

---

# **PDF automatisiert testen**

**PDFUnit-Java**

**Carsten Siedentop**

---



# Inhaltsverzeichnis

Vorwort .....	6
1. Über diese Dokumentation .....	7
2. Quickstart .....	9
3. Funktionsumfang .....	10
3.1. Überblick .....	10
3.2. Aktionen (Actions) .....	12
3.3. Anhänge (Attachments) .....	18
3.4. Anzahl verschiedener PDF-Bestandteile .....	21
3.5. Berechtigungen .....	22
3.6. Bilder in Dokumenten .....	23
3.7. Datum .....	26
3.8. Dokumenteneigenschaften .....	28
3.9. Fast Web View .....	31
3.10. Format .....	32
3.11. Formularfelder .....	33
3.12. Formularfelder, Textüberlauf .....	41
3.13. JavaScript .....	43
3.14. Layer .....	44
3.15. Layout - gerenderte volle Seiten .....	47
3.16. Layout - gerenderte Seitenausschnitte .....	48
3.17. Lesezeichen (Bookmarks) und Sprungziele .....	49
3.18. Passwort .....	53
3.19. Schriften .....	54
3.20. Seitenzahlen als Testziel .....	57
3.21. Signaturen und Zertifikate .....	58
3.22. Sprachinformation (Language) .....	63
3.23. Texte .....	64
3.24. Texte - in Ausschnitten einer Seite .....	70
3.25. Texte - senkrecht, schräg und überkopf .....	70
3.26. Tagging .....	71
3.27. Trapping-Info .....	72
3.28. Version .....	74
3.29. XFA Daten .....	75
3.30. XMP-Daten .....	78
3.31. Zertifiziertes PDF .....	81
4. Vergleiche gegen ein Master-PDF .....	82
4.1. Überblick .....	82
4.2. Aktionen vergleichen .....	83
4.3. Anhänge (Attachments) vergleichen .....	85
4.4. Berechtigungen vergleichen .....	85
4.5. Bilder vergleichen .....	86
4.6. Datumswerte vergleichen .....	87
4.7. Dokumenteneigenschaften vergleichen .....	88
4.8. Formate vergleichen .....	89
4.9. Formularfelder vergleichen .....	89
4.10. JavaScript vergleichen .....	91
4.11. Layout vergleichen (gerenderte Seiten) .....	91
4.12. Lesezeichen (Bookmarks) vergleichen .....	93
4.13. "Named Destinations" vergleichen .....	94
4.14. PDF-Bestandteile vergleichen .....	94
4.15. Schriften vergleichen .....	95
4.16. Signaturnamen vergleichen .....	95
4.17. Text vergleichen .....	96
4.18. XFA-Daten vergleichen .....	97

4.19. XMP-Daten vergleichen .....	98
4.20. Sonstige Vergleiche .....	99
5. Tests mit mehreren Dokumenten .....	101
6. PDFUnit-Monitor .....	103
7. Unicode .....	107
8. XPath-Einsatz .....	112
9. Hilfsprogramme zur Testunterstützung .....	114
9.1. Allgemeine Hinweise für alle Hilfsprogramme .....	114
9.2. Anhänge extrahieren .....	114
9.3. Bilder aus PDF extrahieren .....	116
9.4. Feldeigenschaften nach XML extrahieren .....	117
9.5. JavaScript extrahieren .....	118
9.6. Lesezeichen nach XML extrahieren .....	119
9.7. PDF-Dokument seitenweise in PNG umwandeln .....	120
9.8. PDF-Seite ausschnittsweise in PNG umwandeln .....	121
9.9. Schrifteigenschaften nach XML extrahieren .....	123
9.10. Signaturdaten nach XML extrahieren .....	125
9.11. Sprungziele nach XML extrahieren .....	126
9.12. Unicode-Texte in Hex-Code umwandeln .....	127
9.13. XFA-Daten nach XML extrahieren .....	128
9.14. XMP-Daten nach XML extrahieren .....	128
10. Praxisbeispiele .....	130
10.1. Passt ein Text in vorgefertigte Formularfelder? .....	130
10.2. Neues Logo auf jeder Seite .....	130
10.3. Unterschrift des neuen Vorstandes .....	131
10.4. Name des alten Vorstandes .....	132
10.5. Selenium und PDFUnit in Kombination .....	132
10.6. HTML2PDF - Hat die dynamische PDF-Erstellung funktioniert? .....	133
10.7. Caching von Testdokumenten .....	135
10.8. Schachtelungstiefe von Bookmarks .....	136
11. Installation, Konfiguration, Update .....	137
11.1. Technische Voraussetzungen .....	137
11.2. Installation ohne vorhandenes iText .....	137
11.3. Installation bei existierendem iText .....	139
11.4. Classpath in Eclipse, ANT, Maven definieren .....	139
11.5. Pfade über Systemumgebungsvariablen setzen .....	142
11.6. Einstellungen in der config.properties .....	143
11.7. Überprüfung der Konfiguration .....	144
11.8. Installation eines neuen Releases .....	146
11.9. Deinstallation .....	147
12. PDFUnit für Nicht-Java Systeme .....	148
12.1. Kurzer Blick auf PDFUnit-XML .....	148
12.2. Kurzer Blick auf PDFUnit-Perl .....	148
12.3. Kurzer Blick auf PDFUnit-NET .....	149
13. Anhang .....	150
13.1. Einsatz mit TestNG .....	150
13.2. Instantiierung der PDF-Dokumente .....	150
13.3. Seitenauswahl .....	150
13.4. Textvergleich .....	152
13.5. Behandlung von Whitespaces .....	153
13.6. Anführungszeichen in Suchbegriffen .....	154
13.7. Seitenausschnitt definieren .....	157
13.8. Maßeinheiten - Points, Millimeter, ... ..	158
13.9. Fehlermeldungen .....	159
13.10. Datumsauflösung .....	160
13.11. Default-Namensraum in XML .....	160

---

13.12. Konfiguration überprüfen .....	161
13.13. JAXP-Konfiguration .....	161
13.14. Versionshistorie .....	163
13.15. Nicht Implementiertes, Bekannte Fehler .....	163
Stichwortverzeichnis .....	165

# Vorwort

## Aktuelle Testsituation in Projekten

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

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

```
@Test
public void haveSameText_OnEveryPage() throws Exception {
    String filenameTest = PATH + "master/compareToMaster_copy.pdf";
    String filenameMaster = PATH + "master/compareToMaster.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameText(ON_EVERY_PAGE)
    ;
}
```

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 an Mitarbeiter der Softwareentwicklung, die mit der Entwicklung von PDF-Dokumenten betraut sind, Projektverantwortliche und an Mitarbeiter der Qualitätssicherung.

Es wird davon ausgegangen, dass Sie Grundkenntnisse in Java besitzen. Ein Kontakt mit JUnit oder TestNG und ein Grundverständnis über Testautomatisierung ist hilfreich, aber keine Voraussetzung.

## Code-Beispiele

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

## Javadoc

Die Javadoc-Dokumentation der API ist online verfügbar: <http://www.pdfunit.com/javadoc/index.html>.

## Andere Programmiersprachen

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

## Wenn es Probleme gibt

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

```
@Test
public void hasOnePage_de() throws Exception {
    String filename = PATH + "quickstart/quickstartDemo_de.pdf";

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

@Test
public void hasGreeting_de() throws Exception {
    String filename = PATH + "quickstart/quickstartDemo_de.pdf";

    String expectedGreeting = "Vielen Dank für die Nutzung unserer Serviceleistungen";
    AssertThat.document(filename)
        .hasText(ON_LAST_PAGE)
        .containing(expectedGreeting)
    ;
}

@Test
public void hasExpectedCharge_de() throws Exception {
    String filename = PATH + "quickstart/quickstartDemo_de.pdf";

    double upperLeftX = 172; // in Millimeter
    double upperLeftY = 178;
    double width = 20;
    double height = 9;
    ClippingArea inClippingArea = new ClippingArea(upperLeftX, upperLeftY, width, height);

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE, inClippingArea)
        .containing("29,89 Euro")
        // The value is intentionally wrong to demonstrate the error message.
    ;
}
```

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

#### Unit Test Results.

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

#### Class com.pdfunit.QuickstartTests\_de

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
<a href="#">QuickstartTests_de</a>	3	0	1		0.070	2013-10-14T18:39:26	NOTEBOOK64

#### Tests

Name	Status	Type	Time(s)
hasGreeting_de	Success		0.028
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.assertFoundOnEveryPage(ContentValidator.java:521) at com.pdfunit.validators.ContentValidator.containing(ContentValidator.java:185) at com.pdfunit.validators.ContentValidator.containing(ContentValidator.java:139) at com.pdfunit.QuickstartTests_de.hasExpectedCharge_de(QuickstartTests_de.java:65)	0.036
hasOnePage_de	Success		0.003

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

```
// The following code snippets demonstrates the instantiation of any PDFUnit test:

AssertThat.document(filename) 13.2: „Instantiierung der PDF-Dokumente“ (S. 150)
    .hasText(..)              13.3: „Seitenauswahl“ (S. 150)
    .containing(..)           13.4: „Textvergleich“ (S. 152)

    .hasField(..)              // Switch to one of many test scopes, see next listing.
    ...                       // Use test scope specific test methods.
    ...                       // Concatenation of tests methods is possible and intended.
;

AssertThat.document(filename) ❶
    .and(..)                   4.1: „Überblick“ (S. 82)
    ...
;

AssertThat.document(filename)
    .asRenderedPage(..)        3.15: „Layout - gerenderte volle Seiten“ (S. 47)
    ...
;
```

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

Es können auch mehrere PDF-Dokumente in einen Test einfließen. Solche Tests beginnen mit der Methode `AssertThat.eachDocument(..)`:

```
// The following snippets demonstrates how to instantiate
// tests using multiple documents:
...
File[] files = {file1, file2, file3};
AssertThat.eachDocument(filename) ❷ 5: „Tests mit mehreren Dokumenten“ (S. 101)
    .hasText(..)
    .containing(..)
;
```

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

#### Exception für erwartete Fehler

Tests, die einen Fehler erwarten, müssen die `PDFUnitValidationException` abfangen. In früheren Releases musste die Klasse `PDFUnitError` abgefangen werden. Diese Klasse ist im Release 2015.10 noch enthalten, wurde aber als 'deprecated' markiert. Sie wird im kommenden Release gelöscht.

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

```
@Test(expected=PDFUnitValidationException.class)
public void isSigned_DocumentNotSigned() throws Exception {
    String filename = PATH + "signed/notSigned.pdf";

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

## Testbereiche

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

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

.areBothForFastWebView()           4.20: „Sonstige Vergleiche“ (S. 99)
.asRenderedPage()                  3.15: „Layout – gerenderte volle Seiten“ (S. 47)

.containsImage(...)                3.6: „Bilder in Dokumenten“ (S. 23)
.containsOneImageOf(...)           3.6: „Bilder in Dokumenten“ (S. 23)

.hasAnyAction()                    3.2: „Aktionen (Actions)“ (S. 12)
.hasAuthor()                       3.8: „Dokumenteneigenschaften“ (S. 28)
.hasBookmark()                     3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 49)
.hasBookmarks()                    3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 49)
.hasChainedAction()                3.2: „Aktionen (Actions)“ (S. 12)
.hasCloseAction()                  3.2: „Aktionen (Actions)“ (S. 12)
.hasCreationDate()                 3.7: „Datum“ (S. 26)
.hasCreator()                      3.7: „Datum“ (S. 26)
.hasEmbeddedFile(...)              3.3: „Anhänge (Attachments)“ (S. 18)
.hasEncryptionLength(...)          3.18: „Passwort“ (S. 53)
.hasField(...)                     3.11: „Formularfelder“ (S. 33)
.hasFields()                       3.11: „Formularfelder“ (S. 33)
.hasFont()                         3.19: „Schriften“ (S. 54)
.hasFonts()                       3.19: „Schriften“ (S. 54)
.hasFormat(...)                    3.10: „Format“ (S. 32)
.hasImportDataAction()             3.2: „Aktionen (Actions)“ (S. 12)
.hasJavaScript()                   3.13: „JavaScript“ (S. 43)
.hasJavaScriptAction()             3.2: „Aktionen (Actions)“ (S. 12)
.hasKeywords()                     3.8: „Dokumenteneigenschaften“ (S. 28)
.hasLaunchAction()                 3.2: „Aktionen (Actions)“ (S. 12)
.hasLayer()                        3.14: „Layer“ (S. 44)
.hasLayers()                       3.14: „Layer“ (S. 44)
.hasLessPagesThan()                3.20: „Seitenzahlen als Testziel“ (S. 57)
.hasLocale(...)                    3.22: „Sprachinformation (Language)“ (S. 63)
.hasLocalGotoAction()              3.2: „Aktionen (Actions)“ (S. 12)
.hasModificationDate()             3.7: „Datum“ (S. 26)
.hasMorePagesThan()                3.20: „Seitenzahlen als Testziel“ (S. 57)
.hasNamedAction()                  3.2: „Aktionen (Actions)“ (S. 12)
.hasNamedDestination()             3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 49)

.hasNoAuthor()                     3.8: „Dokumenteneigenschaften“ (S. 28)
.hasNoCreationDate()               3.7: „Datum“ (S. 26)
.hasNoCreator()                    3.8: „Dokumenteneigenschaften“ (S. 28)
.hasNoKeywords()                   3.8: „Dokumenteneigenschaften“ (S. 28)
.hasNoLocale()                     3.22: „Sprachinformation (Language)“ (S. 63)
.hasNoModificationDate()           3.7: „Datum“ (S. 26)
.hasNoProducer()                   3.8: „Dokumenteneigenschaften“ (S. 28)
.hasNoProperty()                   3.8: „Dokumenteneigenschaften“ (S. 28)
.hasNoSubject()                    3.8: „Dokumenteneigenschaften“ (S. 28)
.hasNoText()                       3.23: „Texte“ (S. 64)
.hasNoTitle()                      3.8: „Dokumenteneigenschaften“ (S. 28)
.hasNoXFADData()                   3.29: „XFA Daten“ (S. 75)
.hasNoXMPData()                    3.30: „XMP-Daten“ (S. 78)

.hasNumberOf...()                  3.4: „Anzahl verschiedener PDF-Bestandteile“ (S. 21)

.hasOCG()                          3.14: „Layer“ (S. 44)
.hasOCGs()                         3.14: „Layer“ (S. 44)
.hasOpenAction()                   3.2: „Aktionen (Actions)“ (S. 12)
.hasOwnerPassword(...)             3.18: „Passwort“ (S. 53)
.hasPermission()                   3.5: „Berechtigungen“ (S. 22)
.hasPrintAction()                   3.2: „Aktionen (Actions)“ (S. 12)
.hasProducer()                     3.8: „Dokumenteneigenschaften“ (S. 28)
.hasProperty(...)                  3.8: „Dokumenteneigenschaften“ (S. 28)
.hasRemoteGotoAction()             3.2: „Aktionen (Actions)“ (S. 12)
.hasResetFormAction()              3.2: „Aktionen (Actions)“ (S. 12)
.hasSaveAction()                   3.2: „Aktionen (Actions)“ (S. 12)
.hasSignature(...)                 3.21: „Signaturen und Zertifikate“ (S. 58)
.hasSignatures()                   3.21: „Signaturen und Zertifikate“ (S. 58)
.hasSignedSignatureFields()         3.21: „Signaturen und Zertifikate“ (S. 58)

... continued
```

```

... continuation:

.hasSubject()                3.8: „Dokumenteigenschaften“ (S. 28)
.hasSubmitFormAction()      3.2: „Aktionen (Actions)“ (S. 12)
.hasText(...)               3.23: „Texte“ (S. 64)
.hasTitle()                 3.8: „Dokumenteigenschaften“ (S. 28)
.hasTrappingInfo(...)       3.27: „Trapping-Info“ (S. 72)
.hasUnsignedSignatureFields() 3.21: „Signaturen und Zertifikate“ (S. 58)
.hasURIAction()             3.2: „Aktionen (Actions)“ (S. 12)
.hasUserPassword(...)       3.18: „Passwort“ (S. 53)
.hasVersion()               3.28: „Version“ (S. 74)
.hasXFADData()              3.29: „XFA Daten“ (S. 75)
.hasXMPData()               3.30: „XMP-Daten“ (S. 78)

.haveSame...()              4.1: „Überblick“ (S. 82)

.isCertified()              3.31: „Zertifiziertes PDF“ (S. 81)
.isCertifiedFor(...)        3.31: „Zertifiziertes PDF“ (S. 81)
.isLinearizedForFastWebView() 3.9: „Fast Web View“ (S. 31)
.isSigned()                 3.21: „Signaturen und Zertifikate“ (S. 58)
.isTagged()                 3.26: „Tagging“ (S. 71)

... (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 Testmethoden zur Verfügung:

```

// Simple tests:

.hasAnyAction()
.hasNumberOfActions(...)
.hasNumberOfJavaScriptActions(...)

... continued

```

```

... continuation:

// Action specific tests:

.hasAnyAction()
.hasChainedAction()
.hasCloseAction()
.hasImportDataAction()
.hasJavaScriptAction()
.hasLaunchAction()
.hasLaunchAction().toLaunch(..)
.hasLocalGotoAction()
.hasLocalGotoAction().toDestination(..)
.hasNamedAction()
.hasOpenAction()
.hasOpenAction().withDestinationToPage(..)
.hasPrintAction()
.hasRemoteGotoActionTo(..)
.hasRemoteGotoActionTo(..).toDestination(..)
.hasRemoteGotoActionTo(..).toPage(..)
.hasResetFormAction()
.hasSaveAction()
.hasSubmitFormAction()
.hasURIAction()

```

Alle Tests mit Aktionen vergleichen Zeichenketten, für die es gibt folgende Vergleichsfunktionen gibt:

```

// Comparing text in actions:

.hasXXXAction().containing(..)
.hasXXXAction().containing(.., WhitespaceProcessing) ❶
.hasXXXAction().matchingComplete(..)
.hasXXXAction().matchingComplete(.., WhitespaceProcessing) ❷
.hasXXXAction().matchingRegex(..)

// or in this form:

.hasXXXAction().xxx().containing(..)
.hasXXXAction().xxx().containing(.., WhitespaceProcessing) ❸
.hasXXXAction().xxx().matchingComplete(..)
.hasXXXAction().xxx().matchingComplete(.., WhitespaceProcessing) ❹
.hasXXXAction().xxx().matchingRegex(..)

```

❶❷❸❹ Der Umgang mit Whitespaces wird im Kapitel 13.5: „Behandlung von Whitespaces“ (S. 153) beschrieben.

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

Die folgenden Abschnitte zeigen Beispiele für verschiedene Aktionen.

## Close-Actions

Close-Actions werden beim Schließen eines PDF-Dokumentes ausgeführt. Ein Test sieht so aus:

```

@Test
public void hasCloseAction_WithContent_ContentAsString() throws Exception {
    String filename = PATH + "actions/documentCloseAction.pdf";
    String script = "app.alert('A sample for a 'DOCUMENT_CLOSE'-action');";

    AssertThat.document(filename)
        .hasCloseAction()
        .matchingComplete(script, IGNORE_WHITESPACES)
    ;
}

```

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

```
@Test
public void hasCloseAction_MatchingComplete_ContentFromReader() throws Exception {
    String filenamePdf = PATH + "actions/documentCloseAction.pdf";
    String filenameScript = PATH + "actions/documentCloseAction.js";
    Reader scriptFile = new FileReader(filenameScript);

    AssertThat.document(filenamePdf)
        .hasCloseAction()
        .matchingComplete(scriptFile, IGNORE_WHITESPACES)
        ;
}
```

Der zweite Parameter steuert die Behandlung der Whitespaces. Die Standard-Whitespace-Behandlung ist NORMALIZE\_WHITESPACES.

## ImportData-Actions

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

```
@Test
public void hasImportDataAction_MatchingFilename() throws Exception {
    String filenamePDF = PATH + "actions/chainedActions.pdf";
    String filenameToImport = "build.xml";

    AssertThat.document(filenamePDF)
        .hasImportDataAction()
        .matchingComplete(filenameToImport)
        ;
}
```

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:

```
@Test
public void hasJavaScriptAction_ScriptFromInputStream() throws Exception {
    String filename = PATH + "javascript/bookmarkWithJavaScriptAction_OneSimpleAlert.pdf";
    String scriptFileName = PATH + "javascript/javascriptAction_OneSimpleAlert.js";
    InputStream scriptFileAsInputStream = new FileInputStream(scriptFileName);

    AssertThat.document(filename)
        .hasJavaScriptAction()
        .matchingComplete(scriptFileAsInputStream)
        ;
}
```

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:

```
@Test
public void hasLaunchAction_Notepad_Print() throws Exception {
    String filename = PATH + "actions/launchActionToFile.pdf";
    String withOperation = "print";
    String application = "c:/windows/notepad.exe";

    AssertThat.document(filename)
        .hasLaunchAction()
        .toLaunch(application, withOperation)
        ;
}
```

Während des Tests werden lediglich die Inhalte der Variablen `application` und `withOperation` 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:

```
@Test
public void hasNamedAction_WithName_NextPage() throws Exception {
    String filename = PATH + "actions/namedActionsNextPages.pdf";

    AssertThat.document(filename)
        .hasNamedAction()
        .WithName()
        .matchingComplete("/NextPage"); // note the leading slash
    ;
}
```

## Goto-Actions

Goto-Actions benötigen ein Sprungziel in derselben Datei:

```
@Test
public void hasGotoAction_ToNamedDestination() throws Exception {
    String filename = PATH + "actions/bookmarksWithPdfOutline.pdf";
    String destinationName21 = "destination2.1";

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

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

## GotoRemote-Actions

GotoRemote-Actions benötigen ein Sprungziel in einer Zieldatei:

```
@Test
public void hasGotoRemoteActionTo_NamedDestination() throws Exception {
    String filename = PATH + "actions/gotoRemotePageAction.pdf";
    String remoteFileName = "destination.pdf";
    String destinationName = "destination-3";

    AssertThat.document(filename)
        .hasRemoteGotoActionTo(remoteFileName)
        .toDestination(destinationName)
    ;
}
```

```
@Test
public void hasGotoRemoteAction_ToPage() throws Exception {
    String filename = PATH + "actions/gotoRemotePageAction.pdf";
    String remoteFile = "destination.pdf";
    int pageNumber = 4;

    AssertThat.document(filename)
        .hasRemoteGotoActionTo(remoteFile)
        .toPage(pageNumber)
    ;
}
```

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.

```
@Test
public void hasOpenAction_MultipleInvocation() throws Exception {
    String filename = PATH + "actions/documentOpenAction_Print.pdf";
    String expectedContent = "this.print(true);";

    AssertThat.document(filename)
        .hasOpenAction()
        .containing(expectedContent)
        .matchingRegex("(?ms).*print(.*)" )
    ;
}
```

Zusätzlich zu den Textvergleichsmethoden gibt es für Open-Actions noch die Testfunktion `withDestinationToPage()`:

```
@Test
public void hasOpenAction_GotoPage2() throws Exception {
    String filename = PATH + "actions/documentOpenAction_Goto.pdf";
    int destinationPageNumber = 2;

    AssertThat.document(filename)
        .hasOpenAction()
        .withDestinationToPage(destinationPageNumber)
    ;
}
```

## 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.

```
@Test
public void hasPrintAction_WillPrint() throws Exception {
    String filename = PATH + "actions/documentPrintActions.pdf";
    String scriptWillPrint = "app.alert('A sample for a WILL_PRINT-action');";

    AssertThat.document(filename)
        .hasPrintAction()
        .matchingComplete(scriptWillPrint)
    ;
}
```

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:

```
@Test
public void hasResetFormAction() throws Exception {
    String filename = PATH + "acrofields/javaScriptForFields.pdf";

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

## 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.



```
@Test
public void haveSaveActions() throws Exception {
    String filename = PATH + "actions/documentSaveActions.pdf";
    String expectedContent1 = "app.alert('A sample for a WILL_SAVE-action');";
    String expectedContent2 = "app.alert('A sample for a DID_SAVE-action');";

    AssertThat.document(filename)
        .hasSaveAction()
        .matchingComplete(expectedContent1)
        .matchingComplete(expectedContent2)
    ;
}
```

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:

```
@Test
public void hasSubmitFormAction_ToUri() throws Exception {
    String filename = PATH + "acrofields/javaScriptForFields.pdf";
    String url = "http://www.geek-tutorials.com/java/itext/submit.php";

    AssertThat.document(filename)
        .hasSubmitFormAction()
        .withDestination()
        .matchingComplete(url)
    ;
}
```

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:

```
@Test
public void hasURIAction() throws Exception {
    String filename = PATH + "actions/noBookmarks-manyActions.pdf";

    AssertThat.document(filename)
        .hasURIAction()
        .withURI()
        .matchingComplete("http://www.imdb.com/")
    ;
}
```

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.

Die Methode `withURI()` kann auch weggelassen werden. Sie hat keine eigene Funktionalität und ist nur vorhanden, um die API besser „fließen“ zu lassen.

## Any-Action - beliebige Aktionen

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

```
@Test
public void hasAnyAction_DifferentKindOfActions() throws Exception {
    String filename = PATH + "actions/chainedActions.pdf";
    String javascriptAction = "app.alert('Demo: the first action of five.');"
    String printDialogAction = "this.print(true)";

    String uriAction = "http://www.google.de";
    String launchAction = "c:/windows/notepad.exe";

    String gotoRemoteFile = "build.xml";
    String gotoDestinationInFile = "/1";

    AssertThat.document(filename)
        .hasAnyAction()
        .matchingComplete(javascriptAction)
        .matchingComplete(uriAction)
        .matchingComplete(launchAction)
        .matchingComplete(printDialogAction)
    ;
}
```

## Whitespace-Behandlung in Vergleichen

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

```
@Test
public void hasCloseAction_Containing_ContentFromReader() throws Exception {
    String filename = PATH + "actions/documentCloseAction.pdf";
    Reader scriptFile = new FileReader(PATH + "actions/documentCloseAction.js");

    AssertThat.document(filename)
        .hasCloseAction()
        .containing(scriptFile, IGNORE_WHITESPACES)
    ;
}
```

Das Kapitel 13.5: „Behandlung von Whitespaces“ (S. 153) geht ausführlich auf den flexiblen Umgang mit Whitespaces ein.

## 3.3. Anhänge (Attachments)

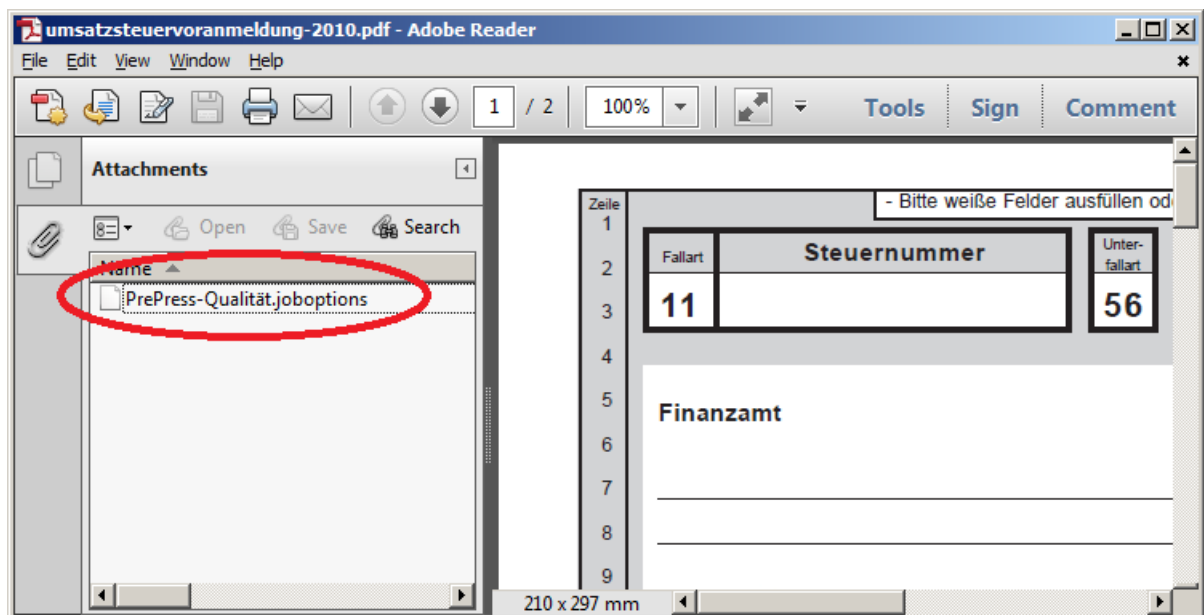
### Überblick

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

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

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

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



## Existenz

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

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

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

## Anzahl

Etwas weiter geht die Prüfung der Anzahl der eingebetteten Dateien:

```
@Test
public void hasNumberOfEmbeddedFiles() throws Exception {
    String filename = PATH + "embeddedfiles/umsatzsteuervoranmeldung-2010.pdf";

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

## Dateiname

Danach kommen die Namen der Dateien:

```
@Test
public void hasEmbeddedFile_WithName() throws Exception {
    String filename = PATH + "embeddedfiles/umsatzsteuervoranmeldung-2010.pdf";

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

## Inhalt

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

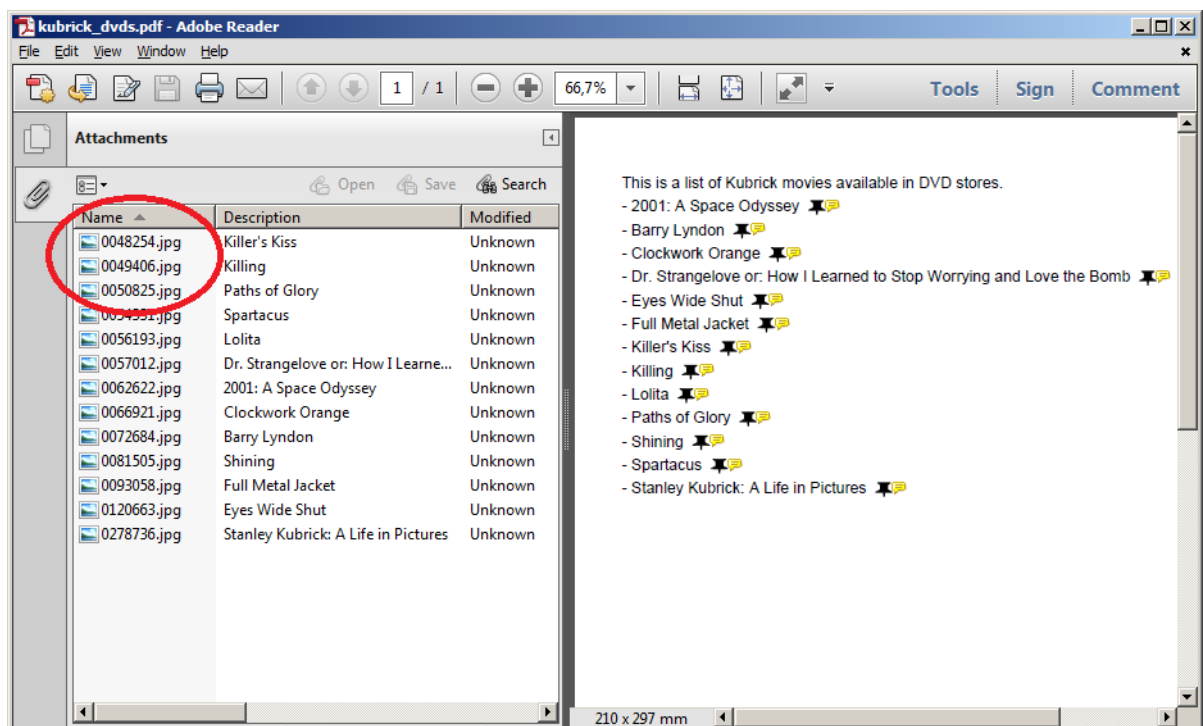
```
@Test
public void hasEmbeddedFile_WithContent() throws Exception {
    String pdfFileName = PATH + "embeddedfiles/umsatzsteuervoranmeldung-2010.pdf";
    String embeddedFileName = PATH + "embeddedfiles/PrePress-Qualität.joboptions";

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

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

## Mehrfachaufrufe

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



Mehrere Dateien können mit einem Test überprüft werden. Wählen Sie aber einen passenderen Namen für die Testmethode:

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

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

Wenn die eingebetteten Dateien nicht als eigene Datei vorliegen, können sie mit dem Hilfsprogramm „ExtractEmbeddedFiles“ aus einem bestehenden PDF-Dokument (Master-PDF) extrahiert werden. Das Programm wird in Kapitel 9.2: „Anhänge extrahieren“ (S. 114) genauer beschrieben.

### 3.4. Anzahl verschiedener PDF-Bestandteile

#### Überblick

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

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

- ❶❸ Prüfungen auf die Anzahl von Bildern werden in Kapitel 3.6: „Bilder in Dokumenten“ (S. 23) beschrieben.
- ❷ Prüfungen auf die Anzahl von Seiten eines PDF-Dokumentes werden in Kapitel 3.20: „Seitenzahlen als Testziel“ (S. 57) beschrieben.

