelegt. Hier ein Beispiel für den Test auf Version „1.4“:

```
<testcase name="hasVersion_v14">
  <assertThat testDocument="version/pdf-version-1.4.pdf">
    <hasVersion matching="1.4" />
  </assertThat>
</testcase>
```

Versionsbereiche

Mit den Attributen `greaterThan` und `lessThan` kann auch ein Versionsbereich überprüft werden:

```
<testcase name="hasVersion_GreaterThanLessThan">
  <assertThat testDocument="version/pdf-version-1.6.pdf" >
    <hasVersion greaterThan="1.3" lessThan="1.7" /> ❶
  </assertThat>
</testcase>
```

❶ Die Ober- und Untergrenzen gelten exklusiv.

Auch zukünftige PDF-Versionen können getestet werden:

```
<testcase name="hasVersion_LessThanFutureVersion">
  <assertThat testDocument="version/pdf-version-1.6.pdf" >
    <hasVersion lessThan="2.0" />
  </assertThat>
</testcase>
```

3.29. 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:

```
<!-- Tags to test XFA data: -->

<hasXFADData />
<hasNoXFADData />

<!-- Inner tags of hasXFADData: -->

<hasXFADData>
  <matchingXPath /> (optional)
  <matchingXML /> (optional)
  <withNode /> (optional)
</hasXFADData>
```

Existenz und Abwesenheit von XFA

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

```
<testcase name="hasXFADData">
  <assertThat testDocument="xfa/xfa-movie.pdf">
    <hasXFADData />
  </assertThat>
</testcase>
```

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

```
<testcase name="hasNoXFADData">
  <assertThat testDocument="xfa/no-xfa.pdf">
    <hasNoXFADData />
  </assertThat>
</testcase>
```

Vergleich gegen eine XML-Datei

Die XFA-Daten eines PDF-Dokumentes können mit dem Hilfsprogramm `ExtractXFADData` in eine XML-Datei exportiert werden. Diese Datei kann vollständig mit den XFA-Daten eines PDF-Dokumentes verglichen werden:

```
<testcase name="hasXFADData_MatchingXML">
  <assertThat testDocument="xfa/xfa-movie.pdf">
    <hasXFADData>
      <matchingXML file="xfa/xfa-movie.xml" /> ❶
    </hasXFADData>
  </assertThat>
</testcase>
```

❶ Whitespaces werden beim Vergleich der XML-Daten nicht berücksichtigt.

Oftmals ist es nicht sinnvoll, die kompletten XFA-Daten eines PDF-Dokumentes gegen eine XML-Vorlage zu vergleichen. Für solche Fälle gibt es sowohl die Möglichkeit, auf einzelne XML-Knoten zu testen, als auch Tests mit XPath-Ausdrücken zu formulieren. Beide Möglichkeiten werden in den folgenden Abschnitten beschrieben.

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:xmpptk="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>
```

Der Wert eines bestimmten XML-Knotens kann mit dem Tag `<withNode />` getestet werden:

```
<testcase name="hasXFADData_WithNode">
  <assertThat testDocument="xfa/xfa-enabled.pdf">
    <hasXFADData>
      <withNode tag="xmp:MetadataDate"
        value="2009-12-03T17:50:52Z"
        defaultNamespace="http://www.xfa.org/schema/xci/2.6/" /> ❶
    </hasXFADData>
  </assertThat>
</testcase>
```

- ❶ PDFUnit analysiert die XFA-Daten des aktuellen PDF-Dokumentes und ermittelt die Namensräume selbständig, lediglich der Default-Namespaces muss angegeben werden.

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

Zur Umsetzung ergänzt PDFUnit intern vor dem Knoten den Pfad-Bestandteil " / ". Aus diesem Grund darf der Knoten im Test kein Pfad sein, der die Wurzel " / " enthält.

Prüfungen auf Attribut-Knoten sind selbstverständlich auch möglich:

```
<testcase name="hasXFADData_WithNode_NamespaceDD">
  <assertThat testDocument="xfa/xfa-enabled.pdf">
    <hasXFADData>
      <withNode tag="dd:dataDescription/@dd:name"
        value="movie"
      />
    </hasXFADData>
  </assertThat>
</testcase>
```

XPath-basierte XFA-Tests

XPath kann mehr, als nur einzelne Knoten zu benennen. Mit dem Tag `<matchingXPath />` können alle Möglichkeiten von XPath genutzt werden.

Die nächsten beiden Beispiele zeigen, was machbar ist:

```
<testcase name="hasXFADData_FunctionStartsWith">
  <assertThat testDocument="xfa/xfa-enabled.pdf">
    <hasXFADData>
      <!-- complete value: 'movie' -->
      <matchingXPath expr="starts-with(//dd:dataDescription/@dd:name, 'mov')" />
    </hasXFADData>
  </assertThat>
</testcase>
```

```
<testcase name="hasXFADData_MatchingXPath_FunctionCount_MultipleInvocation">
  <assertThat testDocument="xfa/xfa-movie.pdf">
    <hasXFADData>
      <matchingXPath expr="//pdf:Producer[.='Adobe LiveCycle Designer ES 8.2']" />
      <matchingXPath expr="count(//processing-instruction()) = 30" />
    </hasXFADData>
  </assertThat>
</testcase>
```

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.

Default-Namensraum in XPath

XML-Namensräume werden automatisch ermittelt, aber der verwendete **Default**-Namensraum muss explizit angegeben werden. Weil der XML-Standard **mehrere, verschiedene** Deklarationen in einem Dokument zulässt, ist nicht automatisch klar, welcher Default-Namensraum benutzt werden soll, wenn tatsächlich mehrere Deklarationen verwendet werden. Deshalb muss er im Test angegeben werden:

```
<testcase name="hasXFADData_DefaultNamespace_matchingXPath">
  <assertThat testDocument="xfa/xfa-movie.pdf">
    <hasXFADData>
      <matchingXPath expr="count(//default:subform[@name='movie']//default:field) = 5"
                    defaultNamespace="http://www.xfa.org/schema/xfa-template/2.6/"
      />
    </hasXFADData>
  </assertThat>
</testcase>
```

Aus dem gleichen Grund muss der Default-Namensraum auch bei der Verwendung des Tags `<withNode />` mitgegeben werden:

```
<!--
  The default namespace has to be declared,
  but any alias can be used for it.
-->
<testcase name="hasXFADData_DefaultNamespace_WithNode_AnyAlias">
  <assertThat testDocument="xfa/xfa-enabled.pdf">
    <hasXFADData>
      <withNode tag="foo:log/foo:to"
                value="memory"
                defaultNamespace="http://www.xfa.org/schema/xci/2.6/" />
    </hasXFADData>
  </assertThat>
</testcase>
```

❶ Der Alias für den Default-Namensraum kann **beliebig** gewählt werden.

3.30. XMP-Daten

Überblick

XMP steht für „Extensible Metadata Platform“ und ist ein von Adobe initiiertes offener Standard, Metadaten in beliebige Dateitypen einzubetten. Nicht nur PDF-Dokumente, auch Bilder können Informationen über den Ort, das Format und andere „Daten über 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 Tags an, wie für XFA-Daten:

```
<!-- Tags to test XMP data: -->

<hasXMPData />
<hasNoXMPData />

<!-- Inner tags of hasXMPData: -->

<hasXMPData>
  <matchingXPath /> (optional)
  <matchingXML /> (optional)
  <withNode /> (optional)
</hasXMPData>
```

Existenz und Abwesenheit von XMP

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

```
<testcase name="hasXMPData">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData />
  </assertThat>
</testcase>
```

```
<testcase name="hasNoXMPData">
  <assertThat testDocument="xmp/bookmarkWithURLAction_noXMP.pdf">
    <hasNoXMPData />
  </assertThat>
</testcase>
```


Vergleich gegen eine XML-Datei

Mit dem Hilfsprogramm `ExtractXMPData` können die XMP-Daten eines PDF-Dokumentes in eine XML-Datei exportiert werden. XMP-Daten eines PDF-Dokumentes können dann gegen diese XML-Datei verglichen werden:

```
<testcase name="hasXMPData_MatchingXML">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <matchingXML file="xmp/metadata-added.xml" /> ❶
    </hasXMPData>
  </assertThat>
</testcase>
```

❶ Beim Vergleich zweier XML-Strukturen werden Leerzeichen ignoriert.

Einzelne XML-Tags validieren

XMP-Daten eines PDF-Dokumentes können gezielt auf einzelne XML-Knoten und ihren Wert überprüft werden. Das nächste Beispiel 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>
```

Für den Test auf einen Knoten aus den XMP-Daten steht das Tag `<withNode />` zur Verfügung. Nachfolgend wird die Existenz zweier Knotens geprüft:

```
<testcase name="hasXMPData_WithNode_ValidateExistence">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <withNode name="xmp:CreateDate" />
      <withNode name="xmp:ModifyDate" />
    </hasXMPData>
  </assertThat>
</testcase>
```

Soll auch der Wert eines Knotens überprüft werden, muss der erwartete Wert im Attribut `value=".."` angegeben werden:

```
<!--
  When the node name occurs multiple times in the document, only
  the first node will be returned.
-->
<testcase name="hasXMPData_WithNodeAndValue">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <withNode name="xmp:CreateDate" value="2011-02-08T15:04:19+01:00" />
      <withNode name="xmp:ModifyDate" value="2011-02-08T15:04:19+01:00" />
    </hasXMPData>
  </assertThat>
</testcase>
```

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

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

Der Knoten darf selbstverständlich auch ein Attribut-Knoten sein.

XPath-basierte XMP-Tests

Mit dem Tag `<matchingXPath />` kann das ganze Potential von XPath genutzt werden:

```
<testcase name="hasXMPData_MatchingXPath_CreateDateWithValue">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <matchingXPath expr="//xmp:CreateDate[node() = '2011-02-08T15:04:19+01:00']" />
    </hasXMPData>
  </assertThat>
</testcase>
```

```
<testcase name="hasXMPData_MatchingXPath_MultipleInvocation">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <matchingXPath expr="count(//xmp:CreateDate) = 1" />
      <matchingXPath
        expr="count(//xmp:CreateDate[1][node()='2011-02-08T15:04:19+01:00']) = 1" />
    </hasXMPData>
  </assertThat>
</testcase>
```

Der Funktionsumfang der Verarbeitung der XPath-Ausdrücke hängt vom verwendeten XML-Parser bzw. der XPath-Engine ab. PDFUnit verwendet die des JDK/JRE und unterliegt damit den Unterschieden der unterschiedlichen JVM-Hersteller.

Default-Namensraum in XPath

Wie schon für XFA-Tests beschrieben, werden XML-Namensräume automatisch ermittelt. Default-Namensräume müssen vorgegeben werden, weil sie im XML-Dokument mehrfach auftreten können und deshalb nicht aus dem Dokument abgeleitet werden können.

Hier die Verwendung des Default-Namensraumes für das Tag `<matchingXPath />`:

```
<testcase name="hasXMPData_MatchingXPath_WithDefaultNamespace">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <matchingXPath expr="//foo:format = 'application/pdf'"
        defaultNamespace="http://purl.org/dc/elements/1.1/"
      />
    </hasXMPData>
  </assertThat>
</testcase>
```

Und noch ein Beispiel für den Default-Namensraum im Tag `<withNode />` mit einem Erwartungswert:

```
<testcase name="hasXMPData_WithDefaultNamespace_XMLNode">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <withNode name="foo:ModifyDate"
        value="2011-02-08T15:04:19+01:00"
        defaultNamespace="http://ns.adobe.com/xap/1.0/"
      />
    </hasXMPData>
  </assertThat>
</testcase>
```

3.31. Zertifiziertes PDF

Überblick

Wer garantiert eigentlich die Einhaltung von Vorgaben für die Verarbeitung von PDF-Dokumenten in Ihrem oder einem anderen Unternehmen? „Zertifizierte PDF-Dokumente“ sind die Antwort auf diese Frage.

Ein „zertifiziertes PDF“ ist ein normales PDF mit Zusatzinformationen. Es enthält Informationen über das Profil, das während der Zertifizierung des Dokumentes verwendet wurde. Außerdem enthält es

auch eine Historie über alle Änderungen, die seit der Zertifizierung an diesem Dokument vorgenommen wurden. Die Änderungen können rückgängig gemacht werden.

Passend zu den Möglichkeiten, die iText für die Erstellung von PDF bietet, bietet PDFUnit ein Tag zum Testen an:

```
<!-- Tag to verify certification: -->
<isCertified for=".." (optional)
/>

<!-- The allowed values are defined as constants: -->
for="NOT_CERTIFIED"
for="NO_CHANGES_ALLOWED"
for="FORM_FILLING"
for="FORM_FILLING_AND_ANNOTATIONS"
```

Die PDFUnit Konstanten entsprechen den Konstanten von iText, PdfSignatureAppearance.CERTIFIED_*.

Achtung, ein „zertifiziertes PDF-Dokument“ darf nicht mit dem „Zertifikat“ einer Signatur verwechselt werden.

Beispiele

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

```
<testcase name="isCertified">
  <assertThat testDocument="certified/sampleCertifiedPDF.pdf">
    <isCertified />
  </assertThat>
</testcase>
```

Als Nächstes kann der Grad der Zertifizierung überprüft werden:

```
<testcase name="isCertifiedFor_NoChangesAllowed"
  errorExpected="YES"
>
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <isCertified for="NO_CHANGES_ALLOWED" />
  </assertThat>
</testcase>
```

Kapitel 4. Vergleiche gegen ein Master-PDF

4.1. Überblick

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

Viele Eigenschaften eines PDF-Dokumentes, die einzeln getestet werden können, können auch als Vergleich gegen ein Master-PDF getestet werden.

Initialisierung

Die Verwendung eines Master-Dokumentes erfolgt über das Attribut `masterDocument`:

```
<testcase name="testInstantiation_NotEncryptedMaster">
  <assertThat testDocument="test/test.pdf"
              testPassword="owner-password"
              masterDocument="master/master.pdf"
  >
  </assertThat>
</testcase>
```

- ❶ Das Test-Dokument ist verschlüsselt und wird mit dem Passwort geöffnet.
- ❷ Das Master-Dokument ist hier nicht verschlüsselt. Falls es verschlüsselt wäre, muss das Passwort im Attribut `masterPassword` angegeben 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. Die Kapitel sind alphabetisch sortiert. Das letzte Kapitel 4.20: „Sonstige Vergleiche“ (S. 91) enthält mehrere Funktionen, die nicht in einzelnen Kapiteln beschrieben sind.

```
<!-- Tags to compare two PDF documents: -->

<areBothForFastWebView />          4.20: „Sonstige Vergleiche“ (S. 91)
<haveSameActions />                4.2: „Aktionen vergleichen“ (S. 77)
<haveSameAppearance />            4.11: „Layout vergleichen (gerenderte Seiten)“ (S. 84)
<haveSameAuthor />                4.7: „Dokumenteneigenschaften vergleichen“ (S. 81)
<haveSameBookmarks />             4.12: „Lesezeichen (Bookmarks) vergleichen“ (S. 85)
<haveSameCreationDate />          4.6: „Datumswerte vergleichen“ (S. 81)
<haveSameCreator />               4.7: „Dokumenteneigenschaften vergleichen“ (S. 81)
<haveSameEmbeddedFiles />         4.3: „Anhänge (Attachments) vergleichen“ (S. 78)
<haveSameFields />                4.9: „Formularfelder vergleichen“ (S. 82)
<haveSameFonts />                 4.15: „Schriften vergleichen“ (S. 87)
<haveSameFormat />                4.8: „Formate vergleichen“ (S. 82)
<haveSameImages />                4.5: „Bilder vergleichen“ (S. 80)
<haveSameJavaScript />            4.20: „Sonstige Vergleiche“ (S. 91)
<haveSameKeywords />              4.20: „Sonstige Vergleiche“ (S. 91)
<haveSameLanguage />              4.20: „Sonstige Vergleiche“ (S. 91)
<haveSameLayerNames />            4.20: „Sonstige Vergleiche“ (S. 91)
<haveSameModificationDate />      4.6: „Datumswerte vergleichen“ (S. 81)
<haveSameText />                  4.17: „Text vergleichen“ (S. 88)

... continued
```

```

... continuation:

<haveSameNumberOfActions />      4.2: „Aktionen vergleichen“ (S. 77)
<haveSameNumberOfBookmarks />    4.12: „Lesezeichen (Bookmarks) vergleichen“ (S. 85)
<haveSameNumberOfEmbeddedFiles /> 4.3: „Anhänge (Attachments) vergleichen“ (S. 78)
<haveSameNumberOfFields />       4.9: „Formularfelder vergleichen“ (S. 82)
<haveSameNumberOfFonts />       4.15: „Schriften vergleichen“ (S. 87)
<haveSameNumberOfImages />      4.5: „Bilder vergleichen“ (S. 80)
<haveSameNumberOfLayers />      4.14: „PDF-Bestandteile vergleichen“ (S. 87)
<haveSameNumberOfPages />       4.14: „PDF-Bestandteile vergleichen“ (S. 87)
<haveSameNumberOfTaggingInfo />  4.14: „PDF-Bestandteile vergleichen“ (S. 87)
<haveSamePermission />          4.4: „Berechtigungen vergleichen“ (S. 79)
<haveSamePermissions />        4.4: „Berechtigungen vergleichen“ (S. 79)
<haveSameProducer />           4.7: „Dokumenteneigenschaften vergleichen“ (S. 81)
<haveSameProperties />          4.7: „Dokumenteneigenschaften vergleichen“ (S. 81)
<haveSameProperty />           4.7: „Dokumenteneigenschaften vergleichen“ (S. 81)
<haveSameSignatureNames />     4.16: „Signaturnamen vergleichen“ (S. 88)
<haveSameSubject />            4.7: „Dokumenteneigenschaften vergleichen“ (S. 81)
<haveSameTaggingInfo />        4.20: „Sonstige Vergleiche“ (S. 91)
<haveSameTitle />              4.7: „Dokumenteneigenschaften vergleichen“ (S. 81)
<haveSameTrappingInfo />       4.20: „Sonstige Vergleiche“ (S. 91)
<haveSameXFADData />           4.18: „XFA-Daten vergleichen“ (S. 89)
<haveSameXMPData />            4.19: „XMP-Daten vergleichen“ (S. 90)

... (end of list)

```

4.2. Aktionen vergleichen

Anzahl

Ein Vergleich der Anzahl von Aktionen in zwei PDF-Dokumenten, sieht so aus:

```

<testcase name="haveSameNumberOfActions">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfActions />
  </assertThat>
</testcase>

```

Eigenschaften von Aktionen

Um die Aktionen zweier PDF-Dokumente miteinander zu vergleichen, gibt es das Tag `<haveSameActions />`:

```

<testcase name="haveSameActions">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameActions />
  </assertThat>
</testcase>

```

Wann zwei Aktionen gleich sind, hängt von ihrem Typ ab. Die folgende Tabelle zeigt für jeden Aktionstyp die Eigenschaften, die für eine Gleichheit relevant sind:

Typ	Relevante Eigenschaft(en) für equals()	
GotoAction	destination	Das Ziel, auf das die Aktion zeigt.
	orientation	Die Richtung der Aktion, beispielsweise „/FIT“
GotoEmbeddedAction	destination	Das Ziel, auf das die Aktion zeigt.
	new window	Gibt an, ob das Ziel in einem neuen Fenster angezeigt werden soll.
GotoRemoteAction	filename	Die Datei, die die Aktion anspricht.
	page number	Die Seitenzahl im Ziel

Typ	Relevante Eigenschaft(en) für equals()	
	remote destination	Ein Ziel innerhalb der angesprochenen Datei.
	new window	Gibt an, ob das Ziel in einem neuen Fenster angezeigt werden soll.
ImportDataAction	filename	Die Datei, die importiert werden soll.
JavaScriptAction	javaScript	Der JavaScript-Code. Leerzeichen werden so reduziert, wie in Kapitel 13.4: „Behandlung von Whitespaces“ (S. 138) beschrieben.
LaunchAction	filename	Die Datei, die die Aktion starten will.
	default directory	Das Verzeichnis, in dem die Datei gesucht.
	operation	Eine Funktion, die der Datei übergeben wird und ausgeführt werden soll, beispielsweise „print“.
	parameters	Parameter, die an die Datei/Funktion übergeben werden
NamedAction	name	Der Name der Aktion
ResetFormAction	fields	Die Namen der Felder, deren Inhalt gelöscht werden sollen.
	flags	Eigenschaften für die zurückzusetzenden Felder haben müssen.
SubmitFormAction	destination	Das Ziel, an das die Feldinhalte gesendet werden sollen.
	fields	Die Felder, deren Werte verschickt werden.
	flags	Einstellung für die Art der Übermittlung, beispielsweise PdfAction.SUBMIT_HTML_GET oder PdfAction.SUBMIT_PDF.
URIAction	destination	Das Ziel in Form einer URI, auf das die Aktion zeigt.

Die folgenden Events sind immer mit JavaScript-Aktionen verknüpft und werden als solche miteinander verglichen:

- document close (/DC)
- document will print (/WP)
- document did print (/DP)
- document will save (/WS)
- document did save (/DS)

Der Zeitpunkt/Event „document open“ (/DocumentOpen) kann mit jeder beliebigen Aktion der oben dargestellten Liste verknüpft werden. Damit ist dann auch klar, dass zwei „Open-Actions“ gemäß dieser Liste miteinander verglichen werden.

4.3. Anhänge (Attachments) vergleichen

Anzahl

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

```
<testcase name="haveSameNumberOfEmbeddedFiles">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameNumberOfEmbeddedFiles />
  </assertThat>
</testcase>
```

Namen und Inhalte

Für einen Vergleich der eingebetteten Dateien nach Name oder Inhalt gibt es das Tag `<haveSameEmbeddedFiles />`:

```
<testcase name="haveSameEmbeddedFiles">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameEmbeddedFiles comparedBy="NAME" />
    <haveSameEmbeddedFiles comparedBy="CONTENT" /> ❶
  </assertThat>
</testcase>
```

- ❶ Die Dateien werden byte-weise verglichen, sodass Dateien jeglicher Art verglichen werden können.

Das Attribut `comparedBy=" . . "` definiert die zwei Konstanten `NAME` und `CONTENT`.

Eingebettete Dateien können mit dem Hilfsprogramm `ExtractEmbeddedFiles` extrahiert werden. Siehe Kapitel 9.2: „Anhänge extrahieren“ (S. 105).

4.4. Berechtigungen vergleichen

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

```
<testcase name="haveSamePermissions">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSamePermissions />
  </assertThat>
</testcase>
```

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

```
<testcase name="haveSamePermissions_MultiplePermissions">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSamePermission permission="ALLOW_EXTRACT_CONTENT" />
    <haveSamePermission permission="ALLOW_COPY" />
    <haveSamePermission permission="ALLOW_MODIFY_CONTENT" />
  </assertThat>
</testcase>
```

Folgende Konstanten stehen zur Verfügung:

```
<!-- Available permissions: -->
permission="ALLOW_ASSEMBLE_DOCUMENTS"
permission="ALLOW_COPY"
permission="ALLOW_DEGRADED_PRINTING"
permission="ALLOW_EXTRACT_CONTENT" ❶
permission="ALLOW_FILL_IN"
permission="ALLOW_MODIFY_ANNOTATIONS"
permission="ALLOW_MODIFY_CONTENT"
permission="ALLOW_PRINTING"
permission="ALLOW_SCREENREADERS" ❷
```

- ❶❷ Die Berechtigungen `ALLOW_EXTRACT_CONTENT` und `ALLOW_SCREENREADERS` sind gleichwertig. Sie werden aus sprachlichen Gründen beide angeboten.

4.5. Bilder vergleichen

Anzahl

Soll die Anzahl der Bilder verglichen werden, sieht der Test so aus:

```
<testcase name="haveSameNumberImages">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfImages />
  </assertThat>
</testcase>
```

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

```
<testcase name="haveSameNumberOfImages_OnPage2">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfImages onPage="2" />
  </assertThat>
</testcase>
```

Die Möglichkeiten der Seitenauswahl sind in Kapitel 13.2: „Seitenauswahl“ (S. 136) beschrieben.

Bildinhalte

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

```
<!--
  The tag <haveSameImages /> does not consider the order of the images.
-->

<testcase name="haveSameImages">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameImages />
  </assertThat>
</testcase>
```

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 als Parameter mitgegeben werden:

```
<testcase name="haveSameImages_OnPage2">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameImages onPage="2" />
  </assertThat>
</testcase>
```

❶

```
<testcase name="haveSameImages_BeforePage2">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameImages onEveryPageBefore="2" />
  </assertThat>
</testcase>
```

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-Dokuments mit dem Hilfsprogramm `ExtractImages` extrahiert werden. Siehe Kapitel 9.3: „Bilder aus PDF extrahieren“ (S. 107).

4.6. Datumswerte vergleichen

Es macht selten Sinn, Datumswerte zweier PDF-Dokumente miteinander zu vergleichen, aber für die wenigen Anwendungsfälle stehen doch Tags zum Datumsvergleich zur Verfügung:

```
<!-- Tags to compare date values: -->

<haveSameCreationDate />
<haveSameModificationDate />
```

Im folgenden Beispiel wird das Änderungsdatum verglichen:

```
<testcase name="haveSameModificationDate">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameModificationDate />
  </assertThat>
</testcase>
```

Die Vergleiche zweier Datumswerten finden immer in der Auflösung `DATE` (`yyyy-MM-dd`) statt.

4.7. Dokumenteneigenschaften vergleichen

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

```
<!-- Tags to compare document properties: -->

<haveSameAuthor />
<haveSameCreationDate />
<haveSameCreator />
<haveSameKeywords />
<haveSameLanguage />
<haveSameModificationDate />
<haveSameProducer />
<haveSameProperties />
<haveSameProperty />
<haveSameSubject />
<haveSameTitle />
```

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

```
<testcase name="haveSameAuthor">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameAuthor />
  </assertThat>
</testcase>
```

Der Vergleich von **Custom-Eigenschaften** ist mit dem Tag `<haveSameProperty />` möglich:

```
<testcase name="haveSameCustomProperty">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameProperty name="Company" />
    <haveSameProperty name="SourceModified" />
  </assertThat>
</testcase>
```

Mit diesem Tag können natürlich auch die Standardeigenschaften verglichen werden.

Um alle Eigenschaften zweier PDF-Dokumente miteinander zu vergleichen, gibt es noch das Tag `<haveSameProperties />`:

```
<testcase name="haveSameProperties_AllProperties">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameProperties />
  </assertThat>
</testcase>
```

4.8. Formate vergleichen

Seitenformate zweier Dokumente sind gleich, wenn Breite und Höhe gleiche Werte haben. Bei diesem Vergleich wird eine durch die ISO 216 definierte Toleranz der Seitenlängen berücksichtigt.

```
<testcase name="haveSameFormat">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameFormat />
  </assertThat>
</testcase>
```

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

```
<testcase name="haveSameFormat_OnPage2">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameFormat onPage="2" />
  </assertThat>
</testcase>
```

```
<testcase name="haveSameFormat_OnEveryPageAfter">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameFormat onEveryPageAfter="2" />
  </assertThat>
</testcase>
```

Die Möglichkeiten, Seiten zu selektieren, sind in Kapitel 13.2: „Seitenauswahl“ (S. 136) ausführlich beschrieben.

4.9. Formularfelder vergleichen

Anzahl

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

```
<testcase name="haveSameNumberOfFields">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfFields />
  </assertThat>
</testcase>
```

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:

```
<testcase name="haveSameFields_ByName">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameFields by="NAME" />
  </assertThat>
</testcase>
```

Feldeigenschaften

Wenn über die Namen von Feldern hinaus auch die weiteren Eigenschaften verglichen werden sollen, muss das Tag `<haveSameFields />` mit dem Attribut `by="PROPERTIES"` verwendet werden:

```
<testcase name="haveSameFields_ByProperties">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameFields by="PROPERTIES" />
  </assertThat>
</testcase>
```

Feldeigenschaften können mit dem Hilfsprogramm `ExtractFieldsInfo` in eine XML-Datei überführt und analysiert werden, siehe Kapitel 9.4: „Feldeigenschaften nach XML extrahieren“ (S. 108).

Feldinhalte

Und als Letztes können mit dem Tag `<haveSameFields />` und dem Attribut `by="VALUE"` auch die Inhalte der Felder verglichen werden. Gleichnamige Felder müssen gleiche Inhalte haben, sonst schlägt der Test fehl:

```
<testcase name="haveSameFields_ByValues">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameFields by="VALUE" /> ❶
  </assertThat>
</testcase>
```

- ❶ Whitespaces werden für den Vergleich „normalisiert“, siehe 13.4: „Behandlung von Whitespaces“ (S. 138).

Kombination mehrerer Tests

In einem Testfall können mehrere Vergleiche durchgeführt werden, allerdings gibt es dann Probleme mit dem Namen des Tests:

```
<testcase name="compareManyItems">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameFields by="PROPERTIES" />
    <haveSameFields by="VALUE" />
    <haveSameFonts />
    <haveSameTitle />
    <haveSameAuthor />
  </assertThat>
</testcase>
```

4.10. JavaScript vergleichen

Zwei PDF-Dokumente können „gleiches“ JavaScript enthalten. Ihr Vergleich erfolgt zeichenweise unter Vernachlässigung der Whitespaces mit dem Tag `<haveSameXMPData />`:

```
<testcase name="haveSameJavaScript">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameJavaScript />
  </assertThat>
</testcase>
```

Wenn Sie den JavaScript Code sehen wollen, können Sie ihn mit dem Hilfsprogramm `ExtractJavaScript` extrahieren. Kapitel 9.5: „JavaScript extrahieren“ (S. 109) beschreibt die nötigen Arbeitsschritte.

4.11. Layout vergleichen (gerenderte Seiten)

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

```
<testcase name="haveSameAppearance_CompleteDocument">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameAppearance on="EVERY_PAGE" />
  </assertThat>
</testcase>
```

Bestimmte Seiten können auf vielfältige Weise selektiert werden. Alle Möglichkeiten werden in Kapitel 13.2: „Seitenauswahl“ (S. 136) beschrieben.

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

```
<testcase name="haveSameAppearance_OnFirstPage_InClippingArea">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameAppearance on="FIRST_PAGE">
      <inClippingArea upperLeftX="50" upperLeftY="755"
                      width="370" height="35"
                      unit="POINTS"
                    />
    </haveSameAppearance>
  </assertThat>
</testcase>
```

Die Werte für die Definition des Seitenausschnitts können in den Einheiten `POINTS`, `MILLIMETER`, `CENTIMETER`, `INCH` und `DPI72` angegeben werden. Die Namen sind als Konstanten für das Attribut `unit="..."` definiert. Wenn keine Einheit angegeben wird, werden die Werte als `MILLIMETER` interpretiert.

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 wird folgendermaßen gebildet:

- Er beginnt mit dem vollständigen Name der Test-PDF.
- Falls die Testdatei ein Stream ist, beginnt der Name mit „_pdfunit_stream_“. Falls es sich um ein Byte-Array handelt, beginnt er mit „_pdfunit_bytearray_“. Beide Zeichenketten werden um eine Zufallszahl erweitert.
- Der zweite Teil ist ein formatiertes Datum im Format „yyyyMMdd-HH:mm:ssSSS“.
- Der letzte Teil des Dateinamens ist die Zeichenkette „.out.png“.

Aufgrund der Standardkonfiguration wird ein Diff-Image für Testdateien im selben Verzeichnis abgespeichert, in dem auch die Testdatei liegt. Diff-Images für Streams und Byte-Arrays werden im Home-Verzeichnis des laufenden Java-Prozess abgelegt. Sie können diese Einstellungen in der Datei `config.properties` ändern.

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.

4.12. Lesezeichen (Bookmarks) vergleichen

Anzahl

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

```
<testcase name="haveSameNumberOfBookmarks">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfBookmarks />
  </assertThat>
</testcase>
```

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:

- title
- namedDestination
- relatedPage
- action

```
<testcase name="haveSameBookmarks">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameBookmarks />
  </assertThat>
</testcase>
```

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. 110).

4.13. "Named Destinations" vergleichen

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

Anzahl

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

```
<testcase name="compareNumberOfNamedDestinations">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfNamedDestinations />
  </assertThat>
</testcase>
```

Namen und interne Position

Wenn die Namen von „Named Destinations“ und deren PDF-interne Positionen für zwei Dokumente gleich sein sollen, kann das auf die folgende Weise getestet werden:

```
<testcase name="compareNamedDestinations">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNamedDestinations />
  </assertThat>
</testcase>
```

4.14. PDF-Bestandteile vergleichen

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

Auch, wenn einige solcher Tests schon in den anderen Kapiteln beschrieben sind, soll an dieser Stelle ein Überblick über alle Tags gegeben werden, mit denen zählbaren Bestandteile verglichen werden können:

```
<!-- Tags to compare countable items in two PDF documents: -->
<haveSameNumberOfActions />
<haveSameNumberOfBookmarks />
<haveSameNumberOfEmbeddedFiles />
<haveSameNumberOfFields />
<haveSameNumberOfFonts />
<haveSameNumberOfImages />
<haveSameNumberOfLayers />
<haveSameNumberOfPages />
<haveSameNumberOfTaggingInfo />
```

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

```
<testcase name="haveSameNumberOfPages">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfPages />
  </assertThat>
</testcase>
```

```
<testcase name="haveSameNumberOfTaggingInfo">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfTaggingInfo />
  </assertThat>
</testcase>
```

```
<testcase name="haveSameNumberOfLayers">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfLayers />
  </assertThat>
</testcase>
```

4.15. Schriften vergleichen

Anzahl

Geht es nur um die Anzahl von Schriften, sieht ein Vergleich so aus:

```
<testcase name="haveSameNumberOfFonts">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameNumberOfFonts />
  </assertThat>
</testcase>
```

Schrifteigenschaften

Schriften in zwei PDF-Dokumenten sind gleich, wenn alle Schriftinformationen gleiche Werte enthalten. Eine Filterung der relevanten Eigenschaften nach Name, Typ usw. ist beim Vergleich nicht vorgesehen:

```
<testcase name="haveSameFonts">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameFonts />
  </assertThat>
</testcase>
```

Informationen über die Schriften eines Dokumentes können mit dem Hilfswerkzeug `ExtractFontsInfo` extrahiert werden. Siehe Kapitel 9.9: „Schrifteigenschaften nach XML extrahieren“ (S. 114).

4.16. Signaturnamen vergleichen

Der Vergleich von Signaturen zweier PDF-Dokumente bezieht sich im Release 2015.10 nur auf die Namen der Signatur:

```
<testcase name="haveSameSignatureNames">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameSignatureNames />
  </assertThat>
</testcase>
```

Weitere Vergleiche sind in zukünftigen Releases denkbar.

Umfangreiche Daten für Signaturen und Zertifikate können mit dem Hilfsprogramm `ExtractSignaturesInfo` nach XML extrahiert werden. Siehe dazu Kapitel 9.10: „Signaturdaten nach XML extrahieren“ (S. 116).

4.17. Text vergleichen

Sie können Texte auf beliebigen Seiten zweier PDF-Dokumente vergleichen. Das folgende Beispiel testet, dass der Text auf jeder Seite eines Test-Dokumentes mit dem Text auf der gleichen Seite des Master-PDF übereinstimmt. Whitespaces werden dabei ignoriert:

```
<testcase name="haveSameText_CompleteDocument">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameText on="EVERY_PAGE" />
  </assertThat>
</testcase>
```

Ein Vergleich kann auf Seiten beschränkt werden. Alle Möglichkeiten, Seiten auszuwählen, werden in Kapitel 13.2: „Seitenauswahl“ (S. 136) beschrieben:

```
<testcase name="haveSameText_OnSinglePage">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameText on="FIRST_PAGE" />
  </assertThat>
</testcase>
```

```
<testcase name="compareText_OnLastPage">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
            >
    <haveSameText on="LAST_PAGE" />
  </assertThat>
</testcase>
```

Zusätzlich kann der Textvergleich noch auf Seitenausschnitte beschränkt werden:


```
<testcase name="haveSameText_CompleteDocument_InClippingArea">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameText on="EVERY_PAGE" >
      <inClippingArea upperLeftX="50" upperLeftY="755"
                      width="370" height="35"
                      unit="POINTS"
      />
    </haveSameText>
  </assertThat>
</testcase>
```

4.18. 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

Folgendes Tag steht für Tests zur Verfügung:

```
<!-- Tag to compare XFA data: -->
<haveSameXFADData />
```

Beispiel - Resulttype Node

Das Ergebnis der XPath-Auswertung muss für beide PDF-Dokumente identisch sein:

```
<testcase name="haveSameXFADData_ResulttypeNode">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameXFADData>
      <matchingXPath expr="//default:pageSet"
                    withResultType="NODE"
                    defaultNamespace="http://www.xfa.org/schema/xfatemplate/2.6/"
      />
    </haveSameXFADData>
  </assertThat>
</testcase>
```

Wie das Beispiel zeigt, muss der erwartete Ergebnistyp angegeben werden. Die verfügbaren Ergebnistypen sind als Konstanten für das Attribut `withResultType` definiert. Es sind:

```
<!-- Result types for XPath-processing: -->
withResultType="BOOLEAN"
withResultType="NUMBER"
withResultType="NODE"
withResultType="NODESET"
withResultType="STRING"
```

Whitespaces bleiben beim Vergleich unberücksichtigt.

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.

Beispiel - Resulttype Boolean

Die XPath-Ausdrücke dürfen auch XPath-Funktionen enthalten:

```
<testcase name="haveSameXFADData_ResulttypeBoolean">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameXFADData>
      <matchingXPath expr="count(//default:field) = 3"
                    withResultType="BOOLEAN"
                    defaultNamespace="http://www.xfa.org/schema/xf-template/2.6/"
      />
    </haveSameXFADData>
  </assertThat>
</testcase>
```

Test mit „boolean“ als erwartetem Ergebnistyp (BOOLEAN) sind insofern kritisch, weil in XPath nicht zwischen „nicht gefunden“ und „false“ unterschieden werden kann.

Eine ausführliche Beschreibung für das Arbeiten mit XPath in PDFUnit steht in Kapitel 8: „XPath-Einsatz“ (S. 103).

Beispiel - Default-Namensraum

Beachten Sie den Umgang mit dem Default-Namensraum. Er muss mit dem Attribut `defaultNamespace=" . . "` deklariert werden, denn eine automatische Erkennung ist nicht möglich, weil ein XML-Dokument prinzipiell mehrere Default-Namensräume haben kann.

Sie können mehrere XPath-Vergleiche in einem Test ausführen, sogar mit wechselndem Default-Namensraum. Überlegen Sie sich aber, ob Sie den folgenden Test nicht lieber in einzelne Tests aufteilen:

```
<!--
  Test with multiple XPath expressions.

  It is not recommended to create tests in this way.
  But PDFUnit has to ensure that it works.
-->
<testcase name="haveSameXFADData_MultipleDefaultNamespaces">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameXFADData>
      <matchingXPath expr="//default:agent/@name[.='designer']"
                    withResultType="BOOLEAN"
                    defaultNamespace="http://www.xfa.org/schema/xci/2.6/"
      />
      <matchingXPath expr="//default:subform/@name[.='movie']"
                    withResultType="BOOLEAN"
                    defaultNamespace="http://www.xfa.org/schema/xf-template/2.6/"
      />
      <matchingXPath expr="//default:locale/@name[.='nl_BE']"
                    withResultType="BOOLEAN"
                    defaultNamespace="http://www.xfa.org/schema/xf-template/2.7/"
      />
    </haveSameXFADData>
  </assertThat>
</testcase>
```

4.19. XMP-Daten vergleichen

Auch die XMP-Daten zweier PDF-Dokumente werden auf XPath-Basis miteinander verglichen. Die Implementierungen der XMP- und XFA-Tests sind gleich und damit auch die Schnittstelle. Weil das vorhergehende Kapitel 3.29: „XFA Daten“ (S. 69) die Tests schon ausführlich beschreibt, soll hier nur 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. 119).

Übersicht

Folgendes Tag steht für Tests zur Verfügung:

```
<!-- Tag to compare XMP data: -->
<haveSameXMPData />
```

Beispiel

```
<testcase name="haveSameXMPData_ResulttypeNode">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameXMPData>
      <matchingXPath expr="//pdf:Producer"
                    withResultType="NODE"
      />
    </haveSameXMPData>
  </assertThat>
</testcase>
```

Die XPath-Ergebnistypen sind die gleichen, wie für XFA-Tests.

Die XPath-Ausdrücke dürfen auch XPath-Funktionen enthalten.

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.

4.20. 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 tags to compare 2 PDF documents, not described before: -->
<areBothForFastWebView />
<haveSameKeywords />
<haveSameLanguage />
<haveSameLayerNames />
<haveSameTrappingInfo />
<haveSameTaggingInfo />
```

Hier zwei Beispiele:

Fast WebView, Tagging

```
<testcase name="compareFastWebView_BothNO">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <areBothForFastWebView />
  </assertThat>
</testcase>
```

```
<testcase name="haveOfTaggingInfo">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameTaggingInfo />
  </assertThat>
</testcase>
```

Kombination von Test

Verschiedene Tests können auch kombiniert werden:

```
<testcase name="haveSameAuthorTitleFonts">
  <assertThat testDocument="test/test.pdf"
              masterDocument="master/master.pdf"
  >
    <haveSameAuthor />
    <haveSameTitle />
    <haveSameFonts />
  </assertThat>
</testcase>
```

Es schwer, für diesen Test einen guten Namen zu finden. Der hier gewählte ist für die Praxis nicht gut genug.

Kapitel 5. Tests mit mehreren Dokumenten

Mehrere Dokument gleichzeitig in einem Test

Eine Validierung kann sich auf mehrere Dokumente gleichzeitig beziehen, wie das nächste Beispiel zeigt. Ein solcher Test bricht mit dem ersten gefundenen Fehler ab.

```
<testcase name="textInMultipleDocuments">
  <assertThatEachDocument>
    <pdf name="multipleDocuments/document_en.pdf" />
    <pdf name="multipleDocuments/document_es.pdf" />
    <pdf name="multipleDocuments/document_de.pdf" />
    <hasText on="FIRST_PAGE" >
      <containing>28.09.2014</containing>
      <containing>XX-123</containing>
    </hasText>
  </assertThatEachDocument>
</testcase>
```

Die Testdokumente können auch als URL angegeben werden:

```
<testcase name="textInMultipleDocuments_AsURL">
  <assertThatEachDocument>
    <pdf name="http://localhost/.../document_en.pdf" isURL="YES" />
    <pdf name="http://localhost/.../document_es.pdf" isURL="YES" />
    <pdf name="http://localhost/.../document_de.pdf" isURL="YES" />
    <hasText on="FIRST_PAGE" >
      <containing>28.09.2014</containing>
      <containing>XX-123</containing>
    </hasText>
  </assertThatEachDocument>
</testcase>
```

Für die Mengen-Tests stehen fast alle Test-Tags zur Verfügung, die auch für Tests mit einzelnen PDF-Dokumenten existieren. Die folgende Liste zeigt die verfügbaren Tags. Ein Link hinter jedem Tag verweist auf die Beschreibung des jeweiligen Tests.

```
<!-- Tags to validate a set of PDF documents: -->

<containsImage />           3.6: „Bilder in Dokumenten“ (S. 21)

<hasAuthor />               3.8: „Dokumenteneigenschaften“ (S. 26)
<hasBookmark />             3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 46)
<hasBookmarks />            3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 46)
<hasCreationDate />         3.7: „Datum“ (S. 24)
<hasCreationDateAfter />    3.7: „Datum“ (S. 24)
<hasCreationDateBefore />   3.7: „Datum“ (S. 24)
<hasCreator />              3.7: „Datum“ (S. 24)

<hasEmbeddedFile />         3.3: „Anhänge (Attachments)“ (S. 17)
<hasEmbeddedFileContent />  3.3: „Anhänge (Attachments)“ (S. 17)
<hasEncryptionLength />     3.18: „Passwort“ (S. 49)
<hasField />                 3.11: „Formularfelder“ (S. 32)
<hasFields />                3.11: „Formularfelder“ (S. 32)
<hasFont />                  3.19: „Schriften“ (S. 50)
<hasFonts />                 3.19: „Schriften“ (S. 50)
<hasFormat />                3.10: „Format“ (S. 30)
<hasJavaScript />            3.13: „JavaScript“ (S. 40)
<hasKeywords />              3.8: „Dokumenteneigenschaften“ (S. 26)
<hasLayer />                 3.14: „Layer“ (S. 42)
<hasLayers />                3.14: „Layer“ (S. 42)
<hasLessPages />             3.20: „Seitenzahlen als Testziel“ (S. 54)
<hasLocale />                3.22: „Sprachinformation (Language)“ (S. 59)
<hasModificationDate />      3.7: „Datum“ (S. 24)
<hasModificationDateAfter /> 3.7: „Datum“ (S. 24)
<hasModificationDateBefore /> 3.7: „Datum“ (S. 24)
<hasMorePages />             3.20: „Seitenzahlen als Testziel“ (S. 54)

... continued
```

```

... continuation:

<hasNoAuthor />          3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoCreationDate />    3.7: „Datum“ (S. 24)
<hasNoCreator />         3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoKeywords />        3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoLocale />          3.22: „Sprachinformation (Language)“ (S. 59)
<hasNoModificationDate /> 3.7: „Datum“ (S. 24)
<hasNoProducer />        3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoProperty />        3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoSubject />         3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoText />            3.23: „Texte“ (S. 60)
<hasNoTitle />           3.8: „Dokumenteneigenschaften“ (S. 26)
<hasNoXFADData />        3.29: „XFA Daten“ (S. 69)
<hasNoXMPData />         3.30: „XMP-Daten“ (S. 72)

<hasOwnerPassword />     3.18: „Passwort“ (S. 49)
<hasPermission />        3.5: „Berechtigungen“ (S. 20)
<hasProducer />          3.8: „Dokumenteneigenschaften“ (S. 26)
<hasProperty />          3.8: „Dokumenteneigenschaften“ (S. 26)
<hasSignature />         3.21: „Signaturen und Zertifikate“ (S. 54)
<hasSignatures />        3.21: „Signaturen und Zertifikate“ (S. 54)
<hasSignedSignatureFields /> 3.21: „Signaturen und Zertifikate“ (S. 54)
<hasSubject />           3.8: „Dokumenteneigenschaften“ (S. 26)
<hasText />              3.23: „Texte“ (S. 60)
<hasTitle />             3.8: „Dokumenteneigenschaften“ (S. 26)
<hasTrappingInfo />      3.27: „Trapping-Info“ (S. 67)
<hasUnsignedSignatureFields /> 3.21: „Signaturen und Zertifikate“ (S. 54)
<hasVersion />           3.28: „Version“ (S. 68)
<hasXFADData />          3.29: „XFA Daten“ (S. 69)
<hasXMPData />           3.30: „XMP-Daten“ (S. 72)

<isCertified />          3.31: „Zertifiziertes PDF“ (S. 74)
<isLinearizedForFastWebView /> 3.9: „Fast Web View“ (S. 29)
<isSigned />             3.21: „Signaturen und Zertifikate“ (S. 54)
<isTagged />             3.26: „Tagging“ (S. 66)

... (end of list)

```

Wünschen Sie weitere Tests, schreiben Sie Ihren Wunsch an [info\[at\]pdfunit.com](mailto:info[at]pdfunit.com).

Kapitel 6. PDFUnit-Monitor

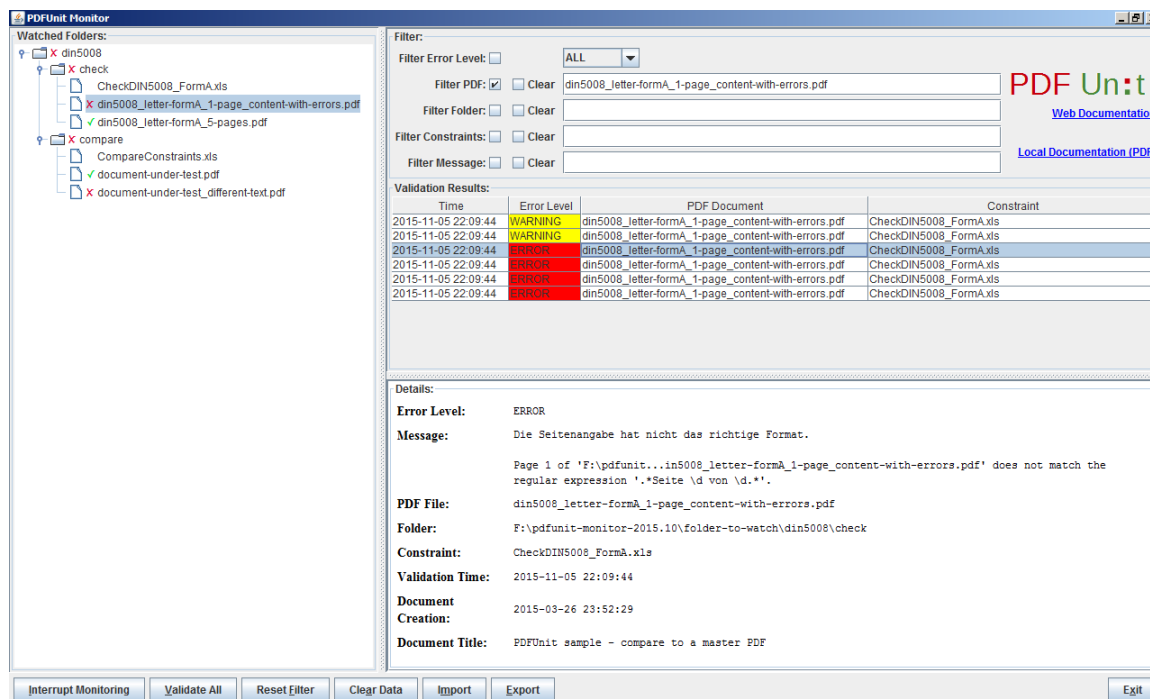
Der PDFUnit-Monitor ist eine graphische Anwendung, um Tests für PDF-Dokumente anzustoßen und das Ergebnis anzeigen zu lassen. Die Zielgruppe für die Anwendung sind Nicht-Programmierer.

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

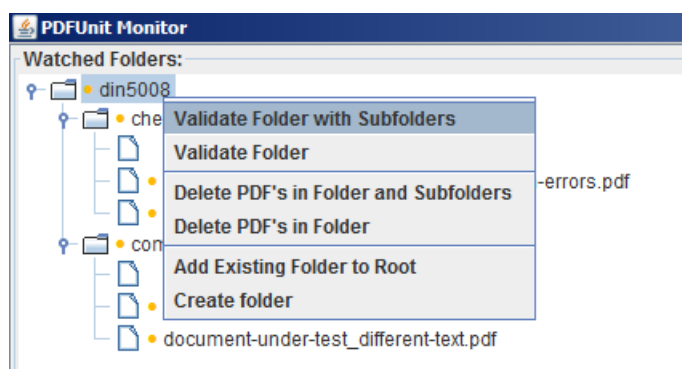
Der PDFUnit-Monitor überwacht alle PDF-Dokumente unterhalb eines definierten Verzeichnisses und prüft die dortigen Dokumente gegen Regeln, die in Excel-Dateien hinterlegt sind, die ebenfalls in den überwachten Verzeichnissen liegen müssen. Erfüllt ein PDF alle Regeln, wird es in der Baumstruktur mit 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.

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.



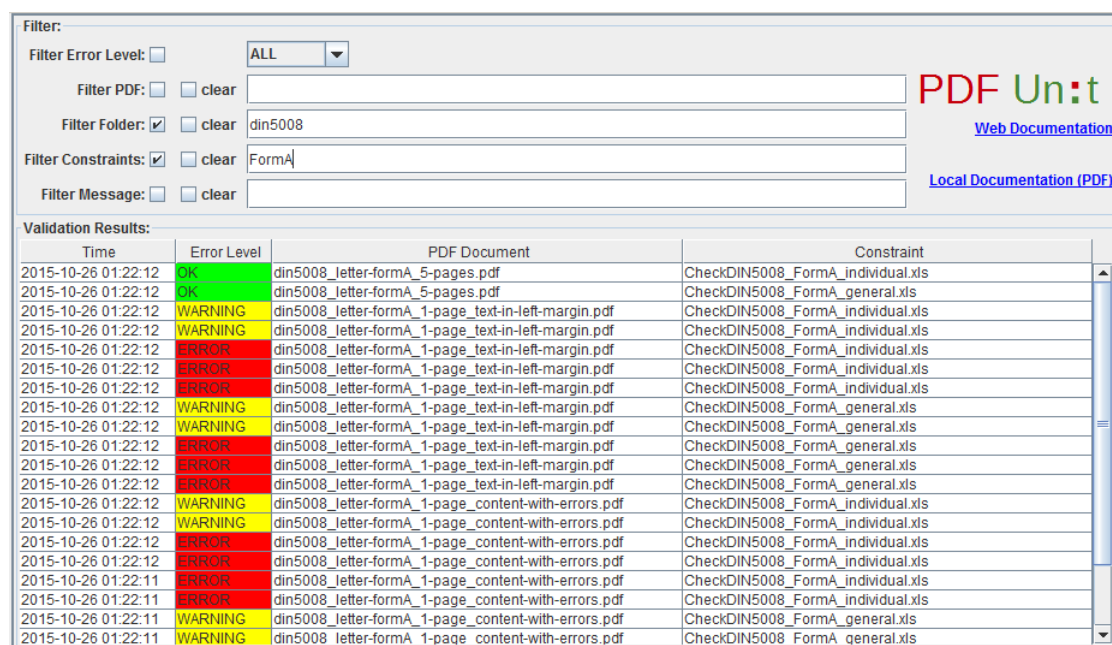
Ein Doppelklick auf ein PDF-Dokument in der Baumstruktur öffnet das Dokument mit der Standardanwendung des Betriebssystems. Gleiches gilt für einen Doppelklick auf eine Excel-Datei.

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 in der Liste selektiert wird.

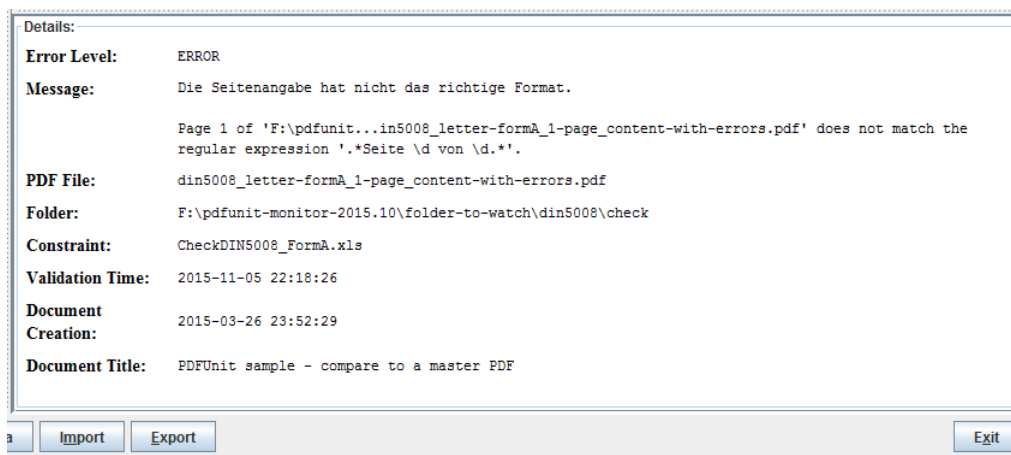


Die Liste der Fehler kann über Filter eingeschränkt werden. Die Filtermöglichkeiten stehen mit der Verzeichnisstruktur in Verbindungen. Wird eine Excel-Datei in der Struktur angeklickt, wird ein Filter mit diesem Namen aktiviert und es werden alle PDF-Dokumente, die mit dieser Excel-Datei geprüft wurden, angezeigt. Filter gibt es für PDF-Dokumente, Verzeichnisse, Excel-Dateien, und Fehlermeldungen.

Wird in der Liste ein PDF-Dokument oder eine Excel-Datei 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.

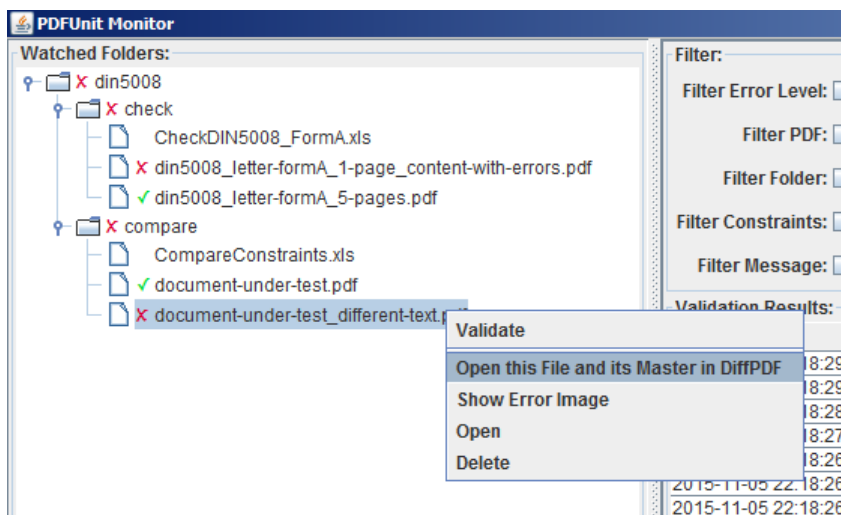


Der erste Teil der Fehlermeldung stammt aus der Excel-Datei und wird von der Person, die die Tests erstellt, geschrieben. Weitere Teile der Meldung stammen vom Testwerkzeug PDFUnit. Neben der eigentlichen Fehlermeldung werden weitere nützliche Informationen über das PDF-Dokument, die Regeldatei und den Testzeitraum angezeigt.

Die Fehlermeldungen von PDFUnit sind momentan auf englisch verfasst, können aber mit wenig Aufwand auch auf deutsch zur Verfügung gestellt werden.

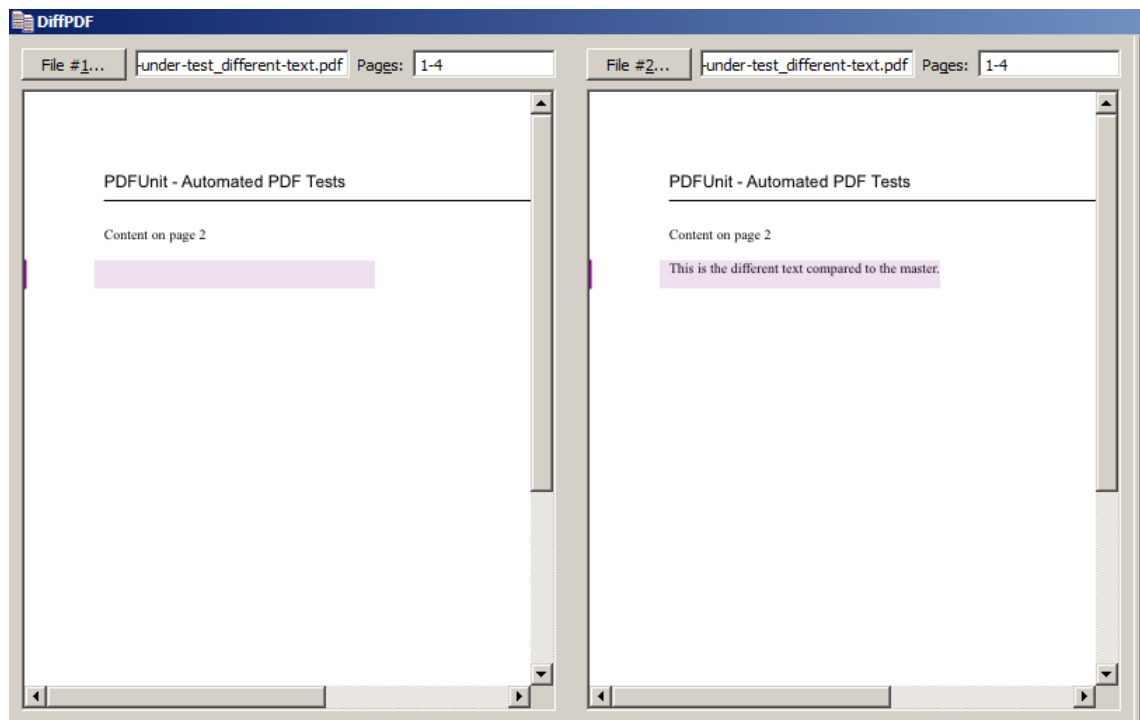
Vergleich eines PDF-Dokumentes gegen eine Vorlage

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 Master-Dokument, wird der Name des Test-Dokuments in der Verzeichnisstruktur mit einem roten Kreuz markiert. Über die rechte Maustaste kann anschließend 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 7. 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 es fehlen noch Tests mit hebräischen und japanischen PDF-Dokumenten. Insofern kann die eingangs gestellte Frage nicht abschließend beantwortet werden.

Prinzipiell tun Sie gut daran, sämtliche Werkzeuge auf UTF-8 zu konfigurieren, wenn Sie Unicode-Daten verarbeiten müssen.

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 mit \uXXXX schreiben, wie hier das Copyright-Zeichen „©“ als \u00A9:

```
<testcase name="hasProducer_CopyrightAsUnicode">
  <assertThat testDocument="unicode/unicode_producer.pdf">
    <hasProducer>
      <!-- 'copyright' -->
      <matchingComplete>txt2pdf v7.3 \u00A9 SANFACE Software 2004</matchingComplete>
    </hasProducer>
  </assertThat>
</testcase>
```

Längere Unicode-Texte

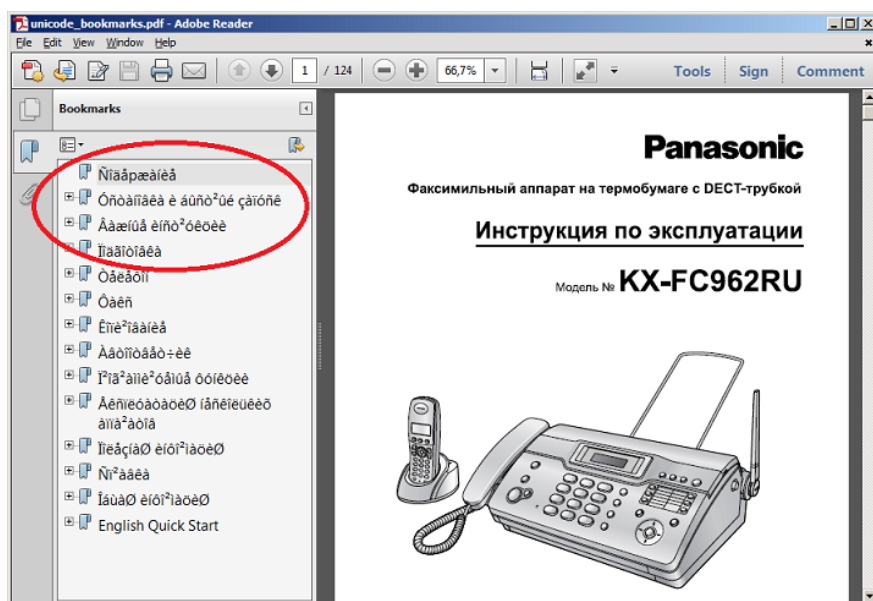
Es wäre nun zu mühsam, für längere Texte den Hex-Code aller Buchstaben herauszufinden. Deshalb stellt PDFUnit das kleine Programm `ConvertUnicodeToHex` zur Verfügung. Übergeben Sie den ausländischen Text als String an das Werkzeug, entnehmen Sie der daraus erzeugten Datei anschließend den Hex-Code und fügen ihn in Ihr Testprogramm ein. Eine genaue Beschreibung steht in Kapitel 9.12: „Unicode-Texte in Hex-Code umwandeln“ (S. 118). Das Test mit Unicode sieht dann so aus:

```
<testcase name="hasSubject_Greek">
  <assertThat testDocument="unicode/unicode_subject.pdf">
    <hasSubject>
      <matchingComplete>
        ##### ## ### ##### / #####
      </matchingComplete>
    </hasSubject>
  </assertThat>
</testcase>
```

- ❶ Wenn Sie an dieser Stelle keinen griechischen Text sehen, liegt das daran, dass das darstellende System (PDF, eBook oder HTML) die Schriftart nicht unterstützt.

PDF mit Unicode gegen XML-Dateien vergleichen

XML- und XPath-basierte Tests funktionieren auch mit Dateien, die Unicode enthalten, wie z.B. die nach XML extrahierten Bookmarks im folgenden Beispiel:



```
<!--
  This test needs the following setting before starting ANT:
  set JAVA_TOOL_OPTIONS=-Dfile.encoding=UTF-8
-->

<testcase name="hasBookmarks_MatchingXML">
  <assertThat testDocument="unicode/unicode_bookmarks.pdf">
    <hasBookmarks>
      <matchingXML file="unicode/unicode_bookmarks.xml" />
    </hasBookmarks>
  </assertThat>
</testcase>
```

- ❶ Die Codepage kann über die Umgebungsvariable „file.encoding“ gesetzt werden.
- ❷ Die Bookmarks wurden mit dem Hilfsprogramm `ExtractBookmarks` nach XML exportiert.

Unicode in XPath-Ausdrücken

In Kapitel 8: „XPath-Einsatz“ (S. 103) wird beschrieben, wie PDFUnit-Tests zusammen mit XPath funktionieren. Auch die XPath-Ausdrücke können Unicode enthalten:

```
<testcase name="hasBookmarks_MatchingXPath">
  <assertThat testDocument="unicode/unicode_bookmarks.pdf">
    <hasBookmarks>
      <!-- The line is wrapped for printing: -->
      <matchingXPath expr="//Title[@Action][.='\'\\u00D1\\u00EE\\u00E4
        \\u00E5p\\u00E6\\u00E0
        \\u00ED\\u00E8\\u00E5']" />
    </hasBookmarks>
  </assertThat>
</testcase>
```

UTF-8 File-Encoding für Shell-Skripte

Jedes Java-Programm, das Dateien verarbeitet, also auch PDFUnit, ist 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 abbrechen. 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**:

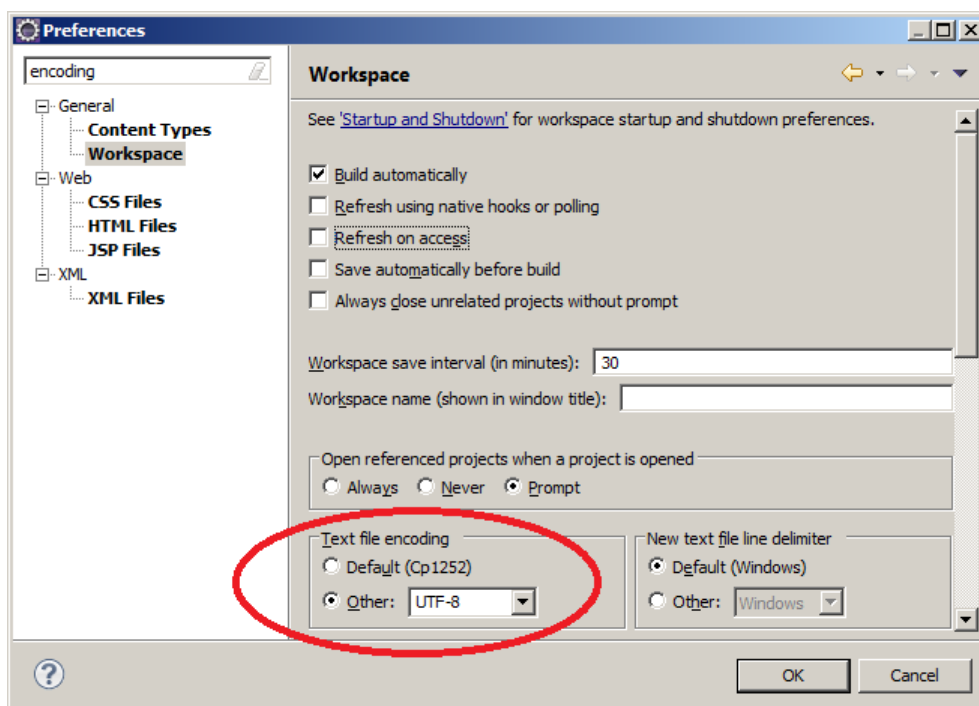
```
// does not work for ANT:  
ant -Dfile.encoding=UTF-8
```

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
```

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 gespeichert werden.

Fehlermeldungen und Unicode

Wenn Tests fehlschlagen, die auf Unicode-Inhalte testen, kann es sein, dass Eclipse oder ein Browser die Fehlermeldung nicht ordentlich darstellen. Ausschlaggebend dafür ist das File-Encoding der Ausgabe, das von PDFUnit selber nicht beeinflusst werden kann. Wenn Sie in ANT 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 erwartete 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.

Adobe Reader® shows: "Ebene 1(4)"
The used String is: "Ebene _XXX"
-->

<testcase name="hasLayer_NameContainingUnicode_UTF16_ErrorIntended"
errorExpected="YES"
>
  <assertThat testDocument="unicode/unicode_layerName.pdf">
    <hasLayer>
      <withName>
        <matchingComplete>
          \u00fe\u00ff\u0000E\u0000b\u0000e\u0000n\u0000e\u0000 \u0000_XXX
        </matchingComplete>
      </withName>
    </hasLayer>
  </assertThat>
</testcase>
```

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)

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:

```
<!--
The content of the node value terminates with the
Unicode value 'non-breaking space'.
-->

<testcase name="nodeValueWithUnicodeValue">
  <assertThat testDocument="xfa/xfaBasicToggle.pdf">
    <hasXFADData>
      <!-- The line is wrapped for printing: -->
      <withNode tag="default:p[7]"
        value="The code for creating the toggle behavior involves
        switching the border between raised and lowered,
        and maintaining the button's\u00A0"
        defaultNamespace="http://www.w3.org/1999/xhtml"
      />
    </hasXFADData>
  </assertThat>
</testcase>
```

Kapitel 8. 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 Spezialkapiteln.

```
<!-- Overview over XPath related test facilities: -->
<hasBookmarks><matchingXPath />... 3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 46)
<hasFields><matchingXPath />... 3.11: „Formularfelder“ (S. 32)
<hasFonts><matchingXPath />... 3.19: „Schriften“ (S. 50)
<hasSignatures><matchingXPath />... 3.21: „Signaturen und Zertifikate“ (S. 54)
<hasXFADData><matchingXPath />... 3.29: „XFA Daten“ (S. 69)
<hasXMPData><matchingXPath />... 3.30: „XMP-Daten“ (S. 72)

<!-- Comparing two documents using XPath: -->
<haveSameXFADData><matchingXPath />... 4.18: „XFA-Daten vergleichen“ (S. 89)
<haveSameXMPData><matchingXPath />... 4.19: „XMP-Daten vergleichen“ (S. 90)
```

Intern verwendet PDFUnit XPath auch für die Implementierung anderer Tests.

Interne Nutzung von XMLUnit

Alle Vergleiche von XML-Werten werden mit XMLUnit (<http://xmlunit.sourceforge.net>) durchgeführt. Dabei werden die Syntaxregeln von XML berücksichtigt werden, wie z.B.:

- Die Reihenfolge von Attributen spielt keine Rolle.
- Whitespaces zwischen Knoten werden ignoriert.

Weitere Regeln für „Canonisches XML“ sind bei Wikipedia (http://de.wikipedia.org/wiki/Canonical_XML) gut beschrieben.

Die allgemeine Konfiguration von XMLUnit wird auf der Projektseite <http://xmlunit.sourceforge.net/userguide/html/index.html#Configuring%20XMLUnit> selber beschrieben. PDFUnit nutzt folgende Konfigurationsparameter:

```
XMLUnit.setXSLTVersion("2.0");
XMLUnit.setNormalizeWhitespace(true);
XMLUnit.setIgnoreWhitespace(true);
XMLUnit.setIgnoreAttributeOrder(true);
XMLUnit.setIgnoreComments(true);
```

Daten als XML extrahieren

Für alle Teile eines PDF-Dokumentes, für die es XML/XPath-Tests gibt, werden Extraktionsprogramme zur Verfügung gestellt:

```
// Utilities to extract XML from PDF:
com.pdfunit.tools.ExtractBookmarks
com.pdfunit.tools.ExtractFieldsInfo
com.pdfunit.tools.ExtractFontsInfo
com.pdfunit.tools.ExtractSignaturesInfo
com.pdfunit.tools.ExtractXFADData
com.pdfunit.tools.ExtractXMPData
```

Die Hilfsprogramme werden im Kapitel 9.1: „Allgemeine Hinweise für alle Hilfsprogramme“ (S. 105) genauer beschrieben.

Namensräume mit Präfix

Namensräume, für die ein Präfix definiert ist, werden von PDFUnit automatisch erkannt. Das gilt sowohl für XML-Dateien, als auch für PDF-internes XML.

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.
-->
<testcase name="hasXFADData_UsingDefaultNamespace">
  <assertThat testDocument="xfa/xfa-enabled.pdf">
    <hasXFADData>
      <withNode tag="foo:log/foo:to"
                value="memory"
                defaultNamespace="http://www.xfa.org/schema/xci/2.6/"
      />
    </hasXFADData>
  </assertThat>
</testcase>
```

Beachten Sie, dass in diesem Beispiel einmal das Präfix `foo` und einmal das Präfix `bar` für den gleichen Namensraum verwendet wird. In der Praxis verwenden Sie bitte nur ein einziges und nicht „foo“ oder „bar“.

Ergebnistypen von XPath-Ausdrücken

Die Auswertung von XPath-Ausdrücken führt zu bestimmten XPath-Datentypen, die beim Vergleich der XFA-Daten oder XMP-Daten zweier PDF-Dokumente mitgegeben werden müssen. Die verfügbaren Ergebnistypen sind als Konstanten für das Attribut `withResultType` definiert:

```
<!-- Result types for XPath-processing: -->
withResultType="BOOLEAN"
withResultType="NUMBER"
withResultType="NODE"
withResultType="NODESET"
withResultType="STRING"
```

Test mit dem Ergebnistypen `BOOLEAN` sind insofern kritisch, weil XPath nicht zwischen „nicht gefunden“ und „false“ unterscheiden kann. Versuchen Sie mit einem anderen XPath-Ausdruck.

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.

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. 118)
ExtractBookmarks         9.6: „Lesezeichen nach XML extrahieren“ (S. 110)
ExtractEmbeddedFiles     9.2: „Anhänge extrahieren“ (S. 105)
ExtractFieldsInfo        9.4: „Feldeigenschaften nach XML extrahieren“ (S. 108)
ExtractFontsInfo         9.9: „Schrifteigenschaften nach XML extrahieren“ (S. 114)
ExtractImages            9.3: „Bilder aus PDF extrahieren“ (S. 107)
ExtractJavaScript         9.5: „JavaScript extrahieren“ (S. 109)
ExtractNamedDestinations 9.11: „Sprungziele nach XML extrahieren“ (S. 117)
ExtractSignaturesInfo    9.10: „Signaturdaten nach XML extrahieren“ (S. 116)
ExtractXFADData          9.13: „XFA-Daten nach XML extrahieren“ (S. 119)
ExtractXMPData           9.14: „XMP-Daten nach XML extrahieren“ (S. 119)
RenderPdfClippingAreaToImage 9.8: „PDF-Seite ausschnittsweise in PNG umwandeln“ (S. 112)
RenderPdfToImages        9.7: „PDF-Dokument seitenweise in PNG umwandeln“ (S. 111)
```

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. Schließlich ist es dann ja keine Ausgabedatei mehr.

In den folgenden Kapiteln werden Batchdateien abgebildet, die zeigen, wie die Programme gestartet werden. Die Batchdateien sind Teil des Releases, Sie müssen aber Teile 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 `support[at]pdfunit.com`.

9.2. Anhänge extrahieren

Das Hilfsprogramm `ExtractEmbeddedFiles` erstellt für jede in einem PDF-Dokument enthaltene Datei eine separate Ausgabedatei.

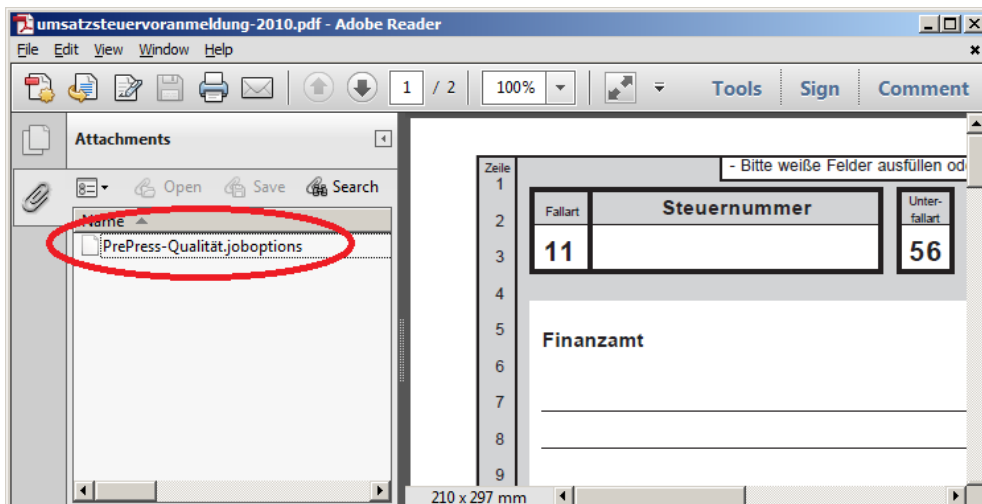
Der Export erfolgt byte-weise, dadurch werden alle Dateiformate unterstützt.

Aufruf

```
::  
:: Extract embedded files from a PDF document. Each in a separate output file.  
::  
  
@echo off  
setlocal  
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%  
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%  
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%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
```

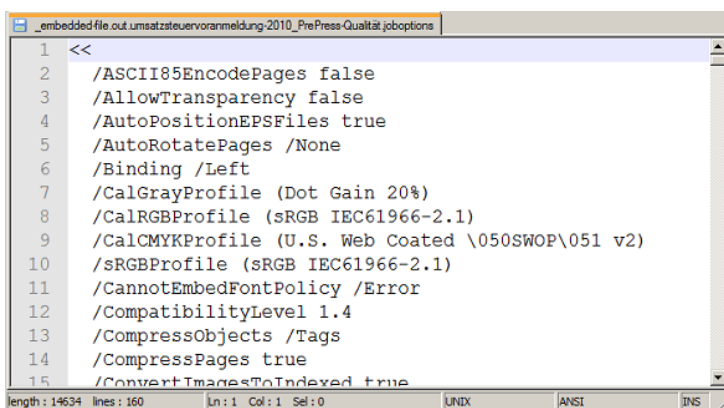
Eingabe

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`.



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 3.6: „Bilder in Dokumenten“ (S. 21) beschrieben.

Aufruf

```
::
:: Extract all images of a PDF document into a PNG file for each image.
::

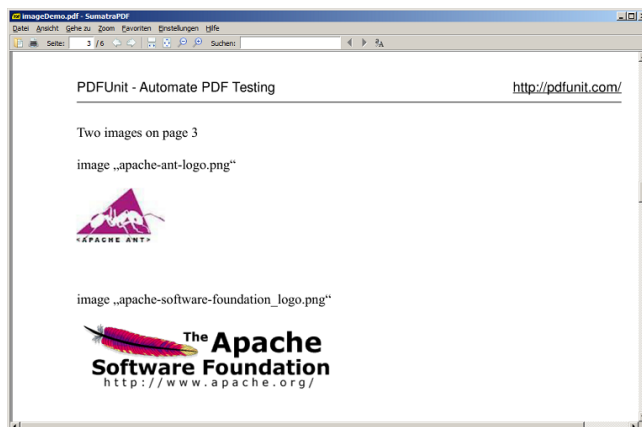
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

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 die zwei Dateien:



```
# created images:

.\tmp\_exported-image_imageDemo_4.out.png ❶
.\tmp\_exported-image_imageDemo_12.out.png ❷
```

❶❷ Die Nummer im Dateinamen entspricht der Objekt-Nummer innerhalb des PDF-Dokumentes.

9.4. Feldeigenschaften nach XML extrahieren

Das Hilfsprogramm `ExtractFieldsInfo` erstellt eine XML-Datei mit zahlreichen Informationen zu allen Formularfeldern. Der Inhalt eines Feldes wird **nicht** extrahiert!

Die Tests auf Feldeigenschaften werden in Kapitel 3.11: „Formularfelder“ (S. 32) beschrieben.

Aufruf

```
::  
:: Extract formular fields from a PDF document into an XML file  
::  
  
@echo off  
setlocal  
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%  
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%  
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%  
  
set TOOL=com.pdfunit.tools.ExtractFieldsInfo  
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" ?>
<fieldlist>
  <!-- Width and height values are given as millimeters. -->
  <field name="ageField" type="text"
    width="30" height="22"
    isEditable="true" isRequired="false"
    isPrintable="false" isVisible="false"
    isHidden="false" isHiddenButPrintable="false"
    isVisibleButNotPrintable="false" isExportable="true"
    isPasswordField="false" isMultiLineField="false"
  />
  <field name="reset" type="button"
    width="35" height="15"
    isEditable="true" isRequired="true"
    isPrintable="false" isVisible="false"
    isHidden="false" isHiddenButPrintable="true"
    isVisibleButNotPrintable="true" isExportable="true"
  />
  <!-- 3 fields deleted for presentation -->
</fieldlist>
```

- ❶ Werte für Breite und Höhe werden in Millimeter angegeben
- ❷❸ Einige Attribute werden typabhängig erstellt. Das Attribut „isMultiLineField“ beispielsweise gibt es nur für Felder vom Typ „text“.

9.5. JavaScript extrahieren

Das Hilfsprogramm `ExtractJavaScript` extrahiert JavaScript aus einem PDF-Dokument und erstellt daraus eine Textdatei. Das Kapitel 3.13: „JavaScript“ (S. 40) 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/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%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. 108) als Beispiel erhalten 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 `_javascript_javascriptForFields.out.txt` 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 im nächsten Release ein neues Hilfsprogramm geben.

9.6. Lesezeichen nach XML extrahieren

PDFUnit enthält das Hilfsprogramm `ExtractBookmarks`. Es exportiert Lesezeichen/Bookmarks von PDF-Dokumenten nach XML. Das Kapitel 3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 46) beschreibt die Verwendung der erzeugten XML-Datei für Bookmarks-Tests.

Aufruf

```
::
:: Extract bookmarks from a PDF document into an XML file
::

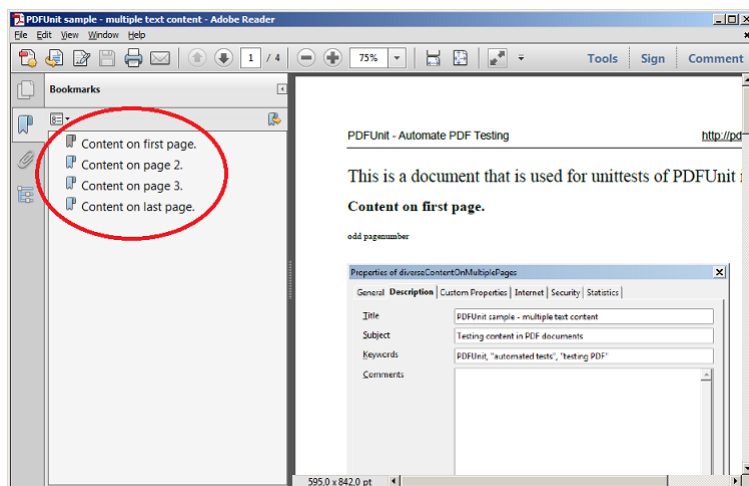
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

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` kann für XML-basierte Tests verwendet werden:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookmark>
  <Title Action="GoTo" Page="1 XYZ 56.7 745 0" >Content on first page.</Title>
  <Title Action="GoTo" Page="2 XYZ 56.7 745 0" >Content on page 2.</Title>
  <Title Action="GoTo" Page="3 XYZ 56.7 733.5 0" >Content on page 3.</Title>
  <Title Action="GoTo" Page="4 XYZ 56.7 733.5 0" >Content on last page.</Title>
</Bookmark>
```

PDFUnit nutzt intern die statische Methode `SimpleBookmark.getBookmark(PdfReader)` von `iText`. Herzlichen Dank an die Entwickler.

9.7. PDF-Dokument seitenweise in PNG umwandeln

Wenn Sie formatierten Text testen wollen, geht das nur so, dass die PDF-Seite in ein Bild gerendert wird und dieses Bild anschließend gegen eine Bildvorlage verglichen wird. Das Kapitel 3.15: „Layout - gerenderte volle Seiten“ (S. 44) beschreibt Layout-Tests unter Verwendung gerendeter Seiten und das Hilfsprogramm `RenderPdfToImages` rendert ein PDF-Dokument seitenweise in PNG-Dateien.

Aufruf

```
::
:: Render PDF into image files. Each page as a file.
::

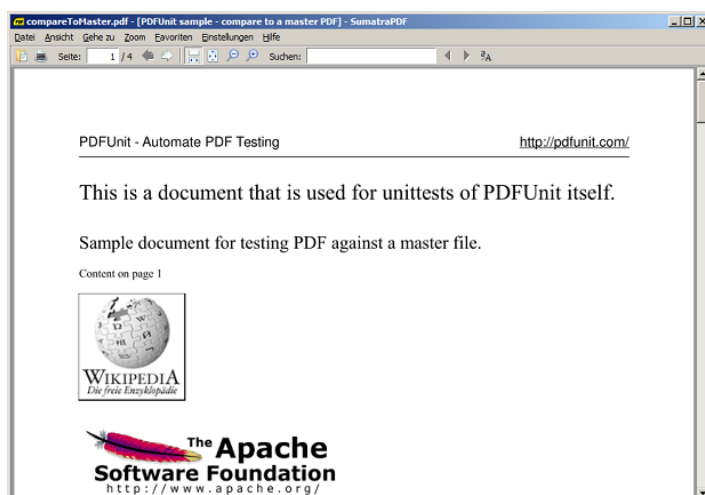
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/jpedal/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.RenderPdfToImages
set OUT_DIR=./tmp
set IN_FILE=compareToMaster.pdf
set PASSWD=

java %TOOL% %IN_FILE% %OUT_DIR% %PASSWD%
endlocal
```

Eingabe

Die Eingabe-Datei `compareToMaster.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:

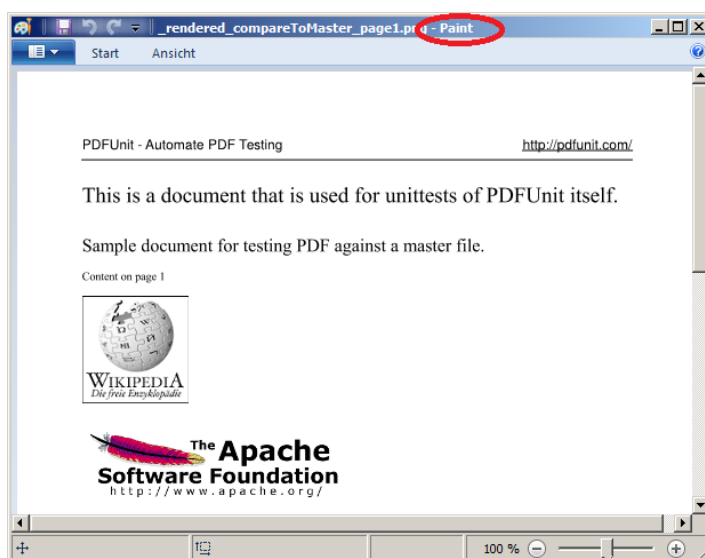


Ausgabe

Nach dem Rendern sind 4 Dateien entstanden:

```
.\tmp\_rendered_compareToMaster_page1  
.\tmp\_rendered_compareToMaster_page2  
.\tmp\_rendered_compareToMaster_page3  
.\tmp\_rendered_compareToMaster_page4
```

Von diesen sieht die erste Datei als Bild genauso aus, wie im PDF-Reader.



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.

PDFUnit nutzt intern die Klasse `org.jpedal.PdfDecoder` aus dem Projekt „jPedal“ (<http://www.idrsolutions.com/>). Danke an die Entwickler.

9.8. PDF-Seite ausschnittsweise in PNG umwandeln

Die Gründe, um Tests mit gerenderten Ausschnitten einer PDF-Seite durchzuführen, sind in Kapitel 3.16: „Layout - gerenderte Seitenausschnitte“ (S. 45) beschrieben. Um den Seitenausschnitt „richtig“

zu ermitteln, stellt PDFUnit das kleine Hilfsprogramm `RenderPdfClippingAreaToImage` 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/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/jpedal/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%
set CLASSPATH=./lib/aspectj-1.8.0/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.RenderPdfClippingAreaToImage
set OUT_DIR=./tmp
set PAGENUMBER=1
set IN_FILE=documentForTextClipping.pdf
set PASSWD=

:: Format unit can only be 'mm' or 'points'
set FORMATUNIT=points

:: Put these values into your test code:
set UPPERLEFTX=50
set UPPERLEFTY=130
set WIDTH=170
set HEIGHT=25

java %TOOL% %IN_FILE% %PAGENUMBER% %OUT_DIR% ❶
    %FORMATUNIT% %UPPERLEFTX% %UPPERLEFTY% %WIDTH% %HEIGHT% %PASSWD%
endlocal

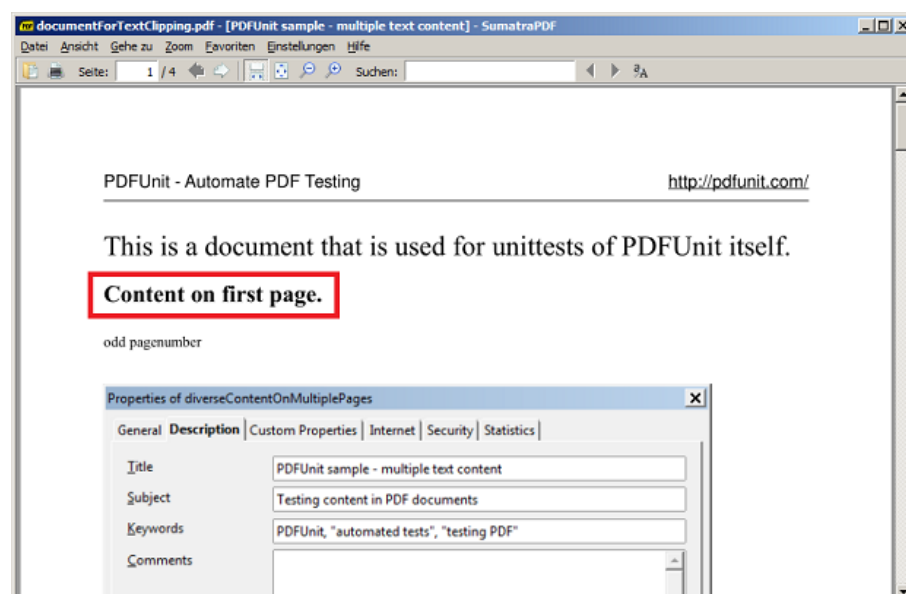
```

❶ Zeilenumbruch nur für diese Dokumentation

Die 4 Werte, die den Ausschnitt beschreiben, müssen entweder Millimeter `mm` oder Points `points` sein. Sie werden einen Taschenrechner bemühen müssen, um an die richtigen Werte zu kommen.

Eingabe

Die Eingabedatei `documentForTextClipping.pdf` enthält im oberen Bereich den Text: „Content on first page.“



Ausgabe

Content on first page.

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 `RenderPdfClippingAreaToImage` 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
```

PDFUnit nutzt für dieses Hilfsprogramm intern Funktionen aus dem Projekt „jPedal“ (<http://www.idrsolutions.com/>). Nochmals Danke an die Entwickler.

9.9. Schrifteigenschaften nach XML extrahieren

Wie in Kapitel 3.19: „Schriften“ (S. 50) beschrieben, bergen Schriften eine Komplexität, die ruhig öfter getestet werden sollte. Sie können alle Informationen über Schriften mit dem Hilfsprogramm `ExtractFontsInfo` als XML-Datei aus PDF extrahieren, um darauf mit XPath vielseitige Tests zu entwickeln.

Der Algorithmus, der die XML-Datei erzeugt, ist der gleiche, der auch von den PDFUnit-Tests verwendet wird.

Aufruf

```
::
:: Extract information about fonts used in a PDF document into an XML file
::

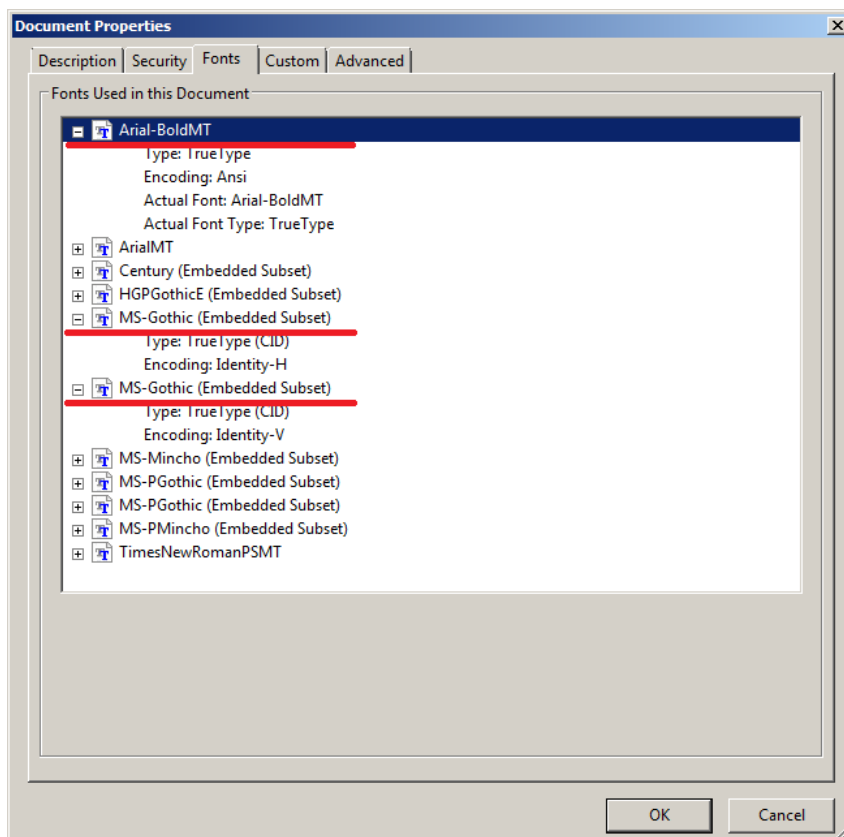
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractFontsInfo
set OUT_DIR=./tmp
set IN_FILE=fonts_11_japanese.pdf
set PASSWD=

java %TOOL% %IN_FILE% %OUT_DIR% %PASSWD%
endlocal
```

Eingabe

Der Adobe Reader® zeigt folgende Schriften des japanischen PDF-Dokumentes `fonts_11_japanese.pdf`:



Ausgabe

Die markierten Namen sind auch in der erzeugten Ausgabedatei `_fontinfo_fonts_11_japanese.out.xml` enthalten:

```
<?xml version="1.0" encoding="UTF-8" ?>
<fontlist>
...
<font name="Arial-BoldMT"          baseFontName="Arial-BoldMT"
      type="TrueType"              embedded="false"
      encoding="WinAnsiEncoding"    convertibleToUnicode="false"
/>
<font name="MDOLLI+MS-Gothic"       baseFontName="MS-Gothic"
      type="CIDFontType2"           embedded="true"
      convertibleToUnicode="false"
/>
<font name="MDOLLI+MS-Gothic"       baseFontName="MS-Gothic"
      type="Type0"                  embedded="false"
      encoding="Identity-H"          convertibleToUnicode="true"
/>
...
</fontlist>
```

Die XML-Datei listet jedes Subset einer Schriftart einzeln auf. Dadurch ergeben sich Abweichungen von der Anzeige durch den Adobe Reader®.

Sie können die Datei beliebig formatieren, ohne dass dadurch die Tests beeinflusst werden, weil Whitespaces zwischen Elementen und Attributen nach den Regeln von XML sowieso keine Rolle spielen.

Auf der Basis dieser Datei können Sie mit einem geeigneten XPath-Ausdruck beliebige Eigenschaften von Schriften testen.

9.10. Signaturdaten nach XML extrahieren

Signaturen und Zertifikate enthalten eine große Zahl an Informationen, von denen nur einige über direkte Tests erreichbar sind. Die restlichen Daten können über XPath getestet werden. Das Kapitel 3.21: „Signaturen und Zertifikate“ (S. 54) beschreibt die Tests mit Signaturen und Zertifikaten.

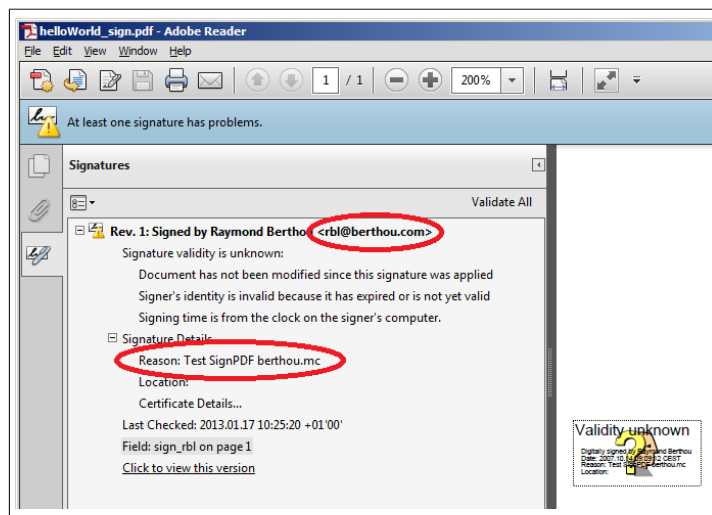
Mit dem folgenden Skript starten Sie die Extraktion:

Aufruf

```
::  
:: Extract infos about signatures and certificates of a PDF document as XMLe  
::  
  
@echo off  
setlocal  
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%  
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%  
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%  
  
set TOOL=com.pdfunit.tools.ExtractSignaturesInfo  
set OUT_DIR=./tmp  
set IN_FILE=signed/helloWorld_sign.pdf  
set PASSWD=  
  
java %TOOL% %IN_FILE% %OUT_DIR% %PASSWD%  
endlocal
```

Eingabe

Der Adobe Reader® zeigt die Signaturdaten für die Datei `helloWorld_sign.pdf` an:



Ausgabe

Die erzeugte Datei `_signatureinfo_helloWorld_sign.out.xml` ist sehr umfangreich, ein Ausschnitt wird hier als Bild gezeigt:



Die Tests zu Signaturen und Zertifikaten unterliegen momentan (Release 2015.10) noch einer größeren Weiterentwicklung. Das kann zu Änderungen der XML-Dateien führen.

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 nach XML extrahieren und anschließend in XPath-basierten Tests verwenden. Das Kapitel 3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 46) 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/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%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"?>
<Destination>
  <Name Page="3 XYZ 36 764 0">destination2.2</Name>
  <Name Page="3 XYZ 36 800 0">destination2_no_blank<</Name>
  <Name Page="3 XYZ 36 782 0">destination2.1</Name>
  <Name Page="2 XYZ 36 800 0">destination1</Name>
  <Name Page="4 XYZ 36 800 0">destination3 with blank</Name>
</Destination>
```

PDFUnit verwendet intern `SimpleNamedDestination.getNamedDestination(...)` von iText (<http://www.itextpdf.com>). Auch hier einen Dank an die Entwickler.

9.12. Unicode-Texte in Hex-Code umwandeln

Java „kann Unicode“. XML „kann Unicode“. Somit kann auch PDFUnit Unicode. In Kapitel 7: „Unicode“ (S. 99) wird das Thema Unicode ausführlich beschrieben.

Dieses Kapitel beschreibt ein kleines Werkzeug, das einen Unicode-String in seinen ASCII-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. Falls Sie das überhaupt dürfen.

Das Programm `ConvertUnicodeToHex` konvertiert eine beliebige Zeichenkette in ASCII-Code und „escaped“ dabei alle 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-2015.10/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ConvertUnicodeToHex
set OUT_DIR=./tmp
set IN_FILE=convert-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 `convert-unicode-to-hex.in.txt` enthält folgende Werte:

```
äöü € @
```

Ausgabe

Der Name wird automatisch aus dem Namen der Eingabedatei abgeleitet. Die Datei `_convert-unicode-to-hex.out.txt` enthält dann den Hex-Code:

```
#Unicode created by com.pdfunit.tools.ConvertUnicodeToHex
#Wed Jan 16 21:50:04 CET 2013
convert-unicode-to-hex.in_as-ascii=\u00E4\u00F6\u00FC \u20AC @
```

Die Ausgabedatei wird im Encoding der Java-Runtime erstellt. Dazu wird der Wert der Umgebungsvariablen `file.encoding` ausgelesen.

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 `ExtractXFADData` können Sie XFA-Daten exportieren und anschließend zusammen mit XPath in Tests verwenden, wie es in Kapitel 3.29: „XFA Daten“ (S. 69) gezeigt wird.

Aufruf

```
::
:: Extract XFA data of a PDF document as XML
::

@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.ExtractXFADData
set OUT_DIR=./tmp
set IN_FILE=xfa-enabled.pdf
set PASSWD=

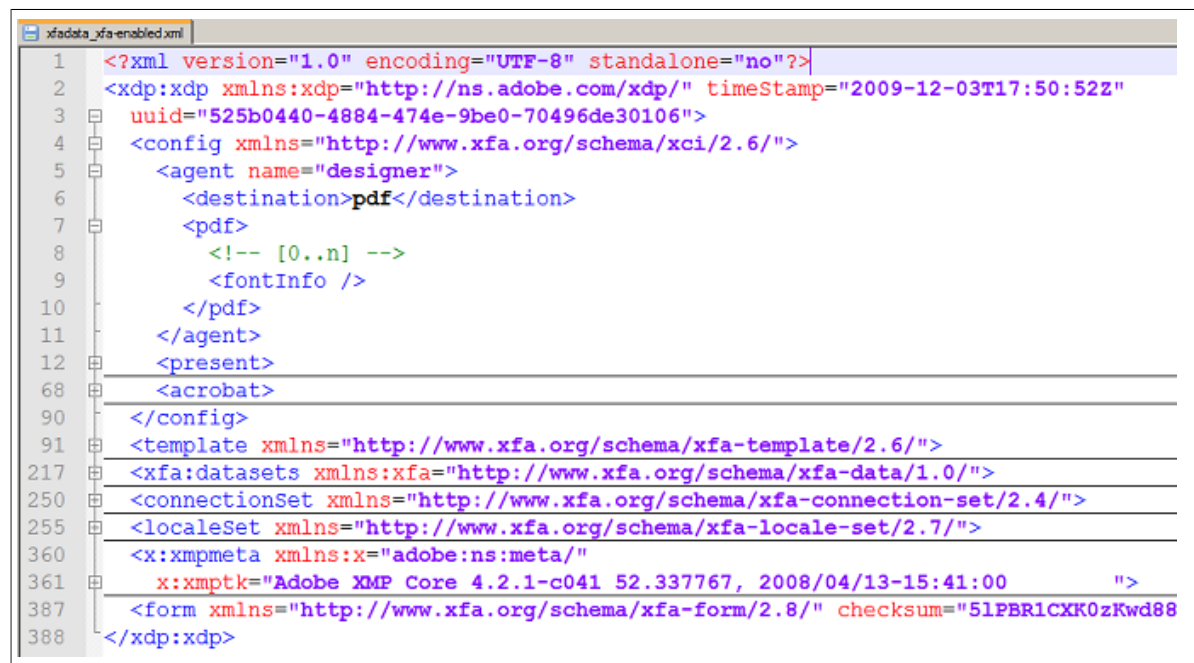
java %TOOL% %IN_FILE% %OUT_DIR% %PASSWD%
endlocal
```

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 zusammengefasst, um einen besseren Eindruck zu vermitteln:



```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/" timeStamp="2009-12-03T17:50:52Z"
3   uuid="525b0440-4884-474e-9be0-70496de30106">
4   <config xmlns="http://www.xfa.org/schema/xci/2.6/">
5     <agent name="designer">
6       <destination>pdf</destination>
7     <pdf>
8       <!-- [0..n] -->
9       <fontInfo />
10    </pdf>
11  </agent>
12  <present>
68  <acrobat>
90  </config>
91  <template xmlns="http://www.xfa.org/schema/xfa-template/2.6/">
217 <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
250 <connectionSet xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
255 <localeSet xmlns="http://www.xfa.org/schema/xfa-locale-set/2.7/">
360 <x:xmpmeta xmlns:x="adobe:ns:meta/"
361   x:xmptk="Adobe XMP Core 4.2.1-c041 52.337767, 2008/04/13-15:41:00" />
387 <form xmlns="http://www.xfa.org/schema/xfa-form/2.8/" checksum="51PBR1CXK0zKwd88
388 </xdp:xdp>
```

Das Extraktionsprogramm nutzt intern die Methode `XfaForm.getDomDocument()` von iText (<http://www.itextpdf.com>).

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 3.30: „XMP-Daten“ (S. 72) 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/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%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-flac-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-flac-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>

```

Bei der Verarbeitung benutzt PDFUnit intern die Methode `PdfReader.getMetadata()` von iText (<http://www.itextpdf.com>).

Kapitel 10. Praxisbeispiele

10.1. Passt ein Text in vorgefertigte Formularfelder

Ausgangssituation

Ein PDF-Dokument wird auf der Basis einer Dokumentenvorlage (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 ein Tag zur Verfügung, mit dem ein **Text-Overflow** festgestellt werden kann.

Lösung

```
<testcase name="noTextOverflow_AllFields">
  <assertThat testDocument="acrofields/fieldsWithAttributes.pdf">
    <hasFields>
      <allWithoutTextOverflow />
    </hasFields>
  </assertThat>
</testcase>
```

Der Test ist auch für einzelne Felder möglich:

```
<testcase name="noTextOverflow_Field_AlignLeft">
  <assertThat testDocument="acrofields/fieldSizeAndText.pdf">
    <hasField withName="Textfield, text inside, align left:" >
      <withoutTextOverflow />
    </hasField>
  </assertThat>
</testcase>
```

In Kapitel 3.12: „Formularfelder, Textüberlauf“ (S. 38) ist dieses Beispiel detailliert beschrieben.

10.2. 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 verwendet.

Lösung

```
<!--  
  This sample shows how to verify that a logo is visible on each page.  
-->  
<testcase name="verifyNewLogoOnEveryPage">  
  <assertThat testDocument="images/imagesWithSameImagesOnOnePage.pdf">  
    <containsImage file="images/exported-image_ant-logo-PNG.png"  
      on="EVERY_PAGE"  
    />  
  </assertThat>  
</testcase>
```

10.3. 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

```
<!--  
  Situation: two companies combine.  
  This sample shows how to verify that the new signature is used.  
-->  
<testcase name="verifyNewSignatureOnLastPage">  
  <assertThat testDocument="images/imagesWithSameImagesOnOnePage.pdf">  
    <containsImage file="images/CEO-signature.png"  
      on="LAST_PAGE"  
    />  
  </assertThat>  
</testcase>
```

10.4. Name des alten Vorstandes

Ausgangssituation

Der Vorstand hat wieder gewechselt.

Problem

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

Lösungsansatz

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

Lösung

```
<!--
This example shows how to verify that an expected text
is not present in the complete document.
-->
<testcase name="verifyOldCEONotPresent">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="EVERY_PAGE">
      <notContaining>NameOfOldCEO</notContaining>
    </hasText>
  </assertThat>
</testcase>
```

10.5. Schachtelungstiefe von Bookmarks

Ausgangssituation

Ein Unternehmen hat für seine PDF-Dokumente einen Styleguide, der eine maximale Verschachtelungstiefe von 2 Hierarchieebenen erlaubt.

Problem

Wie kann diese Vorgabe geprüft werden?

Lösungsansatz

Die Bookmarks werden als XML-Struktur betrachtet. Aufgrund dieser Struktur wird ein geeigneter XPath-Ausdruck entwickelt.

Lösung

Aus den Bookmarks eines PDF-Dokumentes wurde mithilfe des Extraktionsprogramms Extract-Bookmarks folgende XML-Struktur erzeugt:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookmark>
  <Title Action="GoTo" Page="1 FitH 698" Style="bold" >Bookmark to first page
    <Title Action="URI" URI="http://www.wikipedia.org/" Color="0 0 1" >
      Link to Wikipedia
    </Title>
  </Title>
</Bookmark>
```

Der XPath-Ausdruck soll prüfen, dass es **keinen** Title-Knoten gibt, der 2 oder mehr Title-Knoten als Vorgänger hat. Dann sieht der Test folgendermaßen aus:

```
<!--
Testing bookmarks, one nested level allowed.
-->
<testcase name="hasBookmarks_LimitedNestedDepthTo1">
  <assertThat testDocument="bookmarks/bookmarksWithPdfOutline.pdf">
    <hasBookmarks>
      <matchingXPath expr="count(//Title[count(ancestor::Title) > 1]) = 0" />
    </hasBookmarks>
  </assertThat>
</testcase>

<!--
Important hint:
The value of the attribute 'expr' is used as a parameter of type String.
So you have to use double quotes as outer quotes for the attribute.
Otherwise you get a compile error after the transformation from XML to Java.
-->
```

Kapitel 11. Installation, Konfiguration, Update

11.1. Technische Voraussetzungen

PDFUnit benötigt mindestens Java-7 als Laufzeitumgebung.

Weiterhin werden die Bibliotheken von iText ab Version 5.3.3 benötigt, die aus Lizenzgründen von PDFUnit nicht mitgeliefert werden.

Wenn Sie PDFUnit über ANT, Maven oder andere Automatisierungswerkzeuge ausführen, benötigen Sie natürlich noch die Installationen dieser Werkzeuge.

Getestete Umgebungen

Unter den folgenden Umgebungen wurde PDFUnit erfolgreich getestet:

Betriebssystem

- Windows-XP, 32 Bit
- Windows-7, 64 Bit
- Kubuntu Linux 12/04, 32 Bit
- Kubuntu Linux 12/04, 64 Bit
- Mac OS X, 64 Bit

Java Version

- Oracle JDK-1.7.0, 32 + 64 Bit
- Oracle JDK-1.8.0, 64 Bit
- Oracle JDK-1.8.0 b100, Windows, 32 + 64 Bit
- IBM J9, R26_Java726_SR4, Windows 7, 64 Bit
- OpenJDK-1.6.0, 64 Bit
- Apple Inc. 1.6.0, 64 Bit

Mit der JDK-Version „IBM J9, R26_Java726_SR1“ gab es unter „Windows 7, 32 Bit“ Probleme bei der internen XML-Verarbeitung.

Mit der JDK-Version „Oracle 1.8.0“ gibt es bei der Verarbeitung von PNG-Dateien Unterschiede zur Version 1.7.x. Deshalb müssen PNG-Dateien von gerenderten PDF-Seiten ebenfalls mit der Java-Version 1.8.0 gerendert werden, damit die PDFUnit-Tests mit solchen Dateien unter Java 1.8.0 funktionieren.

Weitere Java/Betriebssystem-Kombinationen werden ständig getestet.

Sollte es Probleme mit der Installation geben, schreiben Sie an [problem\[at\]pdfunit.com](mailto:problem[at]pdfunit.com).

11.2. Installation

In den folgenden Abschnitten wird die Installation von PDFUnit-XML, PDFUnit-Java und iText beschrieben. Alle Komponenten werden zur Ausführungszeit von PDFUnit-XML benötigt.

Installation von PDFUnit-XML

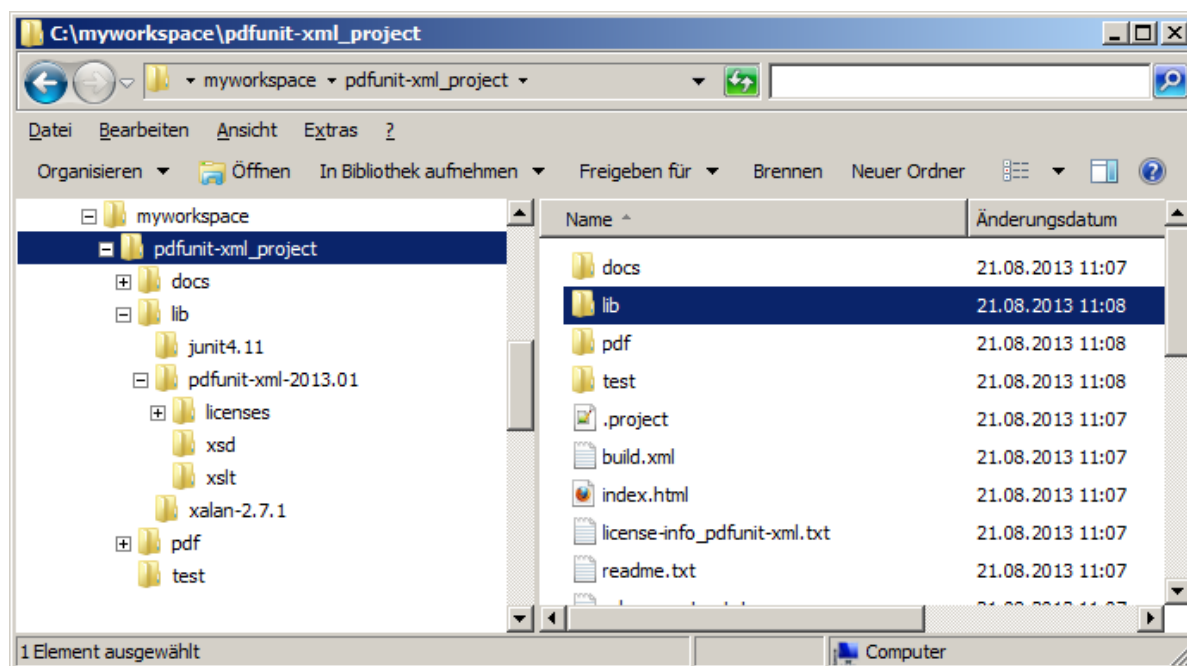
Laden Sie die Datei `pdfunit-xml-VERSION.zip` aus dem Internet: <http://www.pdfunit.com/de/download/index.html>. Wenn Sie eine Lizenz erworben haben, erhalten Sie die ZIP-Datei per Mail.

Entpacken Sie die ZIP-Datei, z.B. in den einen Ordner, der nachfolgend `PROJECT_HOME` genannt.

Die folgende Tabelle beschreibt die Verzeichnisstruktur.

Verzeichnis	Inhalt
PROJECT_HOME	Build-Skript build.xml, Skripte zur Nutzung der Hilfsprogramme *.bat, Projektdatei für Eclipse .project, kleine Einstiegsseite für nützliche Links index.html sowie readme.txt, release-notes.txt, und license-info_pdfunit-xml.txt.
PROJECT_HOME/lib	Runtime-Bibliotheken, die zur Verarbeitung von PDFUnit-XML benötigt werden. Hierhin müssen noch die Bibliotheken von iText und PDFUnit-Java kopiert werden, wie weiter unten beschrieben.
PROJECT_HOME/lib/pdfunit-xml-VERSION/xslt	Stylesheets, die zur Verarbeitung mit PDFUnit-XML benötigt werden
PROJECT_HOME/lib/pdfunit-xml-VERSION/xsd	XML-Schema Datei zur Validierung der PDFUnit-Tests
PROJECT_HOME/lib/pdfunit-xml-VERSION/licenses	Lizenzinformation aller von PDFUnit verwendeten Bibliotheken
PROJECT_HOME/test	Beispiele. Plazieren Sie hier auch Ihre eigenen Tests.
PROJECT_HOME/pdf	PDF-Dokumente für die Beispiel-Tests. Hier speichern Sie Ihre eigenen Test-Dokumente.

Das nachfolgende Bild veranschaulicht diese Struktur.



Die Datei PROJECT_HOME/index.xml enthält Links zu wichtigen Dateien in dem Projekt.

Die Installation liefert das Start-Skript PROJECT_HOME/build.xml mit. Das ANT-Skript ist unter Windows und unter Linux lauffähig. Starten Sie das Skript, wie im nächsten Kapitel beschrieben. Zu den Testergebnissen gelangen Sie über die Einstiegsseite PROJECT_HOME/index.html.

Sollten Sie das Projekt mit anderen Skriptsprachen starten wollen, ist das technisch kein Problem. Schreiben Sie ein Mail an support[at]pdfunit.com, wenn Sie Unterstützung benötigen.

Installation von PDFUnit-Java

PDFUnit-Java wird zur Ausführungszeit benötigt, es kann hier PDFUnit-Java heruntergeladen werden. Achten Sie darauf, dass die PDFUnit-Java-Version zur PDFUnit-XML-Version passt. Die zueinander passenden Versionen sind in der Datei „readme.txt“ dokumentiert.

Entpacken Sie die ZIP-Datei und kopieren Sie den entstandenen Ordner als neues Verzeichnis unterhalb von `PROJECT_HOME/lib`. Dieser Ordner wird weiter unten `PDFUNITJAVA_HOME` genannt.

Installation von iText

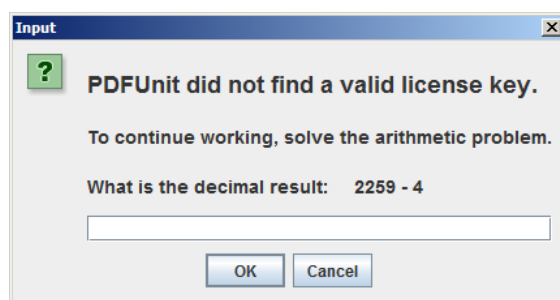
Zur Ausführungszeit wird auch iText in einer Version 5.3.3 oder höher benötigt. Laden Sie sich iText hier (<http://sourceforge.net/projects/itext/files/iText/>) herunter.

Entpacken Sie auch die ZIP-Datei von iText und kopieren den entstandenen Ordner ebenfalls als Verzeichnis unterhalb von `PROJECT_HOME/lib`.

Beachten Sie, dass iText für den kommerziellen Einsatz lizenzpflichtig ist.

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. Sie finden das Dialogfenster, wenn Sie alle Anwendungsfenster minimieren.

PDFUnit bringt Hilfsprogramme zum Extrahieren verschiedener Informationen aus den PDF-Dokumenten mit. Diese benötigen keinen Lizenzschlüssel.

Lizenzschlüssel beantragen

Wenn Sie PDFUnit im kommerziellen Umfeld einsetzen, benötigen Sie eine Lizenz. Schreiben Sie ein Mail an [license\[at\]pdfunit.com](mailto:license[at]pdfunit.com), Sie erhalten dann umgehend 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`. Diese Datei kopieren Sie nach `PDFUNITJAVA_HOME`.

Jede Änderung an der Lizenzdatei macht diese unbrauchbar. Nehmen Sie in einem solchen Falle mit PDFUnit.com Verbindung auf und beantragen Sie eine neue Lizenz.

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 11.5: „Überprüfung der Konfiguration“ (S. 130) ausführlich beschrieben.

11.3. Starten von PDFUnit-XML

Prinzip

Die Ausführung der Tests mit PDFUnit-XML wird über das ANT-Skript `PDFUNITXML_HOME/build.xml` gestartet. Das Skript benötigt ein installiertes ANT und Java. Alle für die Ausführung der PDFUnit-Tests notwendigen Konfigurationen sind in der Datei bereits vorgenommen.

Falls Sie die Datei `build.xml` ändern möchten, achten Sie darauf, dass der Classpath das Verzeichnis `PDFUNITJAVA_HOME` enthält, weil dort die benötigte Datei `config.properties` liegt. Sie wird zur Ausführungszeit von PDFUnit-Java im Classpath gesucht.

PDFUnit-XML von der Konsole starten

Öffnen Sie eine „Shell“, je nach Betriebssystem auch „Command Prompt“, „DOS-Box“, „Eingabeaufforderung“ oder „Kommandozeile“ genannt. Wechseln Sie in das Projektverzeichnis `PROJECT_HOME` und geben Sie dort den Befehl:

```
ant all
```

oder:

```
runPDFUnit.bat
```

ein. Die Skripte `runPDFUnit.bat` bzw. `runPDFUnit.sh` kapseln den ANT-Aufruf und bieten die Möglichkeit, Pfade zu Ihren Java- und ANT-Installationen zu setzen.

Vorausgesetzt, Java und ANT sind auf Ihrem Rechner installiert und im Pfad des Betriebssystems eingetragen, begleitet ein umfangreicher Report die Abarbeitung der Testfälle:

```
Buildfile: C:\pdfunit-xml_demo\build.xml

clean:
  [delete] Deleting directory C:\pdfunit-xml_demo\build_ant

00_resolveDTD:
  [echo] start resolving DTD entities ...
  [mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\xml
  [xslt] Loading stylesheet C:\pdfunit-xml_demo\lib\pdfunit-xml-2015.10\xslt\...
  [xslt] Processing C:\pdfunit-xml_demo\src\test\xml\CompareTestDemo.xml
  ...
  [echo] ... finished
  ...

...

01_verifyXML:
  [echo] start validating PDFUnit test files (xml) ...
  [copy] Copying 1 file to C:\pdfunit-xml_demo\build_ant\xsd
  [echo] ... finished

02_generateJavaSourcesFromXML:
  [echo] start transforming PDFUnit test files (xml) into Java code ...
  [mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\java\org\pdfunit\xml
  [xslt] Loading stylesheet C:\pdfunit-xml_demo\lib\pdfunit-xml-2015.10\xslt\...
  [xslt] Processing C:\pdfunit-xml_demo\src\test\xml\CompareTestDemo.xml
  ...
  [echo] ... finished
  ...
```

```

...
03_compileGeneratedSources:
[echo] start compiling generated sources ...
[mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\classes
[javac] Compiling 9 source files to C:\pdfunit-xml_demo\build_ant\classes
[echo] ... finished

04_runUnittest:
[echo] start running unit tests from compiled sources ...
[mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\junit\data
[junit] Running org.pdfunit.xml.ContentTestDemo
[junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 1.054 sec
...
[echo] ... finished

-testreport:
[echo] start creating HTML report from unit test result ...
[mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\junit\html
[junitreport] Processing C:\pdfunit-xml_demo\build_ant\junit\html\...
[junitreport] Loading stylesheet JAR:file:/C:/environment/share32/tools/...
[junitreport] Transform time: 890ms
[junitreport] Deleting: c:\Temp\null1904905169
[echo] ... finished
[echo] Please look for index.html in subfolder build_ant/junit/html

all:
BUILD SUCCESSFUL

```

Danach steht ein ausführlicher HTML-Report für alle Tests im Verzeichnis `PROJECT_HOME/build_ant/junit/html/index.html` zur Verfügung. Hier ein Blick auf den Report des mitgelieferten Demo-Projektes:

Unit Test Results. Designed for use with [JUnit](#) and [Ant](#).

Summary

Tests	Failures	Errors	Success rate	Time
20	1	0	95.00%	10.539

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)	Time Stamp
org.pdfunit.xml	15	0	1	9.872	2013-08-04T1
org.pdfunit.xml.bookmarks	1	0	0	0.470	2013-08-04T10
org.pdfunit.xml.compare	3	0	0	0.096	2013-08-04T10
org.pdfunit.xml.installation	1	0	0	0.101	2013-08-04T10

Sollten Sie vergessen haben, die Bibliotheken von PDFUnit-Java in das Verzeichnis `PROJECT_HOME/lib/pdfunit-java_VERSION` zu kopieren, erscheint folgende Fehlermeldung:

```

[javac] ... error: cannot find symbol
[javac]     AssertThat.document(filename)
[javac]       ^
[javac] symbol:   variable AssertThat

```

PDFUnit-XML in Entwicklungsumgebungen nutzen

Sie möchten PDFUnit-XML lieber aus Ihrer Entwicklungsumgebung aus starten, als von der Konsole? Das geht, sofern die Entwicklungsumgebung Skripte starten kann, denn der Weg ist, das vorhandene

ANT-Skript `PROJECT_HOME/build.xml` aus der IDE heraus zu starten. Falls Sie Hilfe benötigen, schreiben Sie an `support[at]pdfunit.com`.

11.4. Einstellungen in der config.properties

Normalerweise muss PDFUnit nicht konfiguriert werden, es gibt mit der Datei `config.properties` aber die Möglichkeit dazu, die nachfolgend beschrieben wird.

Formatstring des Datums

Das interne Format des Erstellungsdatums und des Änderungsdatums eines PDF-Dokumentes fällt sehr unterschiedlich aus, je nachdem, mit welchem Werkzeug das PDF-Dokument erstellt wurde. Der Formatstring kann in der Konfigurationsdatei an die jeweiligen Gegebenheiten angepasst werden:

```
#####
# Declaring the default format for dates in PDF documents.
# Use the format strings according to java.util.SimpleDateFormat.
#####
# Using date only:
#dateformat = 'D:'yyyyMMdd
# Using date and time:
dateformat = 'D:'yyyyMMddHHmmss
```

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 gerendeter Seiten eines Testdokumentes und eines Vergleichsdokumentes Unterschiede erkannt werden, wird ein Fehlerbild erstellt. Das Bild enthält auf der linken Seite das vollständige Vergleichsdokument 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 = ./
```

PDF-Dokumente können aber auch als Stream oder Byte-Array verarbeitet werden. Für solche Dokumente wird das Ausgabeverzeichnis für die Diff-Images durch die Property `diffimage.output.path.streams_and_bytearrays` festgelegt:

```
#####
#
# If this property is not defined, the directory of the running process
# is used.
#
#####
diffimage.output.path.streams_and_bytearrays = ./
```

11.5. 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
::

:: Change the installation directories depending on your situation:
set ITEXT_HOME=../itext-5.5.1
set JUNIT_HOME=../junit4.11
set VIP_HOME=../vip-1.0.0
set PDFUNIT_HOME=.

set CLASSPATH=%ITEXT_HOME%/*;%CLASSPATH%
set CLASSPATH=%JUNIT_HOME%/*;%CLASSPATH%
set CLASSPATH=%VIP_HOME%/*;%CLASSPATH%

... (shortened for documentation)

:: 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
```

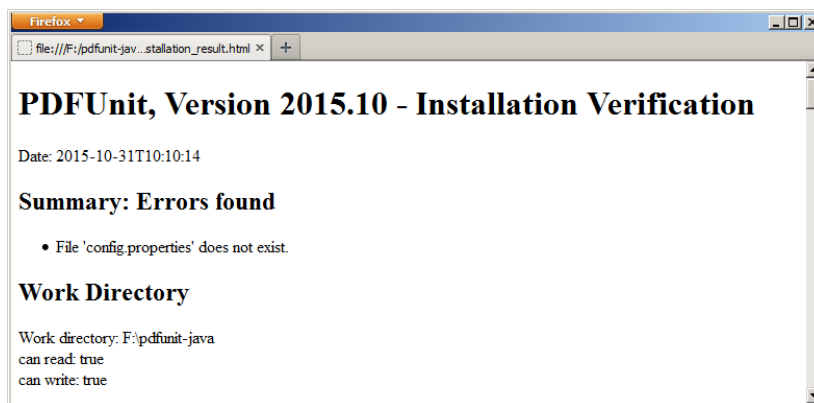
Passen Sie die Pfade an die Verhältnisse Ihrer Installation an.

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 XML-Test

Die gleiche Überprüfung kann auch in der XML-Syntax von PDFUnit-XML durchgeführt werden. Dadurch ist es möglich, die Systemumgebung der tatsächlich laufenden Tests sichtbar zu machen:

```
<comment>
  This tests verifies that all required libraries and files are found.
  Additionally it logs some system properties and writes
  all of them into both an XML file and an HTML formatted file.
</comment>

<testcase name="verifyRequiredFilesAndLibraries">
  <verifyInstallation verificationFile="verifyInstallation.vip"/>
</testcase>
```

Dieser XML-Test 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:

Unit Test Results.

Designed for use with [JUnit](#) and [Ant](#).

Class org.pdfunit.xml_VerifyInstallation

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
VerifyInstallation	1	1	0	0	0.121	2013-10-25T18:04:37	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
verifyRequiredFilesAndLibraries	Error	Configuration ERROR. See output file 'verifyInstallation.vip.out.html'. org.verifyInstallation.VIPEException: Configuration ERROR. See output file 'verifyInstallation.vip.out.html'. at org.verifyInstallation.VIP.verifyAndReport (VIP.java:191) at org.verifyInstallation.VIP.verifyInstallation (VIP.java:133) at com.pdfunit.AssertThat.installationIsClean (SourceFile:177) at org.pdfunit.xml_VerifyInstallation.verifyRequiredFilesAndLibraries (_VerifyInstallation.java:32)	0.119

[Properties >](#)

Die Report-Datei enthält dieselben Informationen (s.o.), als wäre sie über ein Skript erzeugt worden.

11.6. Update von PDFUnit-XML

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: <http://www.pdfunit.com/de/download/index.html>.

Wenn Sie PDFUnit mit Lizenzdatei einsetzen, erhalten Sie das neue Release sowie mit den Lizenzdaten per Mail.

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 in ein neues Verzeichnis.

Achten Sie darauf, dass die neue Version von PDFUnit-XML mit der bestehenden Version von PDFUnit-Java kompatibel ist. Gegebenenfalls installieren Sie auch eine neue Version von PDFUnit-Java. Die Schritte dafür sind im nächsten Kapitel beschrieben.

Übernehmen Sie Ihre Testdateien, PDF-Dokumente und angepasste Konfigurationsdateien aus dem alten Projekt in das neue Release.

Letzter Schritt

Führen Sie Ihre bestehenden Tests mit dem neuen Release durch. Sofern es keine dokumentierten Inkompatibilitäten zwischen dem alten und neuen Release gibt, sollten Ihre Tests erfolgreich durchlaufen. Andernfalls lesen Sie die Release-Informationen.

11.7. Update von PDFUnit-Java

Zur Ausführungszeit von PDFUnit-XML wird im Hintergrund PDFUnit-Java verwendet. Deshalb muss die Version von PDFUnit-XML zu der von PDFUnit-Java passen.

Wenn die Versionsnummern von PDFUnit-XML und -Java gleich sind, passen die Versionen immer zueinander. Für den Fall, dass die Versionsnummern nicht gleich sind, finden Sie weitere Informationen zur Kompatibilität hier: <http://www.pdfunit.com/de/download/index.html>.

Beschaffung des neuen Releases

Wenn Sie PDFUnit mit Lizenzdatei einsetzen, erhalten Sie das neue Release sowie eine Lizenzdatei per Mail.

Ansonsten laden Sie sich die neue ZIP-Datei aus dem Internet: <http://www.pdfunit.com/de/download/index.html>.

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 in einen Ordner, der nachfolgend `PDFUNITJAVA_NEW` genannt wird. Der Ordner des 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_NEW` nach `PROJECT_HOME/lib/pdfunit-NEW-VERSION`.

Sollten Sie im alten Release Änderungen an der `config.properties` vorgenommen haben, so übertragen Sie die Änderungen in die `config.properties` des neuen Releases.

Wenn Sie ein lizenziertes PDFUnit-Java einsetzen, kopieren Sie die neue Lizenzdatei `license-key_pdfunit-java.lic` an den Ort, an dem sie beim alten Release lag. Löschen Sie die alte Lizenzdatei.

Achten Sie darauf, dass die neue Version von PDFUnit-Java mit der bestehenden Version von PDFUnit-XML kompatibel ist. Gegebenenfalls installieren Sie eine neue Version von PDFUnit-XML. Die Schritte dafür sind im vorhergehenden Kapitel beschrieben.

Letzter Schritt

Führen Sie Ihre bestehenden Tests mit dem neuen Release durch. Die Tests sollten erfolgreich laufen, es sei denn, es gibt dokumentierte Inkompatibilitäten zwischen dem alten und neuen Release gibt. Andernfalls lesen Sie die Release-Informationen.

11.8. 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 12. PDFUnit für Nicht-XML Systeme

12.1. Kurzer Blick auf PDFUnit-Java

„PDFUnit-Java“ ist die erste Implementierung von PDFUnit und auch die Basis für die Implementierung in anderen Programmiersprachen. Für die XML-Implementierung ist PDFUnit-Java auch die technische Laufzeitumgebung. Sofern es möglich ist, werden die Schlüsselwörter aller Implementierungen von PDFUnit gleichlautend zu den Schlüsselwörtern in PDFUnit-Java gewählt.

Die API folgt dem „Fluent Interface“ (http://de.wikipedia.org/wiki/Fluent_Interface), wie die folgenden Beispiele zeigen:

```
@Test
public void hasTextOnFirstPageInClippingArea() throws Exception {
    String filename = PATH + "content/documentForTextClipping.pdf";

    int upperLeftX = 50;
    int upperLeftY = 130;
    int width = 170;
    int height = 25;
    ClippingArea inClippingArea = new ClippingArea(upperLeftX, upperLeftY, width, height);

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE, inClippingArea)
        .containing("Content on first page")
    ;
}
```

```
@Test
public void compareFields() throws Exception {
    String filenameTest = PATH + "acrofields/test.pdf";
    String filenameMaster = PATH + "acrofields/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameFieldsByName()
        .haveSameFieldsByProperties()
        .haveSameFieldsByValue()
    ;
}
```

```
@Test
public void hasSignature() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";
    String xpath = "//signature[@name='Signature2']";

    XPathExpression expression = new XPathExpression(xpath);
    AssertThat.document(filename)
        .hasSignatures()
        .matchingXPath(expression)
    ;
}
```

PDFUnit-Java ist in einer eigenen Dokumentation ausführlich beschrieben. Siehe <http://www.pdfunit.com/de/documentation/java/index.html>.

12.2. Kurzer Blick auf PDFUnit-Perl

Für Perl-Umgebungen wird es ab September 2014 „PDFUnit-Perl“ geben. Diese Version von PDFUnit umfasst das Perl-Modul `PDF::PDFUnit`, notwendige Skripte und Laufzeitkomponenten. In Verbindung mit verschiedenen CPAN-Modulen 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()
    ->matchingComplete("wrong-author-intended")
} 'com::pdfunit::errors::PDFUnitValidationException'
,"Test should fail. Demo test with expected exception."
;
```

Die Verwendung von PDFUnit-Perl wird in einer eigenen Dokumentation beschrieben.

12.3. Kurzer Blick auf PDFUnit-NET

Als „PDFUnit-NET“ steht PDFUnit voraussichtlich ab Oktober 2014 auch für eine .NET-Umgebung zur Verfügung. Erste Implementierungen als „Proof-of-Concept“ waren erfolgreich:

```
[TestMethod]
public void HasAuthor()
{
  String filename = path + "resources/pdf/documentInfo/documentInfo_allInfo.pdf";

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

```
[TestMethod]
[ExpectedException(typeof(PDFUnitValidationException))]
public void HasAuthor_StartingWith_WrongString()
{
  String filename = path + "resources/pdf/documentInfo/documentInfo_allInfo.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.

Weil die Entwicklung nicht abgeschlossen ist, kann sich dieser Code noch ändern.

Für PDFUnit-NET wird ebenfalls eine eigene Dokumentation erstellt.

Kapitel 13. Anhang

13.1. Instantiierung der PDF-Dokumente

PDF-Dokumente können als **Datei** oder als **URL** eingelesen werden. Für die Unterscheidung dieser beiden Formate muss das Attribut `testIsURL` bzw. `masterIsURL` verwendet werden. Hier ein Beispiel:

```
<comment>
  This test shows the usage of PDFUnit-XML when comparing two PDF
  documents loaded as a URL.
</comment>

<testcase name="compareDocuments_bothLoadedFromURL">
  <assertThat testDocument="./testdocuments/test.pdf"
              masterDocument="http://localhost/.../master/master.pdf"
              masterIsURL="YES"
            >
    ...
  </assertThat>
</testcase>
```

Wenn die PDF-Dokumente passwort-geschützt sind, wird das „User-Password“ oder das „Owner-Password“ benötigt. Beide werden in einem eigenen Attribut mitgegeben:

```
<testcase name="compareEncryptedTestAgainstEncryptedMaster">
  <assertThat testDocument="master/test.pdf"
              testPassword="secret-test"
              masterDocument="master/master.pdf"
              masterPassword="secret-master"
            >
    ...
  </assertThat>
</testcase>
```

13.2. Seitenauswahl

Vordefinierte Seiten

Für Tests, die sich auf bestimmte Seiten eines PDF-Dokumentes beziehen, existieren mehrere Konstanten, deren Bedeutung sich aus ihrem Namen ergibt:

```
<!-- Possibilities to focus tests to specific pages: -->

<!-- Predefined constants for pages: -->
<xxx on="ANY_PAGE" />
<xxx on="EVEN_PAGES" />
<xxx on="EACH_PAGE" />
<xxx on="EVERY_PAGE" />
<xxx on="FIRST_PAGE" />
<xxx on="LAST_PAGE" />
<xxx on="ODD_PAGES" />

<!-- Attributes for individual pages: -->
<xxx onPage=".." />
<xxx onEveryPageAfter=".." />
<xxx onEveryPageBefore=".." />
<xxx onAnyPageAfter=".." />
<xxx onAnyPageBefore=".." />
```

`on="EACH_PAGE"` und `on="EVERY_PAGE"` sind funktional identisch. Die Redundanz ist aus sprachlichen Gründen beabsichtigt.

Hier ein Beispiel mit einer vordefinierten Seiten-Konstanten im Attribut `on=".."`:


```
<testcase name="hasText_MultipleSearchTokens_EvenPages">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="EVEN_PAGES">
      <containing>Content</containing>
      <containing>even pageNumber</containing>
    </hasText>
  </assertThat>
</testcase>
```

Individuelle Seiten

Das nächste Beispiel zeigt, wie beliebige, individuelle Seiten definiert werden können. Seitenzahlen müssen durch Kommata getrennt werden:

```
<testcase name="hasText_OnMultiplePages_SelectedPages">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onPage="1, 2, 3">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

Offene Seitenbereiche

Auch für zusammenhängende Seiten am Anfang oder am Ende eines Dokumentes gibt es Konstanten:

```
<testcase name="hasText_OnAnyPageAfter1">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onAnyPageAfter="1">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="hasText_OnEveryPageBefore3">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onEveryPageBefore="3">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

Seitenbereiche innerhalb eines Dokumentes

Und als Letztes gibt es die Attribute `<hasText fromPage="1" toPage="2" >`, um Tests auf einen Bereich innerhalb eines Dokumentes zu beschränken. Das folgende Beispiel validiert Text, der zwei Seiten überspannt.

```
<testcase name="hasText_SpanningOver2Pages_matchingRegex">
  <assertThat testDocument="%pdfdir;/content/text-starts-on-page1-continues-on-page2.pdf">
    <hasText fromPage="1" toPage="2" >
      <inClippingArea upperLeftX="18" upperLeftY="30"
                    width="182" height="238"
                    unit="MILLIMETER">
        >
          <matchingRegex>Text starts on page 1.*continues on page 2</matchingRegex>
        </inClippingArea>
      </hasText>
    </assertThat>
  </testcase>
```

Für geschlossene Seitenbereiche stehen nur die Tags `<containing>`, `<matchingRegex>`, `<notContaining>` und `<notMatchingRegex>` zur Verfügung.

Wichtige Hinweise

- Seitenzahlen beginnen mit '1'.
- Die Seitenangaben in den Attributen `onXxxPageBefore` und `onXxxPageAfter` sind jeweils exklusiv gemeint.

- Die Seitenangaben in den Attributen `from` und `to` sind jeweils inklusiv gemeint.
- Das Attribut `onEveryPageXxx` bedeutet, dass der gesuchte Text wirklich auf jeder Seite existieren muss.
- Dagegen reicht es, wenn bei `onAnyPageXxx` der gesuchte Text auf einer Seite existiert.

13.3. Textvergleich

Ein erwarteter Text und der tatsächliche Text einer PDF-Seite können auf folgende Art miteinander verglichen werden:

```
<!-- Comparing text: -->
```

```
<startingWith />
<containing />
<matchingComplete />
<matchingRegex />
<endingWith />

<notStartingWith />
<notContaining />
<notMatchingRegex />
<notEndingWith />

<containing whitespaces=".." />
<matchingComplete whitespaces=".." />
<notContaining whitespaces=".." />
```

❶

❷

❸

❹

❺

❻

- ❶❷❸ Bei diesen Tests werden die Whitespaces „normalisiert“. Das heißt, Leerzeichen am Anfang und Ende werden entfernt und alle Whitespaces innerhalb eines Textes werden auf **ein** Leerzeichen reduziert.
- ❹❺❻ Die Behandlung von Whitespaces wird über das optionale Attribut `whitespaces=".."` gesteuert, für das die Konstanten `KEEP`, `NORMALIZE` und `IGNORE` existieren. Diese Konstanten sind in Kapitel 13.4: „Behandlung von Whitespaces“ (S. 138) separat beschrieben.

Vergleiche mit Regulären Ausdrücken folgen den Regeln und Möglichkeiten der Java-Klasse `java.util.regex.Pattern`:

```
<!-- Using regular expression to compare page content: -->
<testcase name="hasText_MatchingRegex">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="FIRST_PAGE">
      <matchingRegex>.*[Cc]ontent.*</matchingRegex>
    </hasText>
  </assertThat>
</testcase>
```

13.4. 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 drei Konstanten zur Verfügung, von denen `NORMALIZE` die Standardbehandlung ist, falls nichts anderes angegeben wird:

```
<!-- Constants for whitespace processing: -->
```

```
<xxx whitespaces="IGNORE" />
<xxx whitespaces="KEEP" />
<xxx whitespaces="NORMALIZE" />
```

❶

❷

❸

- ❶ Text wird so komprimiert, dass er keine Whitespaces mehr enthält.
- ❷ Alle Whitespaces bleiben erhalten.
- ❸ Whitespaces am Anfang und am Ende eines Textes werden gelöscht. Whitespaces innerhalb eines Textes werden auf ein Leerzeichen reduziert.

Bei diesen Tags kann die Whitespace-Behandlung vorgegeben werden:

```
<!-- Tags which allow to define the whitespace processing -->

<hasXXXAction>
  <containing          whitespaces=".." />
  <matchingComplete    whitespaces=".." />
</hasXXXAction>

<hasText>
  <containing          whitespaces=".." />
  <notContaining        whitespaces=".." />
  <matchingComplete    whitespaces=".." />
</hasText>
```

Ein Beispiel:

```
<testcase name="hasText_WithLineBreaks_UsingIGNORE">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="FIRST_PAGE">
      <matchingComplete whitespaces="IGNORE">
        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
      </matchingComplete>
    </hasText>
  </assertThat>
</testcase>
```

In diesem Beispiel wird die erwartete Zeichenkette mit zahlreichen Zeilenumbrüchen formuliert, die so auf der PDF-Seite nicht existieren. Durch die Angabe `whitespaces="IGNORE"` funktioniert der Test trotzdem bestens.

Tests, die Reguläre Ausdrücke verarbeiten, verändern Whitespaces nicht. Bei Bedarf muss die Behandlung der Whitespaces in den regulären Ausdruck integriert werden. Hier ein Beispiel:

```
(?ms).*print(.*)
```

Der Teilausdruck `(?ms)` bedeutet, dass die Suche über mehrere Zeilen reicht. Zeilenumbrüche werden als 'Character' interpretiert.

13.5. Anführungszeichen in Suchbegriffen

Bedeutung von Double-Quotes in XML, XPath und Java

Zeichenketten in XML-Tags sind durch das Start- und Ende-Tag gekennzeichnet. Insofern dürfen die Inhalte von Tags beliebige Kombinationen von Single- und Double-Quotes enthalten.

Zeichenketten von XML-Attributen werden durch Single- oder Double-Quotes begrenzt. Soll der Wert eines XML-Attributs Double-Quotes enthalten, müssen Single-Quotes als äußere Begrenzer verwendet werden. Es können aber auch die Entitäten `'` bzw. `"` als Teil des Attributinhaltendes verwendet werden.

Eine beliebige Verwendung von Single- und Double-Quotes in XPath-Ausdrücken ist nicht möglich, auch wenn dafür Entitäten benutzt werden. Am besten benutzen Sie Single-Quotes.

In Java werden Zeichenketten alleine durch Double-Quotes gekennzeichnet. Weil die XML-Tests zuerst in Java-Code umgewandelt und dann als Java-Programm ausgeführt werden, müssen Double-Quotes, die Teil einer Zeichenkette sind, in XML mit einem Backslash maskiert werden.

Unterschiedliche Arten von Anführungszeichen

Wichtiger Hinweis: Der Begriff „Anführungszeichen“ wird in Texten unterschiedlich umgesetzt, wie das folgende Bild zeigt:

Example 1: 'single quotes'

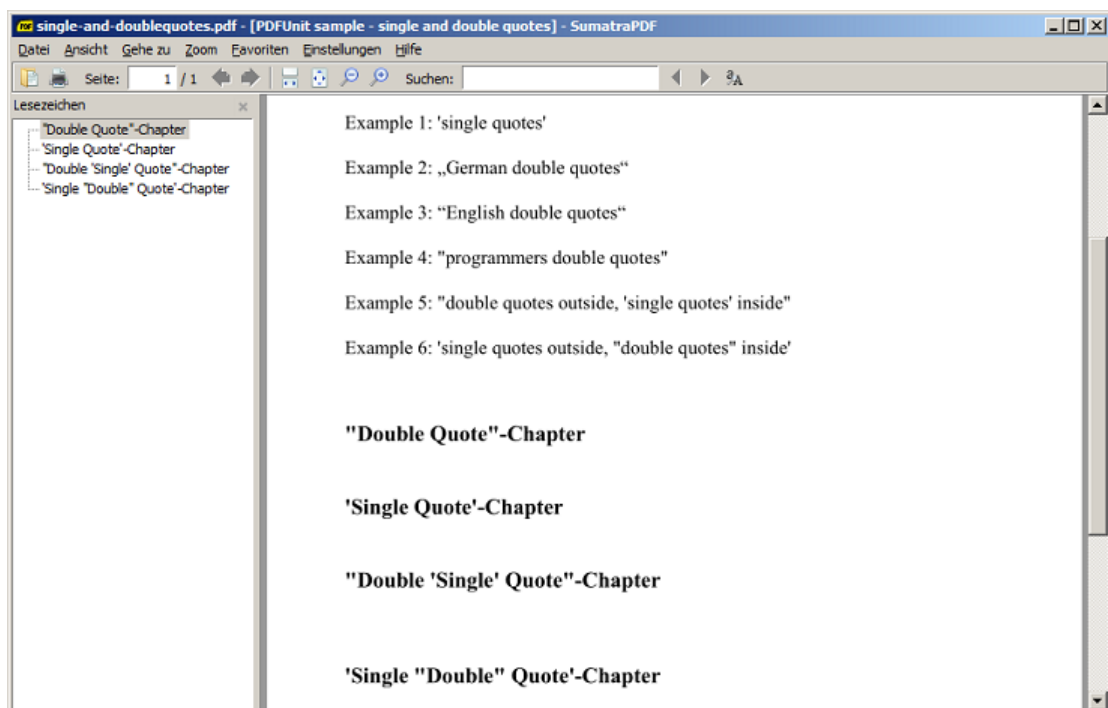
Example 2: „German double quotes“

Example 3: “English double quotes”

Example 4: "programmers double quotes"

“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 Ihren XML-Editor ein.

Die "programmers double quotes" benötigen eine besondere Aufmerksamkeit, weil sie sowohl in XML, als auch in Java als Zeichenkettenbegrenzer dienen. Die nachfolgenden Absätze und Beispiele gehen detailliert darauf ein, sie basieren alle auf dem folgenden Dokument:



Anführungszeichen in Tags

Single-Quotes in PDFUnit-Tags bereiten keine Probleme, Double-Quotes müssen mit einem Backslash maskiert werden:

```
<testcase name="tagWith_SingleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 1: 'single quotes'</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_GermanDoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 2: „German double quotes“</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_EnglishDoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 3: "English double quotes"</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_ProgrammersDoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 4: \"programmers double quotes\"</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_DoubleAndSingleQuotes_1">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>
        Example 5: \"double quotes outside, 'single quotes' inside\"
      </containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_DoubleAndSingleQuotes_2">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 6: 'single quotes outside, \"double quotes\" inside'</containing>
    </hasText>
  </assertThat>
</testcase>
```

Anführungszeichen in Attributen

Single-Quotes in Attributen können ohne Einschränkung benutzt werden, wenn das Attribut außen durch Double-Quotes begrenzt ist. Und umgekehrt: wenn außen Single-Quotes verwendet werden, sind innen Double-Quotes möglich.

Die beiden folgenden Beispiele sind gültige Verwendungen von Single- und Double-Quotes in Attributen:

```
<testcase name="attributeWith_DoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmark withLabel="Double Quote"-Chapter />
  </assertThat>
</testcase>
```

```
<testcase name="attributeWith_SingleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmark withLabel="'Single Quote'-Chapter" />
  </assertThat>
</testcase>
```

Es ist aber **nicht** erlaubt, Single- und Double-Quotes **gleichzeitig** als Inhalt einer Zeichenkette zu benutzen, auch nicht in Form von Entitäten. Das nächste Beispiel ist daher **ungültig**:

```
<!--
  This example fails, because single- and double-quotes were used
  inside the expected label.
-->

<testcase name="attributeWith_SingleDoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmark withLabel="'Single &quot;Double&quot; Quote'-Chapter" />
  </assertThat>
</testcase>
```

Anführungszeichen in XPath-Ausdrücken

Da der XML-Code von PDFUnit-XML zuerst in Java-Code überführt wird und dort die Auswertung von XPath-Ausdrücken stattfindet, bestehen bei XPath-Ausdrücken die stärksten Einschränkungen für die Verwendung von Single- oder Double-Quote.

Das folgende Beispiel lässt sich zwar in gültigen Java-Code umsetzen, liefert dann aber den Laufzeitfehler: One of the parameter contains single and double quote. You may use only one kind of quote.

```
<testcase name="attributeWithXPath_DoubleQuotes_1">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmarks>
      <matchingXPath expr="count(//Title[.='&quot;Double Quote&quot;-Chapter']) = 1" />
    </hasBookmarks>
  </assertThat>
</testcase>
```

Der Fehler lässt sich nur vermeiden, wenn die XPath-Bedingung ohne Double-Quotes formuliert wird:

```
<testcase name="attributeWithXPath_DoubleQuotes_2">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmarks>
      <matchingXPath expr="count(//Title[contains(., 'Double Quote')]) = 1" />
      <matchingXPath expr="count(//Title[contains(., 'Chapter')]) = 4" />
    </hasBookmarks>
  </assertThat>
</testcase>
```

Anführungszeichen in Regulären Ausdrücken

Weil auch die Auswertung von regulären Ausdrücken über den Zwischenschritt von generiertem Java-Code erfolgt, müssen alle Double-Quotes maskiert werden, wie im Tag `matchingRegex` des folgenden Beispiels:

```
<testcase name="regex_tag_DoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <matchingRegex>.*\double.*</matchingRegex>
    </hasText>
  </assertThat>
</testcase>
```

13.6. 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 Ausschnittes:

```
<!-- Defining a clipping area: -->
<inClippingArea upperLeftX=".." upperLeftY=".." width=".." height=".." />
<inClippingArea upperLeftY=".." upperLeftX=".." width=".." height=".." unit=".." />
```

Die verfügbaren Einheiten werden in Kapitel, 13.7: „Maßeinheiten - Points, Millimeter, ...“ (S. 143) beschrieben. Wenn keine Einheit mitgegeben wird, gilt die Einheit `MILLIMETER`.

Hier ein paar Beispiele:

```
<testcase name="hasText_InClippingArea">
  <assertThat testDocument="content/documentForTextClipping.pdf">
    <hasText on="FIRST_PAGE" >
      <inClippingArea upperLeftX="17.6" upperLeftY="45.8"
        width="60.0" height="8.8"
        unit="MILLIMETER"
      >
        <containing>Content on first page.</containing>
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="haveSameAppearance_InClippingArea">
  <assertThat testDocument="master/compareToMaster_sameImagesDifferentOrder.pdf"
    masterDocument="master/compareToMaster.pdf"
  >
    <haveSameAppearance on="FIRST_PAGE">
      <inClippingArea upperLeftX="50" upperLeftY="755"
        width="370" height="35"
        unit="POINTS"
      >
      </inClippingArea>
    </haveSameAppearance>
  </assertThat>
</testcase>
```

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 `RenderPdfClippingAreaToImage` zur Verfügung. Mit diesem können Sie das Rechteck mit den notwendigen Werten auf der Basis der Einheiten `mm` oder `points` als PNG-Datei extrahieren:

```
::
:: Render a part of a PDF page into an image file
::

@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/jpedal/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%
set CLASSPATH=./lib/aspectj-1.8.0/*;%CLASSPATH%

set TOOL=com.pdfunit.tools.RenderPdfClippingAreaToImage
set PAGENUMBER=1
set OUT_DIR=./tmp
set IN_FILE=./content/documentForTextClipping.pdf
set PASSWD=

:: Format unit can only be 'mm' or 'points'
set FORMATUNIT=points
set UPPERLEFTX=50
set UPPERLEFTY=130
set WIDTH=170
set HEIGHT=25

java %TOOL% %IN_FILE% %PAGENUMBER% %OUT_DIR% %FORMATUNIT% %UPPERLEFTX%
%UPPERLEFTY% %WIDTH% %HEIGHT% %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.

13.7. Maßeinheiten - Points, Millimeter, ...

Manche Tests benötigen Werte für Länge und Breite. Damit der Benutzer seine gewohnten Einheiten benutzen kann, stehen in dem Attribut `unit=".."` verschiedene Konstanten zur Verfügung:

```
<!-- Predefined constants for units: -->

<xxx unit="CENTIMETER" />
<xxx unit="DPI72" />
<xxx unit="INCH" />
<xxx unit="MILLIMETER" />
<xxx unit="POINTS" />
```

Wenn das Attribut `unit=".."` weggelassen wird, gilt die Einheit `MILLIMETER` als Default.

Beispiel - Größe von Formularfeldern

```
<testcase name="hasField_WidthAndHeight">
  <assertThat testDocument="acrofields/notExportableAcrofield.pdf">
    <hasField
      withName="Title of 'someField'"
      width="450" height="30"
      unit="POINTS"
    />
  </assertThat>
</testcase>
```

Beispiel - Seitenausschnitt

```
<testcase name="hasTextOnFirstPage_RectangleInInch">
  <assertThat testDocument="content/documentForTextClipping.pdf">
    <hasText on="FIRST_PAGE" >
      <inClippingArea upperLeftX="0.7" upperLeftY="1.8"
        width="2.4" height="0.4"
        unit="INCH"
      >
        <containing>Content on first page.</containing>
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

Beispiel - Größe des Seitenformats

```
<testcase name="hasFormat_HugeFormat">
  <assertThat testDocument="format/physical-map-of-the-world-1999_1117x863mm.pdf">
    <hasFormat width="2448" height="3168" unit="POINTS" />
  </assertThat>
</testcase>
```

Beispiel - Fehlermeldungen

In Fehlermeldungen werden sowohl Millimeter, als auch die ursprünglich verwendete Einheit ausgegeben. Wenn beispielsweise die Breite im letzten Beispiel mit `111 POINTS` erwartet wird, erscheint folgende Fehlermeldung:

```
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')'.
```

13.8. Fehlermeldungen

Fehlermeldungen von PDFUnit sind in englischer Sprache geschrieben und liefern detaillierte Informationen, um die 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;}}]>'.
```


Vergleiche von zwei XML-Strukturen werden intern mit XMLUnit (<http://xmlunit.sourceforge.net>) durchgeführt. Die Original-Fehlermeldung von XMLUnit erscheint auch in der Fehlermeldung von PDFUnit:

```
Content of 'pdf_withDifference.xml' does not match field infos in 'pdfDemo.pdf'.
Message from XMLUnit ==>
  org.custommonkey.xmlunit.Diff [different]
    Expected number of child nodes '1' but was '102' -
    comparing <fieldlist...> at /fieldlist[1] to <fieldlist...> at /fieldlist[1]
<== Extract field infos and compare them with a diff tool against your file.
```

13.9. Datumsauflösung

Der Vergleich eines Datums (Erstellungs- oder Änderungsdatum) mit einem Erwartungswert kann sich entweder nur auf Tag-Monat-Jahr beziehen oder zusätzlich noch auf Stunde-Minute-Sekunde. Für die Unterscheidung dieser beiden Möglichkeiten stellt PDFUnit das Attribut `resolution=".."` mit zwei Konstanten zur Verfügung:

```
<!-- Constants to define the date resolution: -->

resolution="DATE"
resolution="DATETIME"
```

In den folgenden Tags werden diese Konstanten verwendet:

```
<!-- Date resolution in tests: -->

<hasCreationDate           withDate="2013-05-05T09:33:47" resolution="DATETIME" />
<hasCreationDateBefore     withDate="2099-01-01"         resolution="DATE" />
<hasCreationDateAfter      withDate="2013-01-01"         resolution="DATE" />

<hasModificationDate       withDate="2013-05-05T09:33:47" resolution="DATETIME" />
<hasModificationDateBefore withDate="2099-01-01"         resolution="DATE" />
<hasModificationDateAfter  withDate="2013-01-01"         resolution="DATE" />
```

Der Vergleich von Datumswerten zweier PDF-Dokumenten findet immer in der Auflösung `resolution="DATE"` statt.

13.10. Default-Namensraum in XML

Bei den XML- und XPath-basierten Tests werden Namensräume, für die ein Präfix definiert ist, automatisch erkannt. Weil der XML-Standard aber zulässt, dass Namensräume mehrfach definiert werden, ermittelt PDFUnit den Default-Namensraum nicht automatisch. Er muss durch das Attribut `default-Namespace=".."` vorgegeben werden:

```
<!--
  The default namespace has to be declared,
  but any alias can be used for it.
-->
<testcase name="hasXFADData_UsingDefaultNamespace">
  <assertThat testDocument="xfa/xfa-enabled.pdf">
    <hasXFADData>
      <withNode tag="foo:log/foo:to"
                value="memory"
                defaultNamespace="http://www.xfa.org/schema/xci/2.6/"
      />
    </hasXFADData>
  </assertThat>
</testcase>
```

Beachten Sie, dass in diesem Beispiel einmal das Präfix `foo` und einmal das Präfix `bar` für den gleichen Namensraum verwendet wird. Das ist zwar seltsam, aber der Java-Standard verlangt ein **beliebiges** Präfix, das nicht weggelassen werden darf.

Das nächste Beispiel zeigt die Deklaration des Default-Namensraumes für das Tag `<matchingXPath />`:

```
<testcase name="hasXMPData_MatchingXPath_WithDefaultNamespace">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <matchingXPath expr="//foo:format = 'application/pdf'"
                     defaultNamespace="http://purl.org/dc/elements/1.1/"
      />
    </hasXMPData>
  </assertThat>
</testcase>
```

13.11. Konfiguration überprüfen

Wie schon in Kapitel 11.5: „Überprüfung der Konfiguration“ (S. 130) beschrieben, gibt es einen Test, der feststellt, ob alle benötigten Bibliotheken und Dateien im Classpath vorhanden sind:

```
<comment>
  This tests verifies that all required libraries and files are found.
  Additionally it logs some system properties and writes
  everything into both an XML file and an HTML formatted file.
</comment>

<testcase name="verifyRequiredFilesAndLibraries">
  <verifyInstallation verificationFile="verifyInstallation.vip"/>
</testcase>
```

Das Ergebnis wird in eine HTML-Datei geschrieben, deren Name in der Fehlermeldung angegeben wird:

Unit Test Results. Designed for use with [JUnit](#) and [Ant](#).

Class org.pdfunit.xml_VerifyInstallation

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
_VerifyInstallation	1	1	0	0	0.121	2013-10-25T18:04:37	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
verifyRequiredFilesAndLibraries	Error	Configuration ERROR. See output file 'verifyInstallation.vip.out.html'. org.verifyInstallation.VIPException: Configuration ERROR. See output file 'verifyInstallation.vip.out.html'. at org.verifyInstallation.VIP.verifyAndReport (VIP.java:191) at org.verifyInstallation.VIP.verifyInstallation (VIP.java:133) at com.pdfunit.AssertThat.installationIsClean (SourceFile:177) at org.pdfunit.xml._VerifyInstallation.verifyRequiredFilesAndLibraries (_VerifyInstallation.java:32)	0.119

[Properties »](#)

13.12. Versionshistorie

August 2011

Im November 2011 stellte ich meinem Kollegen Jürgen Sieben die damalige Version von PDFUnit-Java vor. Seine Frage, ob dafür auch eine XML-Schnittstelle geplant ist, setzte die Entwicklung von PDFUnit-XML in Gang. Ich begann mit der Ausarbeitung der prinzipiellen Umsetzbarkeit, hatte aber wenig Zeit, substantielle Arbeiten für die neue Idee zu erledigen.

2012

Die XML-Schnittstelle entstand in ihren wesentlichen Teilen im zweiten Quartal. Anschließend verschlang meine Schulungstätigkeit und die Weiterentwicklung von PDFUnit-Java die verfügbare Zeit.

Release 2013.01

Die Release-Bezeichnung ist identisch mit der Bezeichnung für PDFUnit-Java, weil beide die gleiche Funktionalität abdecken. Im zweiten Quartal 2013 entstand im das Handbuch in seiner deutschen Version.

Release 2014.06

Zahlreiche kleine Verbesserungen führten zu einer neuen Version von PDFUnit-Java. Die Funktionen wurden in PDFUnit-XML ebenfalls implementiert. Außerdem steht seit Juni 2014 die englische Version des Handbuches zur Verfügung.

Release 2015.10

Neben einigen neuen Funktionen ist die wesentliche Erweiterung der PDFUnit-Monitor. Er bedient die Zielgruppe der Nicht-Entwickler und entnimmt die Testinformationen aus einer oder mehrerer Excel-Dateien.

13.13. Nicht Implementiertes, Bekannte Fehler

Probleme mit Schreibrichtung RTL (right-to-left)

Momentan werden Texte mit der Schreibrichtung „right-to-left“ (RTL), beispielsweise PDF-Dokumente mit hebräischem Text, nicht richtig verarbeitet. Erfahrungen mit senkrechtem Text liegen noch gar nicht vor.

Textüberlauf in Formularfeldern

Im aktuellen Release 2054.10 wird der Textüberlauf in einem Formularfeld dann nicht erkannt, wenn der Text der letzten Zeile eines Feldes noch **sichtbar innerhalb** des Feldes **beginnt** , aber über das Ende des Feldes hinausgeht.

Formularfelder mit dem Attribut `hidden`

Die Eigenschaft `hidden` eines Formularfeldes wird in einigen Situationen nicht richtig verarbeitet. Anhand geeigneter Testdokumente wird dieses Probleme aktuell untersucht.

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 2054.10 werden keine Farben analysiert. Falls Farben dennoch getestet werden sollen, kann das über gerenderte Seiten geschehen. Die Kapitel 3.15: „Layout - gerenderte volle Seiten“ (S. 44) und 3.16: „Layout - gerenderte Seitenausschnitte“ (S. 45) beschreiben diese Art der Tests.

Ebenenbezogene (Layer) Inhalte

Der Vergleich auf Texte und Bilder bezieht sich noch nicht auf einzelne Ebenen.

PDF/A Validierung

Die Überprüfung eines PDF-Dokumentes auf die Einhaltung der PDF/A-Konformität wird erst im kommenden Release verfügbar sein.

Vollständige XMP-Daten

Im aktuellen Release werden nur die XMP-Daten der Dokument-Ebene extrahiert und ausgewertet. Zukünftig werden alle XMP-Daten extrahiert.

Feldübergreifende Plausibilitäten

Die Feldinhalte mancher XML-Tags sind gegenseitig abhängig. Solche Abhängigkeiten können mit XML Schema nicht spezifiziert und somit auch nicht durch den Parser geprüft werden. Es ist geplant, die feldübergreifenden Plausibilitäten durch Schematron zu implementieren. Einen guten Einstieg in das Thema Schematron finden Sie bei Wikipedia (<http://de.wikipedia.org/wiki/Schematron>).

Stichwortverzeichnis

A

Aktionen, 11
 beliebige Aktionen, 16
 benannte Aktionen, 14
 Close, 13
 Gleichheit von Aktionen, 77
 Goto, 14
 Goto Remote, 15
 Import Data, 14
 JavaScript, 14
 Launch Data, 14
 Open, 15
 Print, 15
 Reset Form, 16
 Save, 16
 Submit Form, 16
 URI, 16
 vergleichen, 77
 Whitespaces-Behandlung, 17
 Änderungsdatum, 24
 Anführungszeichen in Suchbegriffen, 139
 in Attributen, 141
 in Regulären Ausdrücken, 142
 in Tags, 140
 in XPath-Ausdrücken, 142
 Anhang, 17
 extrahieren, 105
 Anhänge, 17
 Anhänge vergleichen, 78
 Anzahl von PDF-Bestandteilen, 19
 Attachments, 17

B

Beispiele, 121
 Name des alten Vorstandes, 122
 Neues Logo auf jeder Seite, 121
 Passt Text in Formularfelder, 121
 Schachtelungstiefe von Bookmarks, 123
 Unterschrift des neuen Vorstandes, 122
 Benutzer-Passwort (user password), 49
 Berechtigungen, 20
 vergleichen, 79
 Bilder, 21
 Anzahl sichtbarer Bilder, 22
 Anzahl unterschiedlicher Bilder, 22
 aus PDF extrahieren, 107
 mit Datei vergleichen, 23
 N-zu-1 Vergleich, 23
 seitenbezogen testen, 23
 vergleichen, 80
 Bookmarks, 46

D

Datum
 Änderungsdatum, 24
 Erstellung eines Zertifikates, 26
 Erstellungsdatum, 24
 Existenz, 24
 Ober- und Untergrenze, 26
 Datumsauflösung, 25, 145
 Datumsformat
 Konfiguration, 25
 Datum vergleichen
 Änderungsdatum, 81
 Erstellungsdatum, 81
 Default-Namensraum, 71, 74, 90, 104, 145
 Deinstallation, 133
 Diff-Image, 84
 Dokumenteneigenschaften, 26
 als Key-Value-Paar testen, 28
 Custom-Property, 29
 vergleichen, 81
 Vergleichsmöglichkeiten, 27

E

Eigentümer-Passwort (owner password), 49
 Eingebettete Dateien
 Anzahl, 18
 Dateiname, 18
 Existenz, 17
 Inhalt, 18
 vergleichen, 78
 Erstellungsdatum, 24
 Erste Seite, 136
 Evaluationsversion, 126

F

Fast Web View, 29
 Feedback, 7
 Fehler
 Erwarteter Fehler, 10
 Fehlerbild, 84
 Fehlermeldungen, 144
 Feldeigenschaften
 nach XML extrahieren, 108
 Fluent Builder, 5
 Fluent Interface, 134
 Format, 30
 individuelle Größe, 31
 Maßeinheiten, 143
 mehrere Formate in einem Dokument, 31
 vergleichen, 82
 Formularfeld, 32
 Anzahl, 33
 Eigenschaften, 35

- Existenz, 32
- Größe, 35
- hidden-Attribut, 147
- Inhalt, 33
- JavaScript-Aktionen, 36
- mit XML-Datei vergleichen, 37
- mit XPath testen, 37
- Name, 33
- Textüberlauf, 38, 147
- Typ, 34
- Unicode, 36
- vergleichen, 82
- Formularfelder vergleichen
 - Anzahl, 82
 - Feldeigenschaften, 83
 - Feldnamen, 82
 - Inhalte, 83

G

- Gerade Seiten, 136
- Gleichheit
 - von Aktionen, 77
 - von Bildern, 80
 - von Dokumenteneigenschaften, 81
 - von Lesezeichen, 86
 - von Schriften, 51, 87
- Gültigkeitsdatum, 56

H

- Hilfsprogramme, 105
 - Anhänge extrahieren, 105
 - Bilder aus PDF extrahieren, 107
 - Feldeigenschaften nach XML extrahieren, 108
 - JavaScript extrahieren, 109
 - Lesezeichen nach XML extrahieren, 110
 - Named Destinations nach XML extrahieren, 117
 - PDF in PNG rendern, 111
 - PDF-Seitenausschnitte in PNG rendern, 112
 - Schrifteigenschaften nach XML extrahieren, 114
 - Signaturdaten nach XML extrahieren, 116
 - Unicode in Hex-Code wandeln, 118
 - XFA-Daten nach XML extrahieren, 119
 - XMP-Daten extrahieren, 119

I

- Installation, 124
 - iText-PDF, 126
 - Lizenzschlüssel, 126
 - Lizenzschlüssel beantragen, 126
 - neues Release, 132
 - new release PDFUnit-Java, 133
 - PDFUnit-Java, 126
 - PDFUnit-XML, 124
- Installation überprüfen, 127
- Instantiierung, 136
- iText Installation, 126

J

- JavaScript, 40
 - Existenz, 40
 - extrahieren, 109
 - Teilstrings vergleichen, 41
 - vergleichen, 83
 - Vergleich gegen eine Textdatei, 40
- Jede Seite, 136

K

- Konfiguration
 - Ausgabeverzeichnis für Fehlerbilder, 129
 - Interne Länderkennung, 129
 - mit Skript prüfen, 130, 146
 - mit Test prüfen, 131
 - PDF-Datumsformat, 129
 - Überprüfung, 130

L

- Language, 59
- Layer, 42
 - Anzahl, 42
 - Doppelte Namen, 43
 - Name, 42
- Layout
 - Seitenausschnitt, 45
 - vergleichen, 84
 - volle Seiten, 44
- Leerzeichen im Text, 17, 62, 138
- Lesezeichen, 46
 - Anzahl, 47
 - Existenz, 47
 - mit XML-Datei vergleichen, 49
 - mit XPath testen, 49
 - nach XML extrahieren, 110
 - ohne Sprungziel, 48
 - Sprungziel (Name einer Sprungmarke), 48
 - Sprungziel (Seitenzahl), 48
 - Sprungziel (URI), 48
 - Sprungziele, 48
 - Text (Label), 48
 - vergleichen, 85
- Letzte Seite, 136
- Lizenzschlüssel
 - beantragen, 126
 - installieren, 126

M

- Maßeinheiten, 143
 - DPI72, 143
 - Inch, 143
 - Millimeter, 143
 - Points, 143
 - Zentimeter, 143
- Mehrere Dokumente, 93

Metadaten (Siehe 'Dokumenteneigenschaften')

N

Named Destination, 46
vergleichen, 86

O

Owner Password, 49

P

Passwort testen, 49
PDF-Bestandteile vergleichen, 87
PdfDIFF, 97
PDFUnit-Java, 134
PDFUnit-Monitor, 95
 Export, 98
 Fehlerdetails, 96
 Filter, 96
 Import, 98
 Vergleich gegen Vorlage, 97
PDFUnit-NET, 135
PDFUnit-Perl, 134
PDF-Version, 68
 Versionsbereiche, 69
 zukünftige Versionen, 69

Q

Quickstart, 8

R

Rechteck definieren, 142
Reguläre Ausdrücke, 138

S

Schreibrichtung (right-to-left), 147
Schrifteigenschaften
 nach XML extrahieren, 114
Schriften, 50
 Anzahl, 50
 mit XML testen, 53
 mit XPath testen, 53
 Namen, 52
 Typen, 52
 vergleichen, 87
 Vergleichskriterien, 51
Schrifttypen, 52
Seiten
 gerendert vergleichen, 84
 in PNG rendern, 111
Seitenangaben mit Unter- und Obergrenze, 64
Seitenausschnitt
 Beispiel, 65
 definieren, 142
 in PNG rendern, 112
 Layout, 45

Layout validieren, 45
 mit Einheit, 144
 Text validieren, 62
Seitenauswahl, 136
 geschlossener Bereich, 137
 individuelle Seiten, 137
 offener Bereich, 137
Seitenzahlen als Testziel, 54
Signatur/Zertifikat, 54
 Anzahl, 55
 Existenz, 55
 Grund (Reason), 56
 Gültigkeitsdatum, 56
 mit XML-Datei vergleichen, 57
 mit XPath testen, 57
 nach XML extrahieren, 116
 Name, 55
 Namen vergleichen, 88
 Revision, 56
 Unterzeichner (Sign Name), 56
Sprachinformation (Language), 59
Sprungziel (Named Destination), 47
 nach XML extrahieren, 117
Starten aus einer IDE, 128
Starten von Konsole, 127
Syntaktischer Einstieg, 9

T

Tagging, 66
Technische Voraussetzungen, 124
Texte in Seitenausschnitten, 65
Texte - senkrecht, schräg, überkopf, 65
Texte validieren, 60
 Abwesenheit von Text, 61
 auf allen Seiten, 61
 auf bestimmten Seiten, 60
 in Seitenausschnitten, 62
 leere Seiten, 62
 mehrfache Suchbegriffe, 63
 Seitenangaben mit Unter- und Obergrenze, 64
 Zeilenumbruch, Leerzeichen, 62
Textüberlauf, 38
 aller Felder, 39
 eines Felder, 38
 technische Randbedingung, 39
Textvergleich, 88, 138
 in Seitenausschnitten, 88
Trapping, 67

U

Überblick
 Hilfsprogramme, 105
 Testbereiche, 10
 Vergleiche gegen ein Master-PDF, 76
Ungerade Seiten, 136
Unicode, 99

- einzelne Zeichen, 99
- in Fehlermeldungen, 101
- in Hex-Code wandeln, 118
- längere Texte, 99
- mit XML-Datei vergleichen, 99
- mit XPath testen, 100
- unsichtbare Zeichen, 102
- UTF-8 (ANT), 101
- UTF-8 (Eclipse), 101
- UTF-8 (Konsole), 100
- Update, 132
- Update PDFUnit-Java, 133
- User Password, 49

V

- Vergleiche gegen ein Master-PDF, 76
 - Aktionen, 77
 - Änderungsdatum, 81
 - Anhänge, 78
 - Anzahl verschiedener PDF-Bestandteile, 87
 - Berechtigungen, 79
 - Bilder, 80
 - Bilder auf bestimmten Seiten, 80, 80
 - Dokumenteneigenschaften, 81
 - Erstellungsdatum, 81
 - Fast WebView, 91
 - Fehlerbild, Diff-Image, 84
 - Formate, 82
 - Formularfelder, 82
 - gerenderte Seiten, 84
 - gerenderte Seitenausschnitte, 84
 - JavaScript, 83
 - Lesezeichen, 85
 - Named Destinations, 86
 - Schriften, 87
 - Signaturnamen, 88
 - Tagging, 91
 - Texte, 88
 - Texte in Seitenausschnitten, 88
 - XFA-Daten, 89
 - XMP-Daten, 90
- Verschlüsselungslänge, 50

W

- Whitespaces-Behandlung, 17, 62, 138
 - IGNORE, KEEP, NORMALIZE, 138, 138
 - Reguläre Ausdrücke, 139

X

- XFA-Daten, 69
 - auf einzelne Knoten prüfen, 70
 - Default-Namensraum, 71
 - Existenz, 69
 - mit XML-Datei vergleichen, 70
 - mit XPath testen, 71
 - nach XML extrahieren, 119

- vergleichen, 89
- XML
 - Daten extrahieren, 103
 - Default-Namensraum, 104
 - Namensraum, 104
- XMLUnit, 103, 145
- XMP-Daten, 72
 - auf einzelne Knoten prüfen, 73
 - Default-Namensraum, 74
 - Existenz, 72
 - mit XML-Datei vergleichen, 73
 - mit XPath testen, 74
 - nach XML extrahieren, 119
 - vergleichen, 90
- XPath
 - allgemeine Erläuterungen, 103
 - Ergebnistyp, 104
 - Kompatibilität, 104
- XPath Ergebnistyp, 89, 91
 - Bookean, 89

Z

- Zeilenumbruch im Text, 17, 62, 138
- Zertifikat (Siehe 'Signatur/Zertifikat')
- Zertifiziertes PDF, 74