
Automated PDF Tests with Java

PDFUnit-Java

Carsten Siedentop

Table of Contents

Preface	6
1. About this Documentation	7
2. Quickstart	9
3. Test Scopes	10
3.1. Overview	10
3.2. Actions	12
3.3. Attachments	18
3.4. Bookmarks and Named Destinations	21
3.5. Certified PDF	24
3.6. Dates	24
3.7. Document Properties	27
3.8. Fast Web View	30
3.9. Fonts	30
3.10. Form Fields	34
3.11. Form Fields - Text Overflow	42
3.12. Format	43
3.13. Images in PDF Documents	45
3.14. JavaScript	48
3.15. Language	49
3.16. Layers	50
3.17. Layout - Entire PDF Pages	52
3.18. Layout - in Clipping Areas	54
3.19. Number of PDF Elements	55
3.20. Page Numbers as Objectives	56
3.21. Passwords	57
3.22. Permissions	58
3.23. Signatures and Certificates	59
3.24. Tagged Documents	64
3.25. Text	65
3.26. Text - in Page Sections	70
3.27. Text - Rotated and Overhead	71
3.28. Trapping Info	71
3.29. Version Info	73
3.30. XFA Data	74
3.31. XMP Data	77
4. Comparing a Test PDF with a Master	81
4.1. Overview	81
4.2. Comparing Form Fields	82
4.3. Comparing Actions	83
4.4. Comparing Attachments	85
4.5. Comparing Bookmarks	86
4.6. Comparing Date Values	86
4.7. Comparing Document Properties	87
4.8. Comparing Fonts	87
4.9. Comparing Format	88
4.10. Comparing Images	89
4.11. Comparing JavaScript	90
4.12. Comparing Layout as Rendered Pages	90
4.13. Comparing Named Destinations	92
4.14. Comparing Permissions	92
4.15. Comparing Quantities of PDF Elements	93
4.16. Comparing Signature Names	94
4.17. Comparing Text	94
4.18. Comparing XFA Data	95

4.19. Comparing XMP Data	97
4.20. More Comparisons	98
5. Tests with Multiple Documents	99
6. PDFUnit-Monitor	101
7. Unicode	105
8. Using XPath	110
9. Utility Programs	112
9.1. Common Remarks for all Utilities	112
9.2. Convert Unicode Text into Hex Code	112
9.3. Extract Field Information to XML	113
9.4. Extract Attachments	114
9.5. Extract Bookmarks to XML	116
9.6. Extract Font Information to XML	117
9.7. Extract Images from PDF	118
9.8. Extract JavaScript to a Text File	119
9.9. Extract Named Destinations to XML	120
9.10. Extract Signature Information to XML	121
9.11. Extract XFA Data to XML	122
9.12. Extract XMP Data to XML	123
9.13. Render Page Sections to PNG	124
9.14. Render Pages to PNG	125
10. Experience from Practice	128
10.1. Does Content Fit in Predefined Form Fields?	128
10.2. New Logo on each Page	128
10.3. Authorized Signature of the new CEO	129
10.4. Name of the Former CEO	129
10.5. Selenium and PDFUnit in Combination	130
10.6. HTML2PDF - Does the Rendering Tool Work Correct?	131
10.7. Caching Test Documents	133
10.8. Nesting Depth of Bookmarks	134
11. Installation, Configuration, Update	135
11.1. Technical Requirements	135
11.2. Installation without an Existing iText	135
11.3. Installation in Addition to an Existing iText	137
11.4. Setting Classpath in Eclipse, ANT, Maven	137
11.5. Set Pathes Using System Properties	140
11.6. Using the config.properties File	140
11.7. Verifying the Configuration	142
11.8. Installation of a New Release	143
11.9. Uninstall	144
12. PDFUnit for Non-Java Systems	145
12.1. A quick Look at PDFUnit-XML	145
12.2. A quick Look at PDFUnit-Perl	145
12.3. A quick Look at PDFUnit-NET	146
13. Appendix	147
13.1. Running PDFUnit with TestNG	147
13.2. Instantiation of PDF Documents	147
13.3. Page Selection	147
13.4. Comparing Text	149
13.5. Whitespace Processing	150
13.6. Single and Double Quotation Marks inside Strings	151
13.7. Defining Page Areas	154
13.8. Format Units	155
13.9. Error Messages	156
13.10. Date Resolution	157
13.11. Using the Default-Namespace	157

13.12. Verify Configuration	158
13.13. JAXP-Configuration	158
13.14. Version History	159
13.15. Unimplemented Features, Known Bugs	160
Index	162

Preface

The Current Situation of Testing PDF in Projects

These days telephone bills, insurance policies, official notifications and many types of contract are delivered as PDF documents. They are the result of a process chain consisting of programs in various programming languages using numerous libraries. Depending on the complexity of the documents to be produced, programming is not easy. Errors can arise at each step.

- Does page 2 contain the expected text?
- Is the new logo visible on every document?
- Are all fonts really embedded as intended?
- Does the layout fulfill the requirements?
- Is the bar code correct?
- Is the PDF signed?

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

It was long overdue to develop an easy-to-use test system.

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

Intuitive API

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

The next example shows the simplicity of the API:

```
@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)
    ;
}
```

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

Time to Start

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

Chapter 1. About this Documentation

Who Should Read it

This documentation is written for software developers developing PDF documents, project managers and members of the quality assurance staff.

It is assumed that you have basic knowledge of Java. Experience with JUnit or TestNG and a basic understanding about test automation is helpful, but not required.

Code Examples

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

Javadoc

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

Other Programming Languages

PDFUnit is available not only for Java, but also for Perl and XML. An implementation in CSharp is still in progress. Separate documentation exists for each language.

If there are Problems

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

New Features Wanted?

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

Responsibility

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

Acknowledgement

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

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

Bruno Lowagie, the founder of iText, read this documentation and sent me critical remarks about some chapters. His deep knowledge of PDF was a great help for me. Thank you, Bruno.

Production of this Documentation

This documentation was created with DocBook-XML and both PDF and HTML are generated from one text source. It is well known that the layout can be improved in both formats, e.g. the pagebreaks in PDF format. And improving the layout is already on my to-do list, but there are other tasks with higher priority.

Feedback

Any kind of feedback is welcomed. Please write to [feedback\[at\]pdfunit.com](mailto:feedback[at]pdfunit.com).

Chapter 2. Quickstart

Quickstart

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

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

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

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

    String expectedGreeting = "Thank you for using our services.";
    AssertThat.document(filename)
        .hasText(ON_LAST_PAGE)
        .containing(expectedGreeting)
    ;
}

@Test
public void hasExpectedCharge_en() throws Exception {
    String filename = PATH + "quickstart/quickstartDemo_en.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.
    ;
}
```

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

Unit Test Results.

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

Class com.pdfunit.QuickstartTests_en

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
QuickstartTests_en	3	0	1		0.061	2013-10-14T18:39:26	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
hasGreeting_en	Success		0.035
hasExpectedCharge_en	Failure	Page 1 of 'C:\daten\p...df\used-for-tests\quickstart\quickstartDemo_en.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_en.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_en.hasExpectedCharge_en(QuickstartTests_en.java:65)	0.020
hasOnePage_en	Success		0.002

[Properties »](#)

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

Chapter 3. Test Scopes

3.1. Overview

An Introduction to the Syntax

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

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

AssertThat.document(filename) 13.2: "Instantiation of PDF Documents" (p. 147)
    .hasText(..)              13.3: "Page Selection" (p. 147)
    .containing(..)           13.4: "Comparing Text" (p. 149)

    .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: "Overview" (p. 81)
    ...
;

AssertThat.document(filename)
    .asRenderedPage(..)        3.17: "Layout - Entire PDF Pages" (p. 52)
    ...
;
```

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

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

```
// The following snippets demonstrates how to instantiate
// tests using multiple documents:
...
File[] files = {file1, file2, file3};
AssertThat.eachDocument(filename) ❷ 5: "Tests with Multiple Documents" (p. 99)
    .hasText(..)
    .containing(..)
;
```

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

Exception

Tests that expect an exception have to catch the class `PDFUnitValidationException`. In previous releases `PDFUnitError` had to be caught, but this class is marked as deprecated in the release 2015.10. It will be deleted in the next release.

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

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

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

Test Scopes

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

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

.areBothForFastWebView()           4.20: "More Comparisons" (p. 98)
.asRenderedPage()                  3.17: "Layout - Entire PDF Pages" (p. 52)

.containsImage(...)                3.13: "Images in PDF Documents" (p. 45)
.containsOneImageOf(...)           3.13: "Images in PDF Documents" (p. 45)

.hasAnyAction()                    3.2: "Actions" (p. 12)
.hasAuthor()                       3.7: "Document Properties" (p. 27)
.hasBookmark()                     3.4: "Bookmarks and Named Destinations" (p. 21)
.hasBookmarks()                    3.4: "Bookmarks and Named Destinations" (p. 21)
.hasChainedAction()                3.2: "Actions" (p. 12)
.hasCloseAction()                  3.2: "Actions" (p. 12)
.hasCreationDate()                 3.6: "Dates" (p. 24)
.hasCreator()                      3.6: "Dates" (p. 24)
.hasEmbeddedFile(...)              3.3: "Attachments" (p. 18)
.hasEncryptionLength(...)          3.21: "Passwords" (p. 57)
.hasField(...)                     3.10: "Form Fields" (p. 34)
.hasFields()                       3.10: "Form Fields" (p. 34)
.hasFont()                         3.9: "Fonts" (p. 30)
.hasFonts()                        3.9: "Fonts" (p. 30)
.hasFormat(...)                    3.12: "Format" (p. 43)
.hasImportDataAction()             3.2: "Actions" (p. 12)
.hasJavaScript()                   3.14: "JavaScript" (p. 48)
.hasJavaScriptAction()             3.2: "Actions" (p. 12)
.hasKeywords()                     3.7: "Document Properties" (p. 27)
.hasLaunchAction()                 3.2: "Actions" (p. 12)
.hasLayer()                        3.16: "Layers" (p. 50)
.hasLayers()                       3.16: "Layers" (p. 50)
.hasLessPagesThan()                3.20: "Page Numbers as Objectives" (p. 56)
.hasLocale(...)                    3.15: "Language" (p. 49)
.hasLocalGotoAction()              3.2: "Actions" (p. 12)
.hasModificationDate()             3.6: "Dates" (p. 24)
.hasMorePagesThan()               3.20: "Page Numbers as Objectives" (p. 56)
.hasNamedAction()                  3.2: "Actions" (p. 12)
.hasNamedDestination()             3.4: "Bookmarks and Named Destinations" (p. 21)

.hasNoAuthor()                     3.7: "Document Properties" (p. 27)
.hasNoCreationDate()               3.6: "Dates" (p. 24)
.hasNoCreator()                    3.7: "Document Properties" (p. 27)
.hasNoKeywords()                   3.7: "Document Properties" (p. 27)
.hasNoLocale()                     3.15: "Language" (p. 49)
.hasNoModificationDate()           3.6: "Dates" (p. 24)
.hasNoProducer()                   3.7: "Document Properties" (p. 27)
.hasNoProperty()                   3.7: "Document Properties" (p. 27)
.hasNoSubject()                    3.7: "Document Properties" (p. 27)
.hasNoText()                       3.25: "Text" (p. 65)
.hasNoTitle()                      3.7: "Document Properties" (p. 27)
.hasNoXFADData()                   3.30: "XFA Data" (p. 74)
.hasNoXMPDData()                   3.31: "XMP Data" (p. 77)

.hasNumberOf...()                  3.19: "Number of PDF Elements" (p. 55)

.hasOCG()                          3.16: "Layers" (p. 50)
.hasOCGs()                         3.16: "Layers" (p. 50)
.hasOpenAction()                   3.2: "Actions" (p. 12)
.hasOwnerPassword(...)             3.21: "Passwords" (p. 57)
.hasPermission()                   3.22: "Permissions" (p. 58)
.hasPrintAction()                  3.2: "Actions" (p. 12)
.hasProducer()                     3.7: "Document Properties" (p. 27)
.hasProperty(...)                  3.7: "Document Properties" (p. 27)
.hasRemoteGotoAction()             3.2: "Actions" (p. 12)
.hasResetFormAction()              3.2: "Actions" (p. 12)
.hasSaveAction()                   3.2: "Actions" (p. 12)
.hasSignature(...)                  3.23: "Signatures and Certificates" (p. 59)
.hasSignatures()                   3.23: "Signatures and Certificates" (p. 59)
.hasSignedSignatureFields()         3.23: "Signatures and Certificates" (p. 59)

... continued
```

```

... continuation:

.hasSubject()                3.7: "Document Properties" (p. 27)
.hasSubmitFormAction()       3.2: "Actions" (p. 12)
.hasText(...)                3.25: "Text" (p. 65)
.hasTitle()                  3.7: "Document Properties" (p. 27)
.hasTrappingInfo(...)        3.28: "Trapping Info" (p. 71)
.hasUnsignedSignatureFields() 3.23: "Signatures and Certificates" (p. 59)
.hasURIAction()              3.2: "Actions" (p. 12)
.hasUserPassword(...)        3.21: "Passwords" (p. 57)
.hasVersion()                3.29: "Version Info" (p. 73)
.hasXFADData()               3.30: "XFA Data" (p. 74)
.hasXMPData()                3.31: "XMP Data" (p. 77)

.haveSame...()               4.1: "Overview" (p. 81)

.isCertified()               3.5: "Certified PDF" (p. 24)
.isCertifiedFor(...)         3.5: "Certified PDF" (p. 24)
.isLinearizedForFastWebView() 3.8: "Fast Web View" (p. 30)
.isSigned()                  3.23: "Signatures and Certificates" (p. 59)
.isTagged()                  3.24: "Tagged Documents" (p. 64)

... (end of list)

```

PDFUnit is continuously being improved and we keep the manual up to date. Wishes and requests for new functions are appreciated. Please send them to [request\[at\]pdfunit.com](mailto:request[at]pdfunit.com).

3.2. Actions

Overview

"Actions" make PDF documents interactive and more complex. "Complex" means that they should be tested, especially when interactive documents are part of a workflow. Those actions need to work correctly.

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

```

// Types of actions:

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

```

For some of these actions PDFUnit provides test methods:

```

// 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()

```

The result of these test methods are strings which can be compared to an expected strings using the following functions:

```

// 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 The chapter 13.5: “Whitespace Processing” (p. 150) describes how to control whitespaces.

The parameter of the methods `containing(..)` and `matchingComplete(..)` can be of type `java.io.Reader`, `java.io.InputStream` or `java.lang.String`.

The following sections show examples for different types of actions.

Close-Actions

Close-Actions are executed when the document is being closed. A test looks like:

```

@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)
    ;
}

```

The content of a Close-Action can also be compared with the content of a file:

```
@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)
        ;
}
```

The second parameter which controls the whitespace processing is optional. The default whitespace processing is `NORMALIZE_WHITESPACES`.

ImportData-Actions

ImportData-Actions import data from a file. They need the filename as a parameter:

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

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

PDFUnit checks whether the action contains the expected filename. The file's existence is not checked.

JavaScript-Actions

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

```
@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)
        ;
}
```

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

Launch-Actions

Launch-Actions are launching applications or scripts. This can be tested like this:

```
@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)
        ;
}
```

PDFUnit compares the content of the variables `application` and `withOperation` with the actual values of the Launch-Action. It is not checked whether the application can be started.

Named-Actions

The name of Named-Actions should be verified:

```
@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 need a destination in the same PDF document:

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

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

The test is successful, when the current test PDF contains the expected destination "destination2.1".

GotoRemote-Actions

GotoRemote-Actions need a destination in another PDF file.

```
@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)
    ;
}
```

PDFUnit checks that the action in the PDF document under test contains an action with the expected destination. PDFUnit does not check whether the remote file or the destination in the remote file exist.

Open-Actions

Open-Actions are executed when the PDF document is loaded. Often they are JavaScript- or 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(.*)" )
    ;
}
```

In addition to the other methods for comparing text, Open-Actions can be tested with the function `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 are JavaScript-Actions which are processed immediately before or after printing a PDF document. They are associated internally with the events `WILL_PRINT` or `DID_PRINT`.

```
@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)
    ;
}
```

As for JavaScript-Actions the content of the Print-Action is compared with the expected string. The whitespaces are normalized before.

ResetForm-Actions

ResetForm-Actions have no parameters and it is unnecessary to compare text. Only the existence will be verified:

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

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

Save-Actions

Save-Actions are JavaScript-Actions which are processed immediately before or after saving a PDF document. The actions are associated with the PDF-Events `WILL_SAVE` or `DID_SAVE`.


```
@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)
    ;
}
```

Again, the comparison is performed only after a normalization of whitespace characters. All known tags can be used to compare the texts.

SubmitForm-Actions

SubmitForm-Actions need a destination to which forms can be sent:

```
@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 does not check whether the destination exists. It only checks that the currently tested action contains a destination with the expected value.

URI-Actions

URI-Actions need a target URI:

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

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

PDFUnit does not access the internet. So this test merely checks that the PDF under test contains a URI with the expected value.

The method `withURI()` is optional. It has no functionality and can be dropped off. It exists to let the API flow better.

Any Action

The following example tests 4 different types of actions which should all exist in one PDF document:

```
@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 Processing in Comparisons

You can control how whitespaces are processed when comparing text. In the following example line breaks and blank lines are ignored:

```
@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)
    ;
}
```

The chapter 13.5: “Whitespace Processing” (p. 150) explains the flexible handling of whitespaces.

3.3. Attachments

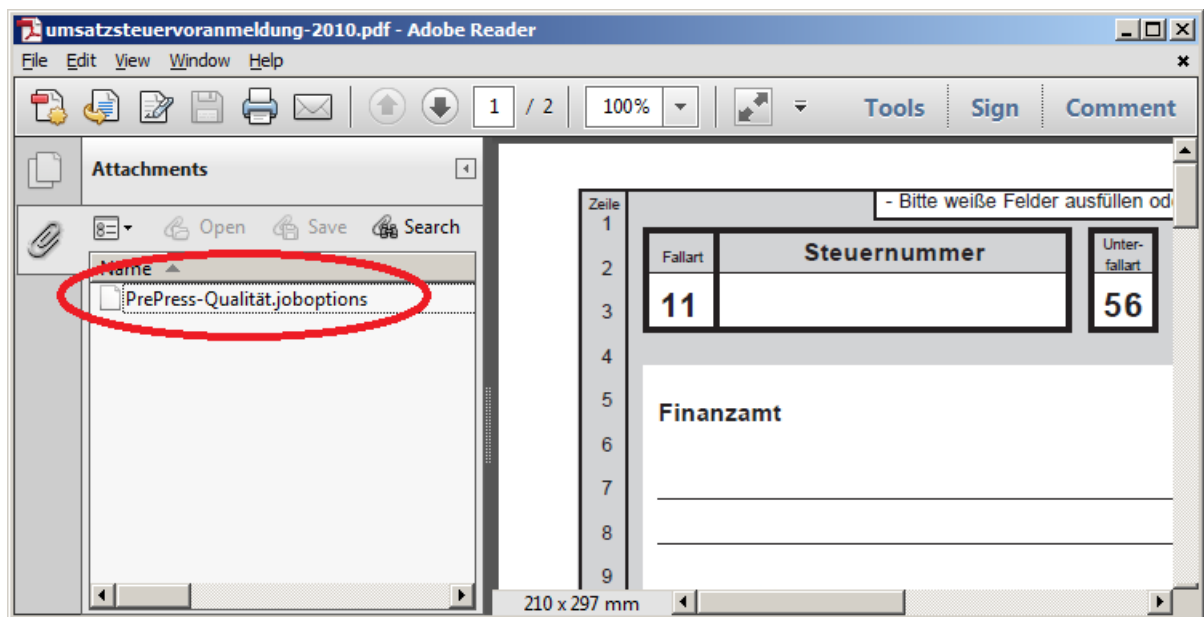
Overview

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

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

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

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



Existence of Attachments

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

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

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

Number of Attachments

The next test verifies the expected number of embedded files:

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

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

Filename

Also the names of embedded files can be tested:

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

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

Content

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

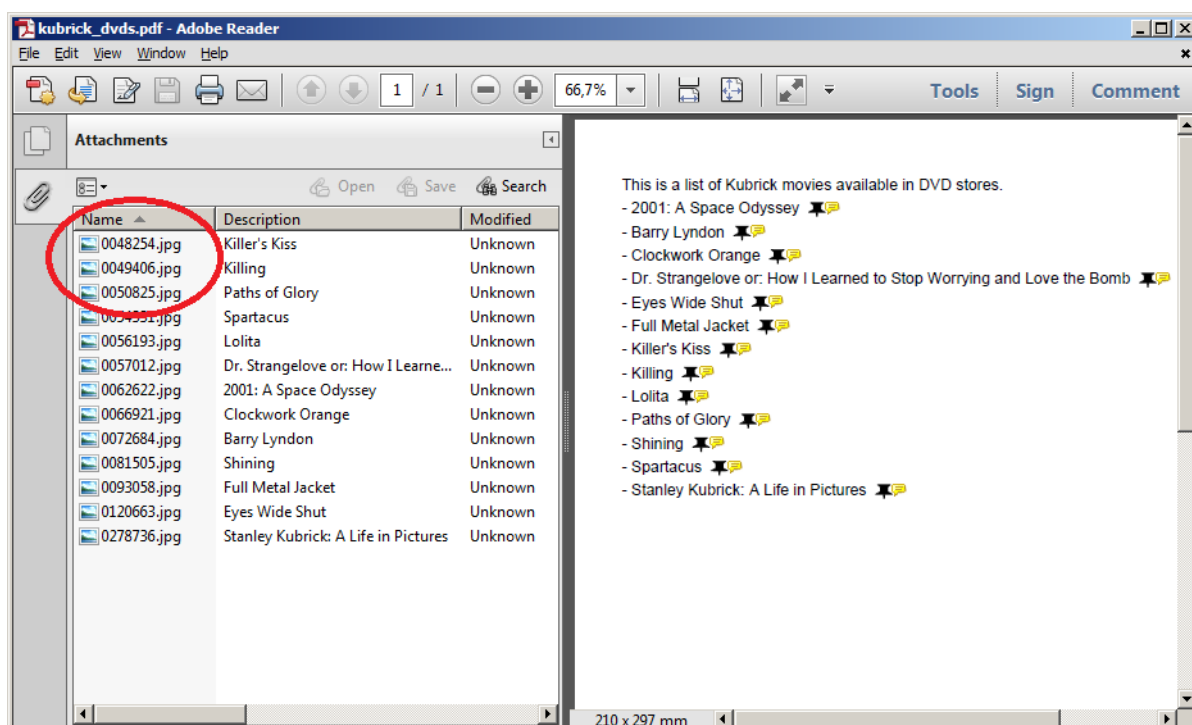
```
@Test
public void hasEmbeddedFile_WithContent() throws Exception {
    String pdfFileName = PATH + "embeddedfiles/umsatzsteuervoranmeldung-2010.pdf";
    String embeddedFileName = PATH + "embeddedfiles/PrePress-Qualität.joboptions";

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

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

Multiple Attachments

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



You can check for multiple attachments in one test. But select a better name for such a test:

```
@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)
    ;
}
```

If embedded files are not available as separate files, they can be extracted from an existing PDF with the utility “ExtractEmbeddedFiles”. This program is described in detail in chapter 9.4: “Extract Attachments” (p. 114):

3.4. Bookmarks and Named Destinations

Overview

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

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

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

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

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

For named destinations, the following test methods are available:

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

Named Destinations

The names of named destinations can be tested easily:

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

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

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

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

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

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

Existence of Bookmarks

It is easy test to verify the existence of bookmarks:

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

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

Number of Bookmarks

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

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

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

Label of a Bookmark

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

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

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

Destinations of Bookmarks

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

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

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

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

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

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

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

Does a bookmark exist which points to an expected destination:

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

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

Is there a bookmark pointing to a URI:

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

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

And finally, we can check that there is no bookmark having a “dead link”:

```
/**
 * 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 does not access websites. So a “dead link” is a bookmark that does not point to a page or any other destination.

Check Bookmarks with XML/XPath

The next tests all use an XML structure which is created with the utility program `ExtractBookmarks`.

The bookmarks of a PDF document can be compared with an existing XML file. Each bookmark in the PDF must match an element in the XML file.

```
@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 When comparing PDF with XML, whitespaces and comments are ignored.

Bookmark information can also be verified using individual XPath expressions:

```
@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)
    ;
}
```

3.5. Certified PDF

Overview

When your workflow relies on certain properties of the processed PDF, who guarantees that the PDF complies with a given specification? A “certified PDF document” gives this guarantee.

A “certified PDF” is a regular PDF with additional information. It contains information about the profile which was used in the certification process and it contains a log of all changes to the PDF after it was certified. These changes can be rolled back.

Don't confuse “certified PDF” with the “certificate” of a signature. For “certified PDF” PDFUnit provides these test methods:

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

Examples

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

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

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

Next you can check the level of certification:

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

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

Certification Level

PDFUnit provides constants for the certification level:

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

3.6. Dates

Overview

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


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

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

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

Existence of a Date

The first test checks that a creation date exists:

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

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

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

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

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

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

Date Resolution

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

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

```
// Constants for date resolution:

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

A test looks like this:

```
@Test
public void hasCreationDate_WithValue() throws Exception {
    String filename = 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 The utility `com.pdfunit.util.DateHelper` can be used to create an instance of `java.util.Calendar` for the expected date.
- ❷❸ Both constants are defined in the enum `com.pdfunit.DateResolution`.

Using an instance of `java.util.Calendar` the expected date is format independent. But the PDF-internal date is formatted. So you have to declare a suitable format string in the file `config.properties`.

Configuring the Date Format in config.properties

Date formats in PDF documents vary widely range depending on the PDF generating tool. That is why you can declare it in the file `config.properties`:

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

Beware: When you define the format `dateformat = 'D:'yyyy` then the “January 1” is assumed for date and month. That is probably not what you intended.

You can only define **one** date format in the config file. But if you want to check a PDF document with another date format, you can test it as a property using `hasProperty()`:

```
@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")
    ;
}
```

Date Tests with Upper and Lower Limit

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

```
@Test
public void hasCreationDate_Before() throws Exception {
    String filename = 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 The lower- or upper-limits are not included in the expected date range.

Creation Date of a Used Certificate

Beside creation and modification date PDF documents contain the date their certificates were issued. Chapter 3.23: “Signatures and Certificates” (p. 59) covers tests for that.

3.7. Document Properties

Overview

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

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

The following methods are available to verify document properties:

```
// Testing document properties:

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

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

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

❶❷ Tests for creation date and modification date are described in chapter 3.6: “Dates” (p. 24) because they differ from tests for the other document properties.

Document properties of a test document can also be compared with the properties of a master document. Such tests are described in chapter 4.7: “Comparing Document Properties” (p. 87).

Testing the Author ...

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

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

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

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

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

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

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

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

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

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

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

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

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

All test methods are is case sensitive.

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

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

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

For every document property the methods `hasXXX()` and `hasNoXXX()` exist.

Of course, methods can be concatenated:

```
@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")
        ;
}
```

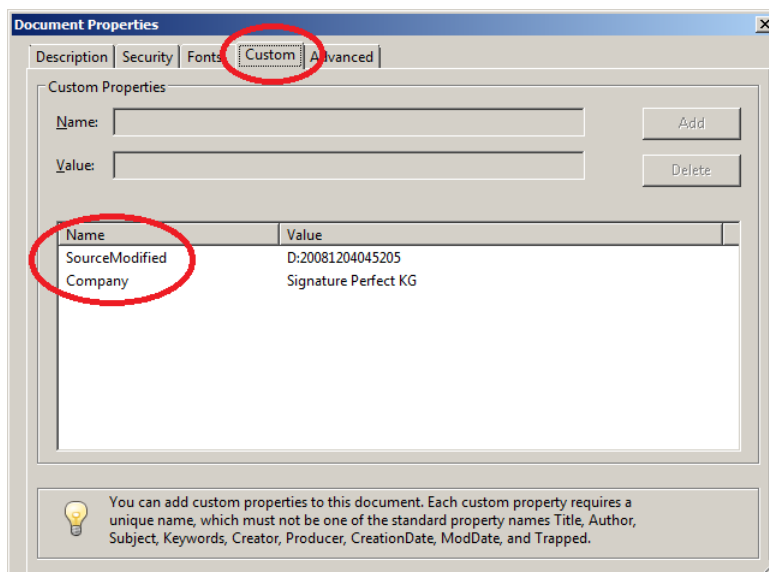
Common Validation as a Key-Value Pair

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

```
@Test
public void hasProperty_StandardProperties() throws Exception {
    String filename = 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'")
    ;
}
```

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



And this is the test for custom properties:

```
@Test
public void hasProperty_CustomProperties() throws Exception {
    String filename = 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)
    ;
}
```

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

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

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

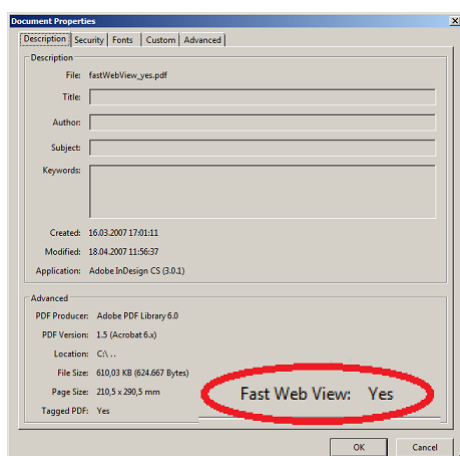
PDF documents of version PDF-1.4 or higher can have metadata as XML (Extensible Metadata Platform, XMP). Chapter 3.31: "XMP Data" (p. 77) explains that in detail.

3.8. Fast Web View

Overview

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

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



PDFUnit looks for a PDF-object (dictionary) with the key `/Linearized` and the value `1`. Additionally, when the document length stored in the same dictionary is the same as the actual file length, the following test method results in a “green bar”.

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

Example

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

3.9. Fonts

Overview

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

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

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

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

Number of Fonts

What is “a font”? Should a “subset” of a font count as a separate font? In most situations this question is irrelevant for developers, but for a testing tool the question has to be answered. And not only a testing tool - every PDF tool has to make this decision. That is the reason why they show different numbers of fonts for the same document. Since a goal of unit testing is that the second run of a test has the same result as the first one, it doesn't really matter how fonts are identified.

In PDFUnit two fonts are “equal” to each other, when the compared criteria have the same values. The criteria you want to test can be set with the attribute `identifiedBy=".."`:

```
// 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
```

The following list explains the available criteria to compare fonts.

Constant	Description
ALLPROPERTIES	All properties of a font are used to identify a font. Two fonts having the same values for all properties considered equal.
BASENAME	Fonts are different when they have different base fonts.
BASENAME_ENCODING	The combination of the name of a base font and the encoding are used to distinguish fonts.
BASENAME_ENCODING_ENCODINGDIFF	Two fonts have to have same values in the properties “basename”, “encoding” and the property “encoding-difference” to be considered equal. The “Encoding-difference” is the value of the PDF object with the key <code>/Differences</code> .
CONVERTIBLE2UNICODE	This filter means that only fonts are considered, which are convertible into Unicode.
EMBEDDED	This filter counts only fonts that are embedded.
EMBEDDED_CONVERTIBLE2UNICODE	In addition to the previous filter the ability of a font to be converted into Unicode is the other distinguishing property.
NAME	Only the fonts' names are relevant to the test.

Constant	Description
NAME_TYPE	Only the font name and font type are used to compare fonts.
TYPE	Only the types of the fonts are considered in the comparison.

The following example shows all filters:

```
@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)
    ;
}
```

Font Names

Testing the names of fonts are easy:

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

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

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

Of course, you can chain multiple methods:

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

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

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

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


Complex tests for font names can be implemented using XPath. They are described later in this chapter:

Font Types

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

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

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

Predefined font types are:

```
// Constants for font types:
com.pdfunit.Constants.FONTTYPE_CID
com.pdfunit.Constants.FONTTYPE_CID_TYPE0
com.pdfunit.Constants.FONTTYPE_CID_TYPE2
com.pdfunit.Constants.FONTTYPE_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 for Font Tests

You can extract all properties of all fonts from a PDF document into an XML file using the utility `ExtractFontsInfo`. This XML file can be used for various tests.

The file contains the following information:

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

Here is a test based on that XML file:

```
@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 are ignored when comparing an XML file with the font properties of a PDF document.

XPath for Font Tests

Sophisticated tests can be implemented using XPath queries:

```

@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)
    ;
}

```

If you have problems with XPath, extract the font information with the utility `ExtractFontsInfo` and verify the XPath expression against the XML file. You can use Eclipse's `has` the "XPath"-View.

Further information about XPath can be found in chapter 8: "Using XPath" (p. 110).

3.10. Form Fields

Overview

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

You can extract all information about form fields with the utility `ExtractFieldsInfo` into an XML file, which can then be used for XML and XPath based tests.

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

```

// Simple tests:
.hasField(..)
.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(..).endsWith(..)
.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 This test is described separately in chapter 3.11: “Form Fields - Text Overflow” (p. 42):

Existence of Fields

The following test verifies whether or not fields exist:

```
@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
    ;
}
```

Number of Fields

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

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

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

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

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

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

Name of Fields

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

```

@Test
public void hasField_MultipleInvocation() throws Exception {
    String filename = 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)
    ;
}

```

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

```

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

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

```

Content of Fields

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

```

@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()
    ;
}

```

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

```

.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

```

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

```

@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 sqlCommentSequence = "--";

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

```

Type of Fields

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

```

import static com.pdfunit.Constants.*;
...
@Test
public void hasField_Withype_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)
        ;
}

```

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

```

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

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

```

Detailed tests for signatures and certificates are described in the chapter 3.23: "Signatures and Certificates" (p. 59):

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

```

// Mapping between PDFUnit-Constants and PDF-internal types.
PDFUnit-Constant    PDF-intern
-----
CHOICE              -> "choice"
COMBO               -> "choice"
LIST                -> "choice"
CHECKBOX            -> "button"
PUSHBUTTON          -> "button"
RADIOBUTTON         -> "button"
SIGNATURE           -> "sig"
TEXT                -> "text"

```

Field Size

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

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

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

Both methods can be invoked with different measuring units, defined as constants:

```
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 The formats POINTS and DPI72 are identical.

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

Whether a text fits into a field or not is not predictable by calculation using font size and field size. In addition to the font size the words at the end of each line determine the required number of rows and the required height. And the calculation has to consider hyphenation. Chapter 3.11: "Form Fields - Text Overflow" (p. 42) deals with this subject in detail.

Field Properties

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

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

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

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

<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>.whichIsMultiSelectable(),</code>	<code>.whichIsNotMultiSelectable()</code>
<code>.whichIsPasswordProtected(),</code>	❷
<code>.whichIsPrintable(),</code>	<code>.whichIsNotPrintable()</code>
<code>.whichIsReadOnly(),</code>	<code>.whichIsNotReadOnly()</code>
<code>.whichIsRequired(),</code>	<code>.whichIsNotRequired()</code>
<code>.whichIsSigned(),</code>	❸
<code>.whichIsVisible(),</code>	<code>.whichIsNotVisible()</code>
<code>.whichIsVisibleButNotPrintable()</code>	❹

❶❷❸❹ The inverse methods are intentionally not provided because they are not needed.

Whitespaces will be ignored when comparing expected and actual field content:

```
@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 Actions for Fields

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

PDFUnit can verify that a field is linked with an action:

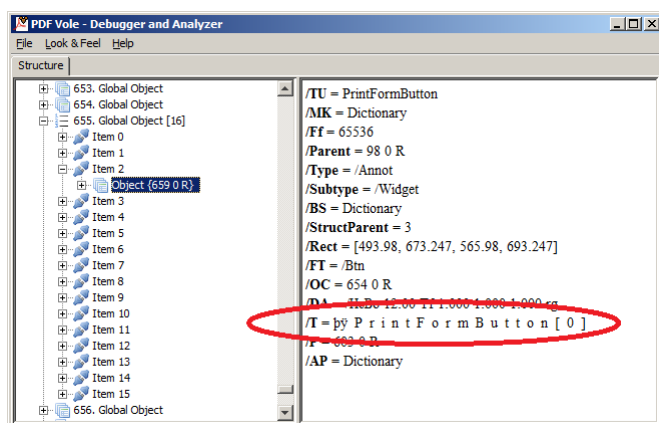
```
@Test
public void hasField_HavingJavaScriptAction_MultipleInvocation() throws Exception {
    String filename = PATH + "acrofields/javaScriptForFields.pdf";

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

Methods to validate the JavaScript itself are described in chapter: 3.14: "JavaScript" (p. 48).

Unicode

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



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

```
/**
 * 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
        //      o          t          [          0          ]
        "\u00fe\u00ff\u0000\u0046\u0000\u006f\u0000\u0072\u0000\u006d\u0000\u0052" +
        //      o          t          [          0          ]
        "\u0000\u006f\u0000\u006f\u0000\u0074\u0000\u005b\u0000\u0030\u0000\u005d" +
        //
        "\u002e"
        //      P          a          g          e          1
        //      [          0          ]
        "\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
        //      F          o          r          m          B          u
        //      t          t          o          n          [          0          ]
        "\u0000\u0046\u0000\u006f\u0000\u0072\u0000\u006d\u0000\u0042\u0000\u0075" +
        //      t          t          o          n          [          0          ]
        "\u0000\u0074\u0000\u0074\u0000\u006f\u0000\u006e\u0000\u005b\u0000\u0030" +
        //      ]
        "\u0000\u005d";

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

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

Validate Field Information against XML

Chapter 9.3: "Extract Field Information to XML" (p. 113) describes how to extract information about all fields in a PDF document. You can compare the field properties of a PDF document with the extracted XML file:

```
@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 The filename can also be given as `java.lang.String`.

Validate Field Information with XPath

The extracted XML data can also be used for XPath based tests. That allows you to test dependencies between multiple fields ("cross-constraints"). The next example gives you an idea of the possibilities:

```
@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)
    ;
}
```

The function `matchingXPath(...)` can be chained:

```
@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)
        .matchingXPath(xpathNumberOfButtonFields)
        .matchingXPath(xpathNumberOfChoiceFields)
        .matchingXPath(xpathNumberOfSignaturFields)
    ;

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

❶❷ Two different kinds of syntax can be used.

The following two example check whether unsigned signature fields exist:

```
@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 uses the XSLT-processor in your Java runtime. Please read the documentation for your JRE or JDK to find out which XPath 2.0 syntax elements and functions are supported. There are no restrictions from PDFUnit itself.

If you didn't expect an exception, this message would appear: Content of field 'Textfield, too much text, multiline:' of 'C:\...\fieldSizeAndText.pdf' does not fit in the available space.

Correct Size of all Fields

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

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

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

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

Technical constraint

Due to technical reasons the content of a field is not detectable by the test method `.hasText(...)`. If you anyhow want to verify it, the PDF document must first be flattened.

Therefore, the following test fails:

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

3.12. Format

Overview

You need to check that the intended format was created successfully: A4-landscape, Letter-portrait or a special format for a poster? You think testing such simple thing is a waste of time? But have you ever printed a "LETTER"-formatted document on an "A4"-printer? It is possible ... ;-)

Deshalb stellt PDFUnit eine Testmethode für Formattests zur Verfügung:

PDFUnit provides one flexible method to test document formats:

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

Documents with Same Page Format

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

```
import static com.pdfunit.Constants.*;
...
@Test
public void hasFormat_A4Landscape() throws Exception {
    String filename = 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 Many popular formats are defined in `com.pdfunit.Constants`

You can also verify individual formats:

```
@Test
public void hasFormat_FreeFormat_1117x836_mm() throws Exception {
    String filename = 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); ❹

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

❸❹ You can use the measurement units POINTS, MILLIMETER, CENTIMETER, INCH and DPI72. All units are defined in the class `com.pdfunit.Constants`. The units DPI72 and POINTS are equivalent.

The topic of different paper sizes and their dimensions in points, millimeters and inches is well documented at www.prepressure.com.

Tolerances of width and height are accepted when comparing expected values with actual values. The standard ISO 216 (http://en.wikipedia.org/wiki/Paper_size) specifies tolerances and also the popular standard DIN 476. PDFUnit uses the stricter tolerances of DIN 476 for all formats.

Documents with Multiple Formats

A document with pages of different sizes can also be checked for its formats:

```
@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)
    ;
}
```

The format tests can be restricted to individual pages or page ranges as described in chapter 13.3: “Page Selection” (p. 147):

```
@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))
    ;
}
```

3.13. Images in PDF Documents

Overview

An outdated image in a document impresses a customer as much as a repeated New Year's speech. You should be sure that the **new** logo is actually shown on the document and not the old one.

Another source of errors with images is that a picture was not found when creating the PDF, so it is missing in the document. Let an automated test detect this error, not your customer.

And finally one kind of error has to be mentioned: images sometimes appear on the wrong page.

All errors can be detected with these test methods:

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

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

Number of different Images inside PDF

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

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

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

How do you know in this example that “2” is the right number? How do you know which images are stored internally for a given PDF? The answer to both questions is given by the utility program `ExtractImages`. You can use it to extract all images from a document into separate files. The chapter 9.7: “Extract Images from PDF” (p. 118) describes this topic in detail.

Number of visible Images inside a PDF

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

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

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

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

The test for the visual images can be limited to specified pages. In the following example, only the images on page 3 are counted:

```
@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)
    ;
}
```

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

The possibilities for limiting tests to specified pages are described in chapter 13.3: “Page Selection” (p. 147).

Validate the Existence of an Expected Image

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

```
@Test
public void 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)
    ;
}
```

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

The picture may pass to the method in different types:

```
// Types for images:
.containsImage(String imageFileName, PagesToUse pagesToUse);
.containsImage(File imageFile, PagesToUse pagesToUse);
.containsImage(InputStream imageStream, PagesToUse pagesToUse);
```

A tool which generates PDF can carry out a format conversion when importing images from a file because not all image formats are supported in PDF. So it might be impossible for PDFUnit to successfully compare an image inside your PDF file with the original image file. If you have such a problem, extract the desired image of a sample document into a new PNG file by following these steps:

- Extract all pictures from a PDF file using `ExtractImages`. All pictures are saved as PNG.
- Verify the picture you want to use.
- Use PDFUnit as demonstrated in the listing above.

Use an Array of Images for Comparison

It might be that a PDF document contains one of three possible logos. Or the signature is one of five possible ones. Use the method `containsOneImageOf(...)` to test such a situation:

```
@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)
        ;
}
```

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

Validate Images on Specified Pages

The tests for images can be restricted to single pages, multiple individual or multiple contiguous pages. All possibilities are described in chapter 13.3: "Page Selection" (p. 147).

Here are some examples:

```
@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)
        ;
}
```

Methods can be invoked multiple times. But it might be better to write two separate tests:

```
@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)
    ;
}
```

All images in a PDF document can be compared to the images of a master PDF. Those tests are described in chapter 4.10: “Comparing Images” (p. 89).

3.14. JavaScript

Overview

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

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

Existence of JavaScript

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

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

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

Comparison against Expected Text

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

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

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

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

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


```

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

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

```

Comparing Substrings

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

```

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 are ignored when comparing JavaScript.

Since extracted JavaScript is plain text, no XML or XPath based tests are provided.

3.15. Language

Overview

PDF documents can be read out by screen readers for visually handicapped users. These programs need documents with a given country or language code.

These methods are available for tests:

```

// Tests for PDF locale:

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

```

Examples

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

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

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

❶ Notation typical for PDF

❷ Notation typical for Java

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

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

```
@Test
public void hasLocale_LocaleInstance_GERMANY() throws Exception {
    String filename = 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)
    ;
}
```

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

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

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

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

3.16. Layers

Overview

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

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

PDFUnit provides the following methods to test layers:

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

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

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

The test method `endsWith()` is not provided because duplicate layer names are expanded internally with a suffix and thus the end of a name is not predictable.

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

Number of Layers

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

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

Layer Names

The next example tests the name of a layer:

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

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

A layer name can be compared using the following methods:

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

The methods `endsWith(..)` and `matchingRegex(..)` are intentionally not provided as explained at the beginning of this chapter.

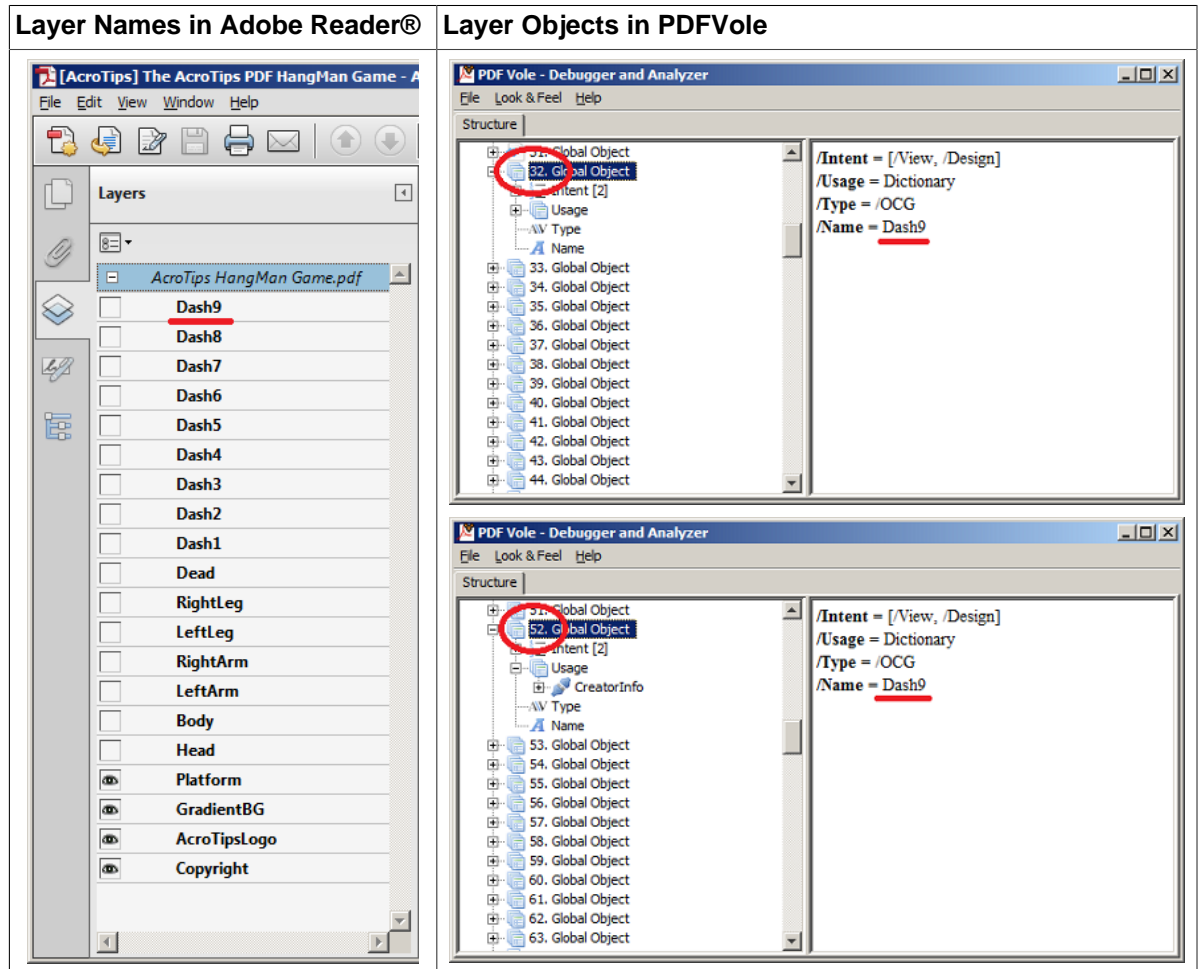
String comparisons are case-insensitive. Whitespaces remain unchanged.

```
@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")
    ;
}
```

Duplicate Layer Names

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



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

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

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

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

In the current release 2015.10 PDFUnit does not provides functions to verify the content of a single layer.

3.17. Layout - Entire PDF Pages

Overview

Text in a PDF document has properties such as font size, font color and lines which should be correct before sending it to the customer. Also paragraphs, alignment of text, images and image descriptions

are important parts of the layout. PDFUnit tests these aspects, first rendering the document page by page and then comparing each page:

- ... with an existing image file. PDFUnit's utility program `RenderPdfToImages` creates images of one or more PDF pages. Chapter 9.14: "Render Pages to PNG" (p. 125) explains how to use it.
- ... with a rendered page of a master document. The chapter 4.12: "Comparing Layout as Rendered Pages" (p. 90) describes tests with master documents.

The following test methods are provided:

```
// 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 3.18: "Layout - in Clipping Areas" (p. 54)
.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 3.18: "Layout - in Clipping Areas" (p. 54)
.asRenderedPage(...).isEqualToImages(upperLeftX, upperLeftY, unit, ...)
```

You can choose a page to be compared. This is described in the chapter 13.3: "Page Selection" (p. 147).

The current chapter deals "only" with the comparison of full pages. If it does not make sense to compare a complete rendered PDF page with an image, the comparison can be limited to a section of a page. The following chapter 3.18: "Layout - in Clipping Areas" (p. 54) describes that topic.

Example - Compare Preselected Pages as Rendered Images

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

```
@Test
public void compareAsRenderedPage_MultipleImages() throws Exception {
    String filename = 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)
    ;
}
```

Image Format of Rendered Pages

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

The images can be passed as parameters of type `File`, `String` or `BufferedImage`.

```
// 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)
```

3.18. Layout - in Clipping Areas

Overview

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

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

```
// 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[])
```

Example - Left Margin on Every Page

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

```
@Test
public void compareAsRenderedPage_LeftMargin() throws Exception {
    String filename = 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)
        ;
}
```

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

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

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

Example - Logo on Page 1 and 2

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

```

@Test
public void verifyLogoOnEachPage() throws Exception {
    String filename = 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 Set the x/y position of the upper left corner of the clipping area
- ❷ Specify the pages, see chapter 13.3: “Page Selection” (p. 147)
- ❸ Invoke the test method

Multiple Comparisons

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

```

@Test
public void compareAsRenderedPage_MultipleInvocation() throws Exception {
    String filename = 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)
    ;
}

```

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

3.19. Number of PDF Elements

Overview

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

```

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

```

- ❶ Tests for the number of images are described in chapter 3.13: “Images in PDF Documents” (p. 45).
- ❷ Tests for the number of pages are described in chapter 3.20: “Page Numbers as Objectives” (p. 56).

Examples

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

```
@Test
public void hasNumberOfFields() throws Exception {
    String filename = 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)
    ;
}
```

All tests can be concatenated:

```
@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)
    ;
}
```

Be careful, this test with a document containing 1370 pages takes about 10 seconds on a contemporary notebook. Separate the long-running test from the fast ones and start them with two Maven scripts or two ANT scripts.

3.20. Page Numbers as Objectives

Overview

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

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

Examples

You can check the number of pages like this:


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

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

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

```
@Test
public void hasLessPagesThan() throws Exception {
    String filename = 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 The values for upper- and lower limits are exclusive.

Of course methods can be concatenated:

```
@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)
    ;
}
```

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

3.21. Passwords

Overview

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

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

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

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

- ② If the second parameter is used, the document is considered protected.

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

Verifying Both Passwords

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

```
// Verify the owner-password of the document:
@Test
public void hasOwnerPassword() throws Exception {
    String filename = 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 Open the file with one password
❷❹ Verify the other password

Usually it's bad practice to hard code passwords in the source code, but it's OK for test passwords in test environments. “Hard coded” also means that the password never changes.

Verifying the Encryption Length

This example shows how to verify the encryption length:

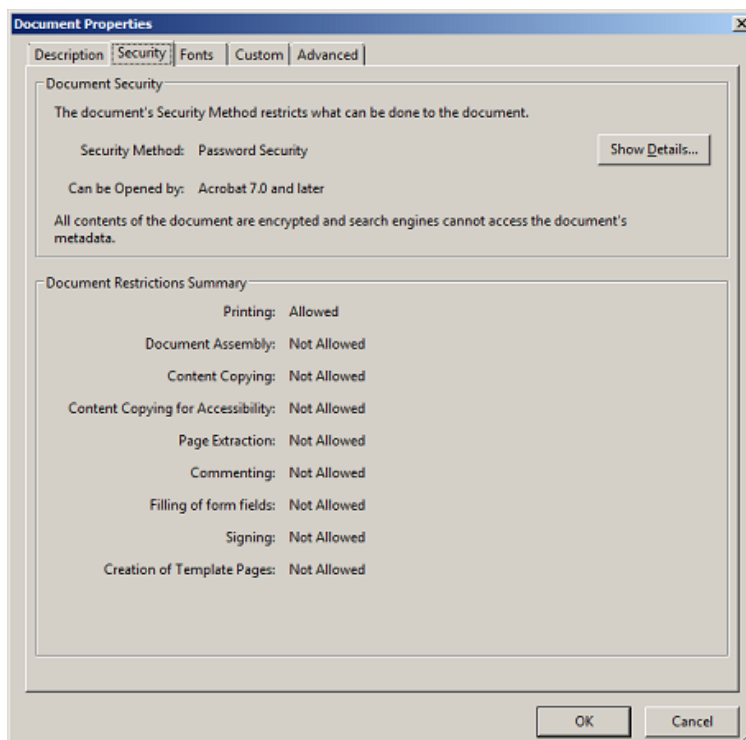
```
@Test
public void hasEncryptionLength() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages_encrypted.pdf";
    String userPassword = "user-password";

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

3.22. Permissions

Overview

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



You can test permissions using the following methods. All methods have one parameter of type boolean:

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

❶❷ The permissions 'extractContent' and 'copyContent' are identical.

Example

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

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

3.23. Signatures and Certificates

Overview

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

PDFUnit provides many test methods for signatures and certificates:

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

When executing the test, PDFUnit only reads the properties of the PDF document. There is no access to remote systems. Thus PDFUnit does not check whether a certificate has been revoked.

A “signed” PDF must not be confused with a “certified” PDF. A “certified” PDF guarantees the compliance with certain properties which are needed to process a document in further workflow steps. Such properties are summarized in “profiles”. Tests for certified PDF documents are described in chapter 3.5: “Certified PDF” (p. 24).

Existence of Signatures

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

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

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

Names and Numbers of Signatures

A document may contain multiple signatures, for example when more than one person has signed it. Thus the next tests check to the number and names of the certificates:

```
@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")
        ;
}
```

Validity Date

Sometimes you need to know that a given date lies within the validity period of a certificate:

```
@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 The comparison is made on the base of year-month-day.

DATE and DATETIME can be specified as the date resolution to compare two date values. Information about this topic can be found in chapter 13.10: “Date Resolution” (p. 157).

```
@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)
    ;
}
```

Another test is to ensure that a certificate is valid within a time period:

```
@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, Reason, Sign-Name

The next examples show how various information of a signature can be tested using special methods:

```
@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)
    ;
}
```

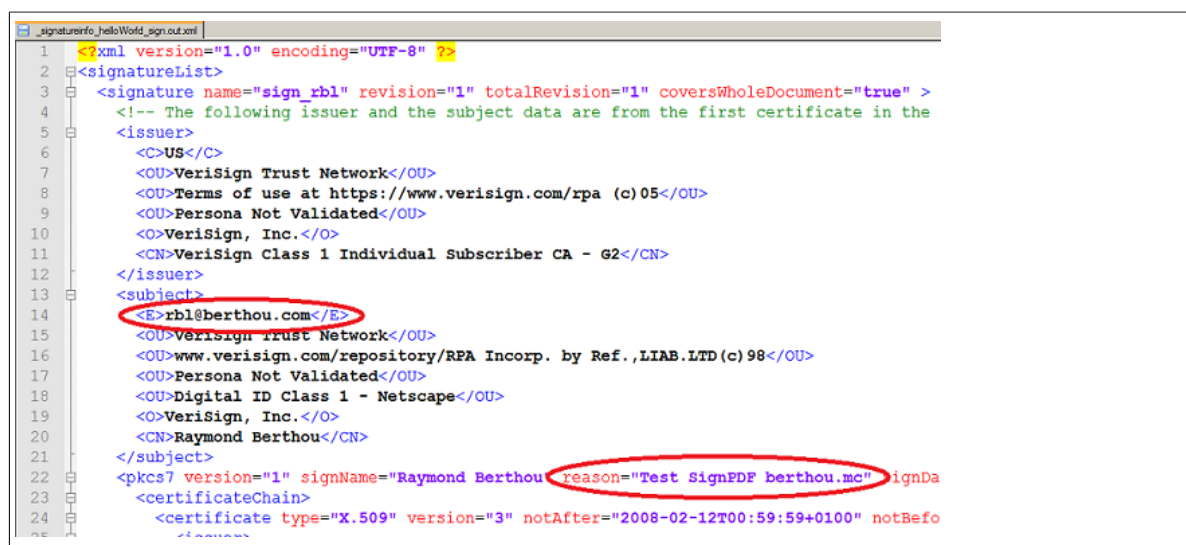
```
@Test
public void hasSignature_WithReason() throws Exception {
    String filename = PATH + "signed/sampleSignedPDFDocument.pdf";

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

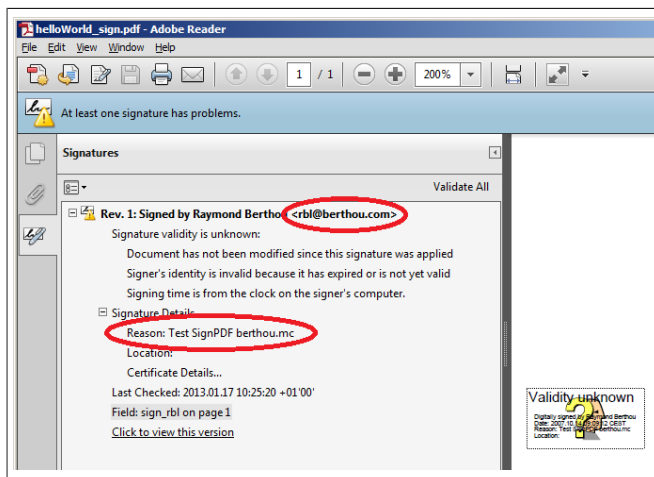
You can check other signature and certificate data using XPath.

Compare Signatures using XML/XPath

Any data of a signature can be verified using XPath expressions. You can see the signature data as an XML structure when you extract it with the utility program `ExtractSignaturesInfo`. The following image shows a small part of the file:



The corresponding image created by Adobe Reader® shows the same information:



The signature information of a document can be compared in its entirety with an XML file:

```
@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)
    ;
}
```

When parts of the XML are the test goal, a matching XPath expression has to be found. The following example checks that the first certificate contains one OU tag with an expected value:

```
@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)
    ;
}
```

Useful hint: Eclipse provides the “XPath-View” to work with XPath expressions.

Multiple Invocation

Of course, test methods can be chained:

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

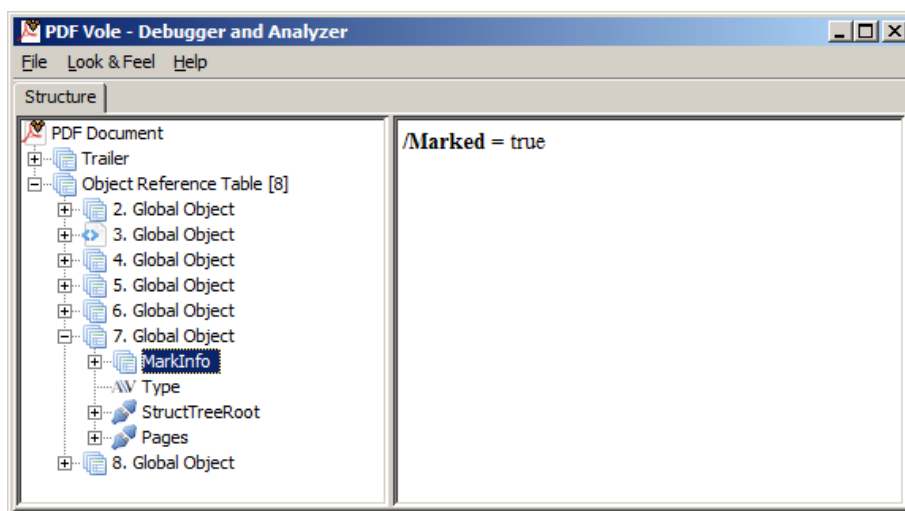
But think of a better name for this test. It would be better to split it into several tests with specific names.

3.24. Tagged Documents

Overview

The PDF standard “ISO 32000-1:2008” says in chapter 14.8.1 “A Tagged PDF document shall also contain a mark information dictionary (see Table 321) with a value of true for the Marked entry.” (Cited from: http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf.)

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



Following test methods are available:

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

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

Examples

The simplest test checks whether a document is tagged.

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

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

Further tests verify the existence of a particular tag.

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

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


And finally you can verify values of tags:

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

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

3.25. Text

Overview

The most common test case for PDF documents is probably to check the presence of expected text.

```
// 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 The chapter 13.5: “Whitespace Processing” (p. 150) describes the different options.

Text on Individual Pages

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

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

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

The next example searches a text on the last page:

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

Also, you can tests individual pages:

```

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

For the specification of pages several constants of the class `com.pdfunit.Constants` are available, e.g. `ON_EVEN_PAGES` and `ON_ODD_PAGES`. Chapter 13.3: “Page Selection” (p. 147) describes more constants and how to use them.

Text on All Pages

There are three constants to search text on multiple pages, `ON_ANY_PAGE`, `ON_EACH_PAGE` and `ON_EVERY_PAGE`. The last two are functionally identical.

```

@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")
    ;
}

```

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

Negated Search

The logic of the two previous examples is clear. But the logic becomes unclear when you negate both statements. In everyday speech, the difference between “Every page does not contain the expected text” and “Any page does not contain the expected text” is unclear. And the last sentence itself has an unclear meaning.

To avoid mistakes, PDFUnit does not allow negated tests with the constant `ON_ANY_PAGE`. The following test is **not** allowed and throws an exception:

```

@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")
    ;
}

```

The error message is:

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

Instead of asking that “any page does NOT contain an expected text” it is better to write “every page contains the expected text” and catch the exception.

Line Breaks in Text

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

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)

The following tests for the marked text use different line breaks. They both succeed:

```
/**
 * 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 Parts of a Page

Text can be searched not only on whole pages, but also on a section of a page. The chapter 13.7: “Defining Page Areas” (p. 154) describes that topic.

Empty Pages

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

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

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

If you want to verify that a page or a section of a page does not contain text, you can use the method `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)
        ;
}
```

Multiple Search Tokens

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

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

    AssertThat.document(filename)
        .hasText(ON_ODD_PAGES)
        .containing("on", "page", "odd pagenumber") // 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")
        ;
}
```

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

Concatenation of Methods

All chained test methods refer to the same pages, i.e. those selected by `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.*")
        .endingWith("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).endingWith("of 4")
        .hasText(ON_ANY_PAGE).matchingRegex(".*[Cc]ontent.*")
    ;
}
```

Using another syntax you can chain test methods that refer to different pages, as the following example shows:

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

This test is not good because the name of the test is not clear enough.

Individual Pages with Upper and Lower Limit

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

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

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

Page numbers start from "1".

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

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

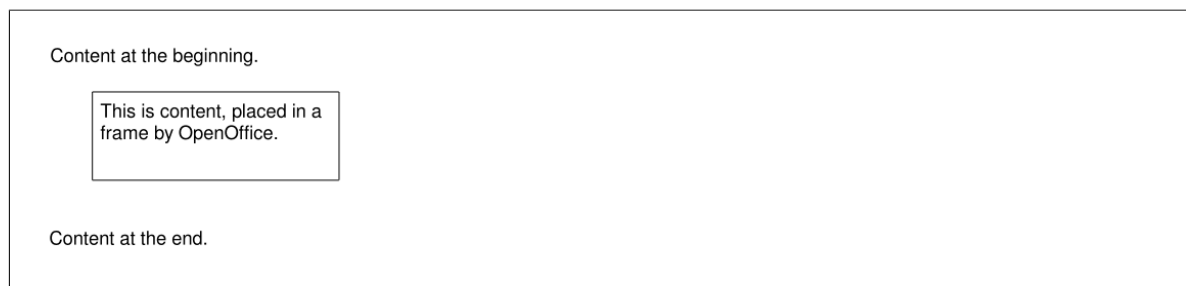
    AssertThat.document(filename)
        .hasText(OnAnyPage.before(99))
        .containing("Content on")
    ;
}
```

Visible Text Order - Potential Problem

The visible sequence of text on a PDF page does not necessarily correspond to the text sequence within the PDF document. This might result in PDFUnit does not recognizing text sequences, but

PDFUnit uses iText's powerful text recognition which assembles text objects based on their positions on a page.

Although the text in the next example is a separate text object in a frame, a test for the text sequence "the beginning. This is content" succeeds:



```
@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)
    ;
}
```

3.26. Text - in Page Sections

Overview

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

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

Example

The following example shows the definition and the usage of a clipping area:

```
// 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 This defines a clipping area. More detailed information is available in chapter 13.7: "Defining Page Areas" (p. 154). The possibility to use measuring units such as `MILLIMETER` is described in chapter 13.8: "Format Units" (p. 155).

- ② In this line the area is passed to the test method.

When comparing text in page areas, all the test methods are available which are available when comparing text on entire pages. These methods are described in section 13.4: “Comparing Text” (p. 149).

A clipping area can also be used when testing images.

3.27. Text - Rotated and Overhead

Overview

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

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

Example

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

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

This is the test using a clipping area:

```
// 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)
    ;
}
```

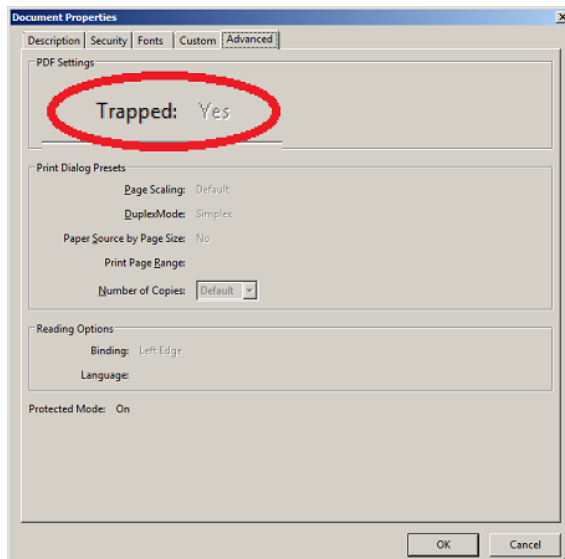
Note that the vertical text is still text with the writing direction LTR (left-to-right). Text with the direction RTL (right-to-left) will be supported by PDFUnit in a future release.

3.28. Trapping Info

Overview

Trapping is described in Wikipedia (http://en.wikipedia.org/wiki/Trap_%28printing%29). In short: Trapping means to define information for the printing process to avoid uncolored paper after printing a multicolored document. The Wikipedia article provides further links to detailed information from Adobe.

A PDF document contains the information about whether it is “trapped” or not. Adobe Reader® shows the property “Trapped” in the document properties dialog:



“Trapped” can have the values Yes, No and Unknown. Only one test method is provided:

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

Examples

```
@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)
    ;
}
```

The parameter is defined as:

```
// Constants for trapping

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


3.29. Version Info

Overview

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

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

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

One Distinct Version

For popular PDF versions constants can be used:

```
// Constants for PDF versions:

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

Here an example to check for version “1.4”

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

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

Ranges of Versions

The existing constants can be used to verify version ranges:

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

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

❶❷ Upper and lower limit values are exclusive.

Also upcoming versions can be tested:

```
@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)
    ;
}
```

3.30. XFA Data

Overview

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

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

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

Existence and Absence of XFA

The first test focuses on the existence of XFA data:

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

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

You can also check that a PDF document does **not contain** XFA data:

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

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

Comparing XFA with an XML File

The XFA data of a document can be extracted into an XML file using the utility `ExtractXFADData`. In a later test the file can be compared with the XFA data of another PDF document:

```
@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 are ignored when comparing XFA data.

Often it makes no sense to compare the complete XFA data from a file with those from a PDF document. So you can test individual XML nodes or use XPath expressions for more detailed tests. Both options are described in the following sections.

Validate Single XML-Tags

The next examples use the following XFA data (extract):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
  ...
  <x:xmpmeta xmlns:x="adobe:meta/"
    x:xmpTk="Adobe XMP Core 4.2.1-c041 52.337767, 2008/04/13-15:41:00"
  >
    <config xmlns="http://www.xfa.org/schema/xci/2.6/">
      ...
      <log xmlns="http://www.xfa.org/schema/xci/2.6/">
        <to>memory</to>
        <mode>overwrite</mode>
      </log>
    </config>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      ...
      <rdf:Description xmlns:xmp="http://ns.adobe.com/xap/1.0/" >
        <xmp:MetadataDate>2009-12-03T17:50:52Z</xmp:MetadataDate>
      </rdf:Description>
    </rdf:RDF>
  </x:xmpmeta>
  ...
</xdp:xdp>
```

To test a particular node, an instance of `com.pdfunit.XMLNode` has to be created. The constructor needs an XPath expression and the expected value of the addressed node:

```
@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 analyzes the XFA data from the current PDF document and determines the namespaces automatically. Only the default namespace has to be specified.

If the XPath expression evaluates to a node set, the first node is used.

When processing the XPath expression PDFUnit internally adds the path element `"/"` to the given XPath expression. For this reason the expression need not contain the document root `"/"`.

Tests on attribute nodes are of course also possible:

```
@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 based XFA Tests

XPath can do more than just identify individual nodes. To take advantage of the full power of XPath, the method `matchingXPath(..)` is provided.

The following two examples give you an idea of what is possible:

```

@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)
        ;
}

```

One limitation has to be mentioned. The evaluation of the XPath expressions depends on the implemented features of the XPath engine you are using. By default PDFUnit uses the JAXP implementation of the your JDK. So the XPath compatibility also depends on the version of your JDK.

The chapter 13.13: “JAXP-Configuration” (p. 158) explains how to use any XPath-Engine, for example from the Xerces-project.

Default Namespaces in XPath

XML namespaces are detected automatically, but the default namespace has to be declared explicitly. Because the XML standard allows multiple declarations of a namespace in a document, it is not automatically clear which default namespace should be used when more than one declaration exists. Therefore, the default namespace must be declared:

```

@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)
        ;
}

```

For the same reason, the default namespace must be given when using XMLNode:

```

/**
 * 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)
        ;
}

```

3.31. XMP Data

Overview

XMP is the abbreviation for “Extensible Metadata Platform”, an open standard initiated by Adobe to embed metadata into files. Not only PDF documents are able to embed data, but also images. For example, metadata can be location and time.

The metadata in a PDF file can be important when processing a document, so they should be correct. PDFUnit provides the same test methods for XMP data as for XFA data:

```

// Methods to test XMP data:

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

```

Existence and Absence of XMP

The following examples show how to verify the existence and absence of XMP data:

```

@Test
public void hasXMPData() throws Exception {
    String filename = 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()
        ;
}

```

Comparing XMP against an XML File

With the utility `ExtractXMPData` you can extract the XMP data from a PDF document into an XML file which can be used later in a test:

```

@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 Whitespaces are ignored when comparing XMP data.

Validate Single XML-Tags

Tests can check a single node of the XMP data and its value. The next example is based on the following XML-snippet:

```

<x:xmpmeta xmlns:x="adobe:ns:meta/">
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    ...
    <rdf:Description rdf:about="" xmlns:xmp="http://ns.adobe.com/xap/1.0/">
      <xmp:CreateDate>2011-02-08T15:04:19+01:00</xmp:CreateDate>
      <xmp:ModifyDate>2011-02-08T15:04:19+01:00</xmp:ModifyDate>
      <xmp:CreatorTool>My program using iText</xmp:CreatorTool>
    </rdf:Description>
    ...
  </rdf:RDF>
</x:xmpmeta>

```

In the example the existence of XML-nodes are validated. The method `withNode(..)` needs an instance of `com.pdfunit.XMLNode` as a parameter:

```

@Test
public void hasXMPData_WithNode_ValidateExistence() throws Exception {
    String filename = 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)
    ;
}

```

When you want to verify the value of a node, you also have to pass the expected value to the constructor of `XMLNode`:

```

@Test
public void hasXMPData_WithNodeAndValue() throws Exception {
    String filename = 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)
    ;
}

```

If an expected node exists multiple times within the XMP data, the first match is used.

The XPath expression may not start with the document root, because PDFUnit adds `//` internally.

Of course, the node may also be an attribute node.

XPath based XMP Tests

With the method `matchingXPath(..)` you can use the full power of XPath:

```
@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)
        ;
}
```

The capability to evaluate XPath expressions depends on the XML parser or more exactly the XPath engine. By default PDFUnit uses the parser in the JDK/JRE. So the capability is vendor dependent.

Chapter 13.13: “JAXP-Configuration” (p. 158) explains how to use any other XML-Parser:

Default Namespaces in XPath

As already described for XFA tests, XML namespaces are detected automatically. But the default namespaces has to be declared by the test because namespaces can occur more than once in an XML document.

The next example shows the default namespaces for an 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)
        ;
}
```

Default namespace for a XMLNode with an expected value:

```
@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)
    ;
}
```


Chapter 4. Comparing a Test PDF with a Master

4.1. Overview

Many tests follow the principle of comparing a newly created test document with a PDF document which has already been validated. Such tests are useful if the process that creates the PDF has to be changed, but the output should be the same.

Initialization

The instantiation of a master document is done with the method `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 If the test document is password protected, the second parameter is needed.
- ❷ If the master document is not password protected, only the filename is needed. Otherwise a password has to be used passed to the method.

Passwords are “only” used to open the documents. They do not influence the tests.

Overview

The following list gives a complete overview of all tests which compare two PDF files. Links after each tag refer to the chapter which describes it in detail. The chapters are sorted alphabetically. The last chapter 4.20: “More Comparisons” (p. 98) collects all the functions which are not described in other chapters.

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

.areBothForFastWebView()           4.20: "More Comparisons" (p. 98)
.haveSameActions()                 4.3: "Comparing Actions" (p. 83)
.haveSameActions(..)               4.3: "Comparing Actions" (p. 83)
.haveSameAppearance(..)            4.12: "Comparing Layout as Rendered Pages" (p. 90)
.haveSameAuthor()                  4.7: "Comparing Document Properties" (p. 87)
.haveSameBookmarks()               4.5: "Comparing Bookmarks" (p. 86)
.haveSameCreationDate()             4.6: "Comparing Date Values" (p. 86)
.haveSameCreator()                  4.7: "Comparing Document Properties" (p. 87)
.haveSameEmbeddedFiles(..)          4.4: "Comparing Attachments" (p. 85)
.haveSameFieldsByName()             4.2: "Comparing Form Fields" (p. 82)
.haveSameFieldsByProperties()        4.2: "Comparing Form Fields" (p. 82)
.haveSameFieldsByValue()            4.2: "Comparing Form Fields" (p. 82)
.haveSameFonts()                   4.8: "Comparing Fonts" (p. 87)
.haveSameFormat()                  4.9: "Comparing Format" (p. 88)
.haveSameFormat(..)                4.9: "Comparing Format" (p. 88)
.haveSameImages()                  4.10: "Comparing Images" (p. 89)
.haveSameImages(..)                4.10: "Comparing Images" (p. 89)
.haveSameJavaScript()               4.20: "More Comparisons" (p. 98)
.haveSameKeywords()                 4.20: "More Comparisons" (p. 98)
.haveSameLanguage()                 4.20: "More Comparisons" (p. 98)
.haveSameLayers()                   4.20: "More Comparisons" (p. 98)
.haveSameText()                     4.17: "Comparing Text" (p. 94)
.haveSameText(..)                   4.17: "Comparing Text" (p. 94)

... continued
```

```

... continuation:

.haveSameModificationDate()      4.6: "Comparing Date Values" (p. 86)
.haveSameNumberOfActions()      4.3: "Comparing Actions" (p. 83)
.haveSameNumberOfBookmarks()    4.5: "Comparing Bookmarks" (p. 86)
.haveSameNumberOfEmbeddedFiles() 4.4: "Comparing Attachments" (p. 85)
.haveSameNumberOfFields()       4.2: "Comparing Form Fields" (p. 82)
.haveSameNumberOfFonts()        4.8: "Comparing Fonts" (p. 87)
.haveSameNumberOfImages()       4.10: "Comparing Images" (p. 89)
.haveSameNumberOfImages(..)     4.10: "Comparing Images" (p. 89)
.haveSameNumberOfLayers()       4.15: "Comparing Quantities of PDF Elements" (p. 93)
.haveSameNumberOfPages()        4.15: "Comparing Quantities of PDF Elements" (p. 93)
.haveSameNumberOfTaggingInfo()  4.15: "Comparing Quantities of PDF Elements" (p. 93)
.haveSamePermissions()          4.14: "Comparing Permissions" (p. 92)
.haveSamePermission(..)         4.14: "Comparing Permissions" (p. 92)
.haveSameProducer()             4.7: "Comparing Document Properties" (p. 87)
.haveSameProperties()            4.7: "Comparing Document Properties" (p. 87)
.haveSameProperty(..)           4.7: "Comparing Document Properties" (p. 87)
.haveSameSignatureNames()       4.16: "Comparing Signature Names" (p. 94)
.haveSameSubject()              4.7: "Comparing Document Properties" (p. 87)
.haveSameTaggingInfo()          4.20: "More Comparisons" (p. 98)
.haveSameTitle()                4.7: "Comparing Document Properties" (p. 87)
.haveSameTrappingInfo()         4.20: "More Comparisons" (p. 98)
.haveSameXFADData()             4.18: "Comparing XFA Data" (p. 95)
.haveSameXMPData()              4.19: "Comparing XMP Data" (p. 97)

```

4.2. Comparing Form Fields

Quantity

The first and simplest test is to check that a document has the same number of form fields as the master document:

```

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

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

```

Field Names

The next test checks that the number of fields and their names are equal in both PDF documents:

```

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

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

```

Field Properties

If you want to compare the fields of two documents including all properties but not the content the following method can be used:

```

@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()
    ;
}

```

By the way, all field properties can be extracted into an XML file using the utility program `Extract-FieldsInfo`, see chapter 9.3: “Extract Field Information to XML” (p. 113). This file can be analyzed later.

Field Content

And finally you can compare the content of form fields of two documents using the method `haveSameFieldsByValue()`. The test fails when a field have different contents in the two files:

```
@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 are “normalized”, see chapter 13.5: “Whitespace Processing” (p. 150).

Concatenated Tests

Multiple and different test methods can be concatenated in one test, but such a test is not recommended because it's hard to find a good name:

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

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameFieldsByProperties()
        .haveSameFieldsByValue()
        .haveSameFonts()
        .haveSameTitle()
    ;
}
```

4.3. Comparing Actions

Quantity

The first example shows how to compare the number of actions of two PDF documents:

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

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

Properties

To compare all actions of two PDF documents, you can use the method `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()
    ;
}

```

The decision whether two actions are equal depends on their type. The following table shows the properties for each type of action which determine whether the actions are equal:

Type	Relevant Property for equals()	
GotoAction	destination	The destination the action is pointing to.
	orientation	The orientation of the destination, for example “/FIT”.
GotoEmbeddedAction	destination	The destination the action is pointing to.
	new window	Whether the destination is shown in a separate window.
GotoRemoteAction	filename	The file the action wants to go to.
	page number	The page in the remote file.
	remote destination	A destination inside the remote file.
	new window	Whether the destination is shown in a separate window.
ImportDataAction	filename	The file the action wants to import.
JavaScriptAction	javaScript	The JavaScript code. Whitespaces are reduced as described in chapter 13.5: “Whitespace Processing” (p. 150).
LaunchAction	filename	The file the action wants to launch.
	default directory	The directory of the file.
	operation	The operation for the filename, for example “print”.
	parameters	Parameters passed to the application.
NamedAction	name	The name of the action.
ResetFormAction	fields	The names of the field which will be reset.
	flags	Settings specifying characteristics of the field.
SubmitFormAction	destination	The destination the field information will be sent to.
	fields	The fields whose values will be transmitted.
	flags	Settings defining the data transfer mechanism. For example PdfAction.SUBMIT_HTML_GET or PdfAction.SUBMIT_PDF.
URIAction	destination	Where the action is pointing to.

The following events are always related to JavaScript actions:

- document close (/DC)

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

The event “document open” (/DocumentOpen) can be linked to any action in the list. The equality of two “document open” actions depends on their actual type.

The whitespace handling can be defined when comparing actions:

```
@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)
    ;
}
```

4.4. Comparing Attachments

Quantity

Compare the number of attachments in two documents:

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

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

Name and Content

There is a parameterized test method to compare the attachments by name or by content:

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

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameEmbeddedFiles(COMPARSED_BY_NAME)
        .haveSameEmbeddedFiles(COMPARSED_BY_CONTENT) ❶
    ;
}
```

- ❶ The attachments are compared byte-by-byte. So two files of any type can be compared.

The two constants are defined in `com.pdfunit.Constants`:

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

com.pdfunit.Constants.COMPARSED_BY_CONTENT
com.pdfunit.Constants.COMPARSED_BY_NAME
```

You can use the utility `ExtractEmbeddedFiles` to extract the attachments. See chapter 9.4: “Extract Attachments” (p. 114).

4.5. Comparing Bookmarks

Quantity

The simplest comparison related to bookmarks is to compare the number of bookmarks in two documents:

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

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

Bookmarks with Properties

Next, the bookmarks and their properties are compared. Bookmarks of two PDF documents are “equal” if the following attributes have the same values:

- 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();
}
```

If you are uncertain about the bookmarks, all bookmark data can be extracted into an XML file using the utility `ExtractBookmarks`. This file can easily be analyzed. See chapter 9.5: “Extract Bookmarks to XML” (p. 116).

4.6. Comparing Date Values

It rarely makes sense to compare date values of two PDF documents, but if it is really needed, you can use the following methods:

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

In the next example, the modification dates of two documents are compared:

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

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

Comparing two dates is always carried out with the time resolution `DateResolution.DATE`.

4.7. Comparing Document Properties

You might want to compare the title or other document information of two PDF documents. Use the following methods:

```
// Comparing document properties:

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

As an example of comparing any document property we compare the “author”:

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

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

Custom properties can be compared using the method `haveSameProperty(...)`:

```
@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")
    ;
}
```

Of course, you can use this method to compare all standard properties.

If you want to compare **all** properties of two documents, you can use the general method `haveSameProperties()`:

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

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

4.8. Comparing Fonts

Quantity

It is simple to compare the number of fonts in two documents:

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

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

Font Properties

Fonts of two PDF documents are equal if all font information contains identical values.

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

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

Information about all fonts of a PDF document can be extracted using the utility `ExtractFontsInfo`. See chapter 9.6: “Extract Font Information to XML” (p. 117).

4.9. Comparing Format

Two documents have the same page format if width and height of all pages have the same values. The tolerance defined by ISO 216 is taken into account.

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

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

The comparison can be restricted to selected pages:

```
@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));
}
```

All possibilities to select pages are explained in chapter 13.3: “Page Selection” (p. 147).

4.10. Comparing Images

Quantity

The first test compares the number of images in two PDF documents:

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

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

The comparison can be limited to selected pages:

```
@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)
    ;
}
```

All possibilities to select pages are described in chapter 13.3: "Page Selection" (p. 147).

Content of Images

The images stored in a test PDF can be compared with those of a master PDF. They are identified as equal when they are equal byte-by-byte.

```
/**
 * The method haveSameImages() does not consider the order of the images.
 */
@Test
public void haveSameImages() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

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

This test method does not care about which pages the images appear on or how often they are used.

You can restrict the comparison of images to individual pages:

```
@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)) ❷
    ;
}
```

❶❷ The order of images is irrelevant for the comparison.

If there are any doubts about the images in a PDF document all images can be extracted using the utility `ExtractImages`. See chapter 9.7: “Extract Images from PDF” (p. 118).

4.11. Comparing JavaScript

Two PDF documents can have the “same” JavaScript. The comparison is done byte-wise using the method `haveSameJavaScript()`. Whitespaces are ignored.

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

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

If you want to see the JavaScript code, you can extract it with the utility `ExtractJavaScript` as described in chapter 9.8: “Extract JavaScript to a Text File” (p. 119).

4.12. Comparing Layout as Rendered Pages

PDF documents can be compared as rendered images (PNG) to ensure that a test document and a master document have the same layout.

```
@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)
    ;
}
```

You can select individual pages in many ways. All possibilities are described in chapter 13.3: “Page Selection” (p. 147).

You can compare the layout of entire pages or you can restrict the comparison to sections of a page:

```

@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)
    ;
}

```

A clipping area can be defined using the measuring units POINTS, MILLIMETER, CENTIMETER, INCH and DPI72. All constants are declared in the class `com.pdfunit.Constants`. If you omit a measuring unit, the values are taken as MILLIMETER.

In case of a test error, PDFUnit creates a **diff image**.



The diff image contains the name of the test in the header. And the name of the diff image is shown in the error message. That allows a cross reference between test and diff image.

Type
<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>

The name of the diff-image file is built as follows:

- The first part is the full name of the test file.
- If the test file is a stream, the name starts with the string “_pdfunit_stream_”. If it is a byte array, the name starts with “_pdfunit_bytearray_”. Both strings are extended with a random number.
- The second part is a formatted date in the format “yyyyMMdd-HH:mm:ssSSS”.

- The last part of the file name is the string “.out.png”.

With the default configuration a diff-image is stored in the same directory as where the test file is located. Diff-images for streams and byte arrays are stored in the home directory of the current Java process. You can change these folders in the file `config.properties`.

If you need to analyze the layout of two PDF documents after a test fails, you can use the very powerful program `DiffPDF`. Information about this Open-Source application by Mark Summerfield is available on the project site <http://soft.rubypdf.com/software/diffpdf>. You can install the program under Linux with a package manager, for example `apt-get install diffpdf`. On Windows you can use it as a “portable application” available from http://portableapps.com/apps/utilities/diffpdf_portable.

4.13. Comparing Named Destinations

“Named Destinations” are seldom a test goal, because until now no test tool is been available which could compare named destinations. With PDFUnit you can verify that two documents have the same named destinations.

Quantity

A simple test is to compare the number of “Named Destinations” in two documents:

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

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

Names and Internal Position

If the names of “Named Destinations” and their PDF-internal positions have to be equal for two documents, the following test can be used:

```
@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. Comparing Permissions

PDFUnit can compare the permissions of two PDF documents. The following example compares **all permissions**:

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

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

If you want to compare a **single permission** you can parameterize the test method with predefined constants:

```
@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)
    ;
}
```

The following constants are available:

```
// 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 The permissions ALLOW_EXTRACT_CONTENT and ALLOW_SCREENREADERS are equivalent.

4.15. Comparing Quantities of PDF Elements

The number of various items in a test document can be compared with the number of the same items in a master document.

Even if some of these tests are already described in other chapters, the following list gives an overview of all comparing methods for countable components:

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

.haveSameNumberOfActions()
.haveSameNumberOfBookmarks()
.haveSameNumberOfEmbeddedFiles()
.haveSameNumberOfFields()
.haveSameNumberOfFonts()
.haveSameNumberOfImages()
.haveSameNumberOfImages(..)
.haveSameNumberOfLayers()
.haveSameNumberOfPages()
.haveSameNumberOfTaggingInfo()
```

Here are some examples which are not shown in other chapters:

```
@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()
    ;
}
```

4.16. Comparing Signature Names

When comparing signatures of two PDF documents using the PDFUnit release 2054.10, only the names of the signatures are compared:

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

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

Further comparisons will be possible in future releases.

Use the `ExtractSignaturesInfo` to extract details of signatures and certificates. See chapter 9.10: “Extract Signature Information to XML” (p. 121) for more information.

4.17. Comparing Text

PDFUnit can compare text on any page of a test document with the corresponding page of a master document. The following simple example shows how to do this (please note that whitespaces are ignored):

```
@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)
    ;
}
```

You can restrict the test to selected pages which is explained in chapter 13.3: “Page Selection” (p. 147):

```
@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)
    ;
}
```

And you can restrict the comparison to a section of a page:

```

@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)
        ;
}

```

The treatment of white space can be controlled in the same kind as in other tests:

```

@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. Comparing XFA Data

It does not make sense to compare the entire XFA data of two PDF documents, because often they contain a creation date and a modification data. For this reason PDFUnit provides an XPath based test method for XFA data.

Overview

Only one powerful method is provided:

```

// Method for tests with XMP data
.haveSameXFAData().matchingXPath(XPathExpression, RESULTTYPE)

```

If you are in doubt about the XFA data, you can extract it with the utility `ExtractXFAData` into an XML file. For more information see chapter 9.11: “Extract XFA Data to XML” (p. 122).

Example - Resulttype Node

The result of the XPath expression has to be the same for both PDF documents:

```

@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/xfatemplate/2.6/");
    XPathExpression expression
        = new XPathExpression("//default:pageSet", defaultNS);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameXFAData()
        .matchingXPath(expression, AS_RESULTTYPE_NODE)
        ;
}

```

As you can see in the example the type of the expected result has to be specified. Possible result types are declared as constants:

```
// 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 are ignored when comparing XFA data.

PDFUnit throws an exception if two documents without any XFA data are compared. It makes no sense to compare things that don't exist.

Example - Resulttype Boolean

The XPath-expressions may contain XPath functions:

```
@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/xfatemplate/2.6/");
    XPathExpression expression
        = new XPathExpression("count(//default:field) = 3", defaultNS);

    AssertThat.document(filenameTest)
        .and(filenameMaster)
        .haveSameXFADData()
        .matchingXPath(expression, AS_RESULTTYPE_BOOLEAN)
    ;
}
```

Tests with an expected result type BOOLEAN are a problem because PDFUnit can't differentiate between "not found" and "false".

A detailed description of how to work with XPath in PDFUnit can be found in chapter 8: "Using XPath" (p. 110).

Example - Default Namespace

Be careful using the default namespace. It has to be declared in the attribute `<haveSameXFADData />`. (Because you can declare namespaces multiple times it could be ambiguous to detect the default namespace automatically.)

You can use the method `matchingXPath(..)` more than once in a test. But it would be better to split the following test:


```

/**
 * Test with multiple XPath expressions and multiple default namespaces.
 * It is not recommended to create tests in this way.
 */
@Test
public void haveSameXFAData_MultipleDefaultNamespaces() throws Exception {
    String filenameTest = PATH + "test/test.pdf";
    String filenameMaster = PATH + "master/master.pdf";

    String nsStringXFATemplate = "http://www.xfa.org/schema/xfatemplate/2.6/";
    String nsStringXFALocale = "http://www.xfa.org/schema/xfatemplate/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)
        .haveSameXFAData()
        .matchingXPath(exprXFATemplate, AS_RESULTTYPE_BOOLEAN)
        .matchingXPath(exprXFALocale, AS_RESULTTYPE_BOOLEAN)
    ;
}

```

4.19. Comparing XMP Data

You can compare the XMP data in two PDF documents using XPath. Comparing XFA and XMP data are basically the same. And because the previous chapter 3.30: “XFA Data” (p. 74) describes the tests in detail, this section is very short.

If you are in doubt about the structure and values of the XMP data you can extract them with the program `ExtractXMPData`. See chapter 9.12: “Extract XMP Data to XML” (p. 123).

Overview

PDFUnit provides the following method:

```

// Method for tests with XMP data:

.haveSameXMPData().matchingXPath(XPathExpression, RESULTTYPE)

```

Example

```

@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)
    ;
}

```

The XPath result types are the same as for XFA tests.

The XPath expression may also contain XPath functions.

If the XMP data is compared in two documents, but neither contains XMP data, PDFUnit throws an exception.

4.20. More Comparisons

In the previous chapters a lot of examples show how to compare various parts of two PDF documents. But PDFUnit provides more methods for comparing PDF. The following list shows the remaining methods. They should be self explanatory:

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

.areBothForFastWebView()
.haveSameKeywords()
.haveSameLanguage()
.haveSameLayers()
.haveSameTrappingInfo()
.haveSameTaggingInfo()
```

Here are two examples:

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()
    ;
}
```

Concatenation of Test Methods

All methods can be concatenated:

```
@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()
    ;
}
```

It's hard to find a good name for this test. So it's better to write three smaller tests.

Chapter 5. Tests with Multiple Documents

A test with more than one PDF document

The following code shows how to use multiple documents in one test. The test stops at the first detected error.

```
@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)
    ;
}
```

For such tests almost all test methods are available, which exist for tests with a single PDF document. The following list shows the available methods. Links refer to the description of each test.

```
// Methods to validate a set of PDF documents:
.containsImage(...)           3.13: "Images in PDF Documents" (p. 45)
.containsOneImageOf(...)      3.13: "Images in PDF Documents" (p. 45)

.hasAuthor()                  3.7: "Document Properties" (p. 27)
.hasBookmark()                3.4: "Bookmarks and Named Destinations" (p. 21)
.hasBookmarks()               3.4: "Bookmarks and Named Destinations" (p. 21)
.hasEncryptionLength(...)     3.21: "Passwords" (p. 57)
.hasField(...)                3.10: "Form Fields" (p. 34)
.hasFields()                  3.10: "Form Fields" (p. 34)
.hasFont()                    3.9: "Fonts" (p. 30)
.hasFonts()                   3.9: "Fonts" (p. 30)
.hasFormat(...)               3.12: "Format" (p. 43)
.hasJavaScript()              3.14: "JavaScript" (p. 48)
.hasKeywords()                3.7: "Document Properties" (p. 27)
.hasLocale(...)               3.15: "Language" (p. 49)

.hasNumberOf...()             3.19: "Number of PDF Elements" (p. 55)

.hasNoAuthor()                3.7: "Document Properties" (p. 27)
.hasNoKeywords()              3.7: "Document Properties" (p. 27)
.hasNoLocale()                3.15: "Language" (p. 49)
.hasNoProducer()              3.7: "Document Properties" (p. 27)
.hasNoProperty()              3.7: "Document Properties" (p. 27)
.hasNoSubject()               3.7: "Document Properties" (p. 27)
.hasNoText()                  3.25: "Text" (p. 65)
.hasNoTitle()                 3.7: "Document Properties" (p. 27)
.hasNoXFADData()              3.30: "XFA Data" (p. 74)
.hasNoXMPData()               3.31: "XMP Data" (p. 77)
.hasOwnerPassword(...)        3.21: "Passwords" (p. 57)
.hasPermission()              3.22: "Permissions" (p. 58)
.hasProperty(...)             3.7: "Document Properties" (p. 27)
.hasSignature(...)            3.23: "Signatures and Certificates" (p. 59)
.hasSignatures()              3.23: "Signatures and Certificates" (p. 59)
.hasSignedSignatureFields()    3.23: "Signatures and Certificates" (p. 59)
.hasSubject()                 3.7: "Document Properties" (p. 27)

... continued
```

```
... continuation:

.hasText(..)                3.25: "Text" (p. 65)
.hasTitle()                 3.7: "Document Properties" (p. 27)
.hasUnsignedSignatureFields() 3.23: "Signatures and Certificates" (p. 59)
.hasUserPassword(..)         3.21: "Passwords" (p. 57)
.hasVersion()                3.29: "Version Info" (p. 73)
.hasXFADData()               3.30: "XFA Data" (p. 74)
.hasXMPData()                3.31: "XMP Data" (p. 77)

.isCertified()               3.5: "Certified PDF" (p. 24)
.isCertifiedFor(..)          3.5: "Certified PDF" (p. 24)
.isLinearizedForFastWebView() 3.8: "Fast Web View" (p. 30)
.isSigned()                  3.23: "Signatures and Certificates" (p. 59)
.isTagged()                  3.24: "Tagged Documents" (p. 64)

... (end of list)
```

If you are looking for more test methods please send your request to [info\[at\]pdfunit.com](mailto:info[at]pdfunit.com).

Chapter 6. PDFUnit-Monitor

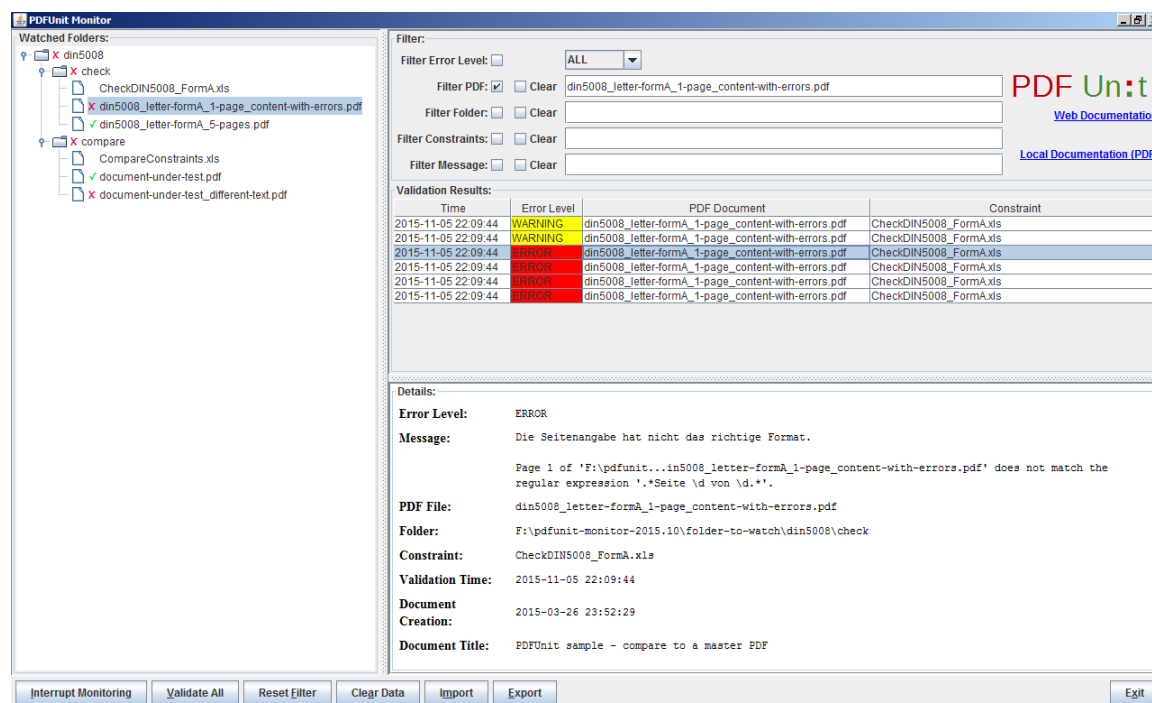
The PDFUnit-Monitor is an application that shows the result of PDFUnit tests. Tests are written in Excel files, so non-developers can create and run the tests.

The functional scope of the PDFUnit-Monitor is large. Therefore a detailed description of it exists as a separate file, also a demonstration video is available. Both can be downloaded with this link (download). The separate documentation provides also information about the installation and configuration of the PDFUnit-Monitor. The following sections briefly describe the main features.

Monitored Folders

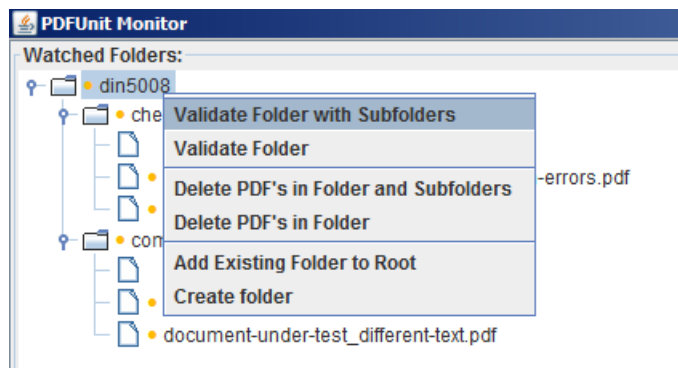
The PDFUnit-Monitor monitors all PDF documents in a defined directory and its subdirectories. It checks the documents against rules which are read from Excel files. The Excel files have to lay also in the monitored directories. If new PDF documents were copied into the monitored directories, the tests started automatically. A manual start is not necessary but can be done. If a PDF document complies with all rules, its name in the folder tree is decorated with a green checkmark. When a PDF test fails, all violations will be listed. Additionally its name is decorated with a red cross. This status is transferred to the directory name. The name of a folder is decorated with a green checkmark only, when the folder itself and all subdirectories contain valid PDF documents. Otherwise the folder is decorated with a red cross.

The following picture shows the PDFUnit-Monitor. On the left side the folder structure with PDF and Excel files can be seen. The right side shows the validation results in the upper half and details of a selected message in the lower half.



A double click on a PDF or Excel document in the folder structure or in the error list opens the standard application of the operating system for the document.

Each element of the folder structure provides a context menu with various functions. The following figure shows an example:



Overview of Test Results with Filter Options

The Monitor shows all validation results as a list in the upper part of the right side. Each constraint validation is one entry in the list. The details of a validation will be shown in the lower part of the right side when a list entry is selected.

Filter:

Filter Error Level: ☐ ALL

Filter PDF: ☐ clear

Filter Folder: ☒ clear din5008

Filter Constraints: ☒ clear FormA

Filter Message: ☐ clear

Validation Results:

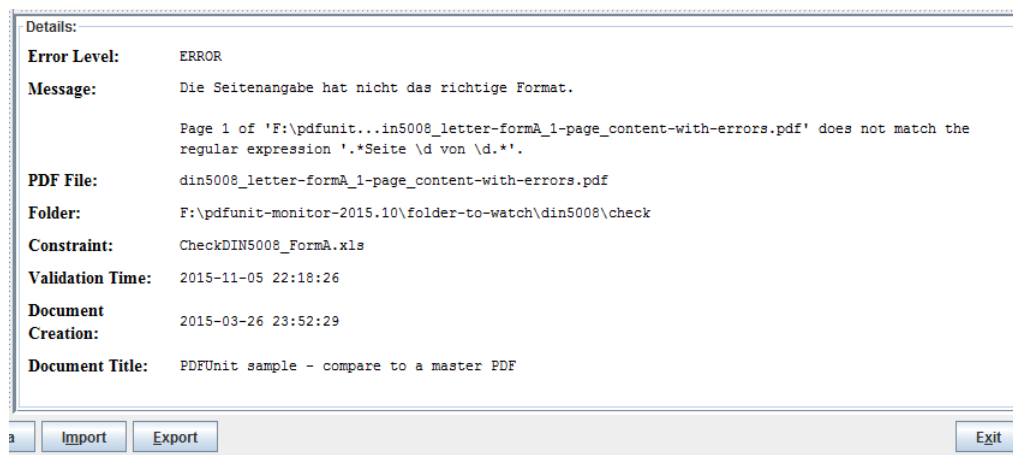
Time	Error Level	PDF Document	Constraint
2015-10-26 01:22:12	OK	din5008_letter-formA_5-pages.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	OK	din5008_letter-formA_5-pages.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:12	WARNING	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	WARNING	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:12	ERROR	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	ERROR	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	ERROR	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	WARNING	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:12	WARNING	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:12	ERROR	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:12	ERROR	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:12	ERROR	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:12	WARNING	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	WARNING	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:11	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:11	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:11	WARNING	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:11	WARNING	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA_general.xls

The error list can be filtered by the names of PDF documents, folders, Excel files and regular expressions on error messages. The folder structure on the left side is connected with the filters on the right side. Each time, when a folder or a document is selected in the structure, a corresponding filter is set.

When a cell with a PDF or Excel document is double-clicked, the standard application of the operating system for that document type starts.

Details of an Error

When a row of the error list is selected all details of that entry will be shown in the lower part of the right side of the Monitor.

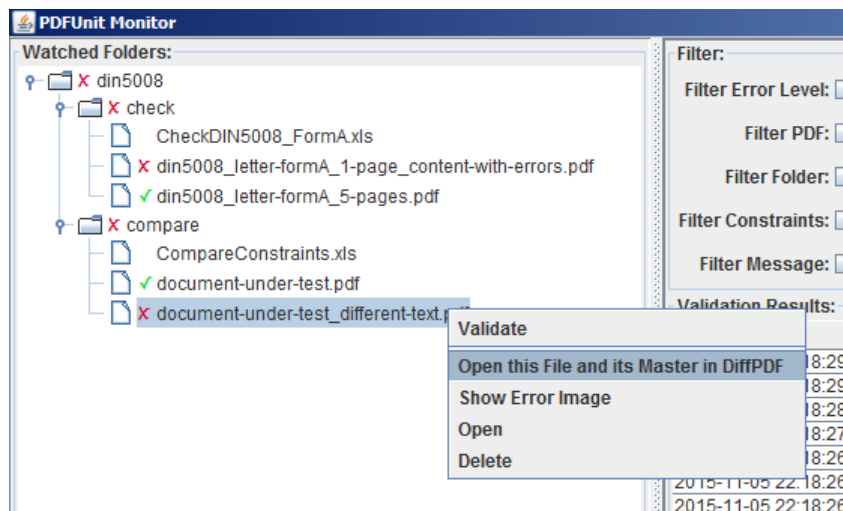


The first part of the error message was read from the Excel file. That part is designed by the person who created the tests. The next part of the message comes from PDFUnit. Additionally to the error message itself useful information about the PDF document, the constraint file and the execution time are provided.

The error messages of PDFUnit are currently in English. They can be provided in other languages with a little work, when requested.

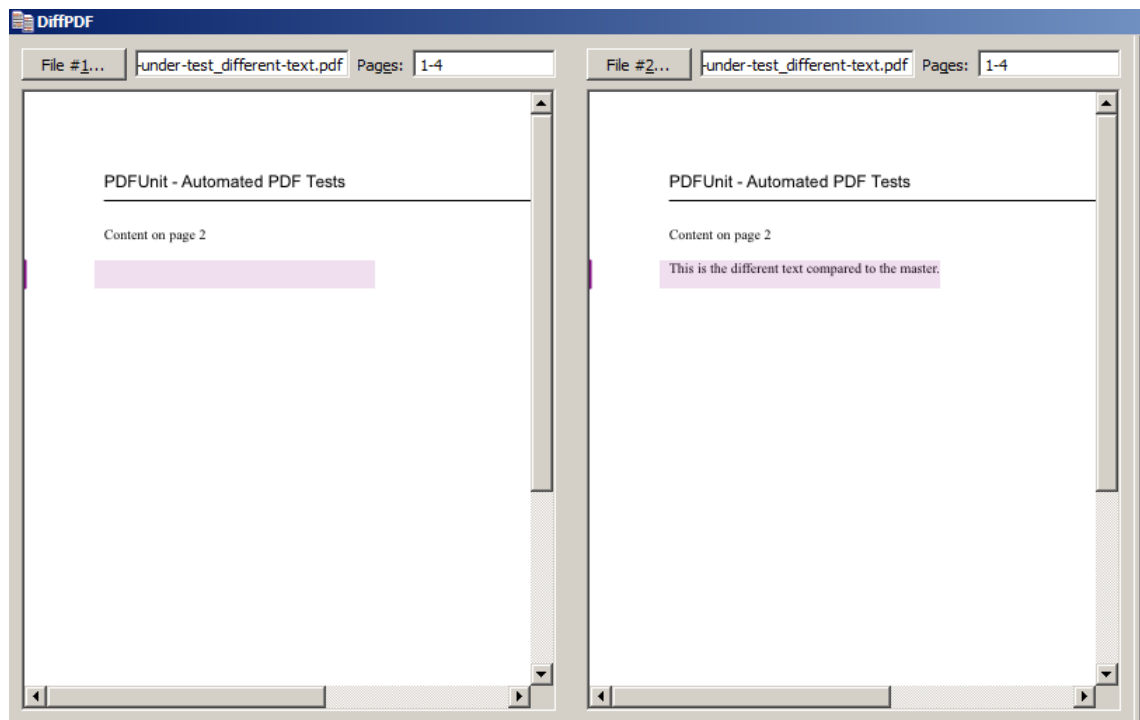
Comparing a PDF with a Master-PDF

PDF document can be compared to a master PDF. The rules for the comparison are read from Excel files. When the PDFUnit-Monitor detects a difference between the test and the master PDF the name of the test document will be decorated with a red cross. Then the program 'DiffPDF 1.5.1, portable' can be started by clicking the right mouse button.



The program was created by Mark Summerfield and is available as a 'portable app' from this link (download). DiffPDF can be used in English, German, French and Czech. Many thanks to all who are involved for their work and the great result.

The next image shows the application DiffPDF just after it was started from the PDFUnit-Monitor. The left side shows the master PDF and the right side the current test PDF. The application point directly to the first difference, in this case to page 2. The differences are marked with a coloured background. The image does not show the buttons to navigate from one mismatch to the next.



Export and Import of Test Results

The test results can be exported as XML over the button 'Export'. So they can be saved for documentation. With XSLT stylesheets the exported files can be converted into HTML reports. With a click on the button 'Import' the can loaded again into the monitor.

Internationalization

The PDFUnit monitor is currently available for the languages German and English. An extension to other languages is structurally prepared and can be realized on demand with little effort.

Chapter 7. Unicode

PDF Documents Containing Unicode

Would the tests described so far also run with content that is not ISO-8859-1, for example with Russian, Greek or Chinese text?

A difficult question. A lot of internal tests are done with Greek, Russian and Chinese documents, but tests are missing for Hebrew and Japanese documents. All in all it is not 100% clear that every available test will work with every language, but it should.

When you need to process Unicode data, it is good practice to configure all your tools to UTF-8.

The following hints may solve problems not only when working with UTF-8 files under PDFUnit. They may also be helpful in other situations.

Single Unicode Characters

Metadata and keywords can contain Unicode characters. If your operating system does not support fonts for foreign languages, you can use Unicode escape sequences in the format `\uXXXX` within strings. For example the copyright character “©” has the Unicode sequence `\u00A9`:

```
@Test
public void hasProducer_CopyrightAsUnicode() throws Exception {
    String filename = PATH + "unicode/unicode_producer.pdf";

    AssertThat.document(filename)
        .hasProducer()
        .matchingComplete("txt2pdf v7.3 \u00A9 SANFACE Software 2004") // 'copyright'
    ;
}
```

Longer Unicode Text

It would be too difficult to figure out the hex code for all characters of a longer text. Therefore PDFUnit provides the small utility `ConvertUnicodeToHex`. Pass the foreign text as a string to the tool, run the program and place the generated hex code into your test. Detailed information can be found in chapter 9.2: “Convert Unicode Text into Hex Code” (p. 112). A test with a longer sequence may look like this:

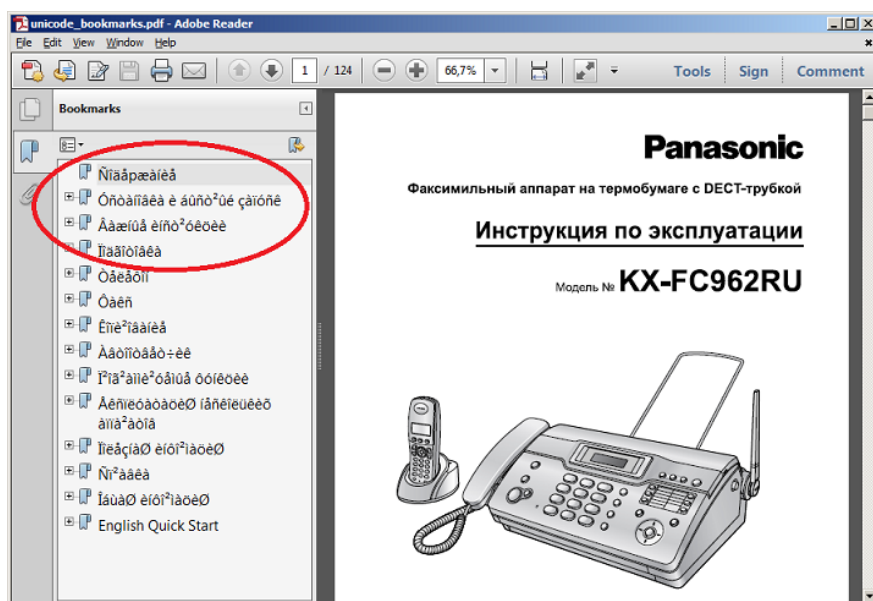
```
@Test
public void hasSubject_Greek() throws Exception {
    String filename = PATH + "unicode/unicode_subject.pdf";
    String expectedSubject = "##### ## ### ##### / #####"; ❶
    //String expectedSubject = "\u0395\u03C1\u03B3\u03B1\u03C3\u03C4\u03AE"
    //                          + "\u03C1\u03B9\u03BF \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 If you don't see Greek text here, then your presentation system (PDF, eBook or HTML) does not support the required Unicode font.

Unicode Content Compared with XML Files

XML and XPath based tests use XML files, which might contain Unicode data, e.g. the bookmarks extracted from the following document:



Although the bookmarks could not be read clearly by Adobe Reader® (they are also not readable in the XML file) the comparison works, because Java is Unicode-enabled:

```
/**
 * 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 The codepage can be set using the environment variable “file.encoding”.
- ❷ The bookmarks were exported to XML by the utility `ExtractBookmarks`.

Using Unicode within XPath Expressions

The chapter 8: “Using XPath” (p. 110) describes how to use XPath in PDFUnit tests. You can also use Unicode sequences in XPath expressions:

```
@Test
public void hasBookmarks_MatchingXPath() throws Exception {
    String filename = PATH + "unicode/unicode_bookmarks.pdf";
    // String xpath = "//Title[@Action][.='1.4 Nĩāāēīāīē']";
    String xpath = "//Title[@Action]"
        + "[.='\\u00D1\\u00EE\\u00E4\\u00E5p\\u00E6\\u00E0\\u00ED\\u00E8\\u00E5']";

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

File Encoding UTF-8 for Shell Scripts

Pay special attention to data read from the file system. Its byte representation depends on the encoding of file system. Every Java program that processes files depends on the environment variable `file.encoding`.

There are multiple possibilities to set the environment for the current shell:

```
set _JAVA_OPTIONS=-Dfile.encoding=UTF8
set _JAVA_OPTIONS=-Dfile.encoding=UTF-8

java -Dfile.encoding=UTF8
java -Dfile.encoding=UTF-8
```

File Encoding UTF-8 for ANT

During the development of PDFUnit there were two tests which ran successfully under Eclipse, but failed with ANT due to the current encoding.

The following command did **not** solve the encoding problem:

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

Instead, the property had to be set using JAVA_TOOLS_OPTIONS:

```
// Used when developing PDFUnit:
set JAVA_TOOL_OPTIONS=-Dfile.encoding=UTF-8
```

Maven - Settings in pom.xml for UTF-8

You can configure “UTF-8” in many places in the 'pom.xml'. The following code snippets show some examples. Choose the right one for your problem:

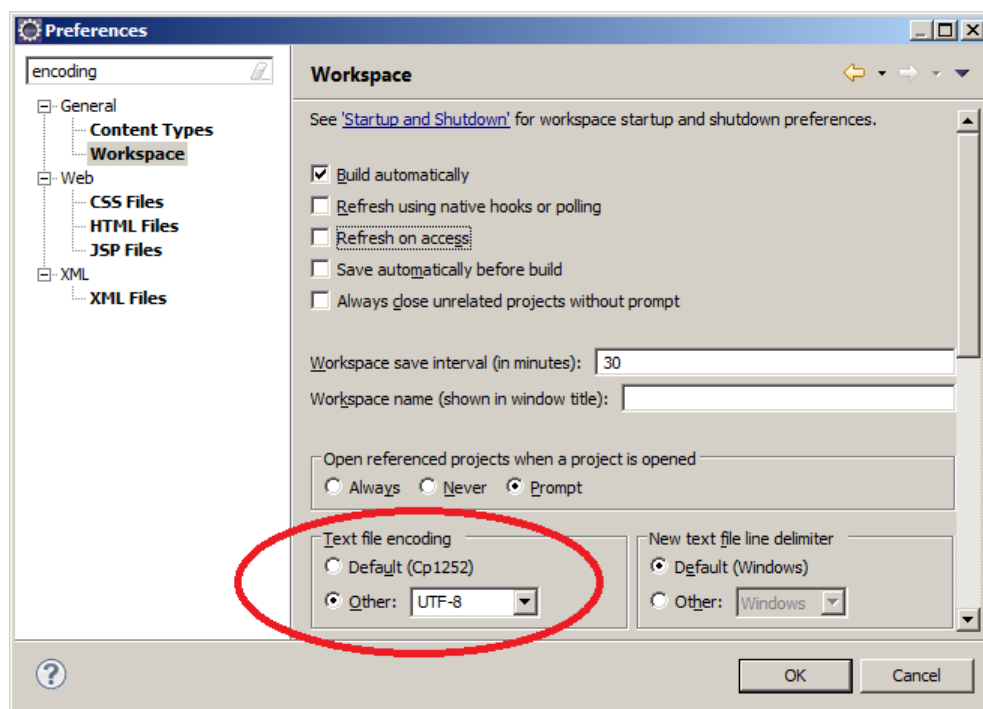
```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.5.1</version>
  <configuration>
    <encoding>UTF-8</encoding>
  </configuration>
</plugin>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <encoding>UTF-8</encoding>
  </configuration>
</plugin>
```

Configure Eclipse to UTF-8

When you are working with XML files in Eclipse, you do not need to configure Eclipse for UTF-8, because UTF-8 is the default for XML files. But the default encoding for other file types is the encoding of the file system. So it is recommended to set the encoding for the entire workspace to UTF-8:



This default can be changed for each file.

Unicode in Error Messages

If tests of Unicode content fail, the error message may be presented incorrectly in Eclipse or in a browser. Again the file encoding is responsible for this behaviour. Configuring ANT to “UTF-8” should solve all your problems. Only characters from the encoding “UTF-16” may corrupt the presentation of the error message.

The PDF document in the next example includes a layer name containing UTF-16BE characters. To show the impact of Unicode characters in error messages, the expected layer name in the test is intentionally incorrect to produce an error message:

```
/**
 * The name of the layers consists of UTF-16BE and contains the
 * byte order mark (BOM). The error message is not complete.
 * It was corrupted by the internal Null-bytes.
 */
@Test
public void hasLayer_NameContainingUnicode_UTF16_ErrorIntended() throws Exception {
    String filename = 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\u0000E\u0000b\u0000e\u0000n\u0000e\u0000 \u0000 _XXX";

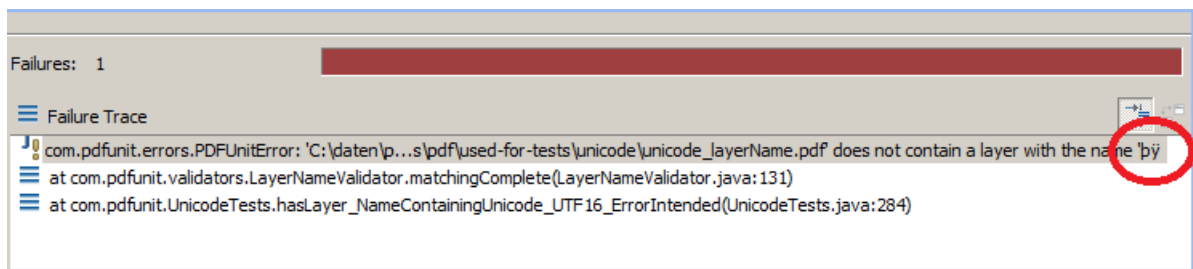
    AssertThat.document(filename)
        .hasLayer()
        .matchingComplete(wrongNameWithUTF16BE);
}
```

When the tests are executed with ANT, a browser shows the complete error message including the trailing string `þÿEbene _XXX`:

'C:\daten\p...s\pdf\used-for-tests\unicode\unicode_layerName.pdf' does not contain a layer with the name 'þÿEbene _XXX'.

```
junit.framework.AssertionFailedError: 'C:\daten\p...s\pdf\used-for-tests\unicode
\unicode_layerName.pdf' does not contain a layer with the name 'þÿEbene _XXX'.
at com.pdfunit.validators.LayerNameValidator.matchingComplete(LayerNameValidator.java:133)
at com.pdfunit.UnicodeTests.hasLayer_NameContainingUnicode_UTF16_ErrorIntended(UnicodeTests.java:274)
```

But the JUnit-View in Eclipse cuts the error message after the internal Byte-Order-Mark. The message '`... \unicode_layerName.pdf`' does not contain a layer with the name '`þÿ`' should end with `Ebene_XXX`:



Unicode for invisible Characters -

A problem can occur due to a “non-breaking space”. Because at first it looks like a normal space, the comparison with a space fails. But when using the Unicode sequence of the “non-breaking space” (`\u00A0`) the test runs successfully. Here's the test:

```
@Test
public void nodeValueWithUnicodeValue() throws Exception {
    String filename = 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)
    ;
}
```

Chapter 8. Using XPath

General Comments about XPath in PDFUnit

Using XPath to evaluate parts of a PDF document opens a wider range of testing capabilities than an API alone can provide.

Several chapters in this manual describe XPath tests. The current chapter gives you an overview with references to the special chapters.

```
// Overview over XPath methods:
.hasBookmarks().matchingXPath(..)      3.4: "Bookmarks and Named Destinations" (p. 21)
.hasFields().matchingXPath(..)         3.10: "Form Fields" (p. 34)
.hasFonts().matchingXPath(..)          3.9: "Fonts" (p. 30)
.hasSignatures().matchingXPath(..)     3.23: "Signatures and Certificates" (p. 59)
.hasXFADData().matchingXPath(..)       3.30: "XFA Data" (p. 74)
.hasXMPData().matchingXPath(..)        3.31: "XMP Data" (p. 77)

// Comparing two documents using XPath:
.haveSameXFADData().matchingXPath(..)  4.18: "Comparing XFA Data" (p. 95)
.haveSameXMPData().matchingXPath(..)   4.19: "Comparing XMP Data" (p. 97)
```

Internally PDFUnit also uses XPath for other methods.

Using XMLUnit for Comparison

PDFUnit uses XMLUnit internally to compare XML structures (<http://xmlunit.sourceforge.net>). This means that the rules of XML syntax are respected, for example:

- The order of attributes doesn't matter.
- Whitespaces between element nodes are ignored.

More rules for "Canonical XML" are well described in Wikipedia (http://de.wikipedia.org/wiki/Canonical_XML).

The general configuration of XMLUnit is documented on the project site <http://xmlunit.sourceforge.net/userguide/html/index.html#Configuring%20XMLUnit>. PDFUnit uses the following:

```
XMLUnit.setXSLTVersion("2.0");
XMLUnit.setNormalizeWhitespace(true);
XMLUnit.setIgnoreWhitespace(true);
XMLUnit.setIgnoreAttributeOrder(true);
XMLUnit.setIgnoreComments(true);
```

Extract Data as XML

PDFUnit provides utility programs for all parts of a PDF document which can be tested using XML/XPath. They extract the information into XML files:

```
// Utilities to extract XML from PDF:

com.pdfunit.tools.ExtractBookmarks
com.pdfunit.tools.ExtractFieldsInfo
com.pdfunit.tools.ExtractFontsInfo
com.pdfunit.tools.ExtractSignaturesInfo
com.pdfunit.tools.ExtractXFADData
com.pdfunit.tools.ExtractXMPData
```

The utilities are described in the chapter 9.1: "Common Remarks for all Utilities" (p. 112):

Namespaces with Prefix

A namespace with an existing prefix will be detected automatically by PDFUnit. This applies to both XML files and PDF-internal XML data.

Default Namespace

The default namespace is not detected automatically because the XML standard allows the definition of namespaces multiple times in an XML document. A default namespace has to be declared and you have to use a prefix:

```
/**
 * The default namespace has to be declared,
 * but any alias can be used for it.
 */
@Test
public void 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)
    ;
}
```

Note that the prefixes in this example are named `foo` for the first and `bar` for the second usage. In real projects please use only one prefix - and not “foo” or “bar”.

XPath Result Types

The evaluation of an XPath expression generally results in distinct node types. The expected result type has to be declared when comparing XFA or XMP data from two PDF documents. All possible result types are available as constants:

```
// 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
```

XPath expressions can use all of XPath's syntax elements and functions. However, the number of available features of the XPath engine is version dependent. PDFUnit uses the XPath engine of your JDK. So your JDK version determines the compatibility to the XPath standard.

XPath Compatibility

XPath expressions can use all syntax elements and functions of XPath. However, the number of available features of the XPath engine is version dependent. PDFUnit uses the XPath engine of the JDK. So the JDK version determines the compatibility to the XPath standard.

The chapter 13.13: “JAXP-Configuration” (p. 158) describes the JAXP configuration of a JRE/JDK and how to use an external XML-parser or XSLT-processor.

Chapter 9. Utility Programs

9.1. Common Remarks for all Utilities

PDFUnit provides utility programs to extract several parts of a PDF document into separate files, mostly XML, which can then be used in tests. The following list gives an overview of the available programs:

```
// Utility programs belonging to PDFUnit:

ConvertUnicodeToHex      9.2: "Convert Unicode Text into Hex Code" (p. 112)
ExtractBookmarks         9.5: "Extract Bookmarks to XML" (p. 116)
ExtractEmbeddedFiles     9.4: "Extract Attachments" (p. 114)
ExtractFieldsInfo        9.3: "Extract Field Information to XML" (p. 113)
ExtractFontsInfo         9.6: "Extract Font Information to XML" (p. 117)
ExtractImages            9.7: "Extract Images from PDF" (p. 118)
ExtractJavaScript         9.8: "Extract JavaScript to a Text File" (p. 119)
ExtractNamedDestinations 9.9: "Extract Named Destinations to XML" (p. 120)
ExtractSignaturesInfo    9.10: "Extract Signature Information to XML" (p. 121)
ExtractXFAData           9.11: "Extract XFA Data to XML" (p. 122)
ExtractXMPData           9.12: "Extract XMP Data to XML" (p. 123)
RenderPdfClippingAreaToImage 9.13: "Render Page Sections to PNG" (p. 124)
RenderPdfToImages        9.14: "Render Pages to PNG" (p. 125)
```

The utility programs generate files. Their names are derived from those of the input files. The following rules are used to avoid naming conflicts with existing files:

- Generated file names start with an underscore.
- The names have two suffices. The penultimate is `.out` and the last one is the typical suffix for the kind of file type.

For example, when you extract bookmarks from `foo.pdf`, the file `_bookmarks_foo.out.xml` is created. Rename it before using it in a test, because then it is no longer an output file.

The Windows batch scripts in the following chapters demonstrate how to start the programs. These scripts are part of the PDFUnit release, but you have to adapt most of their content to your environment anyway: you need to set the classpath, input file and output directory.

When you start a program without parameters or with incorrect parameters, PDFUnit shows a message detailing the correct command line parameters.

The utilities also run on Unix. Unix developers should easily translate the Windows scripts into shell scripts. If you need assistance, please contact us at: [support\[at\]pdfunit.com](mailto:support[at]pdfunit.com).

9.2. Convert Unicode Text into Hex Code

Java “understands Unicode” as does XML. So PDFUnit also “understands” Unicode. The section 7: “Unicode” (p. 105) deals with Unicode in detail.

This section describes a utility program that converts a Unicode string into its ASCII hex code. The hex code can be used in many of your tests. If you are using a small number of Unicode characters it is easier to use ASCII hex code than to install a new font on your computer. And maybe you don't have permission anything.

The utility `ConvertUnicodeToHex` converts any string into ASCII and escapes all non-ASCII characters into their corresponding Unicode hex code. For example, the Euro character is converted into `\u20AC`.

The input file can be of any encoding, but you have to define the right encoding before executing the program.

Program Start

You start the Java program with the parameter `-D`:

```
::
:: Converting Unicode content of the input file to hex code.
::
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-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
```

Input

The input file `convert-unicode-to-hex.in.txt` contains this data:

```
äöü € @
```

Output

The name of the output file is derived from the name of the input file. So `_convert-unicode-to-hex.out.txt` with the following content is generated:

```
#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 @
```

The output file is written in the encoding of the Java Runtime, derived from the environment parameter `file.encoding`.

Leading and trailing whitespaces in the input string will be trimmed! When you need them for your test, add them later by hand.

9.3. Extract Field Information to XML

The program `ExtractFieldsInfo` creates an XML file with numerous items of information about many properties of all form fields. The content of a field is not extracted!

Testing field properties is described in chapter 3.10: “Form Fields” (p. 34).

Program Start

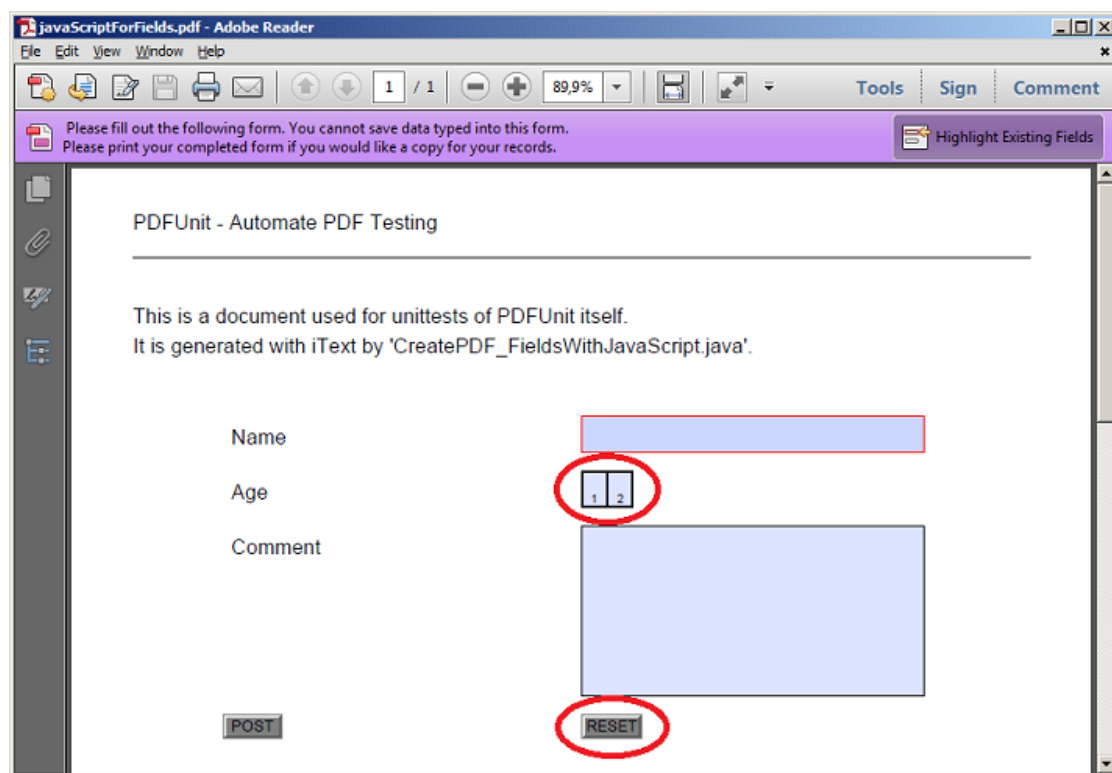
```
::
:: Extract formular fields from a PDF document into an XML file
::
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

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
```

Input

The input file `javaScriptForFields.pdf` is a sample containing 3 input fields and 2 buttons:



Output

And this is a snippet of the generated file `_fieldinfo_javaScriptForFields.out.xml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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 Values for width and height are given in millimeters
- ❷❸ Some attributes are created type dependent. For example the attribute “isMultiLineField” does not makes sense for fields of type “button”.

9.4. Extract Attachments

The utility `ExtractEmbeddedFiles` creates a separate file for every attachment which is embedded in the PDF document.

The attachments are exported byte for byte, so all file formats are supported.

Program Start

```

::
:: Extract embedded files from a PDF document. Each in a separate output file.
::

@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

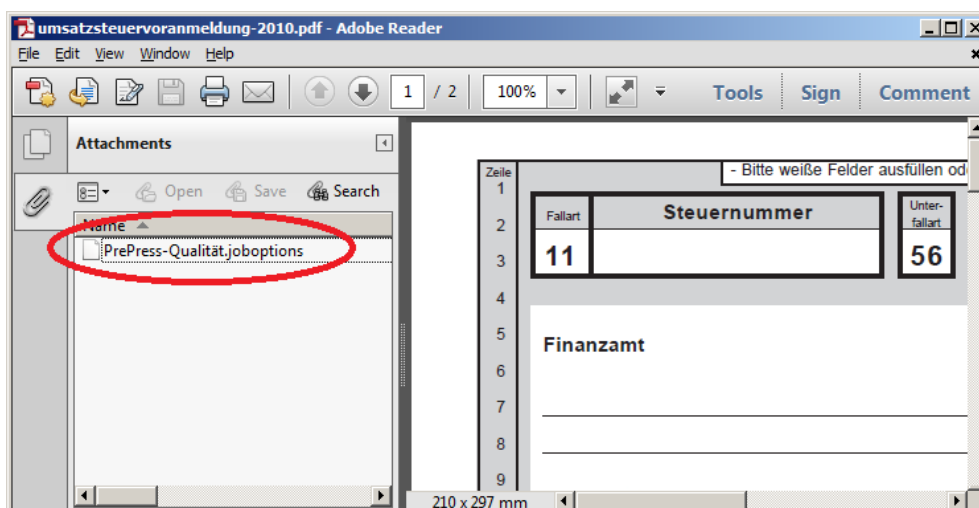
set TOOL=com.pdfunit.tools.ExtractEmbeddedFiles
set OUT_DIR=./tmp
set IN_FILE=umsatzsteuervoranmeldung-2010.pdf
set PASSWD=

java %TOOL% %IN_FILE% %OUT_DIR% %PASSWD%
endlocal

```

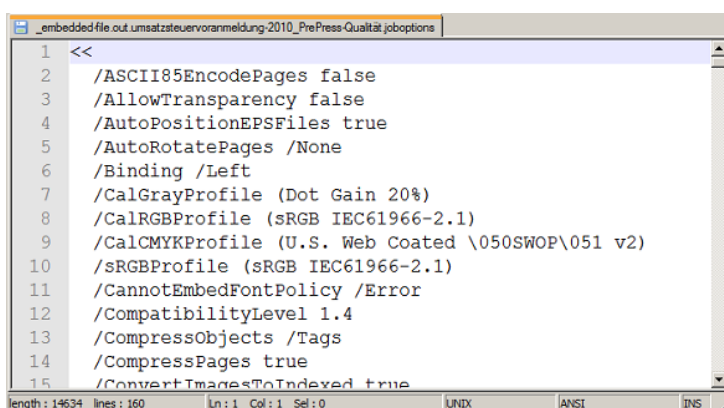
Input

The PDF document `umsatzsteuervoranmeldung-2010.pdf` contains the file `PrePress-Qualität.joboptions`.



Output

The name of the generated file contains both the name of the PDF document and the name of the embedded file: `_embedded-file_umsatzsteuervoranmeldung-2010_PrePress-Qualität.joboptions.out`.



9.5. Extract Bookmarks to XML

PDFUnit comes with the utility `ExtractBookmarks` which exports bookmarks from PDF documents into an XML file. Chapter 3.4: “Bookmarks and Named Destinations” (p. 21) describes how to use this created XML file for bookmark tests.

Program Start

```

::
:: Extract bookmarks from a PDF document into an XML file
::

@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

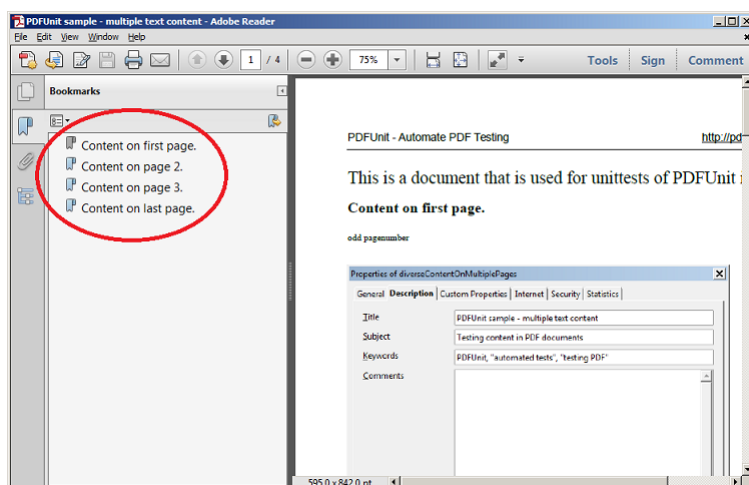
set TOOL=com.pdfunit.tools.ExtractBookmarks
set OUT_DIR=./tmp
set IN_FILE=diverseContentOnMultiplePages.pdf
set PASSWD=

java %TOOL% %IN_FILE% %OUT_DIR% %PASSWD%
endlocal

```

Input

The file `diverseContentOnMultiplePages.pdf` contains 4 bookmarks:



Output

The output file `_bookmarks_diverseContentOnMultiplePages.out.xml` can be used for XML-based tests:

```

<?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>

```

Internally, PDFUnit uses the method `SimpleBookmark.getBookmark(PdfReader)` from iText. Many thanks to the developers of iText.

9.6. Extract Font Information to XML

As described in chapter 3.9: “Fonts” (p. 30) fonts are a topic which need to be tested. All information about fonts can be extracted using the utility `ExtractFontsInfo`. You can use this generated file can for sophisticated tests with XPath.

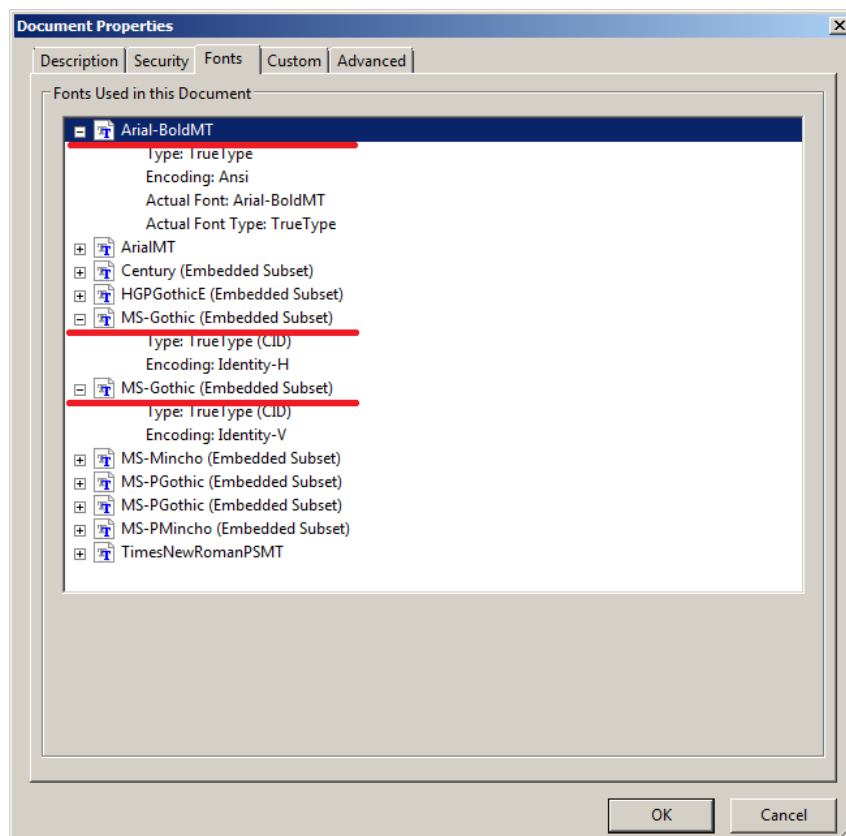
The algorithm that generates the XML file is the same as the one used by the PDFUnit tests.

Program Start

```
::  
:: 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
```

Input

Adobe Reader® shows the following fonts in the Japanese PDF document `fonts_11_japanese.pdf`:



Output

The output file `_fontinfo_fonts_11_japanese.out.xml` contains the underlined names:

```
<?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>
```

Because the XML file contains all subsets of a font it might differ from what Adobe Reader® shows.

You can format the resulting file as desired without affecting the test because whitespaces between elements and attributes are irrelevant to XML.

Based on this file, appropriate XPath expressions can be used to test any complex combination of field properties.

9.7. Extract Images from PDF

This utility extracts images imbedded in PDF document to PNG images. Each image is written to a separate file. Tests with those images are described in section 3.13: "Images in PDF Documents" (p. 45).

Program Start

```
::
:: Extract all images of a PDF document into a PNG file for each image.
::

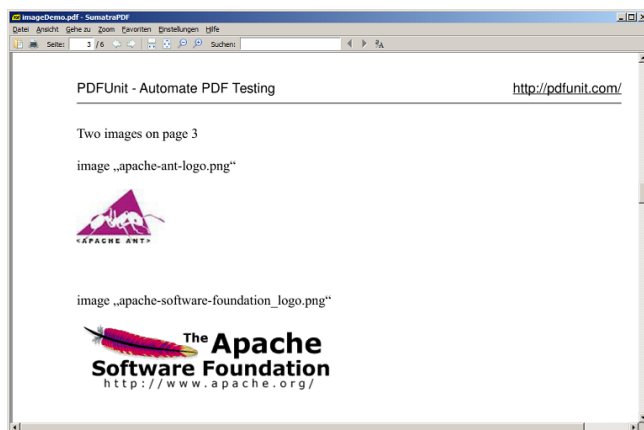
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

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
```

Input

The input file `imageDemo.pdf` contains two images:



Output

After running the utility, two files are created:



```
# created images:
.\tmp\_exported-image_imageDemo_4.out.png ❶
.\tmp\_exported-image_imageDemo_12.out.png ❷
```

❶❷ The number in the file name is the object number within the PDF document.

9.8. Extract JavaScript to a Text File

This utility extracts JavaScript from a PDF document and writes it to a text file, which can be used in PDFUnit tests as described in chapter 3.14: “JavaScript” (p. 48).

Program Start

```
::
:: 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
```

Input

The file `javaScriptForFields.pdf` used in chapter 9.3: “Extract Field Information to XML” (p. 113) contains the fields `nameField`, `ageField` and `comment` which are all associated with JavaScript.

Inside the Java program which creates the PDF document, the following JavaScript code belongs to the field `ageField`:

```
String scriptCodeCheckAge = "var ageField = this.getField('ageField');"
+ "ageField.setAction('Validate','checkAge()');"
+ ""
+ "function checkAge() {"
+ "  if(event.value < 12) {"
+ "    app.alert('Warning! Applicant\\'s age can not be younger than 12.');"
+ "    event.value = 12;"
+ "  }"
+ "}"
;
```

Output

The output file `_javascript_javascriptForFields.out.txt` contains:

```
var nameField = this.getField('nameField');nameField.setAction('Keystroke', ...
var ageField = ...;function checkAge() {  if(event.value < 12) {...
var commentField = this.getField('commentField');commentField.setAction(...
```

You can reformat the file to make it easier to read. Added whitespaces do not affect a PDFUnit test.

Note

JavaScript is also used to implement the document actions `OPEN`, `CLOSE`, `PRINT` and `SAVE`. The discribed extraction utility does only extract JavaScript from document level, but no JavaScript that is bound to actions. A new utility is scheduled for the next release.

9.9. Extract Named Destinations to XML

“Named Destinations” are landing points inside PDF documents. They are difficult to test because they aren’t displayed anywhere. But the utility `ExtractNamedDestinations` extracts all information about “Named Destinations” into an XML file. And that file can be used in tests based on XML and XPath as described in Chapter 3.4: “Bookmarks and Named Destinations” (p. 21).

And that's the extraction script:

Program Start

```
::
:: Extract information about named destinations in a PDF document into an XML file
::
@echo off
setlocal
set CLASSPATH=./lib/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
```

Input

The input file in this sample, `bookmarksWithPdfOutline.pdf`, contains different named destinations.

Output

The output file `_named-destinations_bookmarksWithPdfOutline.out.xml` contains the following data:

```
<?xml version="1.0" encoding="UTF-8"?>
<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 uses `SimpleNamedDestination.getNamedDestination(..)` from iText (<http://www.itextpdf.com>) internally. Again thank you to the developers.

9.10. Extract Signature Information to XML

Signatures and certificates contain a huge amount of data. Only some of them are testable with direct tests. But all of the data can be evaluated using XPath. Section 3.23: "Signatures and Certificates" (p. 59) describes tests with signatures and certificates.

The following script will start the extraction:

Program Start

```
::
:: Extract infos about signatures and certificates of a PDF document as XMLe
::

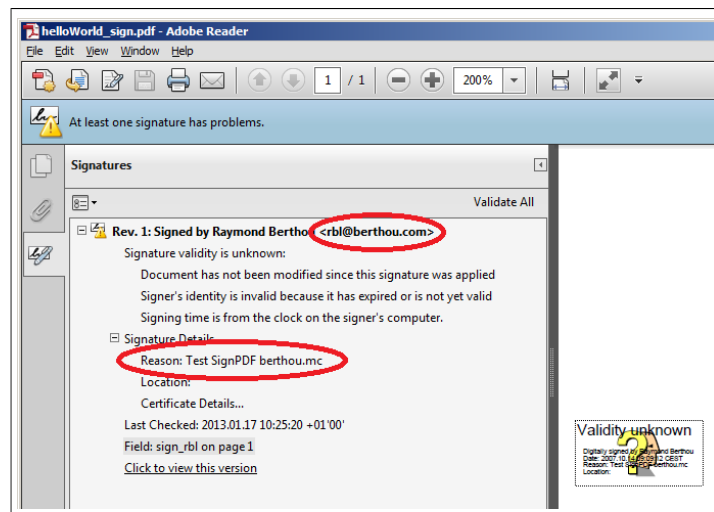
@echo off
setlocal
set CLASSPATH=./lib/pdfunit-2015.10/*;%CLASSPATH%
set CLASSPATH=./lib/itext-5.5.1/*;%CLASSPATH%
set CLASSPATH=./lib/bouncycastle-jdk15on-150/*;%CLASSPATH%

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
```

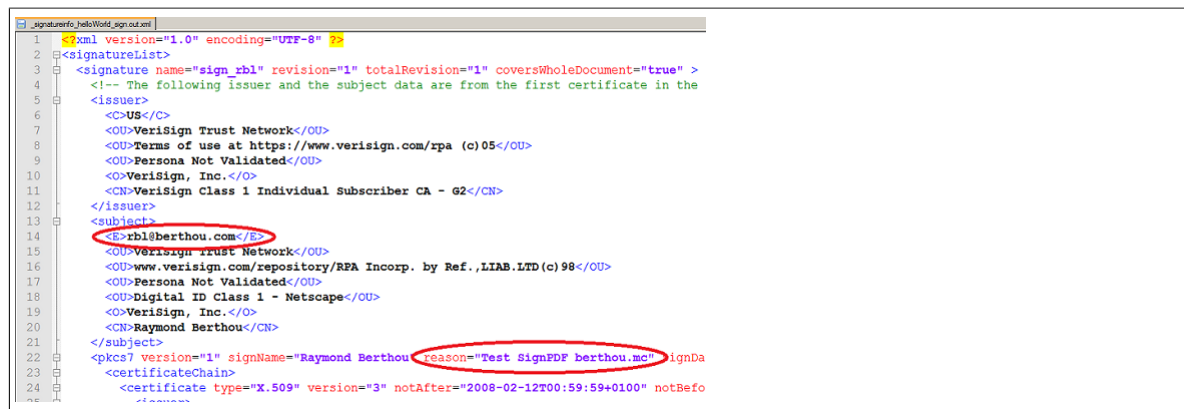
Input

Adobe Reader® shows the signature data for `helloWorld_sign.pdf`:



Output

Here is a snippet of the output file `_signatureinfo_helloWorld_sign.out.xml`:



The tests for signatures and certificates are still under development (release 2015.10). This may lead to changes in the XML structure.

9.11. Extract XFA Data to XML

Using the utility `ExtractXFADData` you can export XFA data from a PDF document and use it in XPath based tests as described in section 3.30: “XFA Data” (p. 74).

Program Start

```

::
:: 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

```

Input

The input file for the script is `xfa-enabled.pdf`, a sample document from iText.

Output

The output XML file `_xfadata_xfa-enabled.out.xml` is quite long. To get a better impression of the generated code, some of the XML-Tags in the next picture are folded:



Internally the extraction program uses the method `XfaForm.getDomDocument()` from the iText (<http://www.itextpdf.com>) project.

9.12. Extract XMP Data to XML

The utility program `ExtractXMPData` writes the **document level** XMP data from a PDF document into an XML file. This file can be used for the PDFUnit tests described in section 3.31: “XMP Data” (p. 77).

XMP data can be found on other places in the PDF than just the document level. Such XMP data is currently not extracted. But it is intended to provide the extraction of all XMP data in the next release of PDFUnit.

Program Start

```

::
:: Extract XMP data from a PDF document as XML
::

@echo off
setlocal
set CLASSPATH=./lib/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

```

Input

The XMP data will be extracted from `LXX_vocab.pdf`.

Output

A part of the output file `xmpdata_LXX_vocab.out.xml` is shown here:

```
<?xpacket begin='' id='W5M0MpCehiHzreSzNTczkc9d'?>
<?adobe-xap-filters esc="CRLF"?>
<x:xmpmeta xmlns:x='adobe:ns:meta/' x:xmptk='XMP toolkit 2.9.1-14, framework 1.6'>
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
        xmlns:ix='http://ns.adobe.com/ix/1.0/'
>
...
<rdf:Description rdf:about='uuid:f6a30687-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>
```

During the processing, PDFUnit uses the method `PdfReader.getMetadata()` from the iText-Project (<http://www.itextpdf.com>).

9.13. Render Page Sections to PNG

The reasons for testing a particular region of a PDF page are described in section 3.18: “Layout - in Clipping Areas” (p. 54). To find the coordinates of the area you want to test, PDFUnit provides the small utility `RenderPdfClippingAreaToImage`. Choose the width, height and the position of the upper left corner and the corresponding region is then extracted into a file. Verify this file then “by eye” and vary the parameters until you got the right region. Once you have found the correct coordinates for your region, use those parameters in your PDFUnit test.

Program Start

```
::
:: Render a part of a PDF page into an image file
::

@echo off
setlocal
set CLASSPATH=./lib/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

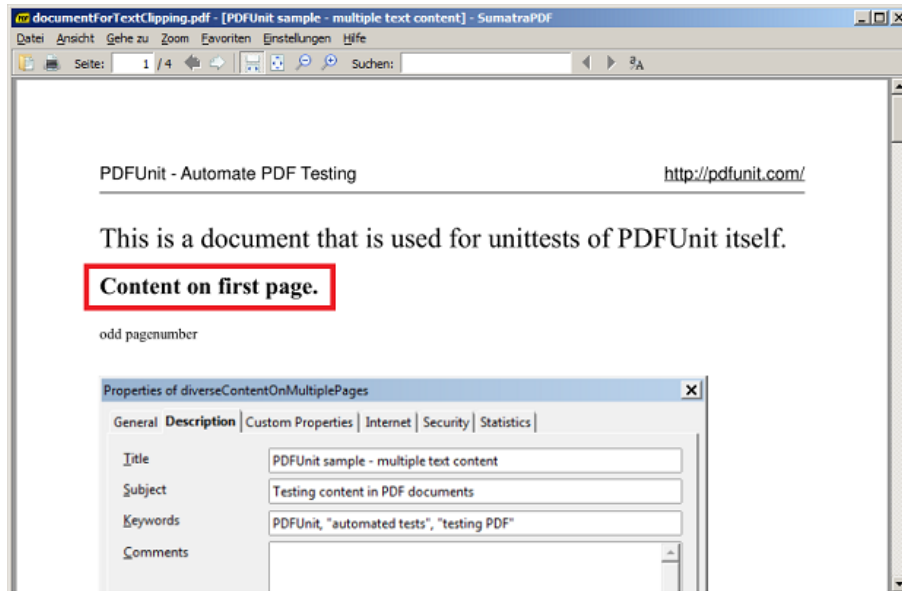
java %TOOL% %IN_FILE% %PAGENUMBER% %OUT_DIR% ❶
    %FORMATUNIT% %UPPERLEFTX% %UPPERLEFTY% %WIDTH% %HEIGHT% %PASSWD%
endlocal
```

❶ The linebreak in this listing is placed here only for documentation purposes.

As in line 17, `mm` (Millimeter) or `points` (Points) can be used as the unit of measurement to describe the clipping area. Maybe, you have to use a calculator to get the right values.

Input

The upper part of the input file `documentForTextClipping.pdf` contains the text: “Content on first page.”



Output

Content on first page.

The generated image file has to be checked.

The name of the generated PNG includes the area's coordinates. Because PDFUnit and the utility program `RenderPdfClippingAreaToImage` use the same algorithm, you can use the parameter values from the script for your test. And later, you can derive them from the file name:

```
#
# Parameters from filename:
#
_rendered_documentForTextClipping_page-1_area-50-130-170-25.out.png
                                     |   |   |   +- height
                                     |   |   +- width
                                     |   +- upperLeftY
                                     +- upperLeftX
```

Internally PDFUnit uses functions from the project “jPedal” (<http://www.idrsolutions.com/>). Thanks to the developer team.

9.14. Render Pages to PNG

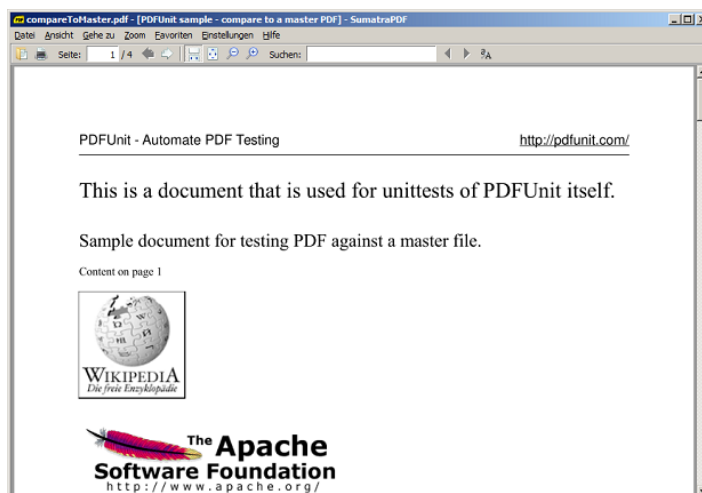
If you want to test formatted text, the only way to do it is to render a PDF page and compare the result with an image of the correctly formatted content. Section 3.17: “Layout - Entire PDF Pages” (p. 52) describes layout-tests using rendered pages. And the utility `RenderPdfToImages` renders a PDF document page by page into PNG files.

Program Start

```
::  
:: Render PDF into image files. Each page as a file.  
::  
  
@echo off  
setlocal  
set CLASSPATH=./lib/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
```

Input

The input file `compareToMaster.pdf` consists of 4 pages with different images and text. The PDF Reader “SumatraPDF” (<http://code.google.com/p/sumatrapdf>) shows the first page:

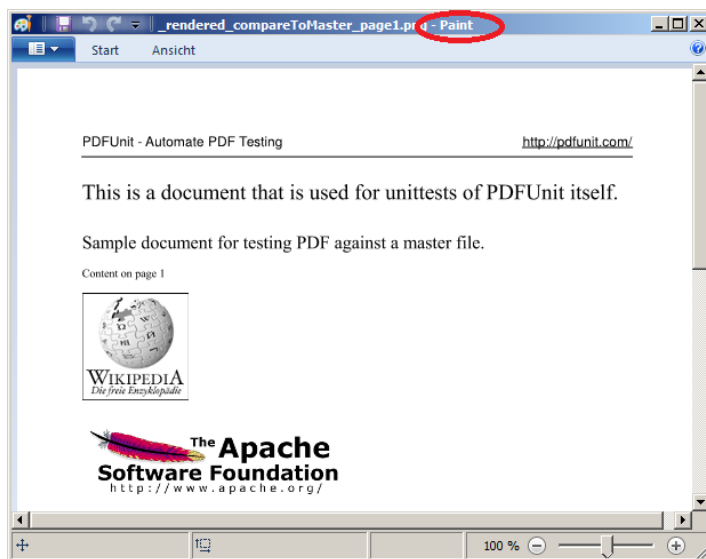


Output

After running the rendering program 4 files are created:

```
.\tmp\_rendered_compareToMaster_page1  
.\tmp\_rendered_compareToMaster_page2  
.\tmp\_rendered_compareToMaster_page3  
.\tmp\_rendered_compareToMaster_page4
```

The first of these files looks the same as seen with the PDF Reader.



Internally, PDFUnit uses the same algorithm to render the pages as the rendering program does. So any difference found by a test is due to a change in the PDF document.

PDFUnit uses the class `org.jpdedal.PdfDecoder` from the project "jPedal" (<http://www.idrsolutions.com/>). Thanks to the developers.

Chapter 10. Experience from Practice

10.1. Does Content Fit in Predefined Form Fields?

Initial Situation

A PDF document is created using a document template with empty fields as place holders for text, e.g. the address of a customer in an advertising letter. At runtime the fields are filled with text.

Problem

The text could be larger than the available size of the field.

Solution Approach

PDFUnit provides methods to detect **text overflow**.

Solution

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

A similar test can be done with one field:

```
@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()
        ;
}
```

Chapter 3.11: "Form Fields - Text Overflow" (p. 42) describes this subject in detail.

10.2. New Logo on each Page

Initial Situation

Two companies merge.

Problem

Some documents need a new logo. This should be visible on each page.

Solution Approach

The new logo exists as an image file. PDFUnit uses this file for tests.

Solution

```
/**
 * 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. Authorized Signature of the new CEO

Initial Situation

The board of executives has changed.

Problem

PDF documents which are used for contracts need to have a valid (visible) signature. So the signature of the new CEO must be used.

The new and the old signature exist as files, but you have to give them the same file name. Otherwise, all programs have to be recompiled the next time the CEO changes.

Solution Approach

The image file is compared byte-wise with the signature image from the PDF documents.

Solution

```
/**
 * 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 of the Former CEO

Initial Situation

The board of executives has changed again.

Problem

The name of the former CEO must be changed in the header of newly created PDF documents.

Solution Approach

PDFUnit provides methods to check documents for the **non-existence** of text.

Solution

```
/**
 * 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 and PDFUnit in Combination

Initial Situation

You are providing generated PDF documents on your web pages. Those documents should be tested.

Problem

But the PDF can only be tested in the context of the web pages. So the input data for a test has to be entered to the browser frontend and the generated PDF has to be selected also through the browser.

Solution Approach

Selenium provides a good way to select a PDF document within a web page. This document will be opened by PDFUnit as a Stream.

Solution

```

/**
 * 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();
    }
}

```

For more information about Selenium contact the project site <http://seleniumhq.org/>.

10.6. HTML2PDF - Does the Rendering Tool Work Correct?

Initial Situation

A web application creates dynamic web pages and additionally it provides the possibility to download the current web page as a PDF document. The PDF generation is done by rendering the HTML page with an appropriate tool.

Problem

How do you know that the complete content of the HTML page is also present in the generated PDF? Do you know the boundary conditions required by the rendering tool? Does your HTML meets these requirements?

Solution Approach

The test starts by reading the HTML page using Selenium. Text is extracted with Selenium and stored in local variables.

Then the PDF generation is started through the web page and the resulting document is selected again with Selenium. Finally the stored text can be used in PDFUnit test methods.

Solution

```
/**
 * This sample shows how to test an HTML page with Selenium,
 * then let it be rendered by the server to PDF and verify that
 * content also appears in PDF.
 *
 * @author Carsten Siedentop, February 2013
 */
public class Html2PDFTest_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 The link with the name “See also” is **not rendered** by the Wikipedia server! But the corresponding link of the German page on the same subject is rendered correctly.

10.7. Caching Test Documents

Initial Situation

You have many tests - that is good.

Problem

The tests are running slowly - that is bad. Maybe it's because the PDF documents are very large, but after all it has to be tested.

Solution Approach

You instantiate the test document only once for all tests in a class using a static instantiation method with the annotation `@BeforeClass`.

Solution

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

Of course, you can concatenate all test methods in one statement. But how would you name the test? Names of test methods should reflect the content as exactly as possible. Otherwise, a test report with many hundreds of tests is difficult to understand. So, `testAll` is a bad name - it means nothing.

PDFUnit is stateless. But it can not be guaranteed that 3rd-party libraries are also stateless. So, if you have problems with tests using cached documents, change the annotation `@BeforeClass` into `@Before`, remove the modifier `static` and the PDF document is instantiated for each test method.

```
@Before // Document will be instantiated for each test. No caching:
public void loadTestDocument() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";
    document = AssertThat.document(filename);
}
```

10.8. Nesting Depth of Bookmarks

Initial Situation

A company has a style-guide for PDF documents which allows a limited nesting depth for bookmarks.

Problem

How can you verify this requirement?

Solution Approach

The bookmarks are analyzed using XPath.

Solution

Extract all bookmarks in a PDF using the extraction program `ExtractBookmarks`. This is the file:

```
<?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>
```

Now run a test containing an XPath expression which checks that there is **no** Title-node having two or more Title-nodes as predecessors.

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

Chapter 11. Installation, Configuration, Update

11.1. Technical Requirements

PDFUnit needs Java 7 or higher.

Also the runtime libraries of iText version 5.3.3 or higher are needed. They are not included in PDFUnit because they need a separate license.

If you run PDFUnit with ANT, Maven or other tools, you have to install those tools independently from PDFUnit.

Tested Environments

PDFUnit was successfully tested in these environments:

Operating System	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

An error occurs with the JDK version "IBM J9, R26_Java726_SR1" under "Windows 7, 32 Bit".

The JDK version "Oracle 1.8.0" processes PNG files differently from version 1.7.x. Therefore image files of rendered PDF pages have to be rendered using the same Java version when they are used in PDFUnit tests.

More combinations of Java and operating systems will be tested in the future.

If you have any problem with the installation, please contact us under [problem\[at\]pdfunit.com](mailto:problem[at]pdfunit.com).

11.2. Installation without an Existing iText

Download of PDFUnit-Java

Download the file `pdfunit-java-VERSION.zip` from the project site: <http://www.pdfunit.com/en/download/index.html>. If you have purchased a license, you get a new ZIP file by mail.

Unzip the file in a folder, for example into `PROJECT_HOME/lib/pdfunit-java-VERSION`. In the following text that folder is referred to as `PDFUNIT_HOME`.

Download iText

During runtime PDFUnit-Java needs the runtime libraries of iText, version 5.3.3 or higher. Download them separately from <http://sourceforge.net/projects/itext/files/>.

Unzip the downloaded ZIP file and copy the JAR files into a separate folder below `PROJECT_HOME/lib`.

Note, that iText needs a license when it is used in commercial projects.

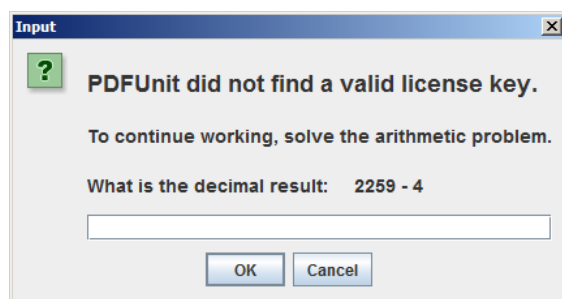
Configuring Classpath

All JAR files which are distributed with PDFUnit and the JAR files of iText need to be included in the classpath. Also the file `config.properties`.

If you have a licensed PDFUnit, add the license key file `license-key_pdfunit-java.lic` to the classpath too.

Using PDFUnit without a License Key

You are welcomed to evaluate PDFUnit. In this case, a message box appears showing a simple math calculation you have to solve. If you calculate successfully the test will run, otherwise you have to restart your test and calculate again.



Sometimes the message box is covered by other applications. Then the ANT or Maven script is blocked. Minimize all applications to look for the message box.

PDFUnit comes with some utility programs `com.pdfunit.tools.*` to extract information from PDF documents. These tools do not need a license key. So you don't have to do your math :-)

Order a License Key

If you use PDFUnit in a commercial context you need a license. Write a mail to [license\[at\]pdfunit.com](mailto:license[at]pdfunit.com) and ask for a license. You will receive an answer as soon as possible.

The license fee is calculated individually. A small company should not pay as much as a big company, and someone testing only a few PDF documents, of course, pays less. And if you want to get a free license, give us some reasons. It is not impossible.

Use License Key

If you have ordered a license you will receive a ZIP file and a separate file `license-key_pdfunit-java.lic`. Install the content of the ZIP file as described above and include the license file in your classpath.

Any change to the license file makes it unusable. If this happens contact PDFUnit.com and ask for a new license file.

Verify the Installation

If you have a problem with the configuration start the script `verifyInstallation.bat` or `verifyInstallation.sh`. You will get a detailed problem analysis. See chapter 11.7: "Verifying the Configuration" (p. 142).

11.3. Installation in Addition to an Existing iText

Using an Existing iText Version for PDFUnit

Both PDFUnit and iText (<http://itextpdf.com>) need libraries from other projects. Due to licensing restrictions PDFUnit is delivered without the iText libraries, but includes all 3rd-party libraries that are royalty free.

When you configure the classpath, make sure that the libraries that are used by iText and by PDFUnit are included to the classpath only once. The versions of the 3rd party libraries have to be compatible with iText and PDFUnit.

If you get a version conflict with the 3rd-party libraries, you have to reconfigure the classpath so that PDFUnit finds the libraries in the classpath that are delivered with PDFUnit, and your iText version uses the libraries that are delivered with iText.

Using a Different iText Version for PDFUnit

Suppose you are using an old version of iText to create PDF documents and you do not want to upgrade this version or it is impossible. In such a case you need two separate classpath configurations. Include an iText version higher than 5.3.3 to the classpath of PDFUnit and use your older version of iText for the programs that create your PDFs. Don't forget that the use of an actual iText version needs a license in commercial projects.

In an ANT script you can define two classpath properties.

```
<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/**/*.*" />
  </fileset>
</path>
```

In Maven, it is best to create a separate pom file, for example `pom_pdfunit.xml`.

11.4. Setting Classpath in Eclipse, ANT, Maven

All development environments need a classpath with these entries:

- all JAR files delivered by PDFUnit
- the JAR files of iText
- the file `config.properties`
- the file `license-key_pdfunit-java.lic` if PDFUnit is used with a license

If PDFUnit does not find the files it will give error messages like these:

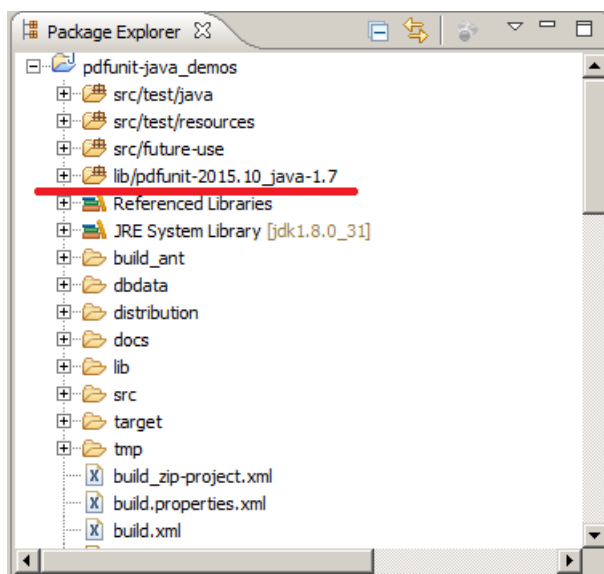
- Could not find 'config.properties'. Verify classpath and installation.
- No valid license key found. Switching to evaluation mode. Contact PDFUnit.com if you are interested in a license.

- 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.

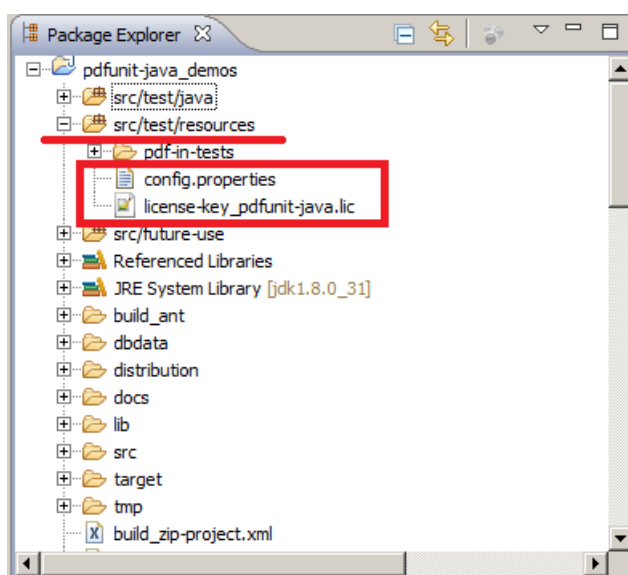
In the following examples show how to configure the classpath in different environments. Additionally chapter 11.5: “Set Pathes Using System Properties” (p. 140) describes how to use system properties to declare the location of the files `config.properties` and `license-key_pdfunit-java.lic`.

Configuring Eclipse

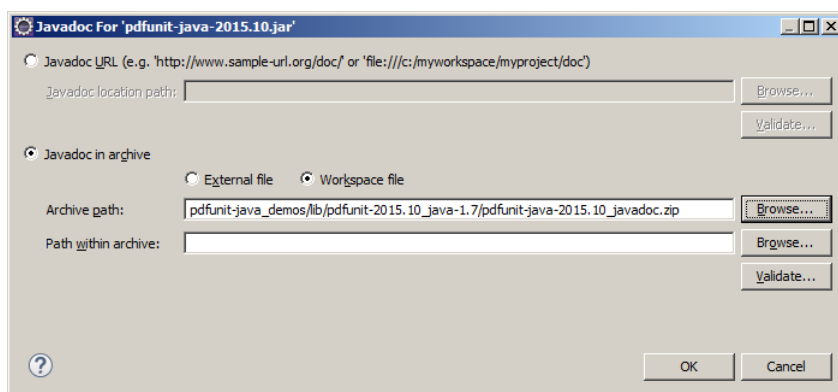
A simple way to configure Eclipse for PDFUnit is to include the installation directory `PDFUNIT_HOME` and all JAR files individually in the classpath:



Another option is to move the file `config.properties` out of `PROJECT_HOME` into another folder, for example `src/test/resources`, and put that folder into the classpath.



PDFUnit includes the file `PDFUNIT_HOME/pdfunit-java-VERSION_javadoc.zip` which can be registered in Eclipse to let Eclipse show the Javadoc comments.



Configuring ANT

Many options exist to configure ANT for PDFUnit. In all options the JAR files of PDFUNIT_HOME and PDFUNIT_HOME/lib/* have to be put into the classpath. Additionally the file config.properties must be included in classpath.

If you have not changed the config.properties, it is simple to include PDFUNIT_HOME itself in the classpath additionally to the JAR files as is shown by the following listing:

```
<!--
  It is important to have the directory of PDFUnit itself in the classpath,
  because the file '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>
```

The file config.properties can be moved to an individual folder, for example into src/test/resources. This way is recommended if you want to change the config file. How to configure PDFUnit is described in the chapter 11.6: "Using the config.properties File" (p. 140). Then the classpath in ANT looks like this:

```
<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>
```

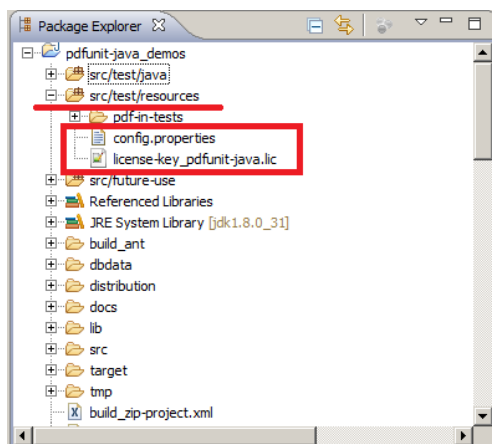
Configuring Maven

The current release of PDFUnit (2015.10) is not provided in a public Maven repository. To use it with Maven despite this fact, you have to install it into a local or company-wide repository. You can do it with the following command. Change to the directory PDFUNIT_HOME\lib and run this command:

```
mvn install:install-file -Dfile=<PATH_TO>pdfunit-java-VERSION.jar -DpomFile=<PATH_TO>pom.xml
```

The next step is to copy `config.properties` into the directory `src/test/resources`.

The following image shows the resulting project layout:



Register the PDFUnit library to your `pom.xml`.

```
<dependency>
  <groupId>com.pdfunit</groupId>
  <artifactId>pdfunit</artifactId>
  <version>2015.10</version>
  <scope>compile</scope>
</dependency>
```

Last Step for Licensed PDFUnit

The license file `license-key_pdfunit-java.lic` must also be included in the classpath. Otherwise the message box with the math calculation is shown.

11.5. Set Pathes Using System Properties

The files `config.properties` and the license key file can be placed outside of the classpath. But then their location has to be declared by Java system properties. The keys of the properties are:

- `-Dpdfunit.configfile`
- `-Dpdfunit.licensekeyfile`

Dependent on your test system (Eclipse, ANT or Maven) these parameters can be set in many ways. Use the common information to see, how Java system properties are set in your specific environment. A less known option is the operating system environment variable `_JAVA_OPTIONS` which works for all test systems:

```
set _JAVA_OPTIONS=-Dpdfunit.configfile=..\myfolder\config.properties
```

If you have questions about this topic, write a mail to [info\[at\]pdfunit.com](mailto:info[at]pdfunit.com).

11.6. Using the config.properties File

Typically PDFUnit does not need to be configured, but the file `config.properties` gives you the chance to do so. The following sections show you how:

Date Format for PDF Internal Dates

In the real world the creation date and modification date of a PDF document can be formatted in many different ways depending on the program which creates the PDF. So the required format string can be set in the config file:

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

Locale of PDF Documents

Java needs a locale when working with date values and for the conversion of strings into lowercases too. PDFUnit reads the value of the locale from `config.properties`. All the constants defined in `java.util.Locale` are allowed. The default value is `en` (English).

```
#####
# Locale of PDF documents, required by some tests.
#####
pdf.locale = en
#pdf.locale = de_DE
#pdf.locale = en_UK
```

You can write all values in lowercase or uppercase. PDFUnit also accepts underscore and hyphen as a delimiter between language and country.

If you delete the key `pdf.locale` from `config.properties` by accident, then the default locale from the Java-runtime is taken (`Locale.getDefault()`).

Output Folder for Error Images

When a difference is detected while comparing rendered pages of two PDF documents, a diff image is created. It shows the master document on the left and differences to the actual test document on the right side. The differences are shown in red. The name of the test is placed above the image.

You can configure the output directory in the `config.properties`. By default the diff images will be stored in the directory containing the test PDF. That might be useful for some projects. But when you want to have a fixed folder for all diff images, you can configure that behaviour by using the property `diffimage.output.path.files`:

```
#####
#
# The path can be absolute or relative. The base of a relative path depends
# on the tool which starts the junit tests (Eclipse, ANT, etc.).
# The path must end with a slash. It must exist before you run the tests.
#
# If this property is not defined, the directory containing the PDF
# files is used.
#
#####
diffimage.output.path.files = ./
```

If your test produces diff images of PDF documents which are processed as streams or byte arrays, you can configure the output folder using the property `diffimage.output.path.streams_and_bytearrays`:

```
#####
#
# If this property is not defined, the directory of the running process
# is used.
#
#####
diffimage.output.path.streams_and_bytearrays = ./
```

11.7. Verifying the Configuration

Verifying with a Script

The installation of PDFUnit can be checked using a special program, started with the script `verifyInstallation.bat` or `verifyInstallation.sh`:

```
::
:: Verify the installation of PDFUnit
::

:: 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
```

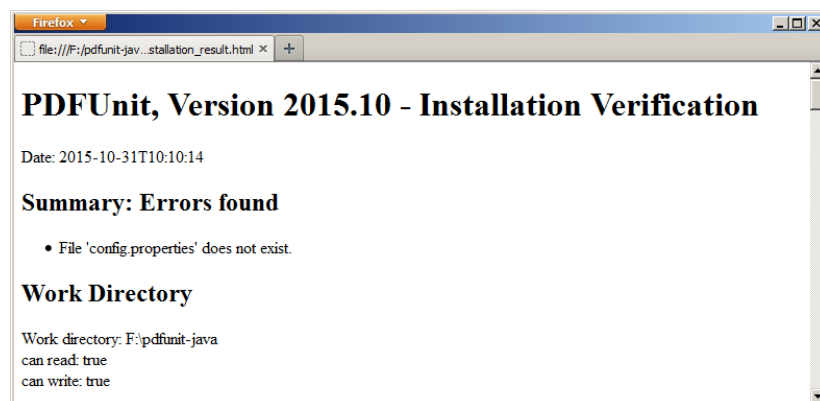
You have to edit the paths depending on your installation.

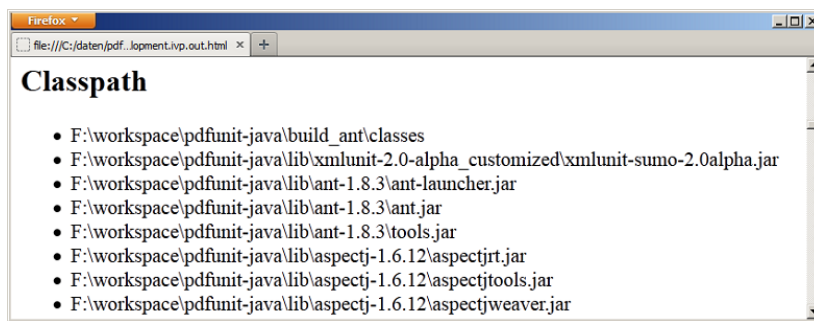
The stylesheet option is provided to use individual stylesheets. If you don't use a stylesheet, a simple one is used automatically.

The script produces the following output on the console:

```
Checking installation ...
... finished. Report created, see 'verifyInstallation_result.html'.
```

The resulting report file shows errors and information about the classpath, environment variables and other runtime related data:





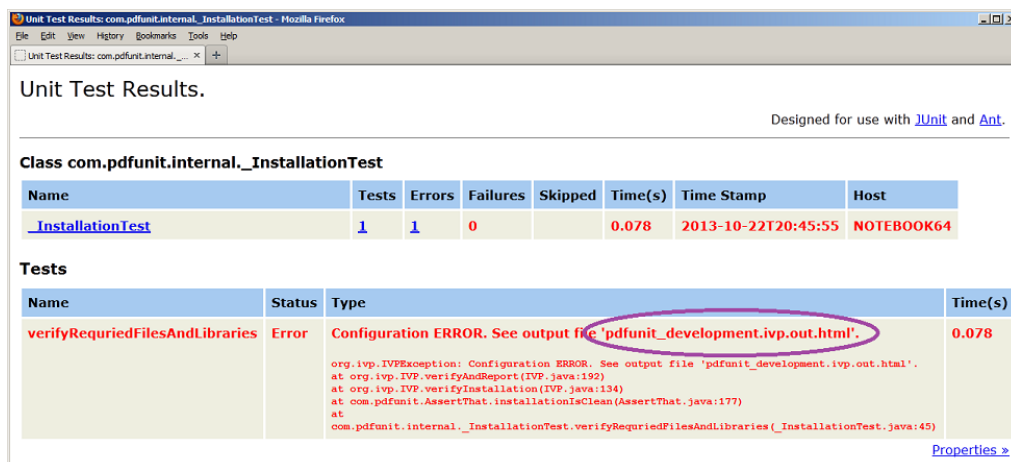
Verifying as a Unit Test

The verification of the installation can also be done as a unit test. That makes it possible to visualize the system environment of the current tests in the context of ANT, Maven or Jenkins.

Within a simple unit test you can use a special method:

```
/*
 * The method verifies that all required libraries and files are found on the
 * classpath. Additionally it logs some system properties and writes
 * all to System.out.
 */
@Test
public void verifyRequiredFilesAndLibraries() throws Exception {
    AssertThat.installationIsClean("pdfunit_development.vip");
}
```

The method performs the same internal checks as the script described above. If a configuration error exists, the test is “red” and the error message points to the created HTML file:



The report file contains the same data as when it was created by a shell script.

11.8. Installation of a New Release

The installation of a new release of PDFUnit-Java runs just like the initial installation, because PDFUnit is always delivered as a full release, never as an incremental release.

Getting the New Release

If you use PDFUnit without a license, download the new ZIP file from the internet: <http://www.pdfunit.com/en/download/index.html>.

If you use PDFUnit with a license, you will receive a new release by mail with an attached ZIP file and a license file.

First Steps for all Development Environments

Before you start installing the new release, run all existing unit tests with the old release. They should be “green”.

Save your project.

Install the Update

Unzip the new release, but not into the existing project folder. In the following text the folder with the new release is called `PDFUNITJAVA_HOME_NEW` and the project folder is called `PROJECT_HOME`.

Delete the folder `PROJECT_HOME/lib/pdfunit-OLD-VERSION`.

Copy the directory `PDFUNITJAVA_HOME_NEW` to the folder, where the old release was located, for example to `PROJECT_HOME/lib/pdfunit-NEW-VERSION`.

If you had placed the file `config.properties` in a different folder than the installation folder, copy the corresponding new file from `PROJECT_HOME/lib/pdfunit-NEW-VERSION` to that folder where it was located in the old release.

If you used the file `config.properties` with individual settings, transfer the changes to the `config.properties` of the new release.

If you use PDFUnit with a license, copy the new license file `license-key_pdfunit-java.lic` to the folder containing the old release.

Next Steps in ANT

No configuration steps are necessary for ANT if you have declared the classpath as described before in this chapter.

Next Steps in Maven

The new release has to be installed into your local or company-wide repository. Open a shell, change to the directory `PROJECT_HOME/lib/pdfunit-NEW-VERSION` and submit the following command:

```
mvn install:install-file -Dfile=pdfunit-java-VERSION.jar -DpomFile=pom.xml
```

Next Steps in Eclipse

Include all new JAR files in the build path. Remove the old JAR files from the build path. Eclipse should not show any build path error.

Register the Javadoc documentation again as described in chapter : “Configuring Eclipse” (p. 138).

Last Step

Run the existing tests with the new release. If there are no documented incompatibilities between the new and the old release, the tests should run “green” again. Otherwise read the release information.

11.9. Uninstall

PDFUnit can be uninstalled cleanly by deleting the installation directories. PDFUnit doesn't create any entries in the registry or system directories, so none need to be removed. Don't forget to remove the references to JAR files and PDFUnit directories from your own scripts.

Chapter 12. PDFUnit for Non-Java Systems

12.1. A quick Look at PDFUnit-XML

It is unnecessary for testers to know Java to write tests for PDF documents. With the name “PDFUnit-XML” a version of PDFUnit exists for an XML-based system. It contains a runtime to execute the tests, scripts to start them, XML Schema to validate the tests and stylesheets as part of the runtime. “PDFUnit-XML” is completely compatible with “PDFUnit-Java”.

The following examples show the idea behind PDFUnit-XML:

```
<testcase name="hasTextOnSpecifiedPages_Containing">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onPage="1, 2, 3" >
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="hasTitle_MatchingRegex">
  <assertThat testDocument="documentInfo/documentInfo_allInfo.pdf">
    <hasTitle>
      <startingWith>PDFUnit sample</startingWith>
      <matchingRegex>.*Unit.*</matchingRegex>
    </hasTitle>
  </assertThat>
</testcase>
```

```
<testcase name="compareText_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>
```

Names of the tags and attributes are mostly the same as the function names in the Java-API. They also follow the idea of “Fluent Interfaces” (http://de.wikipedia.org/wiki/Fluent_Interface).

XML Schema exists to validate the XML syntax.

A detailed description of PDFUnit-XML is available.

12.2. A quick Look at PDFUnit-Perl

“PDFUnit-Perl” will be