#### Beispiele

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

```
@Test
public void hasNumberOfFields() throws Exception {
    String filename = PATH + "acrofields/simpleRegistrationForm.pdf";

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

```
@Test
public void hasNumberOfBookmarks() throws Exception {
    String filename = PATH + "bookmarks/manyBookmarks.pdf";

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

Alle Prüfungen können verkettet werden:

```
@Test
public void testHugeDocument_MultipleInvocation() throws Exception {
    String filename = PATH + "performance/groovy_wiki-snapshot_1370.pdf";

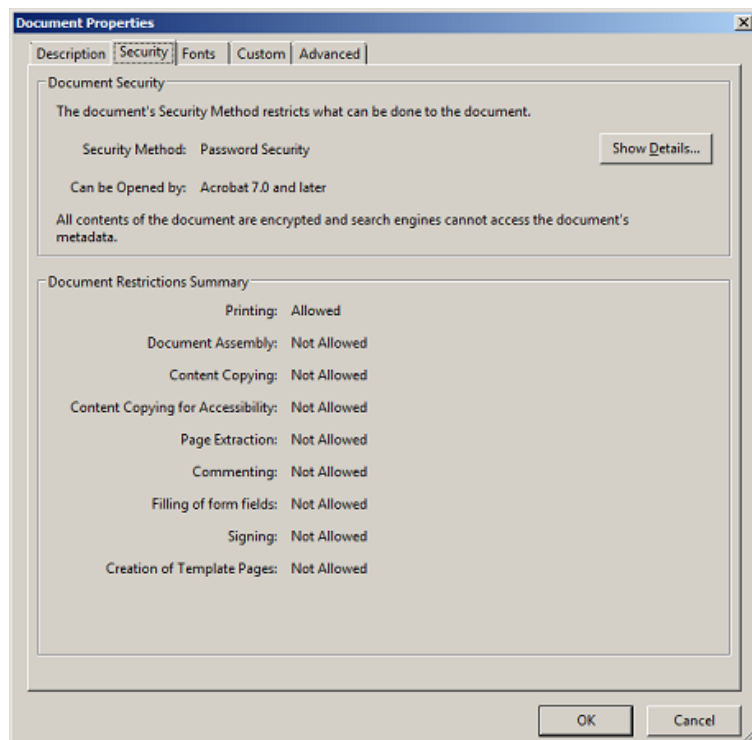
    AssertThat.document(filename)
        .hasNumberOfPages(1370)
        .hasNumberOfBookmarks(565)
        .hasNumberOfActions(1896)
        .hasNumberOfEmbeddedFiles(0)
    ;
}
```

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 Maven-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 passenden Testmethoden überprüfen. Alle Methoden werden mit einem Erwartungswert aufgerufen, der die Werte `true` oder `false` haben kann:

```
// Testing permissions:
.toAllowScreenReaders(..)
.toAssembleDocument(..)
.toCopyContent(..)           ❶
.toExtractContent(..)        ❷
.toFillInFields(..)
.toModifyAnnotations(..)
.toModifyContent(..)
.toPrint(..)
.toPrintInDegradedQuality(..)
```

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

### Beispiel

```
@Test
public void hasPermission() throws Exception {
    String filename = PATH + "permissions/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
        .hasPermission()
        .toPrint(true)
        .toModifyContent(false)
    ;
}
```

## 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 Testmethoden begegnet werden:

```
// Testing images in PDF:
.containsImage(..)
.hasNumberOfDifferentImages(..)
.hasNumberOfVisibleImages(..)
```

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 Testmethoden. Die Methode `hasNumberOfDifferentImages(..)` überprüft die Anzahl der **internen** Bilder während die Methode `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:

```
@Test
public void hasNumberOfDifferentImages() throws Exception {
    String filename = PATH + "images/imageDemo.pdf";

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

Wie kommt man in diesem Beispiel auf die „2“? Wie können Sie für ein bestimmtes PDF wissen, welche Bilder tatsächlich intern gespeichert sind? Zur Beantwortung dieser Fragen extrahieren Sie alle Bilder Ihres PDF-Dokumentes mit dem von PDFUnit mitgelieferten Hilfsprogramm `ExtractImages`. Eine Beschreibung dieses Programms steht in Kapitel 9.3: „Bilder aus PDF extrahieren“ (S. 116).

### Anzahl sichtbarer Bilder im Dokument

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

```
@Test
public void hasNumberOfVisibleImages() throws Exception {
    String filename = PATH + "images/imageDemo.pdf";

    AssertThat.document(filename)
        .hasNumberOfVisibleImages(6)
    ;
}
```

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:

```
@Test
public void hasNumberOfVisibleImages_OnPage3() throws Exception {
    String filename = PATH + "images/imageDemo.pdf";
    PagesToUse ON_PAGE_3 = PagesToUse.getPage(3);

    AssertThat.document(filename)
        .hasNumberOfVisibleImages(2, ON_PAGE_3)
    ;
}
```

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 13.3: „Seitenauswahl“ (S. 150) ausführlich beschrieben.

## Bestimmte Bilder vergleichen

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

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

    AssertThat.document(filename)
        .containsImage(imageFile, ON_ANY_PAGE)
    ;
}
```

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

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

```
// Types for images:

.containsImage(String imageFileName, PagesToUse pagesToUse);
.containsImage(File imageFile, PagesToUse pagesToUse);
.containsImage(InputStream imageStream, PagesToUse pagesToUse);
```

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 die Testmethode `containsOneImageOf(..)`:



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

    String documentSignedByAlex = PATH + "images/letter-signed-by-alex.pdf";
    AssertThat.document(documentSignedByAlex)
        .containsOneImageOf(allPossibleImages, ON_LAST_PAGE)
        ;

    String documentSignedByBob = PATH + "images/letter-signed-by-bob.pdf";
    AssertThat.document(documentSignedByBob)
        .containsOneImageOf(allPossibleImages, ON_LAST_PAGE)
        ;
}
```

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 13.3: „Seitenauswahl“ (S. 150) beschrieben.

Hier ein paar Beispiele:

```
@Test
public void containsImage_OnEveryPageAfter4() throws Exception {
    String filename = PATH + "images/imageDemo.pdf";
    String imageFileName = PATH + "images/apache-software-foundation-logo.png";

    AssertThat.document(filename)
        .containsImage(imageFileName, OnEveryPage.after(4))
        ;
}
```

```
@Test
public void containsImage_OnMultipleSelectedPages() throws Exception {
    String filename = PATH + "images/imageDemo.pdf";
    String imageFileName = PATH + "images/apache-software-foundation-logo.png";
    PagesToUse ON_PAGES_1_5 = PagesToUse.getPages(1, 5);

    AssertThat.document(filename)
        .containsImage(imageFileName, ON_PAGES_1_5)
        ;
}
```

Mehrfache Aufrufe der Testmethode sind möglich. Überlegen Sie aber, ob es nicht sinnvoller ist, für das folgende Beispiel zwei getrennte Tests zu schreiben:

```
@Test
public void containsImage_MultipleInvocation() throws Exception {
    String filename = PATH + "images/imageDemo.pdf";
    String imageFileANTLogo = PATH + "images/apache-ant-logo.png";
    String imageFileASFLogo = PATH + "images/apache-software-foundation-logo.png";
    PagesToUse ON_PAGE_3 = PagesToUse.getPage(3);

    AssertThat.document(filename)
        .containsImage(imageFileASFLogo, OnEveryPage.after(4))
        .containsImage(imageFileANTLogo, ON_PAGE_3)
        ;
}
```

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. 86).

## 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.

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

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

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

### Existenz eines Datums

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

```
@Test
public void hasCreationDate() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";

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

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

```
@Test
public void hasCreationDate_NoDateInPDF() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_noDateFields.pdf";

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

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

### Datumsauflösung

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

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

```
// Constants for date resolution:

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

Ein Test sieht dann so aus:

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

    AssertThat.document(filename)
        .hasCreationDate()
        .matchingComplete(expectedCreationDate, AS_DATE) ❷
        .matchingComplete(expectedCreationDate, AS_DATETIME) ❸
    ;
}
```

- ❶ Die Hilfsklasse `com.pdfunit.util.DateHelper` erleichtert es Ihnen, für den erwarteten Datumswert eine Instanz von `java.util.Calendar` zu erzeugen.
- ❷❸ Beide Konstanten werden durch die Enumeration `com.pdfunit.DateResolution` definiert.

Durch die Verwendung der Klasse `java.util.Calendar` wird das erwartete Datum formatunabhängig gemacht. Da aber das PDF-interne Datum formatiert vorliegt, muss dessen Format-String durch einen Eintrag in der Konfigurationsdatei `config.properties` definiert werden.

## 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, wird für Monat und Tag der „1. Januar“. 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:

```
@Test
public void hasProperty_CreationDate() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";

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

## Datumstest mit Ober- und Untergrenze

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

```

@Test
public void hasCreationDate_Before() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";
    Calendar creationDateUpperLimit = DateHelper.getCalendar("20991231", "yyyyMMdd");

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

```

```

@Test
public void hasCreationDate_After() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";
    Calendar creationDateLowerLimit = DateHelper.getCalendar("19990101", "yyyyMMdd");

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

```

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

## 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. 58) 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 Methoden stehen zur Validierung der Metadaten zur Verfügung:

```

// Testing document properties:

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

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

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

```

- ❶ Tests auf das Erstellungs- und Änderungsdatum werden in einem eigenen Kapitel beschrieben, weil sich die Art der Tests von denen der anderen Metadaten unterscheidet. Siehe Kapitel 3.7: „Datum“ (S. 26).

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. 88) beschrieben.

## Autor validieren ...

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

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

```
@Test
public void hasAuthor() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";

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

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

```
@Test
public void hasNoAuthor() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_noAuthorTitleSubjectKeywordsApplication.pdf";

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

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

```
@Test
public void hasAuthor_MatchingComplete() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";

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

Verschiedene Vergleichsmethoden stehen zur Verfügung. Die Methodennamen sind selbsterklärend:

```
// Comparing text for author, creator, keywords, producer, subject, title:
.containing(expectedValue)
.endsWith(expectedValue)
.matchingComplete(expectedValue)
.matchingRegex(expectedValue)
.notContaining(expectedValue)
.notMatching(expectedValue)
.startingWith(expectedValue)
```

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

Alle Vergleichsmethoden arbeiten case-sensitiv.

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

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

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

Für jede Dokumenteneigenschaft gibt es die Methoden `hasXXX()` und `hasNoXXX()`.

Die Testmethoden können auch verkettet werden:

```
@Test
public void hasKeywords_allTextComparingMethods() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";

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

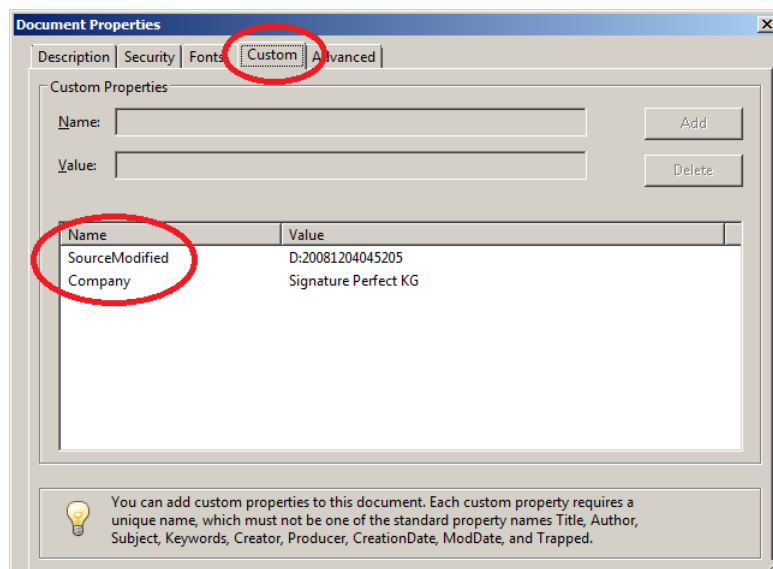
## Allgemeine Prüfung als Key-Value-Paar

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

```
@Test
public void hasProperty_StandardProperties() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";

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

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



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

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

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

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

```
@Test
public void hasNoProperty() throws Exception {
    String filename = PATH + "customproperties/Leitfaden_Elektronische_Signatur.pdf";

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

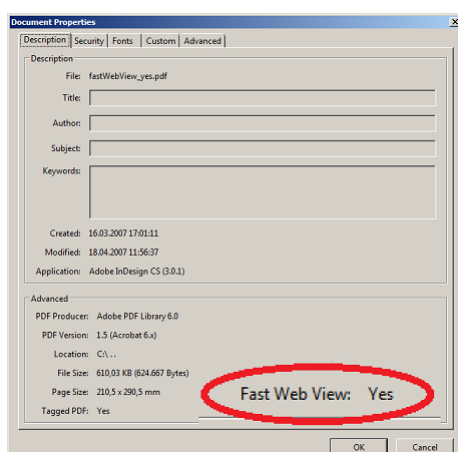
Ab PDF-1.4 existiert die Möglichkeit, Metadaten intern als XML zu speichern (Extensible Metadata Platform, XMP). Das Kapitel 3.30: „XMP-Daten“ (S. 78) geht darauf ausführlich ein.

## 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 die folgende Testmethode einen „grünen Balken“:

```
// Testing linearization:
.isLinearizedForFastWebView()
```

## Beispiel

```
@Test
public void isLinearizedForFastWebView() throws Exception {
    String filename = PATH + "fastWebView/fastWebView_yes.pdf";

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

## 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.

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

### Dokumente mit einheitlichem Seitenformat

Übliche Seitenformate können mit Konstanten geprüft werden:

```
import static com.pdfunit.Constants.*;
...
@Test
public void hasFormat_A4Landscape() throws Exception {
    String filename = PATH + "format/format_A4-Landscape.pdf";

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

```
import static com.pdfunit.Constants.*;
...
@Test
public void hasFormat_LetterPortrait() throws Exception {
    String filename = PATH + "format/format_Letter-Portrait.pdf";

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

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

Auch individuelle Papierformate können geprüft werden:

```
@Test
public void hasFormat_FreeFormat_1117x836_mm() throws Exception {
    String filename = PATH + "format/physical-map-of-the-world-1999_1117x863mm.pdf";
    double heightMM = 1117.6;
    double widthMM = 863.8;
    DocumentFormat formatMM = new DocumentFormat(widthMM, heightMM, MILLIMETER);❸

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



```
@Test
public void hasFormat_FreeFormat_10x15_cm() throws Exception {
    String filename = PATH + "format/format_individual-10x15-cm.pdf";
    double width = 10.0;
    double height = 15.0;
    DocumentFormat formatCM = new DocumentFormat(width, height, CENTIMETER);

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

- ③④ Erlaubte Einheiten sind POINTS, MILLIMETER, CENTIMETER, INCH und DPI72. DPI72 und POINTS sind gleichwertig. Die Einheiten sind in der Klasse `com.pdfunit.Constants` definiert.

Das Thema der verschiedenen Papierformate und deren Maße in Points, Millimeter und Inches wird bei [www.prepressure.com](http://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:

```
@Test
public void hasFormat_DifferentFormatsOnDifferentPages() throws Exception {
    String filename = PATH + "format/format_multiple-formats-on-individual-pages.pdf";
    PagesToUse ON_PAGE_3 = PagesToUse.getPage(3);

    AssertThat.document(filename)
        .hasFormat(A4_LANDSCAPE, ON_FIRST_PAGE)
        .hasFormat(A5_PORTRAIT, ON_PAGE_3)
        ;
}
```

Formattests können auf beliebige einzelne Seiten und Seitenbereiche eingeschränkt werden, wie es in Kapitel 13.3: „Seitenauswahl“ (S. 150) beschrieben ist:

```
@Test
public void hasFormat_OnAnyPageBefore() throws Exception {
    String filename = PATH + "format/format_multiple-formats-on-individual-pages.pdf";

    AssertThat.document(filename)
        .hasFormat(A4_LANDSCAPE, OnAnyPage.before(3))
        ;
}
```

```
@Test
public void hasFormat_OnAllPagesAfter() throws Exception {
    String filename = PATH + "format/format_multiple-formats-on-individual-pages.pdf";

    AssertThat.document(filename)
        .hasFormat(A5_PORTRAIT, OnEveryPage.after(2))
        ;
}
```

## 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:

```
// Simple tests:
.hasField(..)
.hasFields()
.hasNumberOfFields(..)
.hasSignedSignatureFields()
.hasUnsignedSignatureFields()

// Tests belonging to all fields:
.hasFields().allWithoutDuplicateNames()
.hasFields().allWithoutTextOverflow() ❶
.hasFields().matchingXML(..)
.hasFields().matchingXPath(..)

// Content of a field:
.hasField(..).containing(..)
.hasField(..).endingWith(..)
.hasField(..).matchingComplete(..)
.hasField(..).matchingRegex(..)
.hasField(..).notContaining(..)
.hasField(..).notMatchingRegex(..)
.hasField(..).startingWith(..)
.hasField(..).withAnyValue()

// Properties of a field:
.hasField(..).havingJavaScriptAction(..)
.hasField(..).whichHasMultipleLines()
.hasField(..).whichHasSingleLines()
.hasField(..).whichIsEditable()
.hasField(..).whichIsExportable()
.hasField(..).whichIsHidden()
.hasField(..).whichIsMultiSelectable()

... continued
```

```
... continuation:

.hasField(..).whichIsNotEditable()
.hasField(..).whichIsNotExportable()
.hasField(..).whichIsNotHidden()
.hasField(..).whichIsNotMultiSelectable()
.hasField(..).whichIsNotPrintable()
.hasField(..).whichIsReadOnly()
.hasField(..).whichIsRequired()
.hasField(..).whichIsVisible()
.hasField(..).whichIsPasswordProtected()
.hasField(..).whichIsPrintable()
.hasField(..).whichIsReadOnly()
.hasField(..).whichIsRequired()
.hasField(..).whichIsSigned()
.hasField(..).whichIsVisible()
.hasField(..).withHeight(..)
.hasField(..).withoutTextOverflow() ❷
.hasField(..).withType(..)
.hasField(..).withWidth(..)
```

- ❶❷ Dieser Test wird in Kapitel 3.12: „Formularfelder, Textüberlauf“ (S. 41) separat beschrieben.

## Existenz

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

```
@Test(expected=PDFUnitValidationException.class)
public void hasFields_NoFieldsAvailable() throws Exception {
    String filename = PATH + "acrofields/noAcrofieldDemo.pdf";

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

## Anzahl

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

```
@Test
public void hasNumberOfFields() throws Exception {
    String filename = PATH + "acrofields/simpleRegistrationForm.pdf";

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

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

```
@Test
public void hasNumberOfFields_NoFieldsAvailable() throws Exception {
    String filename = PATH + "acrofields/noAcrofieldDemo.pdf";
    int zeroExpected = 0;

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

## 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:

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

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

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

```
@Test
public void hasFields_AllWithoutDuplicateNames() throws Exception {
    String filename = PATH + "acrofields/javascriptForFields.pdf";

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

## Inhalte von Feldern

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

```

@Test
public void hasField_WithAnyValue() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";
    String fieldname = "Text 1";

    AssertThat.document(filename)
        .hasField(fieldname)
        .withAnyValue();
}

```

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

```

.containing(..)
.endsWith(..)
.matchingComplete(..)
.matchingRegex(..)
.notContaining(..)
.notMatchingRegex(..) // useful, because regular expressions are
                        // not designed to find Not-Matches
.startingWith(..)
.withAnyValue(..)     // The field must not be empty

```

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

```

@Test
public void hasField_MatchingComplete() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";
    String fieldname = "Text 1";
    String expectedValue = "Single Line Text";

    AssertThat.document(filename)
        .hasField(fieldname)
        .matchingComplete(expectedValue);
}

```

```

/**
 * This is a small test to protect fields against SQL-Injection.
 */
@Test
public void hasField_NotContaining_SQLComment() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";
    String fieldname = "Text 1";
    String sqlCommandSequence = "--";

    AssertThat.document(filename)
        .hasField(fieldname)
        .notContaining(sqlCommandSequence);
}

```

## 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 eine Testmethode für Typen:

```

import static com.pdfunit.Constants.*;
...
@Test
public void hasField_WithType_MultipleInvocation() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";

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

```

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

```
@Test
public void hasField_WithType_Signature() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";

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

Ausführliche Tests zu Signaturen und Zertifikaten werden in Kapitel 3.21: „Signaturen und Zertifikate“ (S. 58) beschrieben.

Die möglichen Feldtypen sind als Konstanten in `com.pdfunit.Constants` definiert. Die Namen der Konstanten entsprechen den gängigen, sichtbaren Elementen einer graphischen Anwendung. Innerhalb von PDF gibt es aber andere Typen. Weil 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"
```

## Größe von Feldern

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

```
@Test
public void hasField_WidthAndHeight() throws Exception {
    String filename = PATH + "acrofields/notExportableAcrofield.pdf";
    String fieldname = "Title of 'someField'";

    AssertThat.document(filename)
        .hasField(fieldname)
        .withWidth(159) // default is MILLIMETER
        .withHeight(11) // default is MILLIMETER
    ;
}
```

Beide Methoden können mit verschiedenen Maßeinheiten aufgerufen werden, die als Konstante definiert sind:

```
import static com.pdfunit.Constants.*;
...
@Test
public void hasField_Width() throws Exception {
    String filename = PATH + "acrofields/notExportableAcrofield.pdf";
    String fieldname = "Title of 'someField'";

    AssertThat.document(filename)
        .hasField(fieldname)
        .withWidth(450, POINTS)      ❶
        .withWidth(450, DPI72)      ❷
        .withWidth(159, MILLIMETER)
        .withWidth(15.9, CENTIMETER)
        .withWidth(6.26, INCH)
    ;
}
```

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. 41) 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:

```
@Test
public void hasField_Editable() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";
    String fieldnameEditable = "Combo Box 4";

    AssertThat.document(filename)
        .hasField(fieldnameEditable)
        .whichIsEditable()
    ;
}
```

Insgesamt stehen folgende Methoden zur Überprüfung von Feldeigenschaften zur Verfügung:

<code>.whichHasMultipleLines()</code> ,	<code>.whichHasSingleLine()</code>
<code>.whichIsEditable()</code> ,	<code>.whichIsNotEditable()</code>
<code>.whichIsExportable()</code> ,	<code>.whichIsNotExportable()</code>
<code>.whichIsHidden()</code> ,	<code>.whichIsNotHidden()</code>
<code>.whichIsHiddenButPrintable()</code> ,	❶ <code>.whichIsNotMultiSelectable()</code>
<code>.whichIsMultiSelectable()</code> ,	❷ <code>.whichIsNotPrintable()</code>
<code>.whichIsPasswordProtected()</code> ,	<code>.whichIsNotReadOnly()</code>
<code>.whichIsPrintable()</code> ,	<code>.whichIsNotRequired()</code>
<code>.whichIsReadOnly()</code> ,	❸ <code>.whichIsNotVisible()</code>
<code>.whichIsRequired()</code> ,	❹ <code>.whichIsVisibleButNotPrintable()</code>
<code>.whichIsSigned()</code> ,	
<code>.whichIsVisible()</code> ,	
<code>.whichIsVisibleButNotPrintable()</code>	

❶❷❸❹ Die inversen Methoden werden nicht angeboten, weil kein Sinn dazu erkennbar ist.

Bei den Vergleichen spielen Whitespaces keine Rolle:

```
@Test
public void hasMultiLineField_MultipleInvocation() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";
    String fieldnameMultipleLinesField = "Text multi";

    AssertThat.document(filename)
        .hasField(fieldnameMultipleLinesField)
        .matchingComplete("Multiple Line Support:\r\nFirst Line;\r\nSecond Line;")
    ;
}
```

## 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:

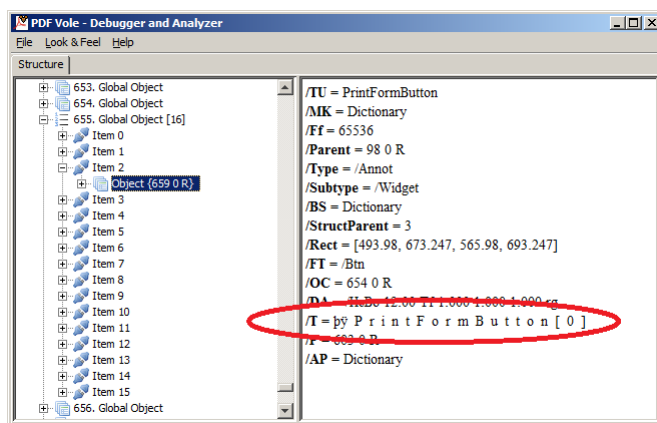
```
@Test
public void hasField_HavingJavaScriptAction_MultipleInvocation() throws Exception {
    String filename = PATH + "acrofields/javaScriptForFields.pdf";

    AssertThat.document(filename)
        .hasField("ageField").havingJavaScriptAction("Validate")
        .hasField("nameField").havingJavaScriptAction("Keystroke")
        .hasField("commentField").havingJavaScriptAction("Keystroke")
    ;
}
```

Testmethoden für die Validierung des eigentlichen JavaScript-Codes sind in Kapitel 3.13: „JavaScript“ (S. 43) 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.
 */
@Test
public void hasField_nameContainingUnicode_UTF16() throws Exception {
    String filename = PATH + "unicode/unicode_inFieldnames.pdf";
    String fieldname =
        //      F      o      r      m      R
        "\u00fe\u00ff\u0000\u0046\u0000\u006f\u0000\u0072\u0000\u006d\u0000\u0052" +
        //      o      o      t      [      0      ]
        "\u0000\u006f\u0000\u006f\u0000\u0074\u0000\u005b\u0000\u0030\u0000\u005d" +
        //
        "\u002e" +
        //      P      a      g      e      l
        "\u00fe\u00ff\u0000\u0050\u0000\u0061\u0000\u0067\u0000\u0065\u0000\u0031" +
        //      [      0      ]
        "\u0000\u005b\u0000\u0030\u0000\u005d" +
        //
        "\u002e" +
        //      P      r      i      n      t
        "\u00fe\u00ff\u0000\u0050\u0000\u0072\u0000\u0069\u0000\u006e\u0000\u0074" +
        //      F      o      r      m      B      u
        "\u0000\u0046\u0000\u006f\u0000\u0072\u0000\u006d\u0000\u0042\u0000\u0075" +
        //      t      t      o      n      [      0
        "\u0000\u0074\u0000\u0074\u0000\u006f\u0000\u006e\u0000\u005b\u0000\u0030" +
        //      ]
        "\u0000\u005d";

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

```

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

## Feldinformationen gegen XML prüfen

Das Kapitel 9.4: „Feldeigenschaften nach XML extrahieren“ (S. 117) beschreibt, wie mit dem Extraktionsprogramm `ExtractFieldsInfo` Informationen über alle Formularfelder eines PDF-Dokuments extrahiert werden können. Die dabei entstehende XML-Datei kann in Tests als Vergleichsinstanz benutzt werden:

```
@Test
public void hasField_MatchingXML() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";
    String xmlFilename = PATH + "acrofields/plugin-pdf_form_maker.xml";
    File fieldInfosAsXML = new File(xmlFilename);

    AssertThat.document(filename)
        .hasFields().matchingXML(fieldInfosAsXML) ❶
    ;
}
```

❶ Der Dateiname kann auch als `java.lang.String` übergeben werden.

## 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:

```
@Test
public void hasField_MatchingXPath_NumberOfTextFields() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";
    String xpath = "count(//field[./@type='text']) = 43"; // 43 fields inside PDF

    AssertThat.document(filename)
        .hasFields().matchingXPath(xpath)
    ;
}
```

Aufrufe können wie gewohnt verkettet werden:

```
@Test
public void hasField_MatchingXPath_MultipleInvocation() throws Exception {
    String filename = PATH + "acrofields/plugin-pdf_form_maker.pdf";

    String xpathNumberOfTextfields = "count(//field[./@type='text']) = 43";
    String xpathNumberOfButtonFields = "count(//field[./@type='button']) = 54";
    String xpathNumberOfChoiceFields = "count(//field[./@type='choice']) = 5";
    String xpathNumberOfSignaturFields = "count(//field[./@type='signatur']) = 0";

    AssertThat.document(filename)
        .hasFields()
        .matchingXPath(xpathNumberOfTextfields) ❶
        .matchingXPath(xpathNumberOfButtonFields)
        .matchingXPath(xpathNumberOfChoiceFields)
        .matchingXPath(xpathNumberOfSignaturFields)
    ;

    // The same test in a different style:
    AssertThat.document(filename)
        .hasFields().matchingXPath(xpathNumberOfTextfields) ❷
        .hasFields().matchingXPath(xpathNumberOfButtonFields)
        .hasFields().matchingXPath(xpathNumberOfChoiceFields)
        .hasFields().matchingXPath(xpathNumberOfSignaturFields)
    ;
}
```

❶❷ Je nach Geschmack können zwei verschiedene Schreibweisen verwendet werden.

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



```
@Test
public void hasField_MatchingXPath_HavingUnsignedSignatureFields_1() throws Exception {
    String filename = PATH + "acrofields/certificateform.pdf";

    AssertThat.document(filename)
        .hasUnsignedSignatureFields()
    ;
}

@Test
public void hasField_MatchingXPath_HavingUnsignedSignatureFields_2() throws Exception {
    String filename = PATH + "acrofields/certificateform.pdf";
    String xpath = "count(//field[./@type='sig'][./@isSigned='false']) > 0";

    AssertThat.document(filename)
        .hasFields().matchingXPath(xpath)
    ;
}
```

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 Testfunktion:

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

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

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

[illegible]

Und so funktioniert der Test auf das letzte Feld:

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

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

Wenn die Exception nicht abgefangen würde, 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:

```
@Test
public void hasFields_AllWithoutTextOverflow() throws Exception {
    String filename = PATH + "acrofields/fieldsWithAttributes.pdf";

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

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

## Technische Randbedingung

Der Inhalt eines Feldes wird aus technischen Gründen nicht von der Testmethode `.hasText( . )` 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.
 */
@Test
public void hasTextOnFirstPage_DocumentWithFields() throws Exception {
    String filename = PATH + "acrofields/fieldSizeAndText.pdf";
    String fieldContent = "middle";
    int upperLeftX = 0; // in points
    int upperLeftY = 0;
    int width = 595;
    int height = 842;

    ClippingArea acrofieldClippingArea =
        new ClippingArea(upperLeftX, upperLeftY, width, height, POINTS);

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE, acrofieldClippingArea)
        .containing(fieldContent)
        ;
}
```

## 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-Unitest-Werkzeug wie beispielsweise „Google JS Test“, aber besser wenig getestet, als überhaupt nicht.

### Existenz von JavaScript

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

```
@Test
public void hasJavaScript() throws Exception {
    String filename = PATH + "javascript/javascriptClock.pdf";

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

### Vergleich gegen eine Vorlage

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

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

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

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

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

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

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

## Teilstrings vergleichen

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

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

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

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

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

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

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

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 Testmethoden rund um das Thema Layer an:

```
// Simple methods:
.hasNumberOfLayers(..) // 'Layer' and ...
.hasNumberOfOCGs(..)  // ...'OCG' are always used the same way
.hasLayer()
.hasOCG()
.hasOCGs()
.hasLayers()

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

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

Die Testmethode `endsWith()` wird nicht angeboten, weil doppelte Layernamen intern mit einem Suffix erweitert werden und das Namensende somit nicht vorhersehbar ist.

Eine Testmethode `matchingRegex()` wird ebenfalls nicht angeboten. Sie wird nicht benötigt werden, weil Layernamen üblicherweise kurz und bekannt sind.

## Anzahl

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

```
@Test
public void hasNumberOfOCGs() throws Exception {
    String filename = PATH + "layer/hang-man-game.pdf";

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

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

## Layernamen

Der nächste Test zielt auf die Namen der Layer:

```
@Test
public void hasLayer_WithName_MatchingComplete() throws Exception {
    String filename = PATH + "layer/simpleLayerDemo.pdf";

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

Für den Namen gibt es folgende Vergleichsmethoden:

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

Die Methoden `endsWith(..)` und `matchingRegex(..)` werden aus den am Anfang des Kapitels erläuterten Gründen nicht angeboten.

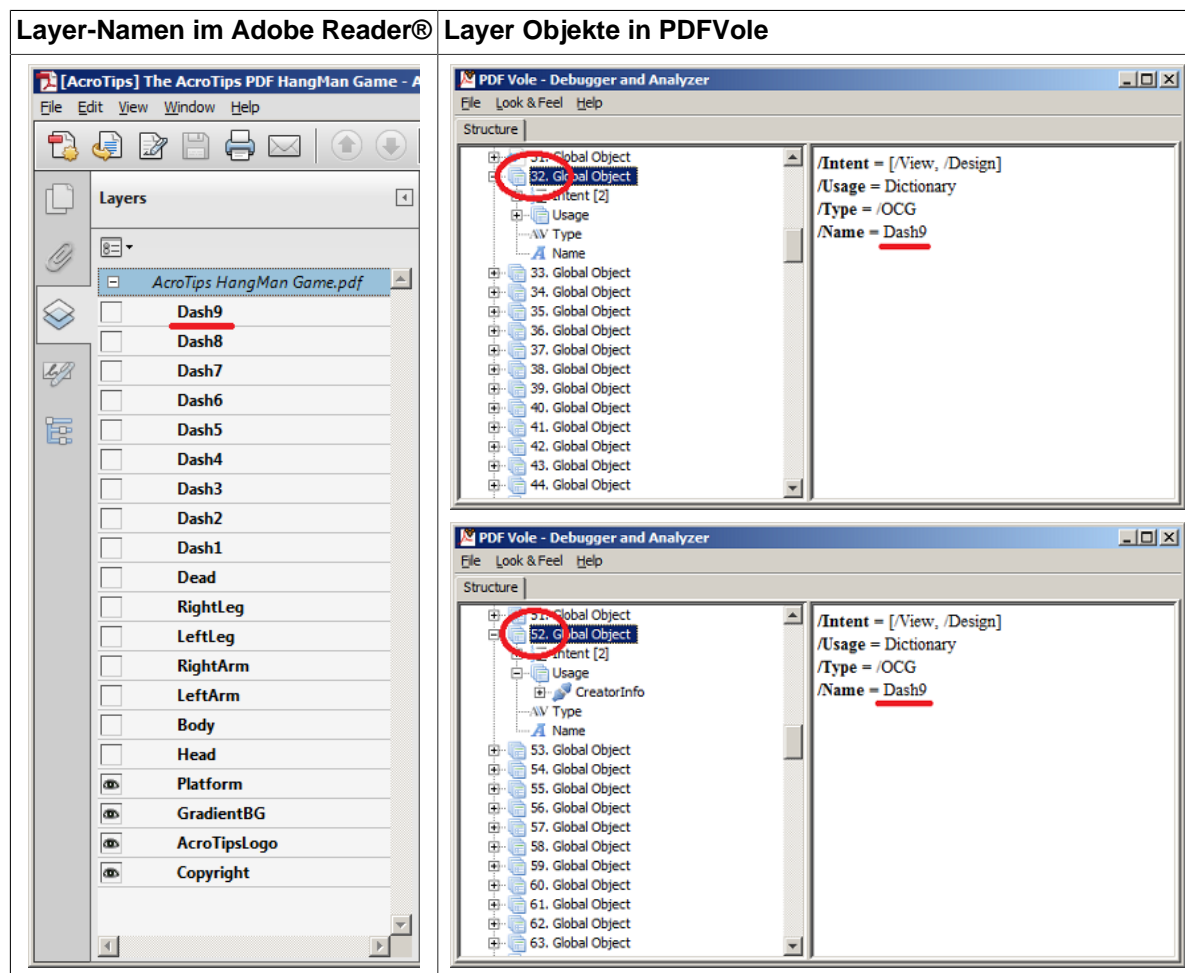
Alle Vergleiche werden case-insensitiv ausgeführt. Leerzeichen bleiben erhalten:

```
@Test
public void hasLayer_WithName_CaseInsensitive() throws Exception {
    String filename = PATH + "layer/simpleLayerDemo.pdf";

    AssertThat.document(filename)
        .hasLayer().withName().matchingComplete("parent layer")
        .hasLayer().withName().matchingComplete("Parent Layer")
    ;
}
```

## 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:



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

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

```
@Test
public void hasLayers_AllWithoutDuplicateNames() throws Exception {
    String filename = PATH + "layer/simpleLayerDemo.pdf";

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

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. 120) 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. 91) beschreibt diesen Vergleich.

Es gibt folgende Testmethoden:

```
// Principles of comparing rendered pages with images:
// Compare the given image with each page of the page array.
// Starting position is 0/0:
.asRenderedPage(...).isEqualToImage(...)

// Compare the given image with each page of the page array,
// Starting position is the given position:
// see 3.16: „Layout - gerenderte Seitenausschnitte“ (S. 48)
.asRenderedPage(...).isEqualToImage(upperLeftX, upperLeftY, unit, ...)

// Compare images from an image array with pages from a page array
// by their corresponding index. Starting position is 0/0:
.asRenderedPage(...).isEqualToImages(...)

// Compare images from an image array with pages from a page array
// by the corresponding index. Starting position is the given position:
// see 3.16: „Layout - gerenderte Seitenausschnitte“ (S. 48)
.asRenderedPage(...).isEqualToImages(upperLeftX, upperLeftY, unit, ...)
```

Für den Vergleich können beliebige Seiten ausgewählt werden. Das Kapitel 13.3: „Seitenauswahl“ (S. 150) 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. 48) geht darauf näher ein.

### Beispiel - Ausgewählte Seiten als Bild vergleichen

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

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

    String page1Rendered = PATH + "master/documentUnderTest_Page1.png";
    String page3Rendered = PATH + "master/documentUnderTest_Page3.png";
    String page4Rendered = PATH + "master/documentUnderTest_Page4.png";
    PagesToUse ON_SELECTED_PAGES = PagesToUse.getPages(1, 3, 4);

    AssertThat.document(filename)
        .asRenderedPage(ON_SELECTED_PAGES)
        .isEqualToImages(page1Rendered, page3Rendered, page4Rendered)
    ;
}
```

## Bildformate der Vergleichsbilder

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

Die Parametertypen `File` und `String` werden von PDFUnit intern in `BufferedImage` überführt:

```
// Possible image file formats
... isEqualToImage(String imageFileNames)
... isEqualToImage(File imageFiles)
... isEqualToImage(java.awt.image.BufferedImage images)

... isEqualToImages(String... imageFileNames)
... isEqualToImages(File... imageFiles)
... isEqualToImages(java.awt.image.BufferedImage... images)
```

## 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.

```
// Principles of testing a section of a rendered page:

// Compare one image with each preselected page:
AssertThat.document(..)
    .asRenderedPage(..)
    .isEqualToImage(upperLeftX, upperLeftY, unit, image)

// Compare the images of the image array with the pages of the page array
// by their corresponding index. Both arrays must have the same size:
AssertThat.document(..)
    .asRenderedPage(..)
    .isEqualToImages(upperLeftX, upperLeftY, unit, image[])
```

### Beispiel - Linker Rand auf jeder Seite

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

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

    String fullImage2cmWidthFromLeft = PATH + "master/marginFullHeight2cmWidth.png";
    int upperLeftX = 0;
    int upperLeftY = 0;

    AssertThat.document(filename)
        .asRenderedPage(ON_EVERY_PAGE)
        .isEqualToImage(upperLeftX, upperLeftY, DPI72, fullImage2cmWidthFromLeft)
        ;
}
```

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:

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

    int upperLeftX = 135;
    int upperLeftY = 35;
    PagesToUse ON_SELECTED_PAGES = PagesToUse.getPages(1, 2);

    AssertThat.document(filename)
        .asRenderedPage(ON_SELECTED_PAGES)
        .isEqualToImage(upperLeftX, upperLeftY, MILLIMETER, logo)
    ;
}
```

- ❶ Die x/y-Koordinaten der linken oberen Ecke des Seitenausschnitts setzen
- ❷ Auswahl der Seiten, siehe Kapitel 13.3: „Seitenauswahl“ (S. 150)
- ❸ Vergleichsmethode aufrufen

## Mehrfache Vergleiche

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

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

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

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

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

    AssertThat.document(filename)
        .asRenderedPage(ON_SELECTED_PAGES)
        .isEqualToImage(ulX_Image1, ulY_Image1, DPI72, fullImage2cmWidthFromLeft)
        .isEqualToImage(ulX_Image2, ulY_Image2, DPI72, subImagePage3And4)
    ;
}
```

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:

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

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

// Tests for all bookmarks, see the plural:
.hasBookmarks().matchingXPath(expression)
.hasBookmarks().matchingXML(xmlFile)
```

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 Testmethoden:

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

## Sprungziele, Named Destinations

Namen von Sprungzielen können einfach getestet werden:

```
@Test
public void hasNamedDestination_WithName() throws Exception {
    String filename = PATH + "namedDestination/manyNamedDestinations.pdf";

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

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

```
@Test
public void hasNamedDestination_ContainingBlanks() throws Exception {
    String filename = PATH + "namedDestination/problem_convert-bookmarks-to-pdf.pdf";

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

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

## Existenz von Lesezeichen (Bookmarks)

Am einfachsten kann die Existenz von Lesezeichen überprüft werden:

```
@Test
public void hasBookmarks() throws Exception {
    String filename = PATH + "bookmarks/manyBookmarks.pdf";

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

## Anzahl

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

```
@Test
public void hasNumberOfBookmarks() throws Exception {
    String filename = PATH + "bookmarks/manyBookmarks.pdf";

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

## Text eines Lesezeichens (Label)

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

```
@Test
public void hasBookmark_WithLabel() throws Exception {
    String filename = PATH + "bookmarks/diverseContentOnMultiplePages.pdf";

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

## Sprungziele von Lesezeichen

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

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

```
@Test
public void hasBookmark_WithLabelLinkingToPage() throws Exception {
    String filename = PATH + "bookmarks/diverseContentOnMultiplePages.pdf";

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

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

```
@Test
public void hasBookmark_WithLinkToPage() throws Exception {
    String filename = PATH + "bookmarks/diverseContentOnMultiplePages.pdf";

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

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

```
@Test
public void hasBookmark_WithLinkToName() throws Exception {
    String filename = PATH + "bookmarks/twoBookmarkToSameDestination.pdf";

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

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

```
@Test
public void hasBookmark_WithLinkToURI() throws Exception {
    String filename = PATH + "bookmarks/bookmarkWithURLAction.pdf";

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

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.
 */
@Test
public void hasBookmark_WithoutDeadLink() throws Exception {
    String filename = PATH + "bookmarks/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
        .hasBookmark().withoutDeadLink()
    ;
}
```

PDFUnit greift während der Tests nicht auf Webseiten zu. Insofern ist ein „toter Link“ ein Lesezeichen, das auf keine Seite oder kein Sprungziel verweist. Es wird nicht geprüft, ob eine Webseite tatsächlich existiert.

## Lesezeichen mit XML/XPath testen

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

Die Lesezeichen eines Test-PDF-Dokumentes können vollständig mit der exportierten XML-Datei verglichen werden:

```
@Test
public void hasBookmarks_MatchingXML() throws Exception {
    String filenamePDF = PATH + "bookmarks/bookmarksWithPdfOutline.pdf";
    String filenameXML = PATH + "bookmarks/bookmarksWithPdfOutline.xml";

    AssertThat.document(filenamePDF)
        .hasBookmarks()
        .matchingXML(filenameXML) ❶
    ;
}
```

- 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:

```
@Test
public void hasBookmarks_MatchingXPath_MultipleInvocation() throws Exception {
    String filename = PATH + "bookmarks/bookmarksWithPdfOutline.pdf";

    String xpathNumberOfBookmarks = "count(//Title) = 5";
    String xpathBookmarkHierarchy2 = "count(//Title[count(ancestor::*) > 2]) = 0";
    XPathExpression exprNumberOfBookmarks = new XPathExpression(xpathNumberOfBookmarks);
    XPathExpression exprBookmarkHierarchy = new XPathExpression(xpathBookmarkHierarchy2);

    AssertThat.document(filename)
        .hasBookmarks()
        .matchingXPath(exprNumberOfBookmarks)
        .matchingXPath(exprBookmarkHierarchy)
    ;
}
```

## 3.18. Passwort

### Überblick

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

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

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

- ❶ Wenn das Dokument unverschlüsselt ist, darf nur der 1. Parameter benutzt werden.
- ❷ Wird der 2. Parameter benutzt, gilt das PDF-Dokument als verschlüsselt.

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

### Tests auf das Password selber

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

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

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

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

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

- ❶❸ Öffnen der Datei mit einem Passwort
- ❷❹ Überprüfen des anderen Passwortes

Passwörter sollten im Source-Code lediglich für Testdokumente hart kodiert werden, wobei auch diese Aussage aus Sicherheitsgründen bedenklich ist. „Hart kodiert“ bedeutet aber auch, dass das Passwort nie wechselt.

### Test auf Verschlüsselungslänge

Mit welcher Verschlüsselungslänge wurde verschlüsselt:

```
@Test
public void hasEncryptionLength() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages_encrypted.pdf";
    String userPassword = "user-password";

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

## 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 Schriften gehört. Auch für die Archivierung von PDF-Dokumenten spielen Schriften eine besondere Rolle. PDFUnit bietet unterschiedliche Testmethoden für Schriften, um verschiedene Bedürfnisse abzudecken:

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

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

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

### Anzahl von Schriften

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

com.pdfunit.Constants.IDENTIFIEDBY_ALLPROPERTIES
com.pdfunit.Constants.IDENTIFIEDBY_BASENAME
com.pdfunit.Constants.IDENTIFIEDBY_BASENAME_ENCODING
com.pdfunit.Constants.IDENTIFIEDBY_BASENAME_ENCODING_ENCODINGDIFF
com.pdfunit.Constants.IDENTIFIEDBY_CONVERTIBLE2UNICODE
com.pdfunit.Constants.IDENTIFIEDBY_EMBEDDED
com.pdfunit.Constants.IDENTIFIEDBY_EMBEDDED_CONVERTIBLE2UNICODE
com.pdfunit.Constants.IDENTIFIEDBY_NAME
com.pdfunit.Constants.IDENTIFIEDBY_NAME_TYPE
com.pdfunit.Constants.IDENTIFIEDBY_TYPE
```

Die folgende Liste erläutert die Vergleichskriterien für Schriften:

Konstante	Beschreibung
ALLPROPERTIES	Alle Eigenschaften eines Fonts gelten als identifizierend. Von zwei verwendeten Schriften, die in allen Eigenschaften gleichwertig sind, wird nur eine gezählt.

Konstante	Beschreibung
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:

```
@Test
public void hasNumberOfFonts_Japanese() throws Exception {
    String filename = PATH + "fonts/fonts_11_japanese.pdf";

    AssertThat.document(filename)
        .hasNumberOfFonts(65, IDENTIFIEDBY_ALLPROPERTIES)
        .hasNumberOfFonts(9, IDENTIFIEDBY_BASENAME)
        .hasNumberOfFonts(16, IDENTIFIEDBY_BASENAME_ENCODING)
        .hasNumberOfFonts(16, IDENTIFIEDBY_BASENAME_ENCODING_ENCODINGDIFF)
        .hasNumberOfFonts(46, IDENTIFIEDBY_CONVERTIBLE2UNICODE)
        .hasNumberOfFonts(6, IDENTIFIEDBY_EMBEDDED)
        .hasNumberOfFonts(0, IDENTIFIEDBY_EMBEDDED_CONVERTIBLE2UNICODE)
        .hasNumberOfFonts(50, IDENTIFIEDBY_NAME)
        .hasNumberOfFonts(55, IDENTIFIEDBY_NAME_TYPE)
        .hasNumberOfFonts(3, IDENTIFIEDBY_TYPE)
    ;
}
```

## Schriftnamen

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

```
@Test
public void hasFont_WithNameContaining() throws Exception {
    String filename = PATH + "fonts/fonts_15_openoffice.pdf";

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

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

Die Testmethoden können verkettet werden:

```
@Test
public void hasFont_WithNameContaining_MultipleInvocation() throws Exception {
    String filename = PATH + "fonts/fonts_15_openoffice.pdf";

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

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

```
@Test
public void hasFont_WithNameNotContaining() throws Exception {
    String filename = PATH + "fonts/fonts_15_openoffice.pdf";
    String wrongFontnameIntended = "ComicSansMS";

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

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:

```
@Test
public void hasFonts_OfThisTypeOnly_TrueType() throws Exception {
    String filename = PATH + "fonts/fonts_15_openoffice.pdf";

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

Die prüfbaren Schrifttypen sind als Konstanten deklariert:

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

## 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 folgende Informationen:



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

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

```
@Test
public void hasFonts_MatchingXML() throws Exception {
    String filenamePDF = PATH + "fonts/fonts_52_itext.pdf";
    String filenameXML = PATH + "fonts/fonts_52_itext.xml";

    AssertThat.document(filenamePDF)
        .hasFonts().matchingXML(filenameXML)
    ;
}
```

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:

```
@Test
public void hasFonts_MatchingXPath_MultipleInvocation() throws Exception {
    String filename = PATH + "fonts/fonts_52_itext.pdf";
    String xpathArial = "count(//font[@baseFontName='ArialMT']) = 1";
    String xpathType1 = "count(//font[@type='Type1']) = 5";

    AssertThat.document(filename)
        .hasFonts()
        .matchingXPath(xpathArial)
        .matchingXPath(xpathType1)
    ;
}
```

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. 112).

## 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 Briefporto anfällt. PDFUnit bietet deshalb Testmethoden an, die sich auf die Anzahl von Seiten beziehen:

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

### Beispiele

Eine konkrete Seitenanzahl wird folgendermaßen überprüft:

```
@Test
public void hasNumberOfPages() throws Exception {
    String filename = PATH + "format/fopPoster_700x500mm.pdf";

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

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

```
@Test
public void hasLessPagesThan() throws Exception {
    String filename = PATH + "format/format_multiple-formats-on-individual-pages.pdf";
    int upperLimitExclusive = 6; // The document has 5 pages

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

```
@Test
public void hasMorePagesThan() throws Exception {
    String filename = PATH + "format/format_multiple-formats-on-individual-pages.pdf";
    int lowerLimitExclusive = 2; // The document has 5 pages

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

❶❷ Die Werte für Ober- und Untergrenze gelten exklusiv.

Die Methoden können verkettet werden:

```
@Test
public void hasNumberOfPages_InRange() throws Exception {
    String filename = PATH + "format/format_multiple-formats-on-individual-pages.pdf";
    // The current document has 5 pages
    int lowerLimit_2 = 2; // the limit is exclusive
    int upperLimit_8 = 8; // the limit is exclusive

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

Verzichten Sie nicht auf Tests mit Seitenzahlen weil Sie denken, sie seien **zu** einfach. Erfahrungsgemäß finden Sie im Umfeld eines einfachen Tests 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 Testmethoden für Signaturen und Zertifikate zur Verfügung:

```
// Simple methods for signatures:
.isSigned()
.hasNumberOfSignatures(..)
.hasSignature(..)

// Detailed tests for one signature:
.hasSignature(..).coveringWholeDocument()
.hasSignature(..).withNumberOfRevisions(..)
.hasSignature(..).withReason(..)
.hasSignature(..).withRevision(..)
.hasSignature(..).withCertificate()
.hasSignature(..).withSigningDate(..)
.hasSignature(..).withSigningName(..)

.hasSignature(..).withCertificate().validFor(..)
.hasSignature(..).withCertificate().validForCurrentDate()
.hasSignature(..).withCertificate().validFrom(..)
.hasSignature(..).withCertificate().validUntil(..)
.hasSignature(..).withCertificate().havingSubjectField(..).withValue(..)

// See the plural form:
.hasSignatures().matchingXML(..)
.hasSignatures().matchingXPath(..)
```

Bei den Testmethoden, 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. 81) beschrieben.

## Existenz

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

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

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

## 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:

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

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

```
@Test
public void hasSignature() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";

    AssertThat.document(filename)
        .hasSignature("Signature2")
    ;
}
```

## Gültigkeitsdatum

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

```
@Test
public void hasSignature_ValidForCurrentDate() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";

    AssertThat.document(filename)
        .hasSignature("Signature2")
        .withCertificate()
        .validForCurrentDate()
    ;
}
```

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

    AssertThat.document(filename)
        .hasSignature("Signature2")
        .withSigningDate(signingDate) ❶
    ;
}
```

❶ Der Vergleich findet auf der Basis von Jahr-Monat-Tag statt.

Für den Vergleich der Datumswerte kann auch eine Auflösung angegeben werden, DATE und DATETIME sind möglich. Informationen dazu liefert Kapitel 13.10: „Datumsauflösung“ (S. 160).

```
@Test
public void hasSignature_WithSigningDate_ResolutionDateTime() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";
    String dateValue = "2009-07-16T16:47:57+0200";
    String datePattern = "yyyy-MM-dd'T'HH:mm:ssZ";
    Calendar signingDate = DateHelper.getCalendar(dateValue, datePattern);

    AssertThat.document(filename)
        .hasSignature("Signature2")
        .withSigningDate(signingDate, AS_DATETIME)
    ;
}
```

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

```
@Test
public void hasSignature_FirstAndLastDate() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";
    Calendar lowerLimit = DateHelper.getCalendar("20060822", "yyyyMMdd");
    Calendar upperLimit = DateHelper.getCalendar("20090904", "yyyyMMdd");

    AssertThat.document(filename)
        .hasSignature("Signature2")
        .withCertificate()
        .validFrom(lowerLimit) // comparing year-month-day
        .validUntil(upperLimit) // comparing year-month-day
    ;
}
```

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

Die nächsten Beispiele zeigen, wie manche Informationen einer Signatur mit speziellen Methoden getestet werden können:

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

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

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

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

```
@Test
public void hasSignature_WithRevision() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";

    AssertThat.document(filename)
        .hasSignature("Signature2")
        .withRevision(1)
    ;
}
```

```
@Test
public void hasSignature_WithNumberOfRevisions() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";

    AssertThat.document(filename)
        .hasSignature("Signature2")
        .withNumberOfRevisions(1)
    ;
}
```

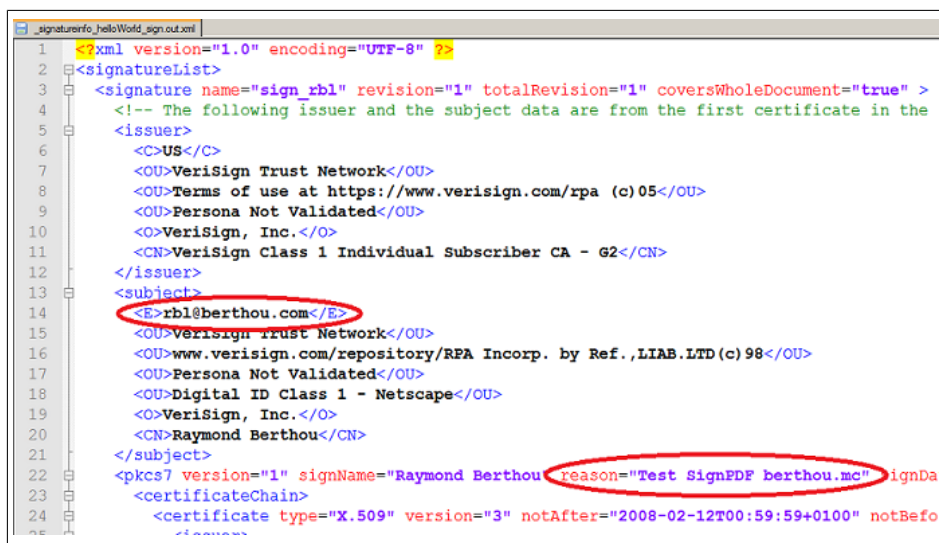
```
@Test
public void hasSignature_WithReason() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";

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

Der Rest der Signatur- und Zertifikatdaten 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):

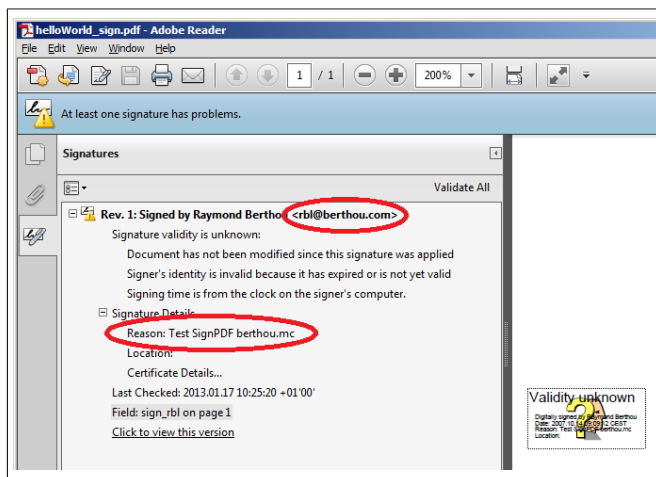


```

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 Incorpor. 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    <pks7 version="1" signName="Raymond Berthou" reason="Test SignPDF berthou.mc" signDa
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:

```

@Test
public void hasSignatures_MatchingXML() throws Exception {
    String filenamePDF = PATH + "signed/helloWorld_sign.pdf";
    String filenameXML = PATH + "signed/helloWorld_sign.xml";

    AssertThat.document(filenamePDF)
        .hasSignatures()
        .matchingXML(filenameXML)
    ;
}

```

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:

```

@Test
public void hasSignatures_MatchingXPath_OneOfManyOU() throws Exception {
    String filename = PATH + "signed/helloWorld_sign.pdf";
    String xpath = "//certificate[1]/subject[OU='Digital ID Class 1 - Netscape']";

    AssertThat.document(filename)
        .hasSignatures()
        .matchingXPath(xpath)
    ;
}

```

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

## Zusammenhängend testen

Mehrere Tests einer Signatur können verkettet werden:

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

    AssertThat.document(filename)
        .hasSignature("sign_rbl")
        .withSigningName("Raymond Berthou")
        .withCertificate()
        .havingSubjectField("O").withValue("VeriSign, Inc.")
    ;
}
```

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 Methoden stehen für Tests der Sprache zur Verfügung:

```
// Tests for PDF locale:

.hasLocale(..)
.hasNoLocale(..)
```

### Beispiele

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

```
@Test
public void hasLocale_CaseInsensitive() throws Exception {
    String filename = PATH + "language/_languageInfo/localeDemo_en-GB.pdf";

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

- ❶ PDF-typische Schreibweise
- ❷ Java-typische Schreibweise

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

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

```
@Test
public void hasLocale_LocaleInstance_GERMANY() throws Exception {
    String filename = PATH + "language/_languageInfo/localeDemo_de.pdf";

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

```
@Test
public void hasLocale_LocaleInstance_GERMAN() throws Exception {
    String filename = PATH + "language/_languageInfo/localeDemo_de.pdf";

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

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:

```
@Test
public void hasNoLocale() throws Exception {
    String filename = PATH + "language/_languageInfo/localeDemo_null.pdf";

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

## 3.23. Texte

### Überblick

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

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

// Comparing content:
.hasText(...).containing(...)
.hasText(...).containing(..., WhitespaceProcessing) ❶
.hasText(...).endingWith(...)
.hasText(...).matchingComplete(...)
.hasText(...).matchingComplete(..., WhitespaceProcessing) ❷
.hasText(...).matchingRegex(...)
.hasText(...).startingWith(...)

// Prove the absence of defined text:
.hasText(...).notContaining(...)
.hasText(...).notContaining(..., WhitespaceProcessing) ❸
.hasText(...).notEndingWith(...)
// notMatchingComplete(...) is intentionally not provided
.hasText(...).notMatchingRegex(...)
.hasText(...).notStartingWith(...)

// Check that a page is completely empty:
.isEmpty()
```

❶❷❸ Das Kapitel 13.5: „Behandlung von Whitespaces“ (S. 153) beschreibt die unterschiedlichen Möglichkeiten.

### Text auf bestimmten Seiten

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

```
@Test
public void hasText_OnFirstPage() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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



Ein Text auf der letzten Seite wird folgendermaßen überprüft:

```
@Test
public void hasText_OnLastPage() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

Auch Tests mit beliebigen individuellen Seiten sind möglich:

```
@Test
public void hasText_OnIndividualPages() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";
    PagesToUse ON_SELECTED_PAGES = PagesToUse.getPages(2, 3); ❶

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

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

Für die Selektion von Seiten stehen mehrere Konstanten zur Verfügung, u.a. `com.pdfunit.Constants.ON_EVEN_PAGES` und `com.pdfunit.Constants.ON_ODD_PAGES`. Das Kapitel 13.3: „Seitenauswahl“ (S. 150) beschreibt die Seitenauswahl ausführlich.

## Text auf allen Seiten

Es gibt drei Konstanten, um Text auf allen Seiten zu suchen, `ON_ANY_PAGE`, `ON_EACH_PAGE` und `ON_EVERY_PAGE`. Die letzten beiden sind funktional identisch.

```
@Test
public void hasText_OnEveryPage() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

```
@Test
public void hasText_OnAnyPage() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

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:

```
@Test // an exception is thrown
public void hasText_NotMatchingRegex() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
        .hasText(ON_ANY_PAGE)
        .notMatchingRegex("wrongValueIntended")
    ;
}
```

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 expected search string does not contain a line break.
 */
@Test
public void hasText_LineBreakInPDF() throws Exception {
    String filename = PATH + "digitalsignatures20121017.pdf";

    // The PDF document has a (visible) line break after the word "The".
    String text = "The technology was conceived";

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

```

/**
 * The expected search string intentionally contains other line breaks.
 */
@Test
public void hasText_LineBreakInExpectedString() throws Exception {
    String filename = PATH + "digitalsignatures20121017.pdf";

    // The PDF document has a (visible) line break after the word "The".
    String text = "The " +
                  "\n " +
                  "technology " +
                  "\n " +
                  "was " +
                  "\n " +
                  "conceived";

    AssertThat.document(filename)
                .hasText(ON_FIRST_PAGE)
                .containing(text)
    ;
}

```

## Text in Seitenausschnitten

Text kann aber nicht nur auf vollständigen Seite gesucht werden, sondern auch auf Seitenausschnitten. Das Kapitel 13.7: „Seitenausschnitt definieren“ (S. 157) beschreibt diesen Aspekt ausführlich.

## Seiten ohne Text

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

```

@Test
public void hasText_AnyPageEmpty() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

```

Wenn Sie explizit prüfen wollen, ob eine Seite oder ein Ausschnitt einer Seite leer ist, geht das mit der Methode `hasNoText()`:

```

@Test
public void hasNoTextInClippingArea() throws Exception {
    String filename = PATH + "emptyPages/pagesPartiallyEmpty.pdf";

    ClippingArea clippingArea = new ClippingArea(70, 80, 90, 60);
    AssertThat.document(filename)
                .hasNoText(ON_FIRST_PAGE, clippingArea)
    ;
}

```

## Mehrere Suchbegriffe gleichzeitig

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

```

@Test
public void hasText_Containing_MultipleTokens() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
                .hasText(ON_ODD_PAGES)
                .containing("on", "page", "odd page number") // multiple search tokens
    ;
}

```

```
@Test
public void hasText_NotContaining_MultipleTokens() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

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

## Verkettung von Textvergleichen

Textvergleiche können verkettet werden, sie beziehen sich dann auf die spezifizierten Seiten der vorhergehenden Methode `hasText(...)`:

```
@Test
public void hasText_MultipleInvocation() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

```
@Test
public void hasText_MultipleInvocation_2ndKind() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

Es können auch Textvergleiche verkettet werden, die sich auf unterschiedliche Seiten beziehen, wie das folgende Beispiel zeigt:

```
/**
 * 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.
 */
@Test
public void hasText_ComplexSearchOverDifferentPages() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
        .hasText(ON_EVERY_PAGE).startingWith("PDFUnit - Automated PDF Tests")
        .hasText(ON_EVEN_PAGES).containing("Content", "even pagenumber")
        .hasText(ON_ODD_PAGES).containing("odd pagenumber")
    ;
}
```

Eine solche Verkettung ist aber nicht sinnvoll, weil der Name der Testmethode 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:

```
@Test
public void hasText_OnAnyPageAfter() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
        .hasText(OnAnyPage.after(1))
        .containing("Content on")
    ;
}
```

Die Zählung der Seitenzahlen beginnt mit „1“.

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

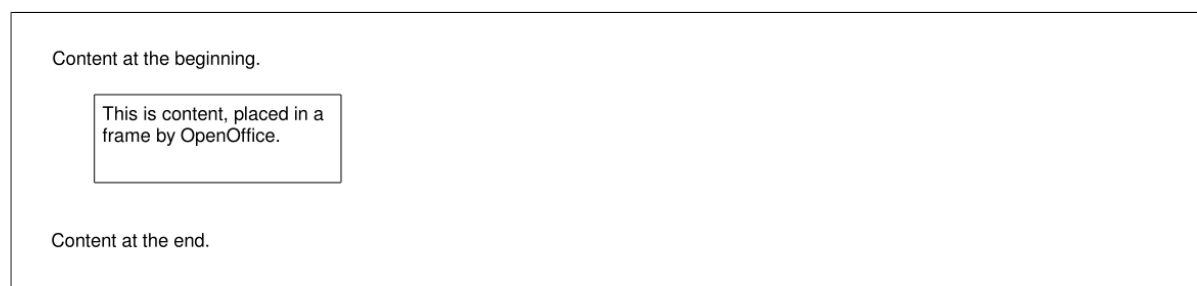
```
/**
 * Attention: The document has the search token on page 1.
 * And '1' is before '99'. So this test ends successfully.
 */
@Test
public void hasText_OnAnyPageBefore_WrongUpperLimit() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

## 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":



```
@Test
public void hasText_TextNotInVisibleOrder() throws Exception {
    String filename = PATH + "content/contentNotInVisibleOrder.pdf";

    String firstAndLastLine = "the beginning. This is content";

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .containing(firstAndLastLine)
    ;
}
```

## 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:

```
// Reducing area of analysis with a rectangle:  
.hasText(PageToUse, ClippingArea)
```

### Beispiel

Das folgende Beispiel zeigt die Definition und Benutzung eines Seitenausschnitts:

```
// Verifying text in a part of a PDF page  
@Test  
public void hasText_OnFirstPage_InClippingArea() throws Exception {  
    String filename = PATH + "content/documentForTextClipping.pdf";  
  
    int ulX    = 17.6; // upper left X  
    int ulY    = 45.8; // upper left Y  
    int width  = 60.0;  
    int height = 8.8;  
    ClippingArea inClippingArea = new ClippingArea(ulX, ulY, width, height); ❶  
  
    AssertThat.document(filename)  
        .hasText(ON_FIRST_PAGE, inClippingArea) ❷  
        .startingWith("Content")  
        .containing("on first")  
        .endingWith("page.")  
    ;  
}
```

- ❶ Hier wird der Seitenausschnitt definiert. Genaue Informationen dazu liefert das Kapitel 13.7: „Seitenausschnitt definieren“ (S. 157). Die Möglichkeiten, Maß-Einheiten wie `MILLIMETER` für die Definition zu benutzen, beschreibt Kapitel 13.8: „Maßeinheiten - Points, Millimeter, ...“ (S. 158).
- ❷ In dieser Code-Zeile wird der Ausschnitt als Parameter übergeben. Alle nachfolgenden Vergleich beschränken sich auf diesen Ausschnitt.

Für Vergleiche von Text in Seitenausschnitten stehen alle Vergleichsmethoden zur Verfügung, die auch für ganze PDF-Seiten zur Verfügung stehen. Sie sind in Abschnitt 13.4: „Textvergleich“ (S. 152) 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 Funktionen zur Verfügung, wie auch für „normalen“ Text.

### Beispiel

Das Beispieldokument enthält die zwei senkrechten Texte:

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

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

```
// Comparing upright text in a part of a PDF page
@Test
public void hasText_RotatedText_InClippingArea() throws Exception {
    String filename = PATH + "writeDirection/verticalText.pdf";
    String textBottomToTop = "Text from bottom to top.";
    String textTopToBottom = "Text from top to bottom.";

    ClippingArea IN_AREA = new ClippingArea(110, 130, 130, 300, FormatUnit.POINTS);

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE, IN_AREA)
        .containing(textBottomToTop)
        .containing(textTopToBottom)
    ;
}
```

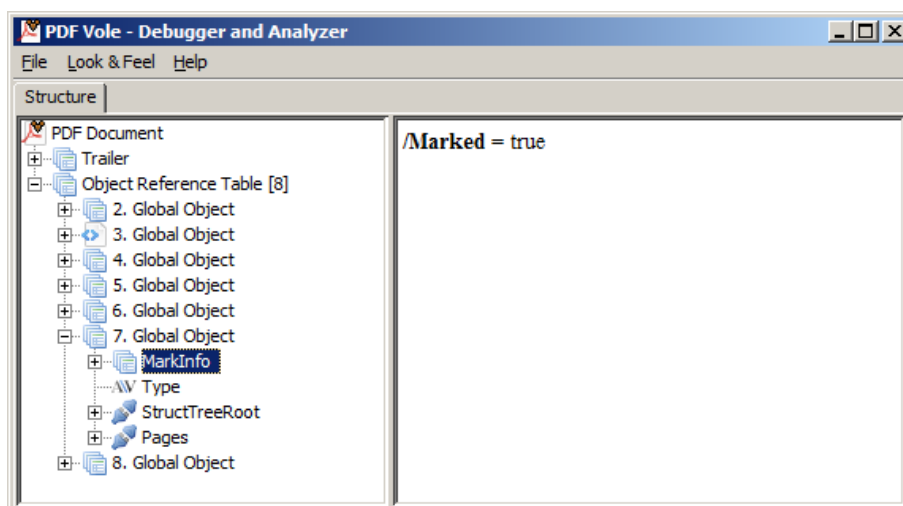
Beachten Sie: Es handelt sich bei diesem kopfwärts gestellten Text immer noch um Text mit der Laufrichtung LTR (left-to-right). Texte mit der Laufrichtung RTL (right-to-left) werden von PDFUnit in zukünftigen Releases unterstützt.

## 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](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 Methoden stehen zur Verfügung:

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

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

## Beispiele

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

```
@Test
public void isTagged() throws Exception {
    String filename = PATH + "tagged/itext-created_tagged.pdf";

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

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

```
@Test
public void isTagged_WithKey() throws Exception {
    String filename = PATH + "tagged/xdp_2.0.pdf";
    String tagName = "LetterspaceFlags";

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

Als Letztes können Werte bestimmter Tags verifiziert werden:

```
@Test
public void isTagged_WithKeyAnValue_MultipleInvocation() throws Exception {
    String filename = PATH + "tagged/xdp_2.0.pdf";

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

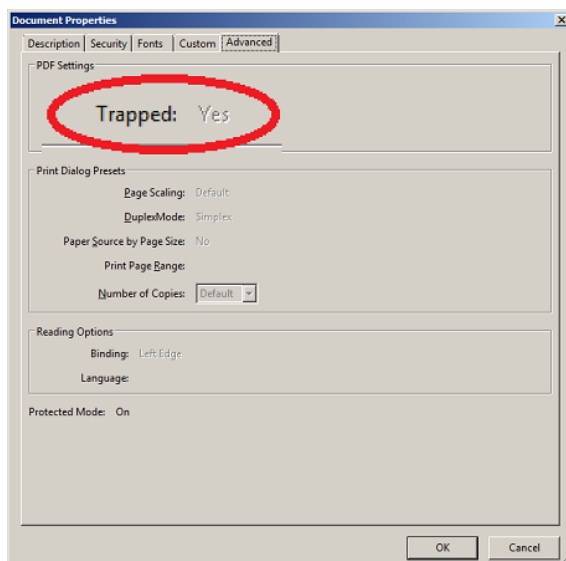
## 3.27. Trapping-Info

### 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:

```
// Testing trapping info:
.hasTrappingInfo(..)
```

## Beispiele

```
@Test
public void hasTrappingInfo_Yes() throws Exception {
    String filename = PATH + "trapping/trapping-info_yes.pdf";

    AssertThat.document(filename)
        .hasTrappingInfo(TRAPPING_YES)
    ;
}
```

```
@Test
public void hasTrappingInfo_No() throws Exception {
    String filename = PATH + "trapping/trapping-info_no.pdf";

    AssertThat.document(filename)
        .hasTrappingInfo(TRAPPING_NO)
    ;
}
```

```
@Test
public void hasTrappingInfo_Unknown() throws Exception {
    String filename = PATH + "trapping/trapping-info_unknown.pdf";

    AssertThat.document(filename)
        .hasTrappingInfo(TRAPPING_UNKNOWN)
    ;
}
```

Der Funktionsparameter ist typisiert. Die Werte sind als Konstanten verfügbar:

```
// Constants for trapping

com.pdfunit.Constants.YES
com.pdfunit.Constants.NO
com.pdfunit.Constants.UNKNOWN
```

## 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:

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

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

### Eine bestimmte Version

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

```
// Constants for PDF versions:

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

Ein Beispiel für den Test auf Version „1.4“:

```
@Test
public void hasVersion_v14() throws Exception {
    String filename = PATH + "version/pdf-version-1.4.pdf";

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

### Versionsbereiche

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

```
@Test
public void hasVersion_GreaterThanLessThan() throws Exception {
    String filename = PATH + "version/pdf-version-1.4.pdf";

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

❶❷ Die Ober- und Untergrenzen gelten exklusiv.

Auch zukünftige PDF-Versionen können getestet werden:

```
@Test
public void hasVersion_LessThanFutureVersion() throws Exception {
    String filename = PATH + "version/pdf-version-1.6.pdf";
    PDFVersion futureVersion = PDFVersion.withName("2.0");

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

## 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:

```
// Methods around XFA data:
.hasNoXFADData()
.hasXFADData()
.hasXFADData().matchingXML(..)
.hasXFADData().matchingXPath(..)
.hasXFADData().withNode(..)
```

### Existenz und Abwesenheit von XFA

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

```
@Test
public void hasXFADData() throws Exception {
    String filename = PATH + "xfa/xfa-movie.pdf";

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

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

```
@Test
public void hasNoXFADData() throws Exception {
    String filename = PATH + "xfa/no-xfa.pdf";

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

### 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:

```
@Test
public void hasXFADData_MatchingXML() throws Exception {
    String filename = PATH + "xfa/xfa-movie.pdf";
    String xmlFilename = PATH + "xfa/xfa-movie.xml";
    File xmlFile = new File(xmlFilename);

    AssertThat.document(filename)
        .hasXFADData()
        .matchingXML(xmlFile) ❶
        ;
}
```

❶ Whitespaces werden beim Vergleich der XFA-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-Tags 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:xmpk="Adobe XMP Core 4.2.1-c041 52.337767, 2008/04/13-15:41:00"
  >
    <config xmlns="http://www.xfa.org/schema/xci/2.6/">
      ...
      <log xmlns="http://www.xfa.org/schema/xci/2.6/">
        <to>memory</to>
        <mode>overwrite</mode>
      </log>
    </config>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      ...
      <rdf:Description xmlns:xmp="http://ns.adobe.com/xap/1.0/" >
        <xmp:MetadataDate>2009-12-03T17:50:52Z</xmp:MetadataDate>
      </rdf:Description>
    </rdf:RDF>
  </x:xmpmeta>
  ...
</xdp:xdp>
```

Um auf einen bestimmten Knoten zu testen, muss eine Instanz von `com.pdfunit.XMLNode` mit dem XPath-Ausdruck für den Knoten und dem Erwartungswert erzeugt werden:

```
@Test
public void hasXFADData_WithNode() throws Exception {
    String filename = PATH + "xfa/xfa-enabled.pdf";
    XMLNode xmpNode = new XMLNode("xmp:MetadataDate", "2009-12-03T17:50:52Z"); ❶

    AssertThat.document(filename)
        .hasXFADData()
        .withNode(xmpNode)
    ;
}
```

- ❶ PDFUnit analysiert die XFA-Daten des aktuellen PDF-Dokumentes und ermittelt die Namensräume selbständig, lediglich der Default-Namespace 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:

```
@Test
public void hasXFADData_WithNode_NamespaceDD() throws Exception {
    String filename = PATH + "xfa/xfa-enabled.pdf";
    XMLNode ddNode = new XMLNode("dd:dataDescription/@dd:name", "movie");

    AssertThat.document(filename)
        .hasXFADData()
        .withNode(ddNode)
    ;
}
```

## XPath-basierte XFA-Tests

XPath kann mehr, als nur einzelne Knoten zu benennen. Um das große Potential zu nutzen, wird eine separate Methode angeboten, die alle Möglichkeiten von XPath unterstützt.

Die nächsten beiden Beispiele zeigen, was machbar ist:

```
@Test
public void hasXFADData_MatchingXPath_FunctionStartsWith() throws Exception {
    String filename = PATH + "xfa/xfa-enabled.pdf";
    String xpathString = "starts-with(//dd:dataDescription/@dd:name, 'mov')";
    XPathExpression expressionWithFunction = new XPathExpression(xpathString);

    AssertThat.document(filename)
        .hasXFADData()
        .matchingXPath(expressionWithFunction)
        ;
}
```

```
@Test
public void hasXFADData_MatchingXPath_FunctionCount_MultipleInvocation() throws Exception {
    String filename = PATH + "xfa/xfa-movie.pdf";

    String xpathProducer = "//pdf:Producer[. = 'Adobe LiveCycle Designer ES 8.2']";
    String xpathPI = "count(//processing-instruction()) = 30";

    XPathExpression exprPI = new XPathExpression(xpathPI);
    XPathExpression exprProducer = new XPathExpression(xpathProducer);

    AssertThat.document(filename)
        .hasXFADData()
        .matchingXPath(exprProducer)
        .matchingXPath(exprPI)
        ;

    // The same test in a different style:
    AssertThat.document(filename)
        .hasXFADData().matchingXPath(exprProducer)
        .hasXFADData().matchingXPath(exprPI)
        ;
}
```

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

Das Kapitel 13.13: „JAXP-Konfiguration“ (S. 161) erläutert am Beispiel von Xerces, wie ein beliebiger XML-Parser genutzt werden kann.

## 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:

```
@Test
public void hasXFADData_WithDefaultNamespace_XPathExpression() throws Exception {
    String filename = PATH + "xfa/xfa-movie.pdf";

    String namespaceURI = "http://www.xfa.org/schema/xfa-template/2.6/";
    String xpathSubform = "count(//default:subform[@name = 'movie']/default:field) = 5";

    DefaultNamespace defaultNS = new DefaultNamespace(namespaceURI);
    XPathExpression exprSubform = new XPathExpression(xpathSubform, defaultNS);

    AssertThat.document(filename)
        .hasXFADData()
        .matchingXPath(exprSubform)
        ;
}
```

Aus dem gleichen Grund muss der Default-Namensraum auch bei der Verwendung der Klasse XMLNode mitgegeben werden:

```
/**
 * The default namespace has to be declared,
 * but any alias can be used for it.
 */
@Test
public void hasXFADData_WithDefaultNamespace_XMLNode() throws Exception {
    String filename = PATH + "xfa/xfa-enabled.pdf";

    String namespaceXCI = "http://www.xfa.org/schema/xci/2.6/";
    DefaultNamespace defaultNS = new DefaultNamespace(namespaceXCI);
    XMLNode aliasFoo = new XMLNode("foo:log/foo:to", "memory", defaultNS);

    AssertThat.document(filename)
        .hasXFADData()
        .withNode(aliasFoo)
        ;
}
```

## 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:

```
// Methods to test XMP data:

.hasNoXMPData()
.hasXMPData()
.hasXMPData().matchingXML(..)
.hasXMPData().matchingXPath(..)
.hasXMPData().withNode(..)
```

### Existenz und Abwesenheit von XMP

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

```
@Test
public void hasXMPData() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";

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

```
@Test
public void hasNoXMPData() throws Exception {
    String filename = PATH + "xmp/bookmarkWithURLAction_noXMP.pdf";

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

### 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:

```
@Test
public void hasXMPData_MatchingXML() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";
    String xmlFilename = PATH + "xmp/metadata-added.xml";

    AssertThat.document(filename)
        .hasXMPData()
        .matchingXML(xmlFilename) ❶
    ;
}
```

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>
```

Nachfolgend wird die Existenz zweier XML-Knoten geprüft. Für den Test auf einen Knoten muss eine Instanz von `com.pdfunit.XMLNode` erzeugt werden:

```
@Test
public void hasXMPData_WithNode_ValidateExistence() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";
    XMLNode nodeCreateDate = new XMLNode("xmp:CreateDate");
    XMLNode nodeModifyDate = new XMLNode("xmp:ModifyDate");

    AssertThat.document(filename)
        .hasXMPData()
        .withNode(nodeCreateDate)
        .withNode(nodeModifyDate)
    ;
}
```

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

```
@Test
public void hasXMPData_WithNodeAndValue() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";
    XMLNode nodeCreateDate = new XMLNode("xmp:CreateDate", "2011-02-08T15:04:19+01:00");
    XMLNode nodeModifyDate = new XMLNode("xmp:ModifyDate", "2011-02-08T15:04:19+01:00");

    AssertThat.document(filename)
        .hasXMPData()
        .withNode(nodeCreateDate)
        .withNode(nodeModifyDate)
    ;
}
```

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 der Testmethode `matchingXPath(..)` kann das ganze Potential von XPath genutzt werden:

```

@Test
public void hasXMPData_MatchingXPath() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";
    String xpathString = "//xmp:CreateDate[node() = '2011-02-08T15:04:19+01:00']";
    XPathExpression expression = new XPathExpression(xpathString);

    AssertThat.document(filename)
        .hasXMPData()
        .matchingXPath(expression)
        ;
}

```

```

@Test
public void hasXMPData_MatchingXPath_MultipleInvocation() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";

    String xpathDateExists = "count(//xmp:CreateDate) = 1";
    String xpathDateValue = "//xmp:CreateDate[node() = '2011-02-08T15:04:19+01:00']";

    XPathExpression exprDateExists = new XPathExpression(xpathDateExists);
    XPathExpression exprDateValue = new XPathExpression(xpathDateValue);

    AssertThat.document(filename)
        .hasXMPData()
        .matchingXPath(exprDateValue)
        .matchingXPath(exprDateExists)
        ;

    // The same test in a different style:
    AssertThat.document(filename)
        .hasXMPData().matchingXPath(exprDateValue)
        .hasXMPData().matchingXPath(exprDateExists)
        ;
}

```

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

Im Kapitel 13.13: „JAXP-Konfiguration“ (S. 161) wird erläutert, wie ein beliebiger XML-Parser genutzt werden kann.

## 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 Default-Namensraum für eine XPathExpression:

```

@Test
public void hasXMPData_MatchingXPath_WithDefaultNamespace() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";

    String xpathAsString = "//foo:format = 'application/pdf'";
    String stringDefaultNS = "http://purl.org/dc/elements/1.1/";
    DefaultNamespace defaultNS = new DefaultNamespace(stringDefaultNS);
    XPathExpression expression = new XPathExpression(xpathAsString, defaultNS);

    AssertThat.document(filename)
        .hasXMPData()
        .matchingXPath(expression)
        ;
}

```

Und noch ein Beispiel für den Default-Namensraum bei der Benutzung eines XMLNode mit Erwartungswert:



```
@Test
public void hasXMPData_WithDefaultNamespace_SpecialNode() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";

    String stringDefaultNS = "http://ns.adobe.com/xap/1.0/";
    DefaultNamespace defaultNS = new DefaultNamespace(stringDefaultNS);
    String nodeName = "foo:ModifyDate";
    String nodeValue = "2011-02-08T15:04:19+01:00";
    XMLNode nodeModifyDate = new XMLNode(nodeName, nodeValue, defaultNS);

    AssertThat.document(filename)
        .hasXMPData()
        .withNode(nodeModifyDate)
    ;
}
```

## 3.31. 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.

Achtung, ein „zertifiziertes PDF-Dokument“ darf nicht mit dem „Zertifikat“ einer Signatur verwechselt werden. Für „zertifizierte PDF-Dokumente“ bietet PDFUnit folgende Testmethoden:

```
// Simple tests:
.isCertified()
.isCertifiedFor(FORM_FILLING)
.isCertifiedFor(FORM_FILLING_AND_ANNOTATIONS)
.isCertifiedFor(NO_CHANGES_ALLOWED)
```

### Beispiele

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

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

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

Als Nächstes kann der Grad der Zertifizierung überprüft werden:

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

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

### Zertifizierungsgrade

PDFUnit stellt die Zertifizierungs-Level als Konstanten zur Verfügung:

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

## 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.

#### Initialisierung

Die Instantiierung eines Master-Dokumentes erfolgt über die Methode `and( . . )`:

```
@Test
public void testInstantiation_NotEncryptedMaster() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";
    String passwordTest = "owner-password";

    AssertThat.document(filenameTest, passwordTest) ❶
        .and(filenameMaster)                        ❷
    ;
}
```

- 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 der Methode `and( )` als zweiter Parameter übergeben werden.

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

#### Überblick

Die folgende Liste gibt einen vollständigen Überblick über die vergleichenden Tests von PDFUnit. Links führen zu Kapiteln, die den jeweiligen Test ausführlich beschreiben. Die Kapitel sind alphabetisch sortiert. Das letzte Kapitel 4.20: „Sonstige Vergleiche“ (S. 99) enthält mehrere Funktionen, die nicht in einzelnen Kapiteln beschrieben sind.

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

.areBothForFastWebView()           4.20: „Sonstige Vergleiche“ (S. 99)
.haveSameActions()                 4.2: „Aktionen vergleichen“ (S. 83)
.haveSameActions(...)              4.2: „Aktionen vergleichen“ (S. 83)
.haveSameAppearance(...)           4.11: „Layout vergleichen (gerenderte Seiten)“ (S. 91)
.haveSameAuthor()                  4.7: „Dokumenteneigenschaften vergleichen“ (S. 88)
.haveSameBookmarks()               4.12: „Lesezeichen (Bookmarks) vergleichen“ (S. 93)
.haveSameCreationDate()            4.6: „Datumswerte vergleichen“ (S. 87)
.haveSameCreator()                 4.7: „Dokumenteneigenschaften vergleichen“ (S. 88)
.haveSameEmbeddedFiles(...)        4.3: „Anhänge (Attachments) vergleichen“ (S. 85)
.haveSameFieldsByName()            4.9: „Formularfelder vergleichen“ (S. 89)
.haveSameFieldsByProperties()       4.9: „Formularfelder vergleichen“ (S. 89)
.haveSameFieldsByValue()           4.9: „Formularfelder vergleichen“ (S. 89)
.haveSameFonts()                   4.15: „Schriften vergleichen“ (S. 95)
.haveSameFormat()                  4.8: „Formate vergleichen“ (S. 89)
.haveSameFormat(...)               4.8: „Formate vergleichen“ (S. 89)
.haveSameImages()                  4.5: „Bilder vergleichen“ (S. 86)
.haveSameImages(...)               4.5: „Bilder vergleichen“ (S. 86)
.haveSameJavaScript()              4.20: „Sonstige Vergleiche“ (S. 99)
.haveSameKeywords()                4.20: „Sonstige Vergleiche“ (S. 99)
.haveSameLanguage()                4.20: „Sonstige Vergleiche“ (S. 99)
.haveSameLayers()                  4.20: „Sonstige Vergleiche“ (S. 99)
.haveSameText()                    4.17: „Text vergleichen“ (S. 96)
.haveSameText(...)                 4.17: „Text vergleichen“ (S. 96)

... continued
```

```

... continuation:

.haveSameModificationDate()      4.6: „Datumswerte vergleichen“ (S. 87)
.haveSameNumberOfActions()      4.2: „Aktionen vergleichen“ (S. 83)
.haveSameNumberOfBookmarks()    4.12: „Lesezeichen (Bookmarks) vergleichen“ (S. 93)
.haveSameNumberOfEmbeddedFiles() 4.3: „Anhänge (Attachments) vergleichen“ (S. 85)
.haveSameNumberOfFields()      4.9: „Formularfelder vergleichen“ (S. 89)
.haveSameNumberOfFonts()       4.15: „Schriften vergleichen“ (S. 95)
.haveSameNumberOfImages()      4.5: „Bilder vergleichen“ (S. 86)
.haveSameNumberOfImages(..)    4.5: „Bilder vergleichen“ (S. 86)
.haveSameNumberOfLayers()      4.14: „PDF-Bestandteile vergleichen“ (S. 94)
.haveSameNumberOfPages()       4.14: „PDF-Bestandteile vergleichen“ (S. 94)
.haveSameNumberOfTaggingInfo() 4.14: „PDF-Bestandteile vergleichen“ (S. 94)
.haveSamePermissions()         4.4: „Berechtigungen vergleichen“ (S. 85)
.haveSamePermission(..)        4.4: „Berechtigungen vergleichen“ (S. 85)
.haveSameProducer()           4.7: „Dokumenteneigenschaften vergleichen“ (S. 88)
.haveSameProperties()          4.7: „Dokumenteneigenschaften vergleichen“ (S. 88)
.haveSameProperty(..)          4.7: „Dokumenteneigenschaften vergleichen“ (S. 88)
.haveSameSignatureNames()      4.16: „Signaturnamen vergleichen“ (S. 95)
.haveSameSubject()            4.7: „Dokumenteneigenschaften vergleichen“ (S. 88)
.haveSameTaggingInfo()        4.20: „Sonstige Vergleiche“ (S. 99)
.haveSameTitle()              4.7: „Dokumenteneigenschaften vergleichen“ (S. 88)
.haveSameTrappingInfo()       4.20: „Sonstige Vergleiche“ (S. 99)
.haveSameXFADData()           4.18: „XFA-Daten vergleichen“ (S. 97)
.haveSameXMPData()            4.19: „XMP-Daten vergleichen“ (S. 98)

```

## 4.2. Aktionen vergleichen

### Anzahl

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

```

@Test
public void haveSameNumberOfActions() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameNumberOfActions()
    ;
}

```

### Eigenschaften von Aktionen

Um die Aktionen zweier PDF-Dokumenten miteinander zu vergleichen, gibt es die Testfunktion `haveSameActions()`:

```

@Test
public void haveSameActions() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameActions()
    ;
}

```

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.

Typ	Relevante Eigenschaft(en) für equals()	
	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
	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 13.5: „Behandlung von Whitespaces“ (S. 153) 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.

Beim Vergleich zweier Aktionen kann die Behandlung der Leerzeichen vorgegeben werden:

```
@Test
public void compareActions_JavascriptActionWithDifferentWhitespaces() throws Exception {
    String filenameTest = PATH + "actions/documentCloseAction.pdf";
    String filenameMaster = PATH + "actions/documentCloseAction_otherWhitespace.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameActions(WhitespaceProcessing.IGNORE)
    ;
}
```

## 4.3. Anhänge (Attachments) vergleichen

### Anzahl

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

```
@Test
public void haveSameNumberOfEmbeddedFiles() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

### Namen und Inhalte

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

```
@Test
public void haveSameEmbeddedFiles() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameEmbeddedFiles(COMPARED_BY_NAME)
        .haveSameEmbeddedFiles(COMPARED_BY_CONTENT) ❶
    ;
}
```

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

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

```
// Constants defining the kind comparing embedded files:
com.pdfunit.Constants.COMPARED_BY_CONTENT
com.pdfunit.Constants.COMPARED_BY_NAME
```

Eingebettete Dateien können mit dem Hilfsprogramm `ExtractEmbeddedFiles` extrahiert werden. Siehe Kapitel 9.2: „Anhänge extrahieren“ (S. 114).

## 4.4. Berechtigungen vergleichen

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

```
@Test
public void haveSamePermissions() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSamePermissions()
    ;
}
```

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

```
@Test
public void haveSamePermissions_MultipleInvocation() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSamePermission(ALLOW_EXTRACT_CONTENT)
        .haveSamePermission(ALLOW_COPY)
        .haveSamePermission(ALLOW_MODIFY_CONTENT)
    ;
}
```

Es stehen folgende Konstanten zur Verfügung:

```
// Available permissions:

com.pdfunit.Constants.ALLOW_ASSEMBLE_DOCUMENTS
com.pdfunit.Constants.ALLOW_COPY
com.pdfunit.Constants.ALLOW_DEGRADED_PRINTING
com.pdfunit.Constants.ALLOW_EXTRACT_CONTENT ❶
com.pdfunit.Constants.ALLOW_FILL_IN
com.pdfunit.Constants.ALLOW_MODIFY_ANNOTATIONS
com.pdfunit.Constants.ALLOW_MODIFY_CONTENT
com.pdfunit.Constants.ALLOW_PRINTING
com.pdfunit.Constants.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:

```
@Test
public void haveSameNumberOfImages() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

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

```
@Test
public void haveSameNumberOfImages_OnPage2() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";
    PagesToUse ON_PAGE_2 = PagesToUse.getPage(2);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameNumberOfImages(ON_PAGE_2)
    ;
}
```

Die Möglichkeiten der Seitenauswahl sind in Kapitel 13.3: „Seitenauswahl“ (S. 150) 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 method haveSameImages() does not consider the order of the images.
 */
@Test
public void haveSameImages() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameImages()
    ;
}
```

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

Wenn aber Bilder auf bestimmten Seiten gleich sein sollen, müssen die Seiten als Parameter mitgegeben werden:

```
@Test
public void haveSameImages_OnPage2() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";
    PagesToUse ON_PAGE_2 = PagesToUse.getPage(2);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameImages(ON_PAGE_2)
    ;
}
```

```
@Test
public void haveSameImages_BeforePage2() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameImages(OnEveryPage.before(2))
    ;
}
```

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. 116).

## 4.6. Datumswerte vergleichen

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

```
// Methods comparing dates:
.haveSameCreationDate()
.haveSameModificationDate()
```

Im folgenden Beispiel wird das Änderungsdatum verglichen:

```
@Test
public void haveSameModificationDate() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

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

## 4.7. Dokumenteneigenschaften vergleichen

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

```
// Comparing document properties:

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

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

```
@Test
public void haveSameAuthor() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

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

```
@Test
public void haveSameCustomProperty() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

Mit dieser Methode können natürlich auch die Standardeigenschaften verglichen werden.

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



```
@Test
public void haveSameProperties_AllProperties() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

## 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.

```
@Test
public void haveSameFormat() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

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

```
@Test
public void haveSameFormat_OnPage2() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    PagesToUse ON_PAGE_2 = PagesToUse.getPage(2);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameFormat(ON_PAGE_2)
    ;
}
```

```
@Test
public void haveSameFormat_OnEveryPageAfter() throws Exception {
    String filename = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filename)
        .and(filenameMaster)
        .haveSameFormat(OnEveryPage.after(2))
    ;
}
```

Die Möglichkeiten, Seiten zu selektieren, sind in Kapitel 13.3: „Seitenauswahl“ (S. 150) ausführlich beschrieben.

## 4.9. Formularfelder vergleichen

### Anzahl

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

```
@Test
public void haveSameNumberOfFields() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

## Feldnamen

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

```
@Test
public void haveSameFields_ByName() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

## Feldeigenschaften

Wenn über die Namen von Feldern hinaus auch die weiteren Eigenschaften verglichen werden sollen, muss die folgende Testmethode verwendet werden:

```
@Test
public void haveSameFields_ByProperties() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameFieldsByProperties()
    ;
}
```

Feldeigenschaften können mit dem Hilfsprogramm `ExtractFieldsInfo` in eine XML-Datei überführt und analysiert werden, siehe Kapitel 9.4: „Feldeigenschaften nach XML extrahieren“ (S. 117).

## Feldinhalte

Und als Letztes können mit der Methode `haveSameFieldsByValue()` zusätzlich zu Anzahl und Namen auch die Inhalte der Felder verglichen werden. Gleichnamige Felder müssen gleiche Inhalte haben, sonst schlägt der Test fehl:

```
@Test
public void haveSameFields_ByValues() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

- ❶ Whitespaces werden für den Vergleich „normalisiert“, siehe 13.5: „Behandlung von Whitespaces“ (S. 153).

## Kombination mehrerer Tests

Unterschiedliche Vergleiche können in einem Testfall verkettet werden, allerdings gibt es dann Probleme mit dem Namen des Tests:

```
@Test
public void compareManyItems() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameFieldsByProperties()
        .haveSameFieldsByValue()
        .haveSameFonts()
        .haveSameTitle()
    ;
}
```

## 4.10. JavaScript vergleichen

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

```
@Test
public void haveSameJavaScript() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

Wenn Sie den JavaScript Code sehen wollen, können Sie ihn mit dem Hilfsprogramm `ExtractJavaScript` extrahieren. Kapitel 9.5: „JavaScript extrahieren“ (S. 118) beschreibt die nötigen Arbeitsschritte.

## 4.11. Layout vergleichen (gerenderte Seiten)

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

```
@Test
public void haveSameAppearance_CompleteDocument() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameAppearance(ON_EVERY_PAGE)
    ;
}
```

Bestimmte Seiten können auf vielfältige Weise selektiert werden. Alle Möglichkeiten werden in Kapitel 13.3: „Seitenauswahl“ (S. 150) beschrieben.

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

```

@Test
public void haveSameAppearance_OnFirstPage_InClippingArea() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    // default unit is MILLIMETER
    // also available POINTS, CENTIMETER, INCH, DPI72

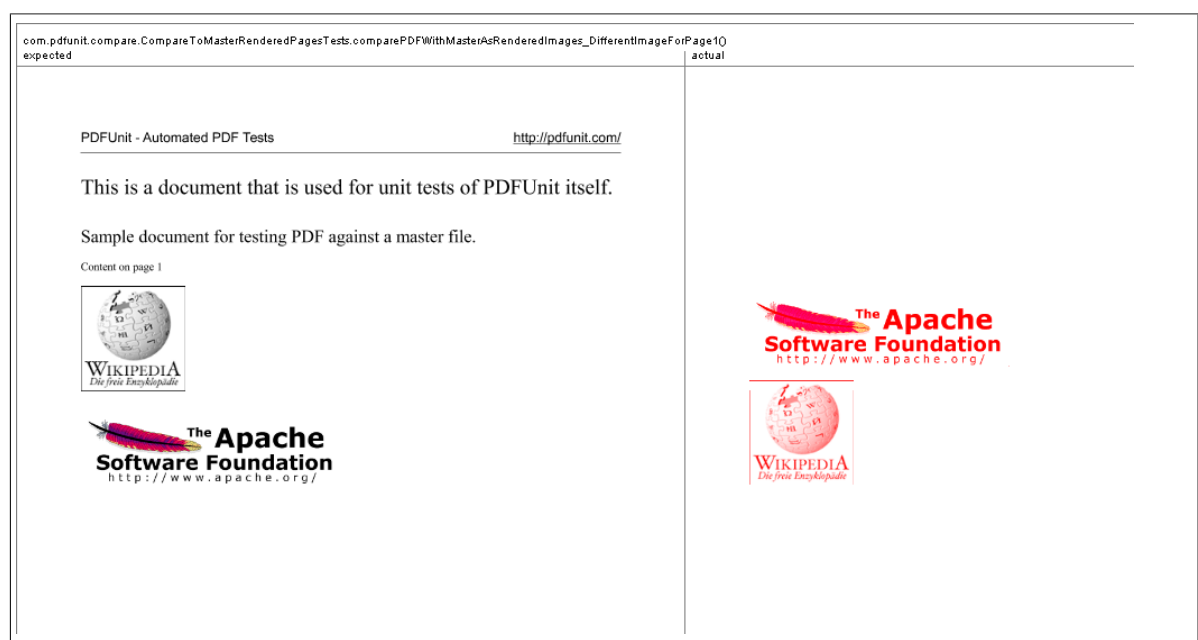
    int upperLeftX = 50;
    int upperLeftY = 755;
    int width = 370;
    int height = 35;
    ClippingArea inClippingArea = new ClippingArea(upperLeftX, upperLeftY,
                                                    width, height, POINTS
                                                    );

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameAppearance(ON_FIRST_PAGE, inClippingArea)
        ;
}

```

Die Werte für die Definition des Seitenausschnitts können in den Einheiten POINTS, MILLIMETER, CENTIMETER, INCH und DPI72 angegeben werden. Die Namen sind als Konstanten in der Klasse `com.pdfunit.Constants` 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
<p>'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'.</p>

Der Dateiname der Diff-Image-Datei wird folgendermaßen gebildet:

- Er beginnt mit dem vollständigen Name der Testdatei.

- 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“.

Die Standardkonfiguration legt ein Diff-Image für Testdateien im selben Verzeichnis ab, in dem auch die Testdatei liegt. Diff-Images für Streams und Byte-Arrays werden im Home-Verzeichnis des laufenden Java-Prozess abgelegt. Sie können die Verzeichnisse über 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](http://portableapps.com/apps/utilities/diffpdf_portable).

## 4.12. Lesezeichen (Bookmarks) vergleichen

### Anzahl

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

```
@Test
public void haveSameNumberOfBookmarks() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

### Lesezeichen mit Eigenschaften

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

- title
- namedDestination
- relatedPage
- action

```
@Test
public void haveSameBookmarks() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameBookmarks()
    ;
}
```

Wenn es Unklarheit über die Lesezeichen gibt, können alle Informationen über Lesezeichen mit dem Hilfsprogramm `ExtractBookmarks` in eine XML-Datei exportiert und dort analysiert werden. Siehe Kapitel 9.6: „Lesezeichen nach XML extrahieren“ (S. 119).

## 4.13. "Named Destinations" vergleichen

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

### Anzahl

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

```
@Test
public void compareNumberOfNamedDestinations() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

### 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:

```
@Test
public void compareNamedDestinations() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

## 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 zählbaren Bestandteile gegeben werden, für die es Vergleichsmethoden gibt:

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

.haveSameNumberOfActions()
.haveSameNumberOfBookmarks()
.haveSameNumberOfEmbeddedFiles()
.haveSameNumberOfFields()
.haveSameNumberOfFonts()
.haveSameNumberOfImages()
.haveSameNumberOfImages(..)
.haveSameNumberOfLayers()
.haveSameNumberOfPages()
.haveSameNumberOfTaggingInfo()
```

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

```
@Test
public void haveSameNumberOfPages() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameNumberOfPages()
    ;
}
```

```
@Test
public void haveSameNumberOfTaggingInfo() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameNumberOfTaggingInfo();
}
```

```
@Test
public void haveSameNumberOfLayers() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameNumberOfLayers();
}
```

## 4.15. Schriften vergleichen

### Anzahl

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

```
@Test
public void haveSameNumberOfFonts() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameNumberOfFonts();
}
```

### 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:

```
@Test
public void haveSameFonts() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "fonts/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameFonts();
}
```

Informationen über die Schriften eines Dokumentes können mit dem Hilfswerkzeug `ExtractFont-sInfo` extrahiert werden. Siehe Kapitel 9.9: „Schrifteigenschaften nach XML extrahieren“ (S. 123).

## 4.16. Signaturnamen vergleichen

Der Vergleich von Signaturen zweier PDF-Dokumente bezieht sich im Release 2054.10 nur auf die Namen der Signatur:

```
@Test
public void haveSameSignatureNames() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameSignatureNames();
}
```

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. 125).

## 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:

```
@Test
public void haveSameText_CompleteDocument() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameText(ON_EVERY_PAGE);
}
```

Ein Vergleich kann auf Seiten beschränkt werden. Alle Möglichkeiten, Seiten auszuwählen, werden in Kapitel 13.3: „Seitenauswahl“ (S. 150) beschrieben:

```
@Test
public void haveSameText_OnSinglePage() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameText(ON_FIRST_PAGE);
};

AssertThat.document(filenameTest)
    .and(filenameMaster)
    .haveSameText(ON_LAST_PAGE);
}
```

Zusätzlich kann der Textvergleich noch auf Seitenausschnitte beschränkt werden:

```
@Test
public void haveSameText_CompleteDocument_InClippingArea() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";
    int upperLeftX = 50;
    int upperLeftY = 720;
    int width = 150;
    int height = 30;

    ClippingArea inClippingArea = new ClippingArea(upperLeftX, upperLeftY, width, height);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameText(ON_EVERY_PAGE, inClippingArea);
}
```



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

```
@Test
public void compareText_SameContentDifferentWhitespaces() throws Exception {
    String filenameTest = PATH + "master/compareToMaster_differentWhitespace.pdf";
    String filenameMaster = PATH + "master/compareToMaster.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameText(ON_FIRST_PAGE, WhitespaceProcessing.IGNORE)
    ;
}
```

## 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

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

```
// Method for tests with XMP data
.haveSameXFAData().matchingXPath(XPathExpression, RESULTTYPE)
```

Falls es Zweifel über die Inhalte der XFA-Daten gibt, können diese mit dem Hilfsprogramm `ExtractXFAData` in eine XML-Datei extrahiert werden. Siehe Kapitel 9.13: „XFA-Daten nach XML extrahieren“ (S. 128).

### Beispiel - Resulttype Node

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

```
@Test
public void haveSameXFAData_ResulttypeNode() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    DefaultNamespace defaultNS
        = new DefaultNamespace("http://www.xfa.org/schema/xfa-template/2.6/");
    XPathExpression expression
        = new XPathExpression("//default:pageSet", defaultNS);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameXFAData()
        .matchingXPath(expression, AS_RESULTTYPE_NODE)
    ;
}
```

Wie das Beispiel zeigt, muss der erwartete Ergebnistyp angegeben werden. Die verfügbaren Ergebnistypen sind als Konstanten in der Klasse `com.pdfunit.Constants` definiert. Es sind:

```
// Result types for XPath-processing:
com.pdfunit.Constants.AS_RESULTTYPE_BOOLEAN
com.pdfunit.Constants.AS_RESULTTYPE_NUMBER
com.pdfunit.Constants.AS_RESULTTYPE_NODE
com.pdfunit.Constants.AS_RESULTTYPE_NODESET
com.pdfunit.Constants.AS_RESULTTYPE_STRING
```

Whitespaces werden beim Vergleich von XFA-Daten ignoriert.

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:

```
@Test
public void haveSameXFADData_ResulttypeBoolean() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    DefaultNamespace defaultNS
        = new DefaultNamespace("http://www.xfa.org/schema/xf-template/2.6/");
    XPathExpression expression
        = new XPathExpression("count(//default:field) = 3", defaultNS);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameXFADData()
        .matchingXPath(expression, AS_RESULTTYPE_BOOLEAN)
    ;
}
```

Test mit „boolean“ als erwartetem Ergebnistyp (AS\_RESULTTYPE\_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. 112).

## Beispiel - Default-Namensraum

Beachten Sie den Umgang mit dem Default-Namensraum. Er muss mit der Klasse `com.pdfunit.DefaultNamespace` deklariert werden, weil ein XML-Dokument prinzipiell mehrere Default-Namensräume haben kann. Eine automatische Erkennung ist somit nicht sicher.

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

```
/**
 * Test with multiple XPath expressions and multiple default namespaces.
 * It is not recommended to create tests in this way.
 */
@Test
public void haveSameXFADData_MultipleDefaultNamespaces() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    String nsStringXFATemplate = "http://www.xfa.org/schema/xf-template/2.6/";
    String nsStringXFALocale = "http://www.xfa.org/schema/xf-locale-set/2.7/";

    DefaultNamespace nsXFATemplate = new DefaultNamespace(nsStringXFATemplate);
    DefaultNamespace nsXFALocale = new DefaultNamespace(nsStringXFALocale);

    String xpathSubform = "//default:subform/@name[.='movie']";
    String xpathLocale = "//default:locale/@name[.='nl_BE']";

    XPathExpression exprXFATemplate = new XPathExpression(xpathSubform, nsXFATemplate);
    XPathExpression exprXFALocale = new XPathExpression(xpathLocale, nsXFALocale);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameXFADData()
        .matchingXPath(exprXFATemplate, AS_RESULTTYPE_BOOLEAN)
        .matchingXPath(exprXFALocale, AS_RESULTTYPE_BOOLEAN)
    ;
}
```

## 4.19. XMP-Daten vergleichen

Auch die XMP-Daten zweier PDF-Dokumente werden auf XPath-Basis miteinander verglichen. Die Implementierungen der XMP- und XFA-Testmethoden sind gleich und damit auch die Schnittstelle.

Weil das vorhergehende Kapitel 3.29: „XFA Daten“ (S. 75) 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. 128).

## Übersicht

Folgende Methode steht zur Verfügung:

```
// Method for tests with XMP data:
.haveSameXMPData().matchingXPath(XPathExpression, RESULTTYPE)
```

## Beispiel

```
@Test
public void haveSameXMPData_ResulttypeNode() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";
    XPathExpression expression = new XPathExpression("//pdf:Producer");

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameXMPData()
        .matchingXPath(expression, AS_RESULTTYPE_NODE)
    ;
}
```

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 methods, comparing PDF. Not described before:
.areBothForFastWebView()
.haveSameKeywords()
.haveSameLanguage()
.haveSameLayers()
.haveSameTrappingInfo()
.haveSameTaggingInfo()
```

Hier zwei Beispiele:

### Fast WebView, Tagging

```
@Test
public void areBothForFastWebView() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .areBothForFastWebView()
    ;
}
```

```
@Test
public void haveSameTaggingInfo() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameTaggingInfo()
    ;
}
```

## Verkettung von Vergleichsmethoden

Vergleichsmethoden können auch verkettet werden:

```
@Test
public void haveSameAuthorTitleFonts() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

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.

```
@Test
public void textInMultipleDocuments() throws Exception {
    String fileName1 = PATH + "multipleDocuments/document_en.pdf";
    String fileName2 = PATH + "multipleDocuments/document_es.pdf";
    String fileName3 = PATH + "multipleDocuments/document_de.pdf";
    File file1 = new File(fileName1);
    File file2 = new File(fileName2);
    File file3 = new File(fileName3);
    File[] files = {file1, file2, file3};

    String expectedDate = "28.09.2014";
    String expectedDocumentID = "XX-123";

    AssertThat.eachDocument(files)
        .hasText(ON_FIRST_PAGE)
        .containing(expectedDate)
        .containing(expectedDocumentID)
    ;
}
```

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

```
// Methods to validate a set of PDF documents:
.containsImage(...) 3.6: „Bilder in Dokumenten“ \(S. 23\)
.containsOneImageOf(...) 3.6: „Bilder in Dokumenten“ \(S. 23\)

.hasAuthor() 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasBookmark() 3.17: „Lesezeichen \(Bookmarks\) und Sprungziele“ \(S. 49\)
.hasBookmarks() 3.17: „Lesezeichen \(Bookmarks\) und Sprungziele“ \(S. 49\)
.hasEncryptionLength(...) 3.18: „Passwort“ \(S. 53\)
.hasField(...) 3.11: „Formularfelder“ \(S. 33\)
.hasFields() 3.11: „Formularfelder“ \(S. 33\)
.hasFont() 3.19: „Schriften“ \(S. 54\)
.hasFonts() 3.19: „Schriften“ \(S. 54\)
.hasFormat(...) 3.10: „Format“ \(S. 32\)
.hasJavaScript() 3.13: „JavaScript“ \(S. 43\)
.hasKeywords() 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasLocale(...) 3.22: „Sprachinformation \(Language\)“ \(S. 63\)

.hasNumberOf...() 3.4: „Anzahl verschiedener PDF-Bestandteile“ \(S. 21\)

.hasNoAuthor() 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasNoKeywords() 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasNoLocale() 3.22: „Sprachinformation \(Language\)“ \(S. 63\)
.hasNoProducer() 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasNoProperty() 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasNoSubject() 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasNoText() 3.23: „Texte“ \(S. 64\)
.hasNoTitle() 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasNoXFADData() 3.29: „XFA Daten“ \(S. 75\)
.hasNoXMPData() 3.30: „XMP-Daten“ \(S. 78\)
.hasOwnerPassword(...) 3.18: „Passwort“ \(S. 53\)
.hasPermission() 3.5: „Berechtigungen“ \(S. 22\)
.hasProperty(...) 3.8: „Dokumenteneigenschaften“ \(S. 28\)
.hasSignature(...) 3.21: „Signaturen und Zertifikate“ \(S. 58\)
.hasSignatures() 3.21: „Signaturen und Zertifikate“ \(S. 58\)
.hasSignedSignatureFields() 3.21: „Signaturen und Zertifikate“ \(S. 58\)
.hasSubject() 3.8: „Dokumenteneigenschaften“ \(S. 28\)

... continued
```

```
... continuation:

.hasText(..)                3.23: „Texte“ (S. 64)
.hasTitle()                 3.8: „Dokumenteigenschaften“ (S. 28)
.hasUnsignedSignatureFields() 3.21: „Signaturen und Zertifikate“ (S. 58)
.hasUserPassword(..)         3.18: „Passwort“ (S. 53)
.hasVersion()                3.28: „Version“ (S. 74)
.hasXFADaata()               3.29: „XFA Daten“ (S. 75)
.hasXMPData()                3.30: „XMP-Daten“ (S. 78)

.isCertified()               3.31: „Zertifiziertes PDF“ (S. 81)
.isCertifiedFor(..)          3.31: „Zertifiziertes PDF“ (S. 81)
.isLinearizedForFastWebView() 3.9: „Fast Web View“ (S. 31)
.isSigned()                  3.21: „Signaturen und Zertifikate“ (S. 58)
.isTagged()                  3.26: „Tagging“ (S. 71)

... (end of list)
```

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

## Kapitel 6. PDFUnit-Monitor

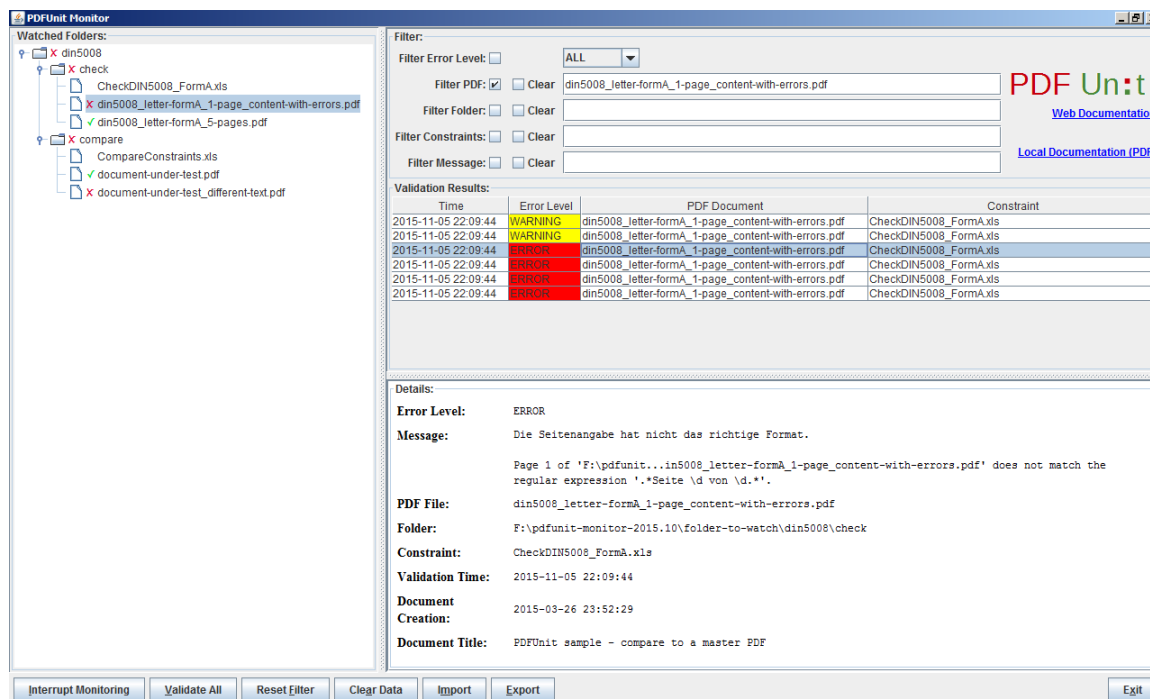
Der PDFUnit-Monitor ist eine graphische Anwendung, um Tests für PDF-Dokumente anzustoßen und das Ergebnis anzeigen zu lassen. Die Zielgruppe für die Anwendung sind Nicht-Programmierer.

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

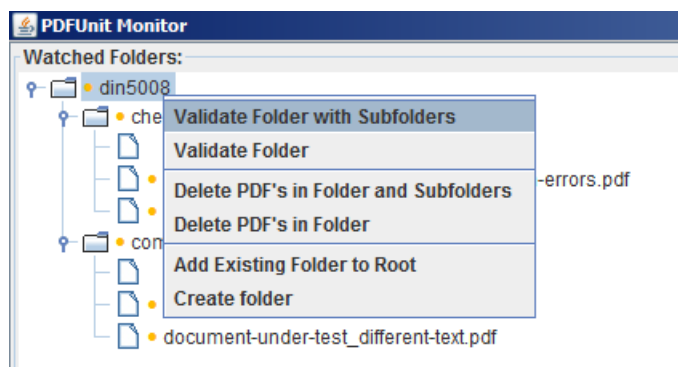
Der PDFUnit-Monitor überwacht alle PDF-Dokumente unterhalb eines definierten Verzeichnisses und prüft die dortigen Dokumente gegen Regeln, die in Excel-Dateien hinterlegt sind, die ebenfalls in den überwachten Verzeichnissen liegen müssen. Erfüllt ein PDF alle Regeln, wird es in der Baumstruktur mit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



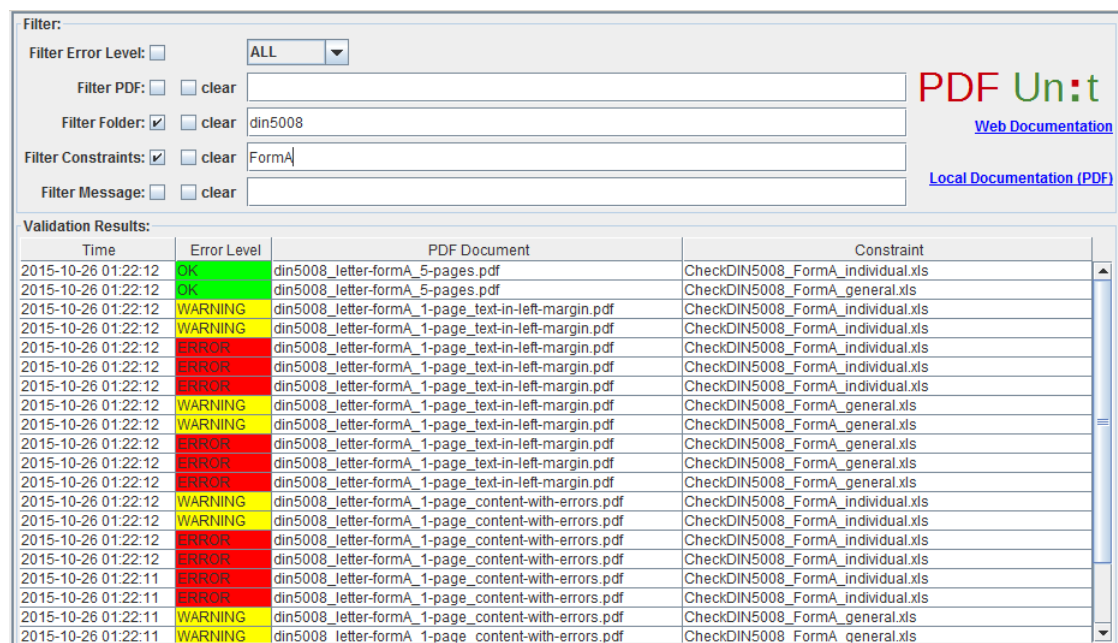
Ein Doppelklick auf ein PDF-Dokument in der Baumstruktur öffnet das Dokument mit der Standardanwendung des Betriebssystems. Gleiches gilt für einen Doppelklick auf eine Excel-Datei.

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.



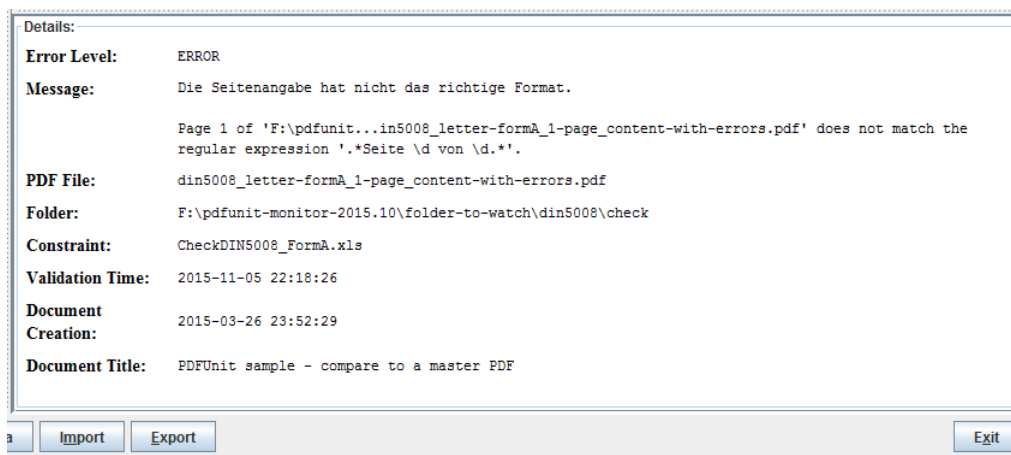
Die Liste der Fehler kann über Filter eingeschränkt werden. Die Filtermöglichkeiten stehen mit der Verzeichnisstruktur in Verbindungen. Wird eine Excel-Datei in der Struktur angeklickt, wird ein Filter mit diesem Namen aktiviert und es werden alle PDF-Dokumente, die mit dieser Excel-Datei geprüft wurden,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



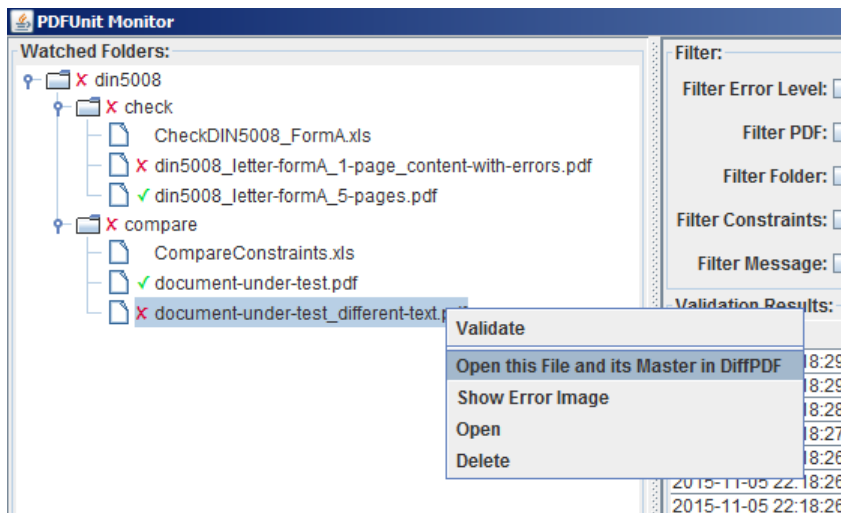


Der erste Teil der Fehlermeldung stammt aus der Excel-Datei und wird von der Person, die die Tests erstellt, geschrieben. Weitere Teile der Meldung stammen vom Testwerkzeug PDFUnit. Neben der eigentlichen Fehlermeldung werden weitere nützliche Informationen über das PDF-Dokument, die Regeldatei und den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

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

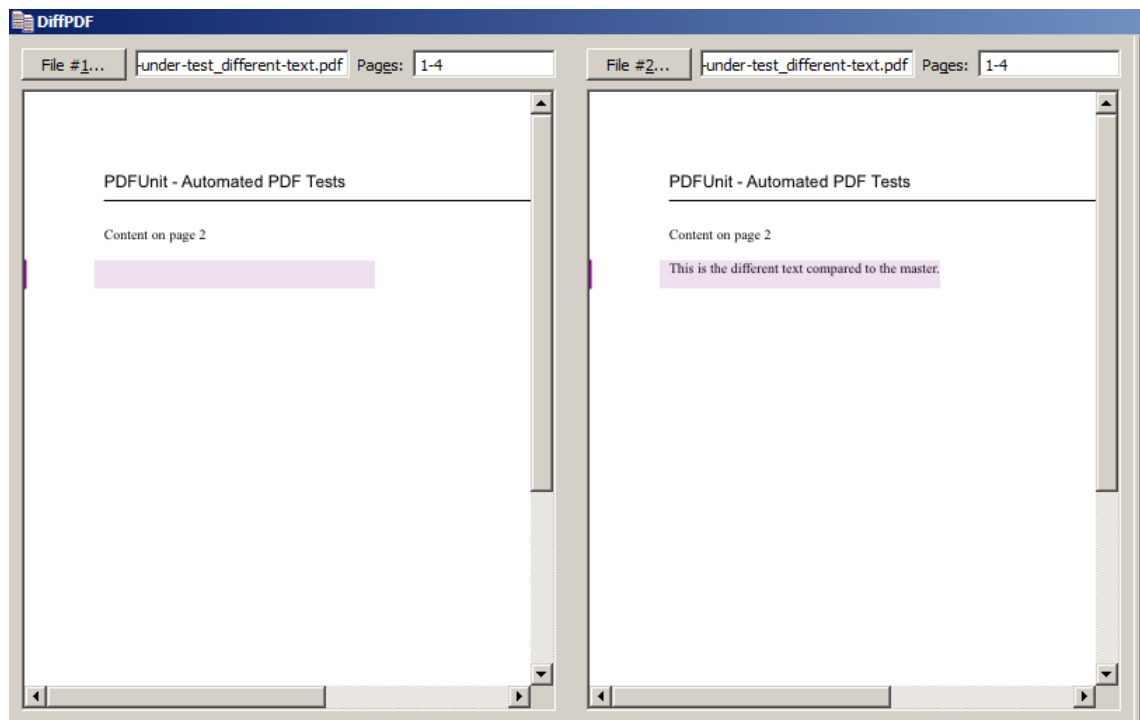
## Vergleich eines PDF-Dokumentes gegen eine Vorlage

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

```
@Test
public void hasProducer_CopyrightAsUnicode() throws Exception {
    String filename = PATH + "unicode/unicode_producer.pdf";

    AssertThat.document(filename)
        .hasProducer()
        .matchingComplete("txt2pdf v7.3 \u00A9 SANFACE Software 2004") // 'copyright'
    ;
}
```

### Längere Unicode-Texte

Es wäre nun zu mühsam, für längere Texte den Hex-Code aller Buchstaben herauszufinden. Deshalb stellt PDFUnit das kleine Programm `ConvertUnicodeToHex` zur Verfügung. Übergeben Sie den ausländischen Text als String an das Werkzeug, entnehmen Sie der daraus erzeugten Datei anschließend den Hex-Code und fügen ihn in Ihr Testprogramm ein. Eine genaue Beschreibung steht in Kapitel 9.12: „Unicode-Texte in Hex-Code umwandeln“ (S. 127). Das Test mit Unicode sieht dann so aus:

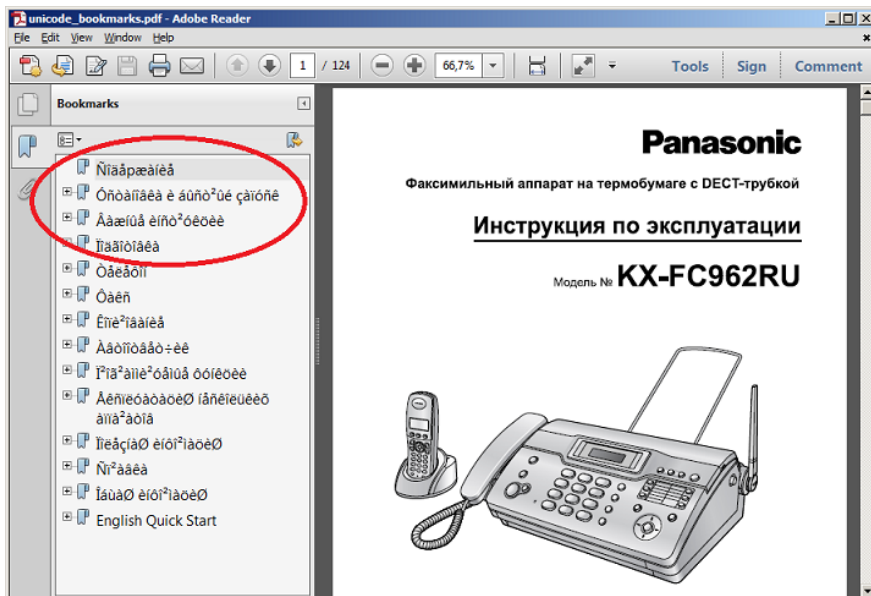
```
@Test
public void hasSubject_Greek() throws Exception {
    String filename = PATH + "unicode/unicode_subject.pdf";
    String expectedSubject = "##### ## ### ##### / #####"; ❶
    //String expectedSubject = "\u0395\u03C1\u03B3\u03B1\u03C3\u03C4\u03AE"
    //      + "\u03C1\u03B9\u03BF \u039C\u03B7\u03C7\u03B1"
    //      + "\u03BD\u03B9\u03BA\u03AE\u03C2 \u0399\u0399 "
    //      + "\u03A4\u0395\u0399 \u03A0\u0395\u0399\u03A1"
    //      + "\u0391\u0399\u0391 / \u039C\u03B7\u03C7\u03B1"
    //      + "\u03BD\u03BF\u03BB\u03CC\u03B3\u03BF\u03B9";

    AssertThat.document(filename)
        .hasSubject()
        .matchingComplete(expectedSubject)
    ;
}
```

- 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:



Obwohl die Bookmarks im Adobe Reader® nicht lesbar sind (in der XML-Datei sind sie ebenso nicht lesbar) funktioniert der Vergleich. Letztendlich liegt es daran, dass Java intern Unicode-fähig ist:

```
/**
 * This test needs the following setting before starting ANT or Maven:
 * set JAVA_TOOL_OPTIONS=-Dfile.encoding=UTF-8
 */
@Test
public void hasBookmarks_MatchingXML() throws Exception {
    String filenamePDF = PATH + "unicode/unicode_bookmarks.pdf";
    String filenameXML = PATH + "unicode/unicode_bookmarks.xml";
    File xmlFile = new File(filenameXML);

    AssertThat.document(filenamePDF)
        .hasBookmarks()
        .matchingXML(xmlFile)
    ;
}
```

- 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. 112) wird beschrieben, wie PDFUnit-Tests zusammen mit XPath funktionieren. Auch die XPath-Ausdrücke können Unicode enthalten:

```
@Test
public void hasBookmarks_MatchingXPath() throws Exception {
    String filename = PATH + "unicode/unicode_bookmarks.pdf";
    // String xpath = "//Title[@Action][.='1.4 Νίαääείάιέ0']";
    String xpath = "//Title[@Action]"
        + "[.='\\u00D1\\u00EE\\u00E4\\u00E5p\\u00E6\\u00E0\\u00ED\\u00E8\\u00E5']";

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

## UTF-8 File-Encoding für Shell-Skripte

Vorsicht bei Daten, die aus dem Dateisystem gelesen werden. Deren Interpretation ist vom Encoding des jeweiligen Dateisystems abhängig. Deshalb ist jedes Java-Programm, das Dateien verarbeitet, also auch PDFUnit, von der Umgebungsvariablen `file.encoding` abhängig.

Es gibt mehrere Möglichkeiten, diese Umgebungsvariable für den jeweiligen Java-Prozess zu setzen:

```
set _JAVA_OPTIONS=-Dfile.encoding=UTF8
set _JAVA_OPTIONS=-Dfile.encoding=UTF-8

java -Dfile.encoding=UTF8
java -Dfile.encoding=UTF-8
```

## UTF-8 File-Encoding für ANT

Während der Entwicklung von PDFUnit gab es zwei Tests, die unter Eclipse fehlerfrei liefen, unter ANT aber mit einem Encoding-Fehler 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
```

## Maven - UTF-8 Konfiguration in der pom.xml

In der `pom.xml` können Sie UTF-8 an vielen Stellen konfigurieren. Die folgenden Code-Ausschnitte zeigen mehrere Beispiele, wählen Sie die passenden für Ihr Problem:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
```

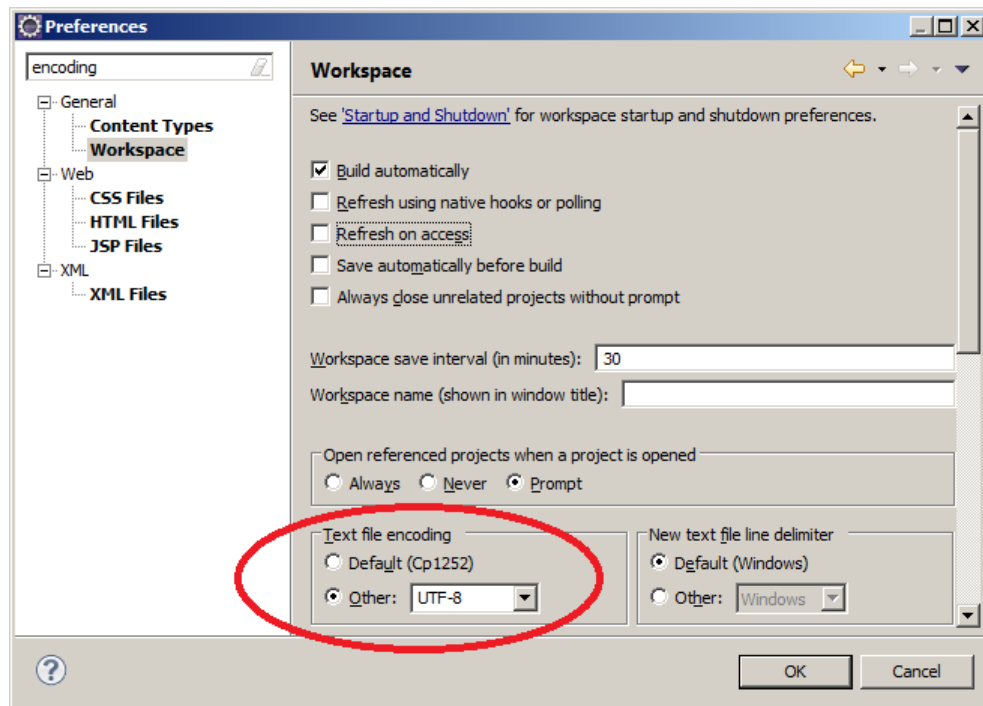
```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.5.1</version>
  <configuration>
    <encoding>UTF-8</encoding>
  </configuration>
</plugin>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <encoding>UTF-8</encoding>
  </configuration>
</plugin>
```

## Eclipse auf UTF-8 einstellen

Wenn Sie XML-Dateien in Eclipse erstellen, ist es nicht unbedingt nötig, Eclipse auf UTF-8 einzurichten, denn XML-Dateien sind auf UTF-8 voreingestellt. Für andere Dateitypen ist aber die Codepage des

Betriebssystems voreingestellt. Sie sollten daher, wenn Sie mit Unicode-Daten arbeiten, das Default-Encoding für den gesamten Workspace auf UTF-8 einstellen:



Abweichend von dieser Standardeinstellung können einzelne Dateien in einem anderen Encoding 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 dargestellt. Ausschlaggebend dafür ist das File-Encoding der Ausgabe, das von PDFUnit selber nicht beeinflusst werden kann. Wenn Sie in Eclipse, ANT oder Maven dafür gesorgt haben, dass „UTF-8“ als Codepage verwendet wird, sind die meisten Probleme beseitigt. Danach können noch Zeichen aus der Codepage „UTF-16“ die Darstellung der Fehlermeldung korrumpieren.

Das PDF-Dokument im nächsten Beispiel enthält einen Layer-Namen, der UTF-16BE-Zeichen enthält. Um die Wirkung der Unicode-Zeichen in der Fehlermeldung zu zeigen, wurde der 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.
 */
@Test
public void hasLayer_NameContainingUnicode_UTF16_ErrorIntended() throws Exception {
    String filename = PATH + "unicode/unicode_layerName.pdf";

    // String layername = "Ebene 1(4)"; // This is shown by Adobe Reader®,
    //                               "Ebene _XXX"; // and this is the used string
    String wrongNameWithUTF16BE =
        "\u00fe\u00ff\u0000\u0000b\u0000e\u0000n\u0000e\u0000 \u0000_XXX";

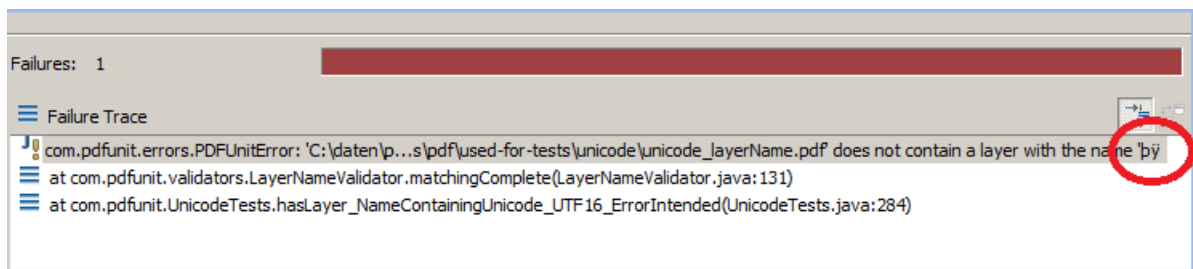
    AssertThat.document(filename)
        .hasLayer()
        .matchingComplete(wrongNameWithUTF16BE);
}
```

Wenn die Tests mit ANT ausgeführt wurden, zeigt ein Browser die von PDFUnit erzeugte Fehlermeldung fehlerfrei an, einschließlich der Zeichenkette `pÿEbene _XXX` am Ende:

'C:\daten\p...s\pdf\used-for-tests\unicode\unicode\_layerName.pdf' does not contain a layer with the name 'p̃Ebene \_XXX'.

```
junit.framework.AssertionFailedError: 'C:\daten\p...s\pdf\used-for-tests\unicode
\unicode_layerName.pdf' does not contain a layer with the name 'p̃Ebene _XXX'.
at com.pdfunit.validators.LayerNameValidator.matchingComplete(LayerNameValidator.java:133)
at com.pdfunit.UnicodeTests.hasLayer_NameContainingUnicode_UTF16_ErrorIntended(UnicodeTests.java:274)
```

Eclipse dagegen hat in der JUnit-View Probleme mit den Null-Bytes. Die Meldung '... \unicode\_layerName.pdf' does not contain a layer with the name 'p̃' endet nicht mit dem Text p̃Ebene \_XXX. Sie wird nach der internen Byte-Order-Markierung abgeschnitten:



## Unicode für unsichtbare Zeichen - &nbsp;

Im praktischen Betrieb trat einmal ein Problem auf, bei dem ein „non-breaking space“ in den Testdaten enthalten war, das zunächst als normales Leerzeichen wahrgenommen wurde. Der String-Vergleich lieferte aber einen Fehler, der erst durch die Verwendung von Unicode beseitigt werden konnte:

```
@Test
public void nodeValueWithUnicodeValue() throws Exception {
    String filename = PATH + "xfa/xfaBasicToggle.pdf";

    DefaultNamespace defaultNS = new DefaultNamespace("http://www.w3.org/1999/xhtml");
    String nodeValue = "The code ... the button's";
    String nodeValueWithNBSP = nodeValue + "\u00A0"; // The content terminates with a NBSP.
    XMLNode nodeP7 = new XMLNode("default:p[7]", nodeValueWithNBSP, defaultNS);

    AssertThat.document(filename)
        .hasXFADData()
        .withNode(nodeP7)
    ;
}
```

## 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 Spezi-alkapitel.

```
// Overview over XPath methods:
.hasBookmarks().matchingXPath(...) 3.17: „Lesezeichen (Bookmarks) und Sprungziele“ (S. 49)
.hasFields().matchingXPath(...)     3.11: „Formularfelder“ (S. 33)
.hasFonts().matchingXPath(...)       3.19: „Schriften“ (S. 54)
.hasSignatures().matchingXPath(...)  3.21: „Signaturen und Zertifikate“ (S. 58)
.hasXFADData().matchingXPath(...)    3.29: „XFA Daten“ (S. 75)
.hasXMPData().matchingXPath(...)     3.30: „XMP-Daten“ (S. 78)

// Comparing two documents using XPath:
.haveSameXFADData().matchingXPath(...) 4.18: „XFA-Daten vergleichen“ (S. 97)
.haveSameXMPData().matchingXPath(...)  4.19: „XMP-Daten vergleichen“ (S. 98)
```

Intern verwendet PDFUnit XPath auch für die Implementierung anderer Testmethoden.

### Interne Nutzung von XMLUnit

Alle Vergleiche von XML-Werte 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](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. 114) 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.
 */
@Test
public void hasXFADData_UsingDefaultNamespace() throws Exception {
    String filename = PATH + "xfa/xfa-enabled.pdf";
    DefaultNamespace defaultNS = new DefaultNamespace("http://www.xfa.org/schema/xci/2.6/");
    XMLNode aliasFoo = new XMLNode("foo:log/foo:to", "memory", defaultNS);
    XMLNode aliasBar = new XMLNode("bar:log/bar:to", "memory", defaultNS);

    AssertThat.document(filename)
        .hasXFADData()
        .withNode(aliasFoo)
        .withNode(aliasBar)
    ;
}
```

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. Für jeden Datentyp existiert eine Konstante:

```
// Result types for XPath-processing:

com.pdfunit.Constants.AS_RESULTTYPE_BOOLEAN
com.pdfunit.Constants.AS_RESULTTYPE_NUMBER
com.pdfunit.Constants.AS_RESULTTYPE_NODE
com.pdfunit.Constants.AS_RESULTTYPE_NODESET
com.pdfunit.Constants.AS_RESULTTYPE_STRING
```

Test mit dem Ergebnistyp `AS_RESULTTYPE_BOOLEAN` sind insofern kritisch, weil XPath nicht zwischen „nicht gefunden“ und „false“ unterscheiden kann.

### XPath-Kompatibilität

Für XPath-Ausdrücke stehen im Prinzip alle Syntaxelemente und Funktionen von XPath zur Verfügung. Allerdings ist die Menge der tatsächlich verfügbaren Funktionen von der Version des verwendeten XML-Parsers und XSLT-Prozessors abhängig. PDFUnit verwendet den vom JDK mitgelieferten XML-Parser bzw. XSLT-Prozessor (Standard JAXP). Insofern bestimmt die jeweils verwendete Java-Engine die Kompatibilität zum XPath-Standard.

Das Kapitel 13.13: „JAXP-Konfiguration“ (S. 161) erläutert die allgemeine JAXP-Konfiguration eines JRE/JDK, um z.B. Xerces als externen XML-Parser zu nutzen.

## 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. 127)
ExtractBookmarks         9.6: „Lesezeichen nach XML extrahieren“ (S. 119)
ExtractEmbeddedFiles     9.2: „Anhänge extrahieren“ (S. 114)
ExtractFieldsInfo        9.4: „Feldeigenschaften nach XML extrahieren“ (S. 117)
ExtractFontsInfo         9.9: „Schrifteigenschaften nach XML extrahieren“ (S. 123)
ExtractImages            9.3: „Bilder aus PDF extrahieren“ (S. 116)
ExtractJavaScript        9.5: „JavaScript extrahieren“ (S. 118)
ExtractNamedDestinations 9.11: „Sprungziele nach XML extrahieren“ (S. 126)
ExtractSignaturesInfo    9.10: „Signaturdaten nach XML extrahieren“ (S. 125)
ExtractXFADData          9.13: „XFA-Daten nach XML extrahieren“ (S. 128)
ExtractXMPData           9.14: „XMP-Daten nach XML extrahieren“ (S. 128)
RenderPdfClippingAreaToImage 9.8: „PDF-Seite ausschnittsweise in PNG umwandeln“ (S. 121)
RenderPdfToImages        9.7: „PDF-Dokument seitenweise in PNG umwandeln“ (S. 120)
```

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%

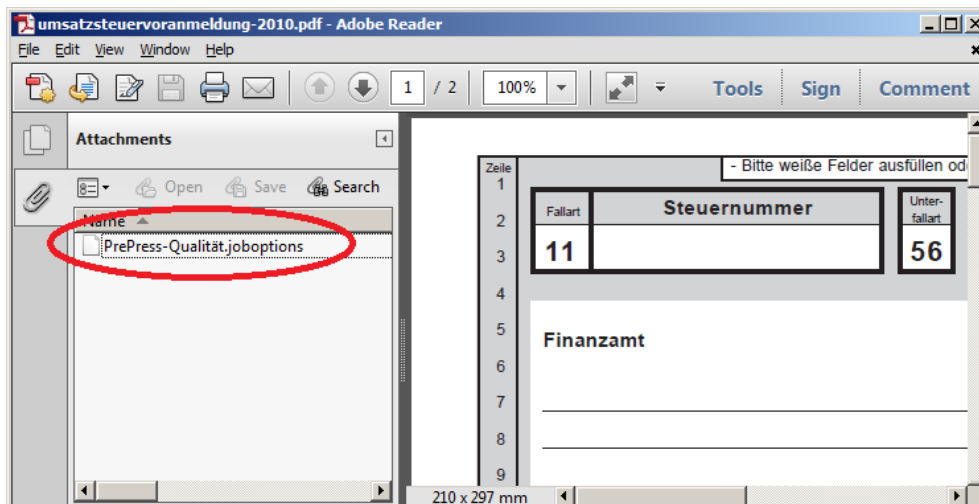
set TOOL=com.pdfunit.tools.ExtractEmbeddedFiles
set OUT_DIR=./tmp
set IN_FILE=umsatzsteuervoranmeldung-2010.pdf
set PASSWD=

java %TOOL% %IN_FILE% %OUT_DIR% %PASSWD%
endlocal

```

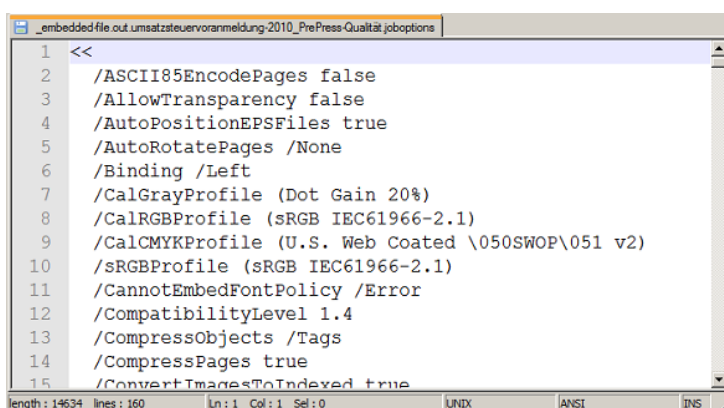
## Eingabe

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. 23) beschrieben.

### Aufruf

```
::
:: Extract all images of a PDF document into a PNG file for each image.
::

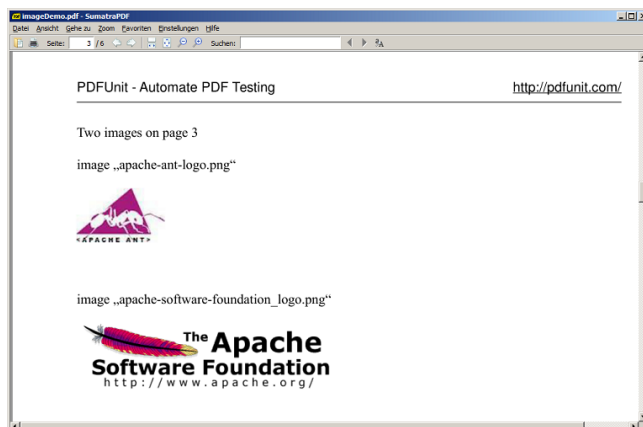
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

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. 33) 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. 43) 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. 117) als Beispiel herhalten musste, enthält für die Felder `nameField`, `ageField` und `comment` auch JavaScript.

Innerhalb des Java-Programms, mit dem das PDF-Dokument erstellt wird, sieht der JavaScript-Code für das Feld „ageField“ so aus:

```
String scriptCodeCheckAge = "var ageField = this.getField('ageField');"
+ "ageField.setAction('Validate','checkAge()');"
+ "
+ "function checkAge() {"
+ "  if(event.value < 12) {"
+ "    app.alert('Warning! Applicant\\'s age can not be younger than 12.');"
+ "    event.value = 12;"
+ "  }"
+ "}"
;
```

## Ausgabe

Die erstellte Datei `_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. 49) beschreibt die Verwendung der erzeugten XML-Datei für Bookmarks-Tests.

## Aufruf

```
::
:: Extract bookmarks from a PDF document into an XML file
::

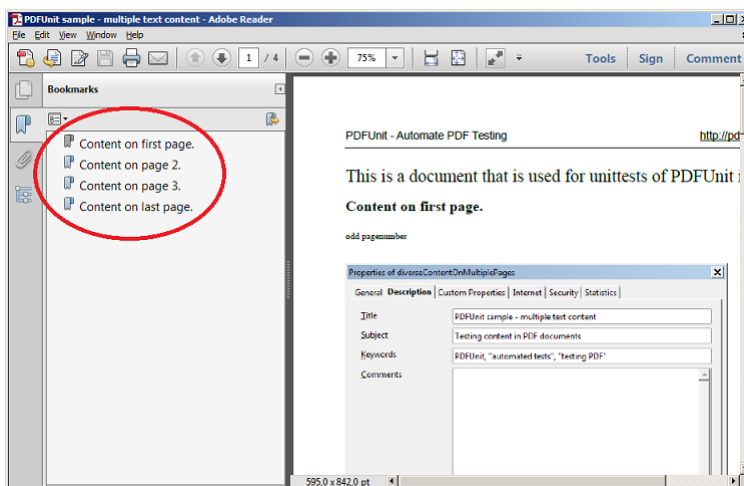
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

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. 47) 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%

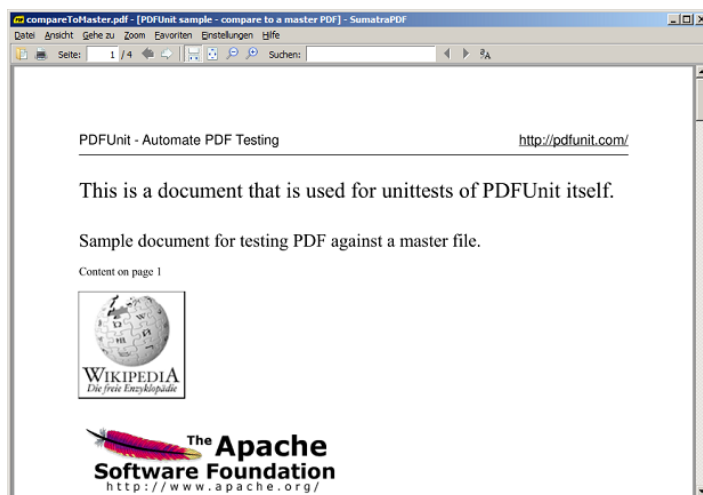
set TOOL=com.pdfunit.tools.RenderPdfToImages
set OUT_DIR=./tmp
set IN_FILE=compareToMaster.pdf
set PASSWD=

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:



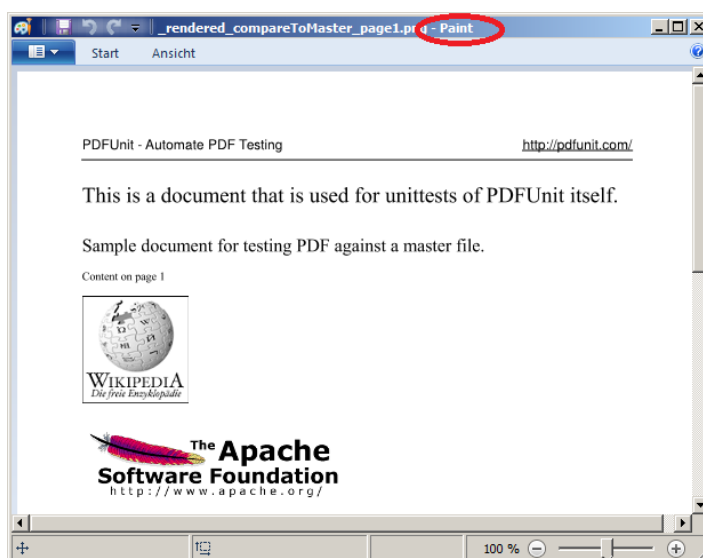


## Ausgabe

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. 48) 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

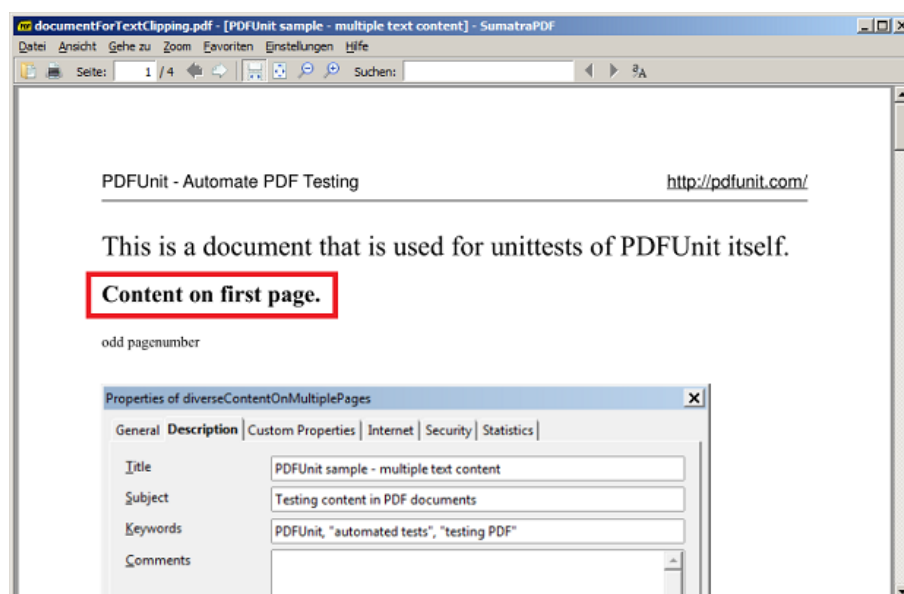
```
::  
:: Render a part of a PDF page into an image file  
::  
  
@echo off  
setlocal  
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%  
set CLASSPATH=./lib/jpedal/*;%CLASSPATH%  
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%  
set CLASSPATH=./lib/aspectj-1.8.0/*;%CLASSPATH%  
  
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. 54) 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
::

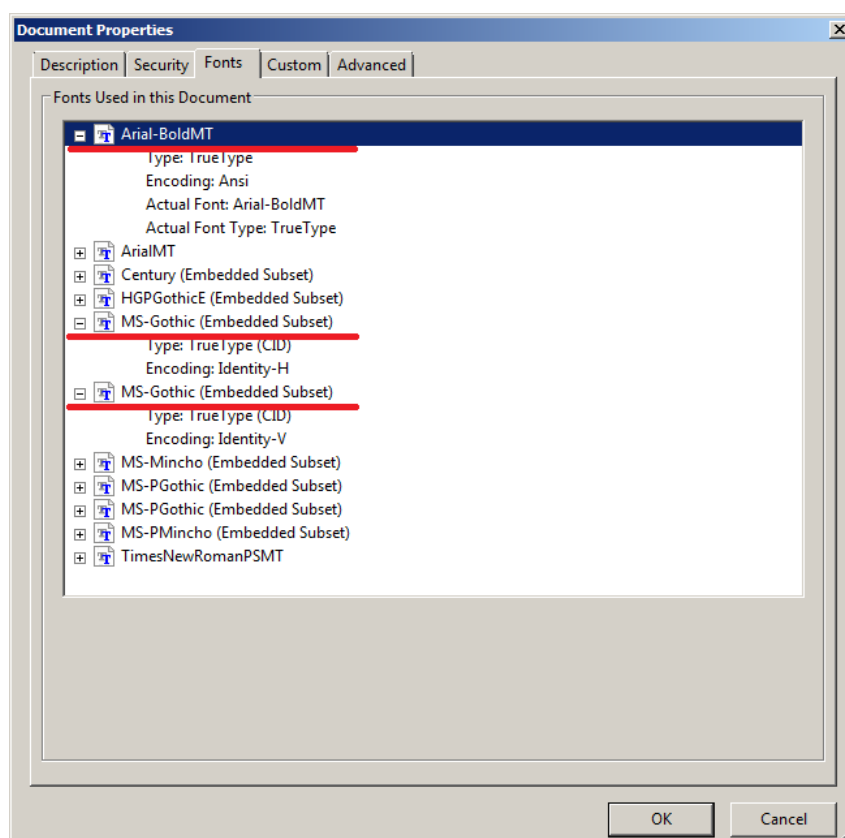
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

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. 58) beschreibt die Tests mit Signaturen und Zertifikaten.

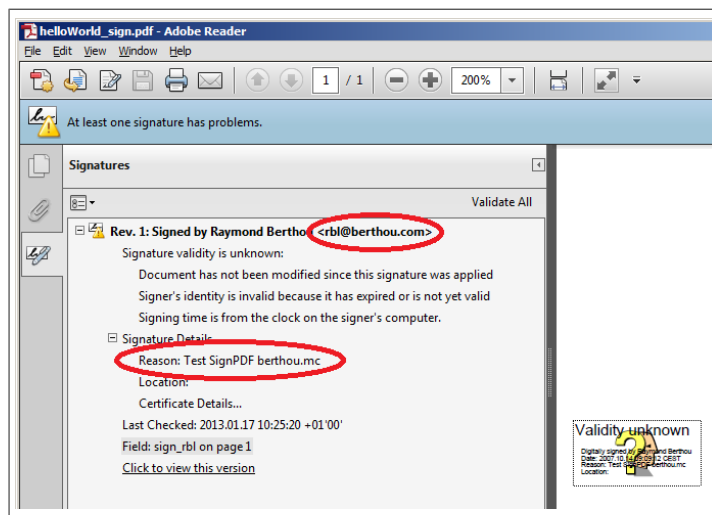
Mit dem folgenden Skript starten Sie die Extraktion:

### 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:

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <signatureList>
3   <signature name="sign_xbl" 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>rb1@berthou.com</E>
15      <OU>VeriSign Trust Network</OU>
16      <OU>www.verisign.com/repository/RPA Incorpor. 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" signDa
23    <certificateChain>
24      <certificate type="X.509" version="3" notAfter="2008-02-12T00:59:59+0100" notBefo

```

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. 49) 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. 107) 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. 75) 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     </agent>
8     <pdf>
9       <!-- [0..n] -->
10      <fontInfo />
11    </pdf>
12  </config>
13  <present>
14    <acrobat>
15  </acrobat>
16</present>
17</template>
18<template xmlns="http://www.xfa.org/schema/xfa-template/2.6/">
19  <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
20    <connectionSet xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
21      <localeSet xmlns="http://www.xfa.org/schema/xfa-locale-set/2.7/">
22        <x:xmpmeta xmlns:x="adobe:ns:meta/"
23          x:xmptk="Adobe XMP Core 4.2.1-c041 52.337767, 2008/04/13-15:41:00"
24        >
25        <form xmlns="http://www.xfa.org/schema/xfa-form/2.8/" checksum="51PBR1CXK0zKwd88"
26      >
27    </form>
28  </xmpmeta>
29</datasets>
30</template>
31</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. 78) 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 Dokumentenvolage (Template) erstellt. Die Platzhalter für unterschiedliche Texte sind Formularfelder, beispielsweise Textbausteine für AGB's.

#### Problem

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

#### Lösungsansatz

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

#### Lösung

```
/**
 * Verifying that all content fits inside a field.
 */
@Test
public void noTextOverflow_AllFields() throws Exception {
    String filename = PATH + "acrofields/fieldsWithAttributes.pdf";

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

Der Test ist auch für einzelne Felder möglich:

```
@Test
public void noTextOverflow_Field_AlignLeft() throws Exception {
    String filename = PATH + "acrofields/fieldSizeAndText.pdf";
    String fieldNameLeftAlign_Fitting = "Textfield, text inside, align left:";

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

In Kapitel 3.12: „Formularfelder, Textüberlauf“ (S. 41) 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

```
/**
 * Tests for the situation when two companies merge.
 *
 * @author Carsten Siedentop, February 2013
 */
public class NewCompanyTests {

    /**
     * This sample shows how to verify that a logo is visible on each page.
     */
    @Test
    public void verifyNewLogoOnEveryPage() throws Exception {
        String filename = PATH + "images/imagesWithSameImagesOnOnePage.pdf";
        String newLogoImage = PATH + "images/newLogo.png";

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

## 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

```
/**
 * Tests for the situation when two companies merge.
 *
 * @author Carsten Siedentop, February 2013
 */
public class NewCompanyTests {

    /**
     * This sample shows how to verify that the new signature is used.
     */
    @Test
    public void verifyNewSignatureOnLastPage() throws Exception {
        String filename = PATH + "images/imagesWithSameImagesOnOnePage.pdf";
        String newSignatureImage = PATH + "images/CEO-signature.png";

        AssertThat.document(filename)
            .containsImage(newSignatureImage, ON_LAST_PAGE)
        ;
    }
}
```

## 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 eine Methode, Inhalte von PDF-Dokumenten auf ihre **Nicht-Existenz** zu überprüfen.

### Lösung

```
/**
 * This examples shows how to verify that an expected text
 * is not present in the complete document.
 *
 * @author Carsten Siedentop, February 2013
 */
public class NewCEOTests {

    @Test
    public void verifyOldCEONotPresent() throws Exception {
        String filename = PATH + "content/diverseContentOnMultiplePages.pdf";
        String oldCEO = "NameOfOldCEO";

        AssertThat.document(filename)
            .hasText(ON_EVERY_PAGE)
            .notContaining(oldCEO)
        ;
    }
}
```

## 10.5. Selenium und PDFUnit in Kombination

### Ausgangssituation

Sie bieten auf Ihrer Webseite benutzerspezifisch generierte PDF-Dokumente an. Diese sollen getestet werden.

### Problem

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

### Lösungsansatz

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

## Lösung

```
/**
 * This sample shows how to test a PDF document with Selenium and PDFUnit.
 *
 * @author Carsten Siedentop, March 2012
 */
public class PDFFromWebsiteTest {

    private WebDriver driver;

    /**
     * When the url of the pdf document inside an HTML page is generated dynamically,
     * you have to find the link (href) first.
     * Input data for the web page can also be typed with Selenium (not shown here).
     */
    @Test
    public void verifyPDF_LoadedBySeleniumWebdriver() throws Exception {
        // arrange, navigate to web site:
        String startURL = "http://www.unicode.org/charts/";
        driver.get(startURL);
        WebElement element = driver.findElement(By.linkText("Basic Latin (ASCII)"));
        String hrefValue = element.getAttribute("href");

        // act, load PDF web site:
        URL url = new URL(hrefValue);
        InputStream is = url.openStream();
        InputStream bis = new BufferedInputStream(is);

        // assert:
        String expectedTitle = "The Unicode Standard, Version 6.3";

        AssertThat.document(bis)
            .hasTitle().matchingComplete(expectedTitle)
            .hasText(Constants.ON_FIRST_PAGE)
            .containing("0000", "007F")
        ;
    }

    @Before
    public void createDriver() throws Exception {
        driver = new HtmlUnitDriver();
        Logger htmlunitLogger = Logger.getLogger("com.gargoylesoftware.htmlunit");
        htmlunitLogger.setLevel(java.util.logging.Level.SEVERE);
    }

    @After
    public void closeAll() throws Exception {
        driver.close();
    }
}
```

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

## 10.6. HTML2PDF - Hat die dynamische PDF-Erstellung funktioniert?

### Ausgangssituation

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

### Problem

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

### Lösungsansatz

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

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

## Lösung

```
/**
 * This sample shows how to test an HTML page with Selenium,
 * then let it be rendered by the server to PDF and verify that
 * content also appears in PDF.
 *
 * @author Carsten Siedentop, February 2013
 */
public class Html2PDFTest_English {

    private WebDriver driver;

    @Test
    public void testHtml2PDFRenderer() throws Exception {
        String urlWikipediaSelenium = "http://en.wikipedia.org/wiki/Selenium_%28software%29";
        driver.get(urlWikipediaSelenium);

        String section1 = "History";
        String section2 = "Selenium components";
        String section3 = "See also";
        String section4 = "References";
        String section5 = "External links";

        assertLinkPresent(section1);
        assertLinkPresent(section2);
        assertLinkPresent(section3);
        assertLinkPresent(section4);
        assertLinkPresent(section5);

        String linkName = "Download as PDF";
        InputStream bis = loadPDF(linkName);

        AssertThat.document(bis)
            .hasText(Constants.ON_ANY_PAGE)
            .containing(section1)
            .containing(section2)
            .containingIgnoringWhitespaces(section3) ❶
            .containing(section4)
            .containing(section5);
    }

    private void assertLinkPresent(String partOfLinkText) {
        driver.findElement(By.xpath("//a[.//span = '" + partOfLinkText + "']"));
    }

    private InputStream loadPDF(String linkName_LoadAsPDF) throws Exception {
        driver.findElement(By.linkText(linkName_LoadAsPDF)).click();
        String title = "Rendering finished - Wikipedia, the free encyclopedia";
        assertEquals(title, driver.getTitle());
        WebElement element = driver.findElement(By.linkText("Download the file"));
        String hrefValue = element.getAttribute("href");

        URL url = new URL(hrefValue);
        InputStream is = url.openStream();
        InputStream bis = new BufferedInputStream(is);
        return bis;
    }

    @After
    public void tearDown() throws Exception {
        driver.quit();
    }

    @Before
    public void createDriver() throws Exception {
        driver = new HtmlUnitDriver();
    }
}
```

- ❶ Der Link mit dem Namen „See also“ wird durch Wikipedia serverseitig **nicht** gerendert! Eventuell ist das ein Bug, denn in der deutschen Version der Webseite zum selben Stichwort, werden alle Links in PDF gerendert.

## 10.7. Caching von Testdokumenten

### Ausgangssituation

Sie haben viele Tests - das ist gut.

### Problem

Die Tests laufen zu langsam - das ist schlecht. Vielleicht liegt es daran, dass die PDF-Dokumente sehr groß sind, aber die müssen schließlich auch getestet werden.

### Lösungsansatz

Sie instantiieren das Testdokument nur einmal für alle Tests einer Klasse in einer statischen Methode, die mit `@BeforeClass` annotiert ist.

### Lösung

```
/**
 * This sample shows how to cache a test document for many tests.
 *
 * @author Carsten Siedentop, February 2013
 */
public class CachedDocumentTestDemo {

    private static DocumentValidator document;

    @BeforeClass // Document will be instantiated once for all tests:
    public static void loadTestDocument() throws Exception {
        String filename = PATH + "content/diverseContentOnMultiplePages.pdf";
        document = AssertThat.document(filename);
    }

    @Test
    public void testNumberOfBookmarks() throws Exception {
        document.hasNumberOfBookmarks(4);
    }

    @Test
    public void testCreationDate() throws Exception {
        Calendar expectedCreationDate = DateHelper.getCalendar("21.04.2013", "dd.MM.yyyy");
        document.hasCreationDate().matchingComplete(expectedCreationDate, AS_DATE);
    }

    @Test
    public void testNoModificationDate() throws Exception {
        document.hasNoModificationDate();
    }

    @Test // bad designed test
    public void testAll() throws Exception {
        Calendar expectedCreationDate = DateHelper.getCalendar("21.04.2013", "dd.MM.yyyy");
        document.hasNumberOfBookmarks(4)
            .hasCreationDate().matchingComplete(expectedCreationDate, AS_DATE)
            .hasNoModificationDate();
    }
}
```

Aus Performance-Gründen könnten Sie zwar auch alle Testmethoden in einem Test verketteten. Aber wie würden Sie den Test dann nennen? Der Inhalt einer Testmethode sollte sich möglichst in ihrem Namen widerspiegeln, sonst werden Testreports mit mehreren hundert Tests schwer verständlich. `testAll` ist daher ein schlechter Name. Er ist bedeutungslos.

PDFUnit arbeitet intern zustandslos. Da aber auch verschiedene externe Bibliotheken verwendet werden, kann die Zustandslosigkeit nicht zu 100% garantiert werden. Sollte es Probleme geben, ändern Sie die Annotation `@BeforeClass` in `@Before` und entfernen den `static` Modifier. Dann wird das PDF-Dokument wieder für jeden Test neu instantiiert.

```
@Before // Document will be instantiated for each test. No caching:
public void loadTestDocument() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";
    document = AssertThat.document(filename);
}
```

## 10.8. 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.
 */
@Test
public void hasBookmarks_NestedDepthLimitedTo1() throws Exception {
    String filename = PATH + "namedDestination/manyNamedDestinations.pdf";
    String xpathNoNestedBookmarks = "count(//Title[count(ancestor::Title) > 1]) = 0";
    XPathExpression expression = new XPathExpression(xpathNoNestedBookmarks);

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



# 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	Java Version
• Windows-XP, 32 Bit	• Oracle JDK-1.7.0, 32 + 64 Bit
• Windows-7, 64 Bit	• Oracle JDK-1.8.0, 64 Bit
• Kubuntu Linux 12/04, 32 Bit	• Oracle JDK-1.8.0 b100, Windows, 32 + 64 Bit
• Kubuntu Linux 12/04, 64 Bit	• IBM J9, R26_Java726_SR4, Windows 7, 64 Bit
• Mac OS X, 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 ohne vorhandenes iText

### Download und Entpacken von PDFUnit-Java

Laden Sie die Datei `pdfunit-java-VERSION.zip` aus dem Internet: <http://www.pdfunit.com/de/download/index.html>. Wenn Sie eine Lizenz erworben haben, erhalten Sie eine neue ZIP-Datei per Mail.

Entpacken Sie die ZIP-Datei, z.B. in den Projektordner `PROJECT_HOME/lib/pdfunit-java-VERSION`. Dieser Ordner wird nachfolgend `PDFUNIT_HOME` genannt.

### Download und Entpacken von iText

Zur Ausführungszeit von PDFUnit-Java wird iText in einer Version 5.3.3 oder höher benötigt. Laden Sie sich iText (<http://sourceforge.net/projects/itext/files/>) herunter.

Entpacken Sie die ZIP-Datei von iText und kopieren Sie den entstandenen Ordner als Verzeichnis unterhalb von `PROJECT_HOME/lib`.

Beachten Sie, dass iText für den kommerziellen Einsatz lizenzpflichtig ist.

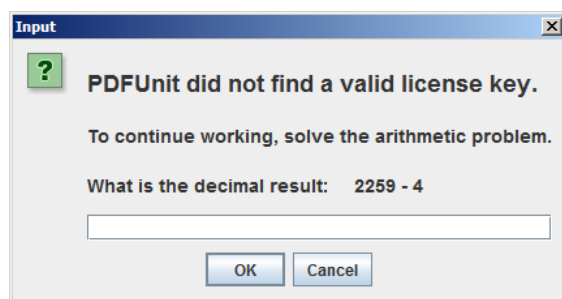
## Classpath konfigurieren

Alle JAR-Dateien, die von PDFUnit mitgeliefert werden, und die JAR-Dateien von iText müssen in den Classpath aufgenommen werden. Ebenso die Datei `config.properties`.

Sofern Sie ein lizenziertes PDFUnit verwenden, muss die Lizenzschlüsseldatei `license-key_pdfunit-java.lic` auch im Classpath liegen.

## PDFUnit ohne Lizenzschlüssel nutzen

Es ist erlaubt, PDFUnit zu Evaluationszwecken ohne Lizenz zu verwenden. Wenn Sie dann einen Test starten, erscheint ein kleines Fenster mit einer leichten Rechenaufgabe, die Sie lösen müssen. Mit der richtigen Lösung laufen die Tests durch, andernfalls nicht und Sie müssen sie neu starten.



Das Fenster mit der Rechenaufgabe ist gelegentlich durch andere Anwendungen verdeckt. Dann „hängt“ das ANT-Skript oder Maven-Skript. Sie finden das Dialogfenster, wenn Sie alle Anwendungsfenster minimieren.

PDFUnit bringt Hilfsprogramme `com.pdfunit.tools.*` zum Extrahieren verschiedener Informationen aus den PDF-Dokumenten mit. Diese benötigen keinen Lizenzschlüssel, Sie müssen also auch keine Rechenaufgabe lösen :-)

## 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 umgehend eine Antwort.

Die Lizenzkosten werden individuell gestaltet. Ein kleines Unternehmen muss nicht genauso viel zahlen, wie ein großes Unternehmen. Und wer nur wenige PDF-Dokumente testet, zahlt selbstverständlich auch weniger. Sollten Sie in den Besitz einer kostenlosen Lizenz kommen wollen, lassen Sie sich Argumente einfallen - es ist möglich.

## Lizenzschlüssel installieren

Wenn Sie eine Lizenz beantragt haben, erhalten Sie eine ZIP-Datei mit PDFUnit-Java und eine separate Datei `license-key_pdfunit-java.lic`. Installieren Sie die ZIP-Datei, wie oben beschrieben, und sorgen Sie dafür, dass auch die Lizenzdatei im Classpath aufgenommen wird.

Jede Änderung an der Lizenzdatei macht diese unbrauchbar. Nehmen Sie in einem solchen Falle mit [PDFUnit.com](http://PDFUnit.com) Verbindung auf und beantragen Sie eine neue Lizenzdatei.

## Überprüfung der Installation

Wenn Sie Probleme mit der Konfiguration haben, starten Sie das Skript zur Überprüfung der Installation: `verifyInstallation.bat` oder `verifyInstallation.sh`. Es ist in Kapitel 11.7: „Überprüfung der Konfiguration“ (S. 144). ausführlich beschrieben.

## 11.3. Installation bei existierendem iText

### PDFUnit mit vorhandener iText-Version nutzen

PDFUnit nutzt iText (<http://itextpdf.com>) und andere Bibliotheken. iText seinerseits benötigt auch weitere Bibliotheken. Aus Lizenzgründen wird PDFUnit ohne die iText-Bibliotheken ausgeliefert, aber mit allen sonstigen Zusatzbibliotheken.

Achten Sie bei der Konfiguration des Classpath darauf, dass Bibliotheken, die sowohl iText als auch PDFUnit nutzen, nur einmal registriert sind. Auch müssen die Versionen der Fremdbibliotheken sowohl für iText, als auch für PDFUnit passen.

Sollte es bei diesen Fremdbibliotheken zu Versionskonflikten kommen, müssen Sie über eine geeignete Classpath-Konfiguration dafür sorgen, dass PDFUnit die Fremdbibliotheken verwendet, die PDFUnit ausliefert, und Ihre PDF-erstellenden Programme die Fremdbibliotheken nutzen, die zu Ihrer iText-Version passen.

### PDFUnit mit zusätzlicher iText-Version nutzen

Angenommen, Sie verwenden für die Erstellung ihrer PDF-Dokumente eine alte Version von iText und möchten oder können diese Version nicht aktualisieren. In diesem Falle benötigen Sie zwei getrennte Classpath-Konfigurationen. PDFUnit muss mit einer passenden neueren Version von iText benutzt werden. Beachten Sie aber, dass iText ab Version 5.3.3 für den kommerziellen Einsatz lizenzpflichtig ist.

In ANT können Sie diese Unterscheidung über eine eigene Classpath-Property vornehmen:

```
<path id="project.classpath.test">
  <pathelement location="${dir.source.test.resources}" />
  <pathelement location="${dir.build.classes}" />

  <fileset dir="${dir.external.tools}/pdfunit-2015.10">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

```
<path id="project.classpath.prod">
  <pathelement location="${dir.build.classes}" />

  <fileset dir="${dir.external.tools}">
    <include name="**/*.jar"/>
    <exclude name="pdfunit-2015.10/**/*.jar"/>
  </fileset>
</path>
```

In Maven ist es am günstigsten, eine separate pom-Datei anzulegen, z.B. `pom_pdfunit.xml`.

## 11.4. Classpath in Eclipse, ANT, Maven definieren

Alle Entwicklungsumgebungen benötigen die folgenden Dateien im Classpath:

- alle JAR-Dateien, die von PDFUnit ausgeliefert werden
- die JAR-Dateien von iText
- die Datei `config.properties`
- die Datei `license-key_pdfunit-java.lic`, falls eine Lizenz verwendet wird

Sollten die Dateien nicht auffindbar sein, gibt es entsprechende Fehlermeldungen:

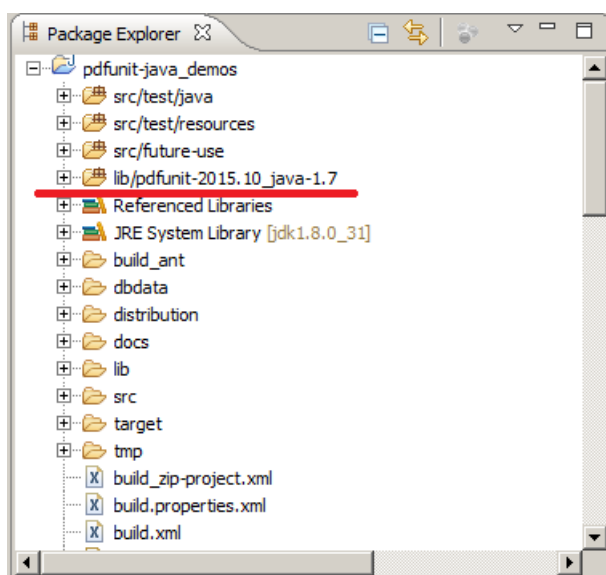
- `Could not find 'config.properties'. Verify classpath and installation.`
- `No valid license key found. Switching to evaluation mode. Contact PDFUnit.com if you are interested in a license.`

- A field of the license-key-file could not be parsed. Do you have the correct license-key file? Check your classpath and PDFUnit version. Please read the documentation.

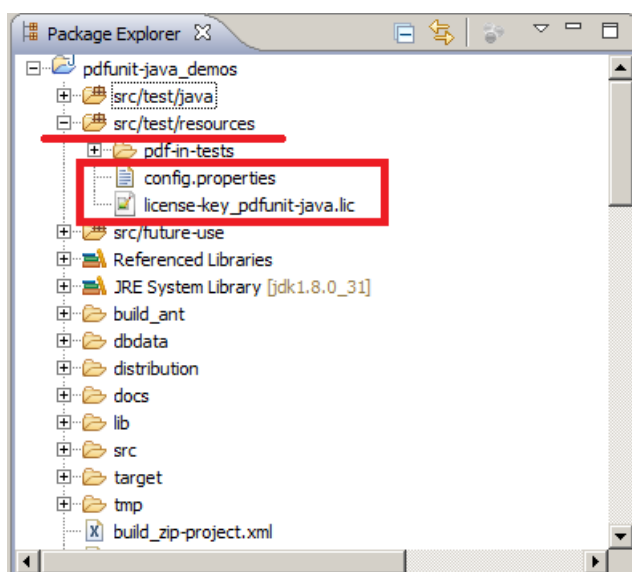
Nachfolgend werden Beispiele gezeigt, wie der Classpath in verschiedenen Umgebungen konfiguriert werden kann. Zusätzlich wird in Kapitel 11.5: „Pfade über Systemumgebungsvariablen setzen“ (S. 142) eine Alternative beschrieben, den Ort der Dateien `config.properties` und `license-key_pdfunit-java.lic` über Systemumgebungsvariablen der Java-Runtime zu deklarieren.

## Eclipse konfigurieren

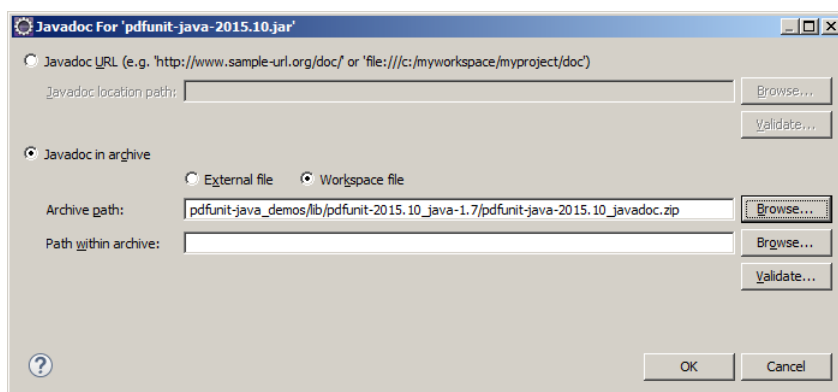
Die einfachste Konfiguration für Eclipse ist, das komplette Installationsverzeichnis `PDFUNIT_HOME` und alle JAR-Dateien einzeln in den Classpath aufzunehmen:



Eine andere Möglichkeit ist die, die Datei `config.properties` in das Verzeichnis `src/test/resources` zu verschieben und das Verzeichnis dann in den Classpath aufzunehmen:



Als Letztes können Sie noch die mitgelieferte Javadoc-Datei in Eclipse registrieren, damit die Javadoc-Kommentare von PDFUnit in Eclipse angezeigt werden. Im Verzeichnis `PDFUNIT_HOME` befindet sich dazu eine Datei mit dem Namensmuster `pdfunit-java-VERSION_javadoc.zip`.



## ANT konfigurieren

Es gibt verschiedene Möglichkeiten, ANT für PDFUnit zu konfigurieren. In allen Varianten müssen die JAR-Dateien des Installationsverzeichnis `PDFUNIT_HOME` und der Verzeichnisse `PDFUNIT_HOME/lib/*` in den Classpath aufgenommen werden. Ebenso muss die Datei `config.properties` im Classpath liegen.

Wenn Sie keine Änderungen in der `config.properties` benötigen, ist es am einfachsten, zusätzlich zu den JAR-Dateien `PDFUNIT_HOME` selbst in den Classpath aufzunehmen, wie es das folgende Listing zeigt:

```
<!--
It is important to have the directory of PDFUnit itself in the classpath,
because the file 'config.properties' must be found.
-->
<property name="dir.build.classes" value="build/classes" />
<property name="dir.external.tools" value="lib-ext" />
<property name="dir.external.tools.pdfunit" value="lib-ext/pdfunit-2015.10" />

<path id="project.classpath">
  <pathelement location="${dir.external.tools.pdfunit}" />
  <pathelement location="${dir.build.classes}" />

  <!-- If there are problems with duplicate JARs, use more detailed filesets: -->
  <fileset dir="${dir.external.tools}">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

Sie können die Datei `config.properties` aber auch in ein beliebiges Verzeichnis legen, beispielsweise in `src/test/resources`. Diese Variante wird empfohlen, wenn Sie Änderungen an der Konfiguration vornehmen. Die Konfigurationsdatei wird in Kapitel 11.6: „Einstellungen in der `config.properties`“ (S. 143) beschrieben. Der Classpath in ANT sieht dann folgendermaßen aus:

```
<path id="project.classpath">
  <!--
The file 'config.properties' should not be located more than once in
the classpath, because it hurts the DRY principle.
-->
  <pathelement location="src/test/resources" />
  <pathelement location="${dir.external.tools.pdfunit}" />
  <pathelement location="${dir.build.classes}" />

  <!-- If there are problems with duplicate JARs, use more detailed fileset: -->
  <fileset dir="${dir.external.tools}">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

## Maven konfigurieren

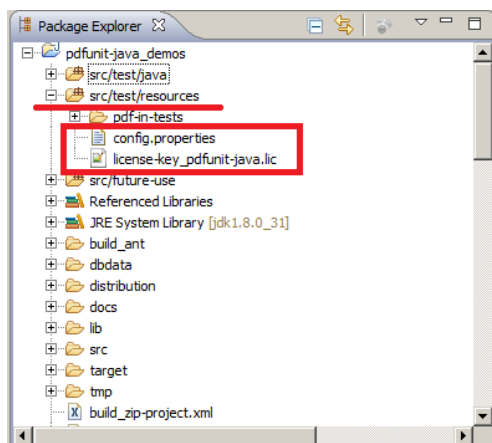
PDFUnit wird noch nicht über ein zentrales Repository zur Verfügung gestellt. Um es dennoch mit Maven zu nutzen, muss es selber in ein lokales oder unternehmenszentrales Repository eingestellt

werden. Dazu wechseln Sie in das Verzeichnis `PDFUNIT_HOME\lib` und führen dort den folgenden Maven-Befehl aus:

```
mvn install:install-file -Dfile=<PATH_TO>pdfunit-java-VERSION.jar -DpomFile=<PATH_TO>pom.xml
```

Anschließend kopieren Sie die Datei `config.properties` in das Verzeichnis `src/test/resources`.

Das folgende Bild zeigt die Projektstruktur nach dem Kopieren:



In der `pom.xml` Ihres Projektes nehmen Sie diese Abhängigkeit auf:

```
<dependency>
  <groupId>com.pdfunit</groupId>
  <artifactId>pdfunit</artifactId>
  <version>2015.10</version>
  <scope>compile</scope>
</dependency>
```

## Letzter Schritt für lizenziertes PDFUnit

Die Lizenzdatei `license-key_pdfunit-java.lic` muss immer im Classpath liegen, sonst erscheint die oben beschriebene Message-Box mit der Rechenaufgabe.

## 11.5. Pfade über Systemumgebungsvariablen setzen

Die Dateien `config.properties` und die Lizenzdatei können auch außerhalb des Classpath's liegen, wenn deren Orte über entsprechende Java-Runtime Umgebungsvariablen deklariert werden. Die Umgebungsvariablen lauten:

- `-Dpdfunit.configfile`
- `-Dpdfunit.licensekeyfile`

Abhängig vom Testsystem (Eclipse, ANT, Maven) können diese Parameter auf vielfältige Weise gesetzt werden. Nutzen Sie die allgemeinen Informationen dieser Systeme, um zu erfahren, wie Java System-Properties jeweils gesetzt werden. Eine weniger bekannte Möglichkeit, die für alle Umgebungen funktioniert, ist die Betriebssystem-Umgebungsvariable `_JAVA_OPTIONS`:

```
set JAVA_OPTIONS=-Dpdfunit.configfile=..\myfolder\config.properties
```

Sollten Sie zu diesem Thema Fragen haben, schreiben Sie ein Mail an: [info\[at\]pdfunit.com](mailto:info[at]pdfunit.com).

## 11.6. 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 = ./
```

## 11.7. 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.

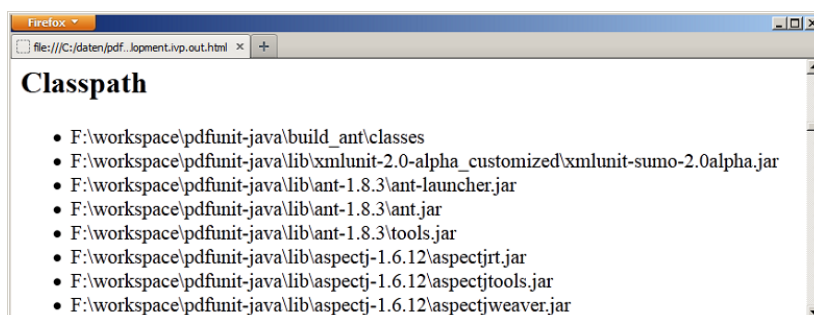
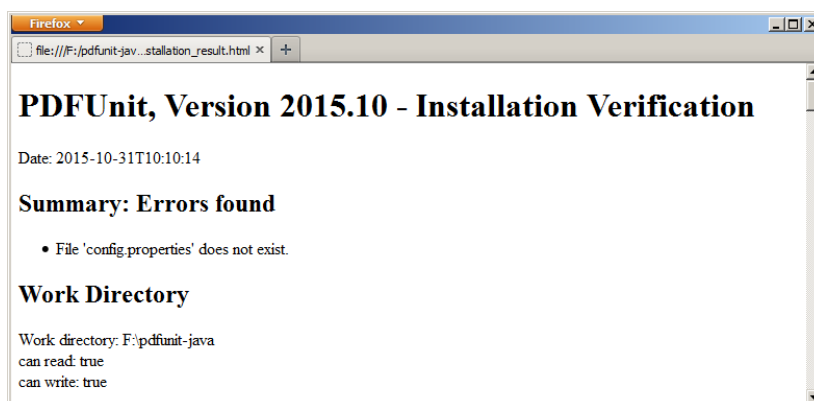
Die Stylesheet-Option kann entfallen. Sie dient vor allem dazu, die Verwendung eigener Stylesheets zu ermöglichen.

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:





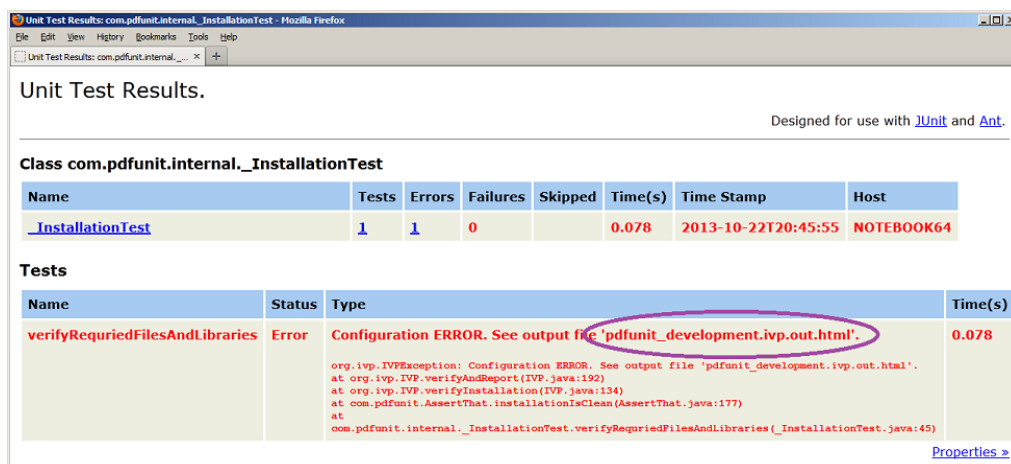
## Überprüfung als Unittest

Die Überprüfung der Installation kann auch als Unittest durchgeführt werden. Dadurch ist es möglich, die Systemumgebung der laufenden Tests im Kontext von ANT, Maven oder Jenkins sichtbar zu machen.

Innerhalb eines Unittests wird eine spezielle Testmethode aufgerufen:

```
/*
 * The method verifies that all required libraries and files are found on the
 * classpath. Additionally it logs some system properties and writes
 * all to System.out.
 */
@Test
public void verifyRequriedFilesAndLibraries() throws Exception {
    AssertThat.installationIsClean("pdfunit_development.vip");
}
```

Die Methode führt die gleichen Prüfungen aus, wie das zuvor beschriebene Skript. Falls ein Konfigurationsfehler vorliegt, wird der Test „rot“ und verweist in der Fehlermeldung auf die Report-Datei:



Die Report-Datei enthält dieselben Informationen (s.o.), als wäre sie über ein Skript erzeugt worden.

## 11.8. Installation eines neuen Releases

Die Installation eines neuen Releases von PDFUnit-Java verläuft genauso, wie die Erstinstallation, weil Releases immer vollständig zur Verfügung gestellt werden, nie als Differenz zum vorhergehenden Release.

### Beschaffung des neuen Releases

Wenn Sie PDFUnit ohne Lizenzdatei einsetzen, laden Sie sich die neue ZIP-Datei aus dem Internet: <http://www.pdfunit.com/de/download/index.html>.

Wenn Sie PDFUnit mit Lizenzdatei einsetzen, erhalten Sie das neue Release per Mail mit der ZIP-Datei als Anhang sowie einer separaten Datei mit den Lizenzdaten.

### Vorbereitende Schritte für alle Umgebungen

Bevor Sie mit dem Releasewechsel beginnen, führen Sie alle vorhandenen Unittests mit dem alten Release durch. Diese sollten „grün“ sein.

Sichern Sie Ihr Projekt.

### Durchführung des Updates

Entpacken Sie das neue Release, jedoch nicht in das bestehende Projekt. Nachfolgend wird der Ordner mit dem neuen Release `PDFUNITJAVA_HOME_NEW` genannt. Der Ordner 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_HOME_NEW` an die Stelle des alten Releases, beispielsweise nach `PROJECT_HOME/lib/pdfunit-NEW-VERSION`.

Sollten Sie im alten Release die Datei `config.properties` in einem anderem Verzeichnis, als dem Installationsverzeichnis verwendet haben, so kopieren Sie die neue Datei jetzt aus dem Verzeichnis `PROJECT_HOME/lib/pdfunit-NEW-VERSION` an Ihren gewünschten Ort.

Sollten Sie im alten Release Änderungen an der `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.

### Weitere Schritte für ANT

Für ANT sind keine weiteren Schritte notwendig, wenn Sie den Classpath so konfiguriert haben, wie weiter oben beschrieben.

### Weitere Schritte für Maven

Das neue Release muss in Ihr lokales oder unternehmensweites Repository eingetragen werden. Öffnen Sie dazu eine Konsole, wechseln in das Verzeichnis `PROJECT_HOME/lib/pdfunit-NEW-VERSION` und führen dort diesen Befehl aus:

```
mvn install:install-file -Dfile=pdfunit-java-VERSION.jar -DpomFile=pom.xml
```

## Weitere Schritte für Eclipse

Nehmen Sie die neuen JAR-Dateien in den Build-Path auf. Entfernen Sie die alten JAR-Dateien aus dem Build-Path, damit Eclipse keinen Build-Fehler mehr anzeigt.

Verknüpfen Sie die Javadoc-Dokumentation in Eclipse erneut so, wie in Kapitel : „Eclipse konfigurieren“ (S. 140) beschrieben.

## Letzter Schritt

Führen Sie Ihre bestehenden Tests mit dem neuen Release durch. Sofern es keine dokumentierten Inkompatibilitäten zwischen dem alten und neuen PDFUnit-Release gibt, sollten Ihre Tests erfolgreich durchlaufen. Andernfalls lesen Sie die Release-Informationen.

## 11.9. 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-Java Systeme

### 12.1. Kurzer Blick auf PDFUnit-XML

Tester müssen keine Java-Kenntnisse besitzen, um PDF-Dokumente automatisiert zu testen. Für eine auf XML basierende Systemlandschaft gibt es unter der Bezeichnung „PDFUnit-XML“ Laufzeitkomponenten, Skripte, XML Schema und Stylesheets zum Testen von PDF-Dokumenten. Die Funktionalität ist voll kompatibel zu „PDFUnit-Java“.

Die folgenden Beispiele geben einen Einblick in PDFUnit-XML:

```
<testcase name="hasTextOnSpecifiedPages_Containing">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onPage="1, 2, 3" >
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="hasTitle_MatchingRegex">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasTitle>
      <startingWith>PDFUnit sample</startingWith>
      <matchingRegex>.*Unit.*</matchingRegex>
    </hasTitle>
  </assertThat>
</testcase>
```

```
<testcase name="compareText_InClippingArea">
  <assertThat testDocument="master/test.pdf"
    masterDocument="master/master.pdf"
  >
    <haveSameText on="EVERY_PAGE" >
      <inClippingArea upperLeftX="50" upperLeftY="720" width="150" height="30" />
    </haveSameText>
  </assertThat>
</testcase>
```

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

Die Namen der Tags und Attribute stimmen überwiegend mit der Java-API überein und folgen ebenfalls der Idee des „Fluent Interfaces“ ([http://de.wikipedia.org/wiki/Fluent\\_Interface](http://de.wikipedia.org/wiki/Fluent_Interface)).

Die XML-Syntax ist mit passenden XML Schema Dateien abgesichert.

Eine genaue Beschreibung steht als eigenständige Dokumentation zur Verfügung.

### 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. Einsatz mit TestNG

PDFUnit läuft auch mit TestNG.

Wenn Sie lediglich die einfache Annotation `@Test` verwenden, ist ja sowieso kein Unterschied erkennbar. Erst wenn z.B. Exceptions erwartet werden, ist TestNG zu erkennen:

```
@Test(expectedExceptions=PDFUnitValidationException.class)
public void hasAuthor_NoAuthorInPDF() throws Exception {
    String filename = PATH + "documentInfo/documentInfo_allInfo.pdf";

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

### 13.2. Instantiierung der PDF-Dokumente

PDF-Dokumente können in unterschiedlichen Formaten eingelesen werden. Die folgende Liste zeigt alle Möglichkeiten auf:

```
// Possibilities to instantiate PDFUnit with a Test-PDF
AssertThat.document(String      pdfDocument)...
AssertThat.document(File       pdfDocument)...
AssertThat.document(URL        pdfDocument)...
AssertThat.document(InputStream pdfDocument)...
AssertThat.document(byte[]     pdfDocument)...

// The same with a password when the PDF is encrypted:
AssertThat.document(String      pdfDocument, String password)...
AssertThat.document(File       pdfDocument, String password)...
AssertThat.document(URL        pdfDocument, String password)...
AssertThat.document(InputStream pdfDocument, String password)...
AssertThat.document(byte[]     pdfDocument, String password)...
```

Auch ein Vergleichs-PDF als `String`, `File`, `URL`, `InputStream` oder `byte[]` eingelesen werden.

Wenn die PDF-Dokumente passwort-geschützt sind, benötigt PDFUnit entweder das „User-Password“ oder das „Owner-Password“ als zusätzlichen Parameter.

### 13.3. Seitenauswahl

#### Vordefinierte Seiten

Für Tests, die sich auf bestimmte Seiten eines PDF-Dokumentes beziehen, existieren in der Klasse `com.pdfunit.Constants` vorgefertigte Konstanten, deren Bedeutung sich aus ihrem Namen ergibt:

```
// Possibilities to focus tests to specific pages:

com.pdfunit.Constants.ON_ANY_PAGE
com.pdfunit.Constants.ON_EVEN_PAGES
com.pdfunit.Constants.ON_EACH_PAGE
com.pdfunit.Constants.ON_EVERY_PAGE
com.pdfunit.Constants.ON_FIRST_PAGE
com.pdfunit.Constants.ON_LAST_PAGE
com.pdfunit.Constants.ON_ODD_PAGES

com.pdfunit.Constants.FIRST_PAGE
com.pdfunit.Constants.LAST_PAGE
```

`ON_FIRST_PAGE` und `FIRST_PAGE` sind funktional identisch, ebenso `ON_LAST_PAGE` und `LAST_PAGE`. Die Redundanz ist aus sprachlichen Gründen beabsichtigt, damit die API besser fließt, je

nach dem, welche nachfolgenden Funktionen verwendet werden. Das gleiche gilt für `ON_EACH_PAGE` und `ON_EVERY_PAGE`.

Hier ein Beispiel mit einer vordefinierten Seiten-Konstanten:

```
@Test
public void hasText_MultipleSearchTokens_EvenPages() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
        .hasText(ON_EVEN_PAGES)
        .containing("Content", "even pagenumber")
    ;
}
```

## Individuelle Seiten

Das nächste Beispiel zeigt, wie beliebige, individuelle Seiten definiert werden können.

```
@Test
public void hasText_OnMultiplePages() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";
    PagesToUse ON_SELECTED_PAGES = PagesToUse.getPages(1, 2, 3);

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

Es stehen zwei Methoden zur Verfügung, um einzelne oder mehrere Seiten auszuwählen:

```
// How to define individual pages:

PagesToUse.getPage(2);
PagesToUse.getPages(1, 2, 3);
```

## Offene Seitenbereiche

Für Tests, die auf Bereiche am Ende oder am Anfang eines Dokumentes zielen, gibt es weitere Methoden. Das nachfolgende Beispiel prüft das Format ab Seite 2 bis zum Ende des Dokumentes.

```
// How to define open ranges:

OnAnyPage.after(2);
OnAnyPage.before(3);

OnEveryPage.after(2);
OnEveryPage.before(2);
```

```
@Test
public void compareFormat_OnEveryPageAfter() throws Exception {
    String filename = PATH + "master/compareToMaster.pdf";
    String filenameMaster = PATH + "master/compareToMaster.pdf";

    AssertThat.document(filename)
        .and(filenameMaster)
        .haveSameFormat(OnEveryPage.after(2))
    ;
}
```

## Seitenbereiche innerhalb eines Dokumentes

Und als Letztes gibt es die Syntax `PagesToUse.spanningFrom().to()`, um Tests auf einen Bereich innerhalb eines Dokumentes zu beschränken. Das folgende Beispiel validiert Text, der zwei Seiten überspannt.

```

@Test
public void hasText_SpanningOver2Pages_matchingRegex() throws Exception {
    String filename = PATH + "content/text-starts-on-page1-continues-on-page2.pdf";
    String textOnPage1 = "Text starts on page 1 and ";
    String textOnPage2 = "continues on page 2.";
    String anyText = ".*";
    String expectedText = textOnPage1 + anyText + textOnPage2;
    PagesToUse onPages1to2 = PagesToUse.spanningFrom(1).to(2);

    // Mark the section without header and footer:
    double upperLeftY = 30;
    double upperLeftX = 18;
    double height = 238;
    double width = 182;
    ClippingArea sectionWithoutHeaderAndFooter = new ClippingArea(upperLeftX,
                                                                    upperLeftY,
                                                                    width,
                                                                    height,
                                                                    FormatUnit.MILLIMETER);

    AssertThat.document(filename)
        .hasText(onPages1to2, sectionWithoutHeaderAndFooter)
        .matchingRegex(expectedText)
        ;
}

```

Für geschlossene Seitenbereiche stehen nur die Methoden `containing()`, `matchingRegex()`, `notContaining()` und `notMatchingRegex()` zur Verfügung.

## Wichtige Hinweise

- Seitenzahlen beginnen mit '1'.
- Die Seitenangaben in `before(int)` und `after(int)` sind jeweils exklusiv gemeint.
- Die Seitenangaben in `from(int)` und `to(int)` sind jeweils inklusiv gemeint.
- `OnEveryPage` bedeutet, dass der gesuchte Text wirklich auf jeder Seite existieren muss.
- Dagegen reicht es, wenn bei `OnAnyPage` der gesuchte Text auf einer Seite existiert.

## 13.4. Textvergleich

Ein erwarteter Text und der tatsächliche Text einer PDF-Seite können auf folgende Art miteinander verglichen werden:

```

// Comparing text:
.containing(String[] searchTokens)           ❶
.containing(searchToken, WhitespaceProcessing) ❷
.endsWith(expectedText)
.matchingComplete(expectedText)              ❸
.matchingComplete(expectedText, WhitespaceProcessing) ❹
.matchingRegex(regex)
.startingWith(expectedText)

.notContaining(String[] searchTokens)         ❺
.notContaining(searchToken, WhitespaceProcessing) ❻
.notEndingWith(expectedText)
.notMatchingRegex(regex)
.notStartingWith(expectedText)

```

- ❶❸❺** Bei diesen Methoden werden die Whitespaces „normalisiert“. Das heißt, Leerzeichen am Anfang und Ende werden entfernt und alle Whitespaces innerhalb eines Textes werden auf ein Leerzeichen reduziert.
- ❷❹❻** Die Behandlung von Whitespaces wird über den zweiten Parameter gesteuert, für den die Konstanten `KEEP_WHITESPACES`, `NORMALIZE_WHITESPACES` und `IGNORE_WHITESPACES` existieren. Diese Konstanten sind in Kapitel 13.5: „Behandlung von Whitespaces“ (S. 153) separat beschrieben.

Vergleiche mit Regulären Ausdrücken folgen den Regeln und Möglichkeiten der Klasse `java.util.regex.Pattern`:



```
// Using regular expression to compare page content
@Test
public void hasText_MatchingRegex() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .matchingRegex(".*[Cc]ontent.*")
    ;
}
```

Die Methoden `containing(String[])` und `notContaining(String[])` können mit mehreren Suchbegriffen aufgerufen werden. Ein Test mit `containing(String[])` gilt als erfolgreich, wenn jeder Suchbegriff auf jeder ausgewählten Seiten auftaucht. Ein Test mit `notContaining(String[])` ist erfolgreich, wenn alle Suchbegriffe auf jeder ausgewählten Seite nicht vorhanden sind:

```
@Test
public void hasText_NotContaining_MultipleSearchTokens() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

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

## 13.5. Behandlung von Whitespaces

In fast allen Tests werden Texte verglichen. Viele Vergleiche würden nicht funktionieren, wenn die Whitespaces eines Textes „so, wie sie sind“ Teil des Vergleiches wären. Deshalb gibt es für Tests, bei denen eine flexible Behandlung von Whitespaces sinnvoll ist, die Möglichkeit einer benutzergesteuerten Whitespace-Behandlung. PDFUnit stellt drei Konstanten zur Verfügung:

```
// Constants for whitespace processing:

com.pdfunit.Constants.IGNORE_WHITESPACES           ❶
com.pdfunit.Constants.KEEP_WHITESPACES             ❷
com.pdfunit.Constants.NORMALIZE_WHITESPACES // default 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 den folgenden Methoden kann die Whitespace-Behandlung mitgegeben werden:

```
// defining whitespace processing:
.hasXXXAction().containing(.., WhitespaceProcessing)
.hasXXXAction().matchingComplete(.., WhitespaceProcessing)
.hasText(..).containing(.., WhitespaceProcessing)
.hasText(..).matchingComplete(.., WhitespaceProcessing)
.hasText(..).notContaining(.., WhitespaceProcessing)
```

Ein Beispiel:

```

@Test
public void hasText_WithLineBreaks_UsingIGNORE() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

    String expected = "PDFUnit - Automated PDF Tests http://pdfunit.com/" +
        "This is a document that is used for unit tests of PDFUnit itself." +
        "Content on first page." +
        "odd pagenumber" +
        "Page # 1 of 4";

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .matchingComplete(expected, IGNORE_WHITESPACES)
    ;
}

```

In diesem Beispiel wird die erwartete Zeichenkette vollständig ohne Zeilenumbrüche formuliert, obwohl die PDF-Seite mehrere davon enthält. Durch die Angabe `IGNORE_WHITESPACES` funktioniert der Test trotzdem bestens.

`NORMALIZE_WHITESPACES` ist die Standardbehandlung, falls nichts anderes angegeben wird. Testmethoden, bei denen eine flexible Behandlung von Whitespaces nicht sinnvoll ist, bieten keine Möglichkeit für eine benutzergesteuerte Whitespace-Behandlung. PDFUnit arbeitet intern dennoch mit `NORMALIZE_WHITESPACES`. Beispiele hierfür sind:

```

// Examples for default whitespace processing,
// The list is not complete:

.hasText(..).startsWith(..)
.hasText(..).endsWith(..)
.hasText(..).matchingComplete(..)
.hasBookmark().withLinkToName(..)
.hasAuthor().containing(..)

```

Testmethoden, die Reguläre Ausdrücke verarbeiten, verändern Whitespaces nicht. Bei Bedarf muss die Behandlung der Whitespaces in den regulären Ausdruck integriert werden, beispielsweise so:

```
(?ms).*print(.*)
```

Der Teilausdruck `(?ms)` bedeutet, dass die Suche über mehrere Zeilen reicht. Zeilenumbrüche werden als 'Character' interpretiert.

## 13.6. Anführungszeichen in Suchbegriffen

### Unterschiedliche Arten von Anführungszeichen

**Wichtiger Hinweis:** Der Begriff „Anführungszeichen“ wird in Texten unterschiedlich umgesetzt, wie das folgende Bild zeigt:

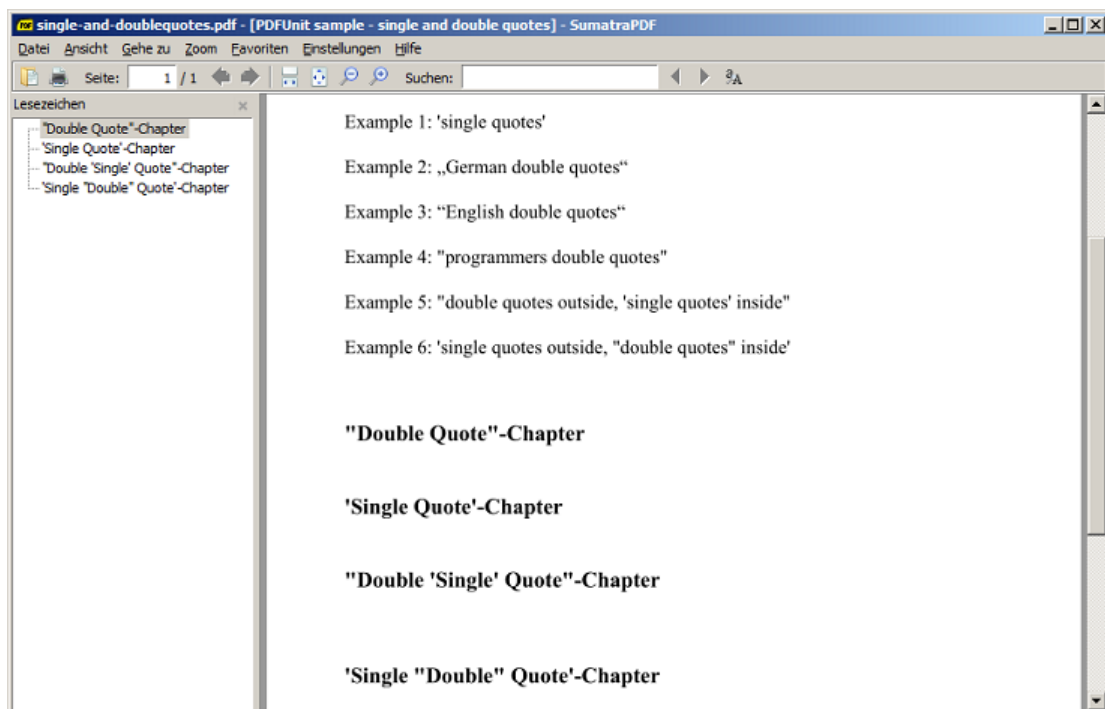
```

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 Ihre Datei ein.

Die "programmers double quotes" benötigen eine besondere Aufmerksamkeit, weil sie in Java als Zeichenkettenbegrenzer dienen. Die nachfolgenden Absätze und Beispiele gehen detailliert darauf ein, sie basieren alle auf dem folgenden Dokument:



## Gültige Beispiele ohne XPath

'Single-Quotes', "englische" und „deutsche“ Anführungszeichen innerhalb von Zeichenketten bereiten alle keine Probleme, lediglich "Double-Quotes" müssen mit einem Backslash maskiert werden:

```
@Test
public void hasText_SingleQuotes() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";

    String expected = "Example 1: 'single quotes'";
    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .containing(expected)
    ;
}
```

```
@Test
public void hasText_GermanDoubleQuotes() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";

    String expected = "Example 2: „German double quotes“";
    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .containing(expected)
    ;
}
```

```
@Test
public void hasText_EnglishDoubleQuotes() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";

    String expected = "Example 3: \"English double quotes\"";
    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .containing(expected)
    ;
}
```

```
@Test
public void hasText_ProgrammersDoubleQuotes() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";

    String expected = "Example 4: \"programmers double quotes\"";
    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .containing(expected)
    ;
}
```

```
@Test
public void hasText_DoubleAndSingleQuotes_1() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";

    String expected = "Example 5: \"double quotes outside, 'single quotes' inside\"";
    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .containing(expected)
    ;
}
```

```
@Test
public void hasText_DoubleAndSingleQuotes_2() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";
    String expected = "Example 6: 'single quotes outside, \"double quotes\" inside'";

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .containing(expected)
    ;
}
```

```
@Test
public void matchingRegex_DoubleQuotes() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .matchingRegex(".*\"double.*\".*")
    ;
}
```

## Gültige Beispiele mit XPath

Zeichenketten, die im Laufe der Verarbeitung mit XPath weiterverarbeitet werden, dürfen **nicht gleichzeitig** Single- und Double-Quotes enthalten. Diese Bedingung ist in den folgenden Beispielen erfüllt:

```
@Test
public void hasBookmarkWithLabel_DoubleQuotes() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";
    String expectedLabel = "\"Double Quote\"-Chapter";

    AssertThat.document(filename)
        .hasBookmark()
        .withLabel(expectedLabel)
    ;
}
```

```
@Test
public void hasBookmarkWithLabel_SingleQuotes() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";
    String expectedLabel = "'Single Quote'-Chapter";

    AssertThat.document(filename)
        .hasBookmark()
        .withLabel(expectedLabel)
    ;
}
```

## Ungültige Beispiele mit XPath

Die nächsten beiden Beispiele enthalten sowohl Single- als auch Double-Quotes. Deshalb erscheint zur Laufzeit diese Fehlermeldung: "One of the parameters contains single and double quote. You may use only one kind of quote."

```
@Test
public void hasBookmarkWithLabel_SingleDoubleQuotes() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";
    String expectedLabel = "'Single \"Double\" Quote'-Chapter";

    AssertThat.document(filename)
        .hasBookmark()
        .withLabel(expectedLabel)
    ;
}
```

```
@Test
public void matchingXPath_DoubleQuotes_1() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";
    String xpath = "count(//Title[.='\"Double Quote\"'-Chapter']) = 1";
    XPathExpression xpathExpression = new XPathExpression(xpath);

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

Der Fehler lässt sich nur vermeiden, wenn die XPath-Bedingung anders formuliert wird:

```
@Test
public void matchingXPath_DoubleQuotes_2() throws Exception {
    String filename = PATH + "quotes/single-and-doublequotes.pdf";
    String xpath1 = "count(//Title[contains(., 'Double Quote')]) = 1";
    String xpath2 = "count(//Title[contains(., 'Chapter')]) = 4";
    XPathExpression xpathExpression1 = new XPathExpression(xpath1);
    XPathExpression xpathExpression2 = new XPathExpression(xpath2);

    AssertThat.document(filename)
        .hasBookmarks()
        .matchingXPath(xpathExpression1)
        .matchingXPath(xpathExpression2)
    ;
}
```

## 13.7. 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:

```
// Instantiating a clipping area
public ClippingArea(double upperLeftX, double upperLeftY, double width, double height)
public ClippingArea(double ulX, double ulY, double width, double height, FormatUnit init)
```

Die verfügbaren Einheiten werden in Kapitel, 13.8: „Maßeinheiten - Points, Millimeter, ...“ (S. 158) beschrieben. Wenn keine Einheit mitgegeben wird, gilt die Einheit MILLIMETER.

Hier ein Beispiel:

```
@Test
public void hasText_InClippingArea() throws Exception {
    String filename = PATH + "content/documentForTextClipping.pdf";

    double ulX    = 17.6; // in millimeter
    double ulY    = 45.8;
    double width  = 60.0;
    double height = 8.8;

    ClippingArea inClippingArea = new ClippingArea(ulX, ulY, width, height, MILLIMETER);

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE, inClippingArea)
        .startingWith("Content")
        .containing("on first")
        .endingWith("page.")
    ;
}
```

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.

## 13.8. Maßeinheiten - Points, Millimeter, ...

Manche Tests benötigen Werte für Länge und Breite. Damit ein Programmierer seine gewohnten Einheiten benutzen kann, stehen in der Klasse `com.pdfunit.Constants` folgende Konstanten zur Verfügung:

```

// Predefined format units:

com.pdfunit.Constants.CENTIMETER
com.pdfunit.Constants.DPI72 // same as POINTS
com.pdfunit.Constants.MILLIMETER
com.pdfunit.Constants.POINTS // same as DPI72
com.pdfunit.Constants.INCH

```

Wenn die Einheit nicht angegeben wird, gelten `MILLIMETER` als Default.

### Beispiel - Größe von Formularfeldern

```

@Test
public void hasField_WidthAndHeight() throws Exception {
    String filename = PATH + "acrofields/notExportableAcrofield.pdf";
    String fieldname = "Title of 'someField'";

    AssertThat.document(filename)
        .hasField(fieldname)
        .withWidth(450, POINTS)
        .withHeight(30, POINTS)
    ;
}

```

## Beispiel - Seitenausschnitt

```
@Test
public void hasTextOnFirstPage_RectangleInInch() throws Exception {
    String filename = PATH + "content/documentForTextClipping.pdf";

    double upperLeftX = 0.7; // in inch
    double upperLeftY = 1.8;
    double width = 2.4;
    double height = 0.4;

    ClippingArea inClippingArea
        = new ClippingArea(upperLeftX, upperLeftY, width, height, INCH);

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE, inClippingArea)
        .startingWith("Content")
        .containing("on first")
        .endingWith("page.")
    ;
}
```

## Beispiel - Größe des Seitenformats

```
@Test
public void hasHugeFormat() throws Exception {
    String filename = PATH + "format/physical-map-of-the-world-1999_1117x863mm.pdf";
    double width = 2448;
    double height = 3168;
    DocumentFormat formatPoints = new DocumentFormat(width, height, POINTS);

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

## 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')'.
```

## 13.9. 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.
```

## 13.10. Datumsauflösung

Der Vergleich eines Datums (Erstellungs- oder Änderungsdatum) mit einem Erwartungswert kann sich entweder nur auf Jahr-Monat-Tag beziehen oder zusätzlich noch auf Stunde-Minute-Sekunde. Für die Unterscheidung dieser beiden Möglichkeiten stellt PDFUnit zwei Konstanten zur Verfügung:

```
// Constants for date resolutions:

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

In den folgenden Methoden werden diese Konstanten verwendet:

```
// Date resolution in test methods:

.hasCreationDate().after(..., DateResolution)
.hasCreationDate().before(..., DateResolution)
.hasCreationDate().matchingComplete(..., DateResolution)
.hasModificationDate().after(..., DateResolution)
.hasModificationDate().before(..., DateResolution)
.hasModificationDate().matchingComplete(..., DateResolution)
.hasSignature(...).withSigningDate(..., DateResolution)

// Internal used resolution DATE:
.hasSignature(...).validFor()
.hasSignature(...).validFrom()
.hasSignature(...).validUntil()

// Comparing two PDF documents, using DATE:
.haveSameCreationDate()
.haveSameModificationDate()
```

Der Vergleich von Datumswerten zweier PDF-Dokumenten findet immer in der Auflösung `DateResolution.DATE` statt.

## 13.11. 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 die Klasse `com.pdfunit.DefaultNamespace` vorgegeben werden:

```
/**
 * The default namespace has to be declared,
 * but any alias can be used for it.
 */
@Test
public void hasXFADData_UsingDefaultNamespace() throws Exception {
    String filename = PATH + "xfa/xfa-enabled.pdf";

    DefaultNamespace ns = new DefaultNamespace("http://www.xfa.org/schema/xci/2.6/");
    XMLNode aliasFoo = new XMLNode("foo:log/foo:to", "memory", ns);

    AssertThat.document(filename)
        .hasXFADData()
        .withNode(aliasFoo)
        ;
}
```

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 eine XPathExpression:

```
@Test
public void hasXMPData_MatchingXPath_WithDefaultNamespace() throws Exception {
    String filename = PATH + "xmp/metadata-added.pdf";

    String xpathAsString = "//foo:format = 'application/pdf'";
    String stringDefaultNS = "http://purl.org/dc/elements/1.1/";
    DefaultNamespace defaultNS = new DefaultNamespace(stringDefaultNS);
    XPathExpression expression = new XPathExpression(xpathAsString, defaultNS);

    AssertThat.document(filename)
        .hasXMPData()
        .matchingXPath(expression)
    ;
}
```

## 13.12. Konfiguration überprüfen

Wie schon in Kapitel 11.7: „Überprüfung der Konfiguration“ (S. 144) beschrieben, gibt es eine Testmethode, die feststellt, ob alle benötigten Bibliotheken und Dateien im Classpath vorhanden sind. Sie kann als JUnit-Test ausgeführt werden:

```
/*
 * The method verifies that all required libraries and files are found on the
 * classpath. Additionally it logs some system properties and writes
 * everything into both an XML file and an HTML formatted file.
 */
@Test
public void verifyRequiredFilesAndLibraries() throws Exception {
    AssertThat.installationIsClean("pdfunit_development.vip");
}
```

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 com.pdfunit.internal.\_InstallationTest

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
<a href="#">InstallationTest</a>	1	1	0		0.078	2013-10-22T20:45:55	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
verifyRequiredFilesAndLibraries	Error	Configuration ERROR. See output file 'pdfunit_development.ivp.out.html'.	0.078

org.ivp.IVPException: Configuration ERROR. See output file 'pdfunit\_development.ivp.out.html'.  
 at org.ivp.IVP.verifyAndReport(IVP.java:192)  
 at org.ivp.IVP.verifyInstallation(IVP.java:134)  
 at com.pdfunit.AssertThat.installationIsClean(AssertThat.java:177)  
 at com.pdfunit.internal.\_InstallationTest.verifyRequiredFilesAndLibraries(\_InstallationTest.java:45)

[Properties »](#)

## 13.13. JAXP-Konfiguration

Die Standard-JAXP-Konfiguration des JDK kann über die allgemeinen Mechanismen der JAXP-Konfiguration verändert werden. Weil die aber nicht allgemein bekannt sind, werden sie am Beispiel von Xerces und Xalan nachfolgend erläutert.

Die Java-Runtime liest die Werte folgender JAXP-Umgebungsvariablen ein und lädt dann die dort angegebene Java-Klasse:

```
"javax.xml.parsers.DocumentBuilderFactory"
"javax.xml.parsers.SAXParserFactory"
"javax.xml.transform.TransformerFactory"
```

Diese Umgebungsvariablen können auf unterschiedliche Weise gesetzt werden, wie die nachfolgende Liste zeigt. Eine Konfigurationsmöglichkeit, die eine andere übersteuert, steht in der Aufzählung weiter oben.

1. Die JAXP Umgebungsvariablen können im Programm selber gesetzt werden:

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
    "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
System.setProperty("javax.xml.parsers.SAXParserFactory",
    "org.apache.xerces.jaxp.SAXParserFactoryImpl");
System.setProperty("javax.xml.transform.TransformerFactory",
    "org.apache.xalan.processor.TransformerFactoryImpl");
```

2. Die Konfigurationsdaten können mit der Startoption `-D` in der Umgebungsvariablen `_JAVA_OPTIONS` bereitgestellt werden. Hier wird nur ein Wert als Beispiel dargestellt, drei sind natürlich auch möglich:

```
set _JAVA_OPTIONS=-Djavax.xml.transform.TransformerFactory=
    org.apache.xalan.processor.TransformerFactoryImpl
```

3. Die Konfigurationsdaten können mit der Startoption `-D` in der Umgebungsvariablen `JAVA_TOOL_OPTIONS` bereitgestellt werden. Diese Umgebungsvariable wird von einigen JDK-Implementierungen ausgewertet. Auch hier wird nur ein Wert als Beispiel:

```
set JAVA_TOOL_OPTIONS=-Djavax.xml.transform.TransformerFactory=
    org.apache.xalan.processor.TransformerFactoryImpl
```

4. Die Konfigurationsdaten können in der Datei `jaxp.properties` im Verzeichnis `JAVA_HOME/jre/lib` zur Verfügung gestellt werden:

```
#
# Sample configuration, file %JAVA_HOME%\jre\lib\jaxp.properties.
#
# Defaults in Java 1.7.0, Windows:
#
#javax.xml.parsers.DocumentBuilderFactory = \
#    com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl
#javax.xml.parsers.SAXParserFactory      = \
#    com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl
#javax.xml.transform.TransformerFactory   = \
#    com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl
#
# Values for Xerces and Xalan:
#
#javax.xml.parsers.DocumentBuilderFactory = \
#    org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
#javax.xml.parsers.SAXParserFactory      = \
#    org.apache.xerces.jaxp.SAXParserFactoryImpl
#javax.xml.transform.TransformerFactory   = \
#    org.apache.xalan.processor.TransformerFactoryImpl
```

5. Die JAXP-Umgebungsvariablen können ANT über die Umgebungsvariable `ANT_OPTS` gesetzt werden:

```
set ANT_OPTS=-Djavax.xml.transform.TransformerFactory=
    org.apache.xalan.processor.TransformerFactoryImpl
set ANT_OPTS=-Djavax.xml.parsers.DocumentBuilderFactory=
    org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
set ANT_OPTS=-Djavax.xml.parsers.SAXParserFactory=
    org.apache.xerces.jaxp.SAXParserFactoryImpl
```

Die deklarierten XML/XSLT-Klassen müssen entweder im Classpath oder im Verzeichnis `JAVA_HOME/jre/lib/ext` liegen. Letzteres ist nicht zu bevorzugen, weil kaum jemand diese Möglichkeit kennt und eine eventuelle Fehlersuche wegen dieses Nicht-Wissens unnötig lange dauern würde.

Beachten Sie, dass Eclipse seine Umgebungsvariablen beim Start einliest und anschließend nicht verändert. Insofern müssen Sie Eclipse nach der Änderung einer Umgebungsvariablen neu starten.

## 13.14. Versionshistorie

### Wie alles anfang...

Im Rahmen eines kundenspezifisches Seminares im August 2010 kam die Sprache auf das Thema des „Testens von PDF-Dokumenten“. Damals habe ich jPdfUnit 1.1 (<http://jpdfunit.sourceforge.net>) vorgestellt, aber die Kundenwünsche konnten durch diese Version nicht abgedeckt werden. Also schickte ich nach dem Seminar eine Wunschliste für neue Testfunktionen an die Projektverantwortlichen und kurze Zeit später befand ich mich selber in der Rolle eines OpenSource-Entwicklers. Nach einer Analyse der vorhandenen Sourcen (Dezember 2010) war schnell klar, dass das intern verwendete Apache Projekt PDFBox (<http://pdfbox.apache.org>) für die geplanten neuen Funktionen nicht ausreichen würde, wohl aber iText (<http://itextpdf.com>).

### 2011

Im Laufe des Jahres 2011 sammelten sich Ideen, mein Wissen über PDF und iText wuchs (notwendigerweise) und sofern mir mein beruflicher Alltag Zeit ließ, brachte ich das Projekt voran.

Ende des Jahres 2011 entschloss ich mich nach reiflicher Überlegung und dem Studium von Lizenzbestimmungen, eine vollständige Neuentwicklung auf der Basis von iText zu starten. Das Projekt begleitete meine Schulungen. Jetzt konnte ich endlich mal gute Konzepte, Prinzipien und Tools an einem größeren Projekt darstellen, als an einem üblichen „Hello-World-Projekt“.

### Release 2012.07

Mit der Zeit entwickelte sich daraus das Produkt, jedoch stoppte die Entwicklung in der zweiten Jahreshälfte, weil die beruflichen Verpflichtungen keine Zeit für eine Weiterentwicklung ließen.

### Release 2013.01

Zu Beginn des Jahres 2013 gab es weniger Schulungsaufträge. In dieser Zeit entstand die Dokumentation und bestehende Funktionen wurden abgerundet, fehlerbereinigt und ergänzt.

### Release 2014.06

Zahlreiche kleine Verbesserungen rechtfertigen eine neue Version von PDFUnit-Java.

### 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.

## 13.15. 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 2015.10 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. 47) und 3.16: „Layout - gerenderte Seitenausschnitte“ (S. 48) 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.

# Stichwortverzeichnis

## A

- Aktionen, 12
  - beliebige Aktionen, 17
  - benannte Aktionen, 15
  - Close, 13
  - Gleichheit von Aktionen, 83
  - Goto, 15
  - Goto Remote, 15
  - Import Data, 14
  - JavaScript, 14
  - Launch Data, 14
  - Open, 15
  - Print, 16
  - Reset Form, 16
  - Save, 16
  - Submit Form, 17
  - URI, 17
  - vergleichen, 83
  - Whitespaces-Behandlung, 18
- Änderungsdatum, 26
- Anführungszeichen in Suchbegriffen, 154
- Anhang, 18
  - extrahieren, 114
- Anhänge, 18
- Anhänge vergleichen, 85
- ANT konfigurieren, 141
- Anzahl von PDF-Bestandteilen, 21
- Attachments, 18

## B

- Beispiele, 130
  - Caching von Testdokumenten, 135
  - HTML2PDF validieren, 133
  - Name des alten Vorstandes, 132
  - Neues Logo auf jeder Seite, 130
  - Passt Text in Formularfelder, 130
  - Schachtelungstiefe von Bookmarks, 136
  - Selenium und PDFUnit in Kombination, 132
  - Unterschrift des neuen Vorstandes, 131
- Benutzer-Passwort (user password), 53
- Berechtigungen, 22
  - vergleichen, 85
- Bilder, 23
  - Anzahl sichtbarer Bilder, 23
  - Anzahl unterschiedlicher Bilder, 23
  - aus PDF extrahieren, 116
  - mit Datei vergleichen, 24
  - N-zu-1 Vergleich, 24
  - seitenbezogen testen, 25
  - vergleichen, 86
- Bookmarks, 49

## C

- Classpath, 139
  - in ANT, 141
  - in Eclipse, 140
  - in Maven, 141
- Classpath konfigurieren, 138

## D

- Datum
  - Änderungsdatum, 26
  - Erstellung eines Zertifikates, 28
  - Erstellungsdatum, 26
  - Existenz, 26
  - Ober- und Untergrenze, 27
- Datumsauflösung, 26, 160
- Datumsformat
  - Konfiguration, 27
- Datum vergleichen
  - Änderungsdatum, 87
  - Erstellungsdatum, 87
- Default-Namensraum, 77, 80, 98, 113, 160
- Deinstallation, 147
- Diff-Image, 92
- Dokumenteneigenschaften, 28
  - als Key-Value-Paar testen, 30
  - Custom-Property, 31
  - vergleichen, 88
  - Vergleichsmöglichkeiten, 29

## E

- Eclipse konfigurieren, 140
- Eigentümer-Passwort (owner password), 53
- Eingebettete Dateien
  - Anzahl, 19
  - Dateiname, 19
  - Existenz, 19
  - Inhalt, 19
  - vergleichen, 85
- Erstellungsdatum, 26
- Erste Seite, 150
- Evaluationsversion, 138

## F

- Fast Web View, 31
- Feedback, 8
- Fehler
  - Erwartete Exception, 10
- Fehlerbild, 92
- Fehlermeldungen, 159
- Feldeigenschaften
  - nach XML extrahieren, 117
- Fluent Builder, 6
- Format, 32

- individuelle Größe, 32
- Maßeinheiten, 158
- mehrere Formate in einem Dokument, 33
- vergleichen, 89

**Formularfeld, 33**

- Anzahl, 35
- Eigenschaften, 38
- Existenz, 34
- Größe, 37
- hidden-Attribut, 164
- Inhalt, 35
- JavaScript-Aktionen, 38
- mit XML-Datei vergleichen, 40
- mit XPath testen, 40
- Name, 35
- Textüberlauf, 41, 163
- Typ, 36
- Unicode, 39
- vergleichen, 89

**Formularfelder vergleichen**

- Anzahl, 89
- Feldeigenschaften, 90
- Feldnamen, 90
- Inhalte, 90

**G**

Gerade Seiten, 150

**Gleichheit**

- von Aktionen, 83
- von Bildern, 87
- von Dokumenteneigenschaften, 88
- von Lesezeichen, 93
- von Schriften, 54, 95

Gültigkeitsdatum, 60

**H****Hilfsprogramme, 114**

- Anhänge extrahieren, 114
- Bilder aus PDF extrahieren, 116
- Feldeigenschaften nach XML extrahieren, 117
- JavaScript extrahieren, 118
- Lesezeichen nach XML extrahieren, 119
- Named Destinations nach XML extrahieren, 126
- PDF in PNG rendern, 120
- PDF-Seitenausschnitte in PNG rendern, 121
- Schrifteigenschaften nach XML extrahieren, 123
- Signaturdaten nach XML extrahieren, 125
- Unicode in Hex-Code wandeln, 127
- XFA-Daten nach XML extrahieren, 128
- XMP-Daten extrahieren, 128

**I****Installation, 137**

- bei existierendem iText, 139
- Classpath konfigurieren, 138
- iText, 137

- Lizenzschlüssel, 138
- Lizenzschlüssel beantragen, 138
- mit unterschiedlicher iText-Version, 139
- neues Release, 146
- ohne vorhandenes iText, 137
- PDFUnit-Java, 137

Installation überprüfen, 138

Instantiierung, 150

iText Installation, 137

**J****JavaScript, 43**

- Existenz, 43
- extrahieren, 118
- Teilstrings vergleichen, 44
- vergleichen, 91
- Vergleich gegen eine Textdatei, 43

Jede Seite, 150

**K****Konfiguration**

- \_JAVA\_OPTIONS, 162
- ANT\_OPTS, 162
- Ausgabeverzeichnis für Fehlerbilder, 143
- Interne Länderkennung, 143
- JAVA\_TOOL\_OPTIONS, 162
- JAXP, 161
- jaxp.properties, 162
- mit Skript prüfen, 144, 161
- mit Test prüfen, 145
- PDF-Datumsformat, 143
- Überprüfung, 144

**L**

Language, 63

**Layer, 44**

- Anzahl, 45
- Doppelte Namen, 46
- Name, 45

**Layout**

- Seitenausschnitt, 48
- vergleichen, 91
- volle Seiten, 47

Leerzeichen im Text, 18, 66, 153

**Lesezeichen, 49**

- Anzahl, 51
- Existenz, 50
- mit XML-Datei vergleichen, 52
- mit XPath testen, 52
- nach XML extrahieren, 119
- ohne Sprungziel, 52
- Sprungziel (Name einer Sprungmarke), 51
- Sprungziel (Seitenzahl), 51
- Sprungziel (URI), 51
- Sprungziele, 51
- Text (Label), 51

- vergleichen, 93
- Letzte Seite, 150
- Lizenzschlüssel
  - beantragen, 138
  - Classpath, 142
  - installieren, 138

## M

- Maßeinheiten, 158
  - DPI72, 158
  - Inch, 158
  - Millimeter, 158
  - Points, 158
  - Zentimeter, 158
- Maven konfigurieren, 141
- Mehrere Dokumente, 101
- Metadaten (Siehe 'Dokumenteneigenschaften')

## N

- Named Destination, 49
  - vergleichen, 94

## O

- Owner Password, 53

## P

- Passwort testen, 53
- PDF-Bestandteile vergleichen, 94
- PdfDIFF, 105
- PDFUnit-Monitor, 103
  - Export, 106
  - Fehlerdetails, 104
  - Filter, 104
  - Import, 106
  - Vergleich gegen Vorlage, 105
- PDFUnit-NET, 149
- PDFUnit-Perl, 148
- PDFUnit-XML, 148
- PDF-Version, 74
  - Versionsbereiche, 74
  - zukünftige Versionen, 74

## Q

- Quickstart, 9

## R

- Rechteck definieren, 157
- Reguläre Ausdrücke, 152

## S

- Schreibrichtung (right-to-left), 163
- Schrifteigenschaften
  - nach XML extrahieren, 123
- Schriften, 54
  - Anzahl, 54

- mit XML testen, 56
- mit XPath testen, 57
- Namen, 55
- Typen, 56
- vergleichen, 95
- Vergleichskriterien, 54

- Schrifttypen, 56

- Seiten

- gerendert vergleichen, 91
  - in PNG rendern, 120

- Seitenangaben mit Unter- und Obergrenze, 68

- Seitenausschnitt

- Beispiel, 70
  - definieren, 157
  - in PNG rendern, 121
  - Layout, 48
  - Layout validieren, 48
  - mit Einheit, 159
  - Text validieren, 67

- Seitenauswahl, 150

- geschlossener Bereich, 151
  - individuelle Seiten, 151
  - offener Bereich, 151

- Seitenzahlen als Testziel, 57

- Signatur/Zertifikat, 58

- Anzahl, 59
  - Existenz, 59
  - Grund (Reason), 60
  - Gültigkeitsdatum, 60
  - mit XML-Datei vergleichen, 61
  - mit XPath testen, 61
  - nach XML extrahieren, 125
  - Name, 59
  - Namen vergleichen, 95
  - Revision, 60
  - Unterzeichner (Sign Name), 60

- Sprachinformation (Language), 63

- Sprungziel (Named Destination), 50
  - nach XML extrahieren, 126

- Syntaktischer Einstieg, 10

- Systemumgebungsvariablen, 142

## T

- Tagging, 71
- Technische Voraussetzungen, 137
- TestNG, 150
- Texte in Seitenausschnitten, 70
- Texte - senkrecht, schräg, überkopf, 70
- Texte validieren, 64
  - Abwesenheit von Text, 65
  - auf allen Seiten, 65
  - auf bestimmten Seiten, 64
  - in Seitenausschnitten, 67
  - leere Seiten, 67
  - mehrfache Suchbegriffe, 67
  - Seitenangaben mit Unter- und Obergrenze, 68



- Zeilenumbruch, Leerzeichen, 66
- Textüberlauf, 41
  - aller Felder, 42
  - eines Felder, 41
  - technische Randbedingung, 42
- Textvergleich, 96, 152
  - in Seitenausschnitten, 96
  - Leerzeichen, 97
- Trapping, 72

## U

- Überblick
  - Hilfsprogramme, 114
  - Testbereiche, 11
  - Vergleiche gegen ein Master-PDF, 82
- Umgebungsvariablen, 142
- Ungerade Seiten, 150
- Unicode, 107
  - einzelne Zeichen, 107
  - in Fehlermeldungen, 110
  - in Hex-Code wandeln, 127
  - längere Texte, 107
  - mit XML-Datei vergleichen, 108
  - mit XPath testen, 108
  - unsichtbare Zeichen, 111
  - UTF-8 (ANT), 109
  - UTF-8 (Eclipse), 109
  - UTF-8 (Konsole), 109
  - UTF-8 (Maven), 109
- Update, 146
- User Password, 53

## V

- Vergleiche gegen ein Master-PDF, 82
  - Aktionen, 83
  - Änderungsdatum, 87
  - Anhänge, 85
  - Anzahl verschiedener PDF-Bestandteile, 94
  - Berechtigungen, 85
  - Bilder, 86
  - Bilder auf bestimmten Seiten, 86, 87
  - Dokumenteneigenschaften, 88
  - Erstellungsdatum, 87
  - Fast WebView, 99
  - Fehlerbild, Diff-Image, 92
  - Formate, 89
  - Formularfelder, 89
  - gerenderte Seiten, 91
  - gerenderte Seitenausschnitte, 91
  - JavaScript, 91
  - Leerzeichen, 97
  - Lesezeichen, 93
  - Named Destinations, 94
  - Schriften, 95
  - Signaturnamen, 95
  - Tagging, 99

- Texte, 96
- Texte in Seitenausschnitten, 96
- XFA-Daten, 97
- XMP-Daten, 98
- Verschlüsselungslänge, 53

## W

- Whitespaces-Behandlung, 18, 66, 153
  - IGNORE, KEEP, NORMALIZE, 153, 153

## X

- XFA-Daten, 75
  - auf einzelne Knoten prüfen, 76
  - Default-Namensraum, 77
  - Existenz, 75
  - mit XML-Datei vergleichen, 75
  - mit XPath testen, 76
  - nach XML extrahieren, 128
  - vergleichen, 97

### XML

- Daten extrahieren, 112
- Default-Namensraum, 113
- Namensraum, 113

- XMLUnit, 112, 159

### XMP-Daten, 78

- auf einzelne Knoten prüfen, 79
- Default-Namensraum, 80
- Existenz, 78
- mit XML-Datei vergleichen, 78
- mit XPath testen, 79
- nach XML extrahieren, 128
- vergleichen, 98

### XPath

- allgemeine Erläuterungen, 112
- Ergebnistyp, 113
- Kompatibilität, 113
- XPath Ergebnistyp, 97, 99
  - Bookean, 98

## Z

- Zeilenumbruch im Text, 18, 66, 153
- Zertifikat (Siehe 'Signatur/Zertifikat')
- Zertifiziertes PDF, 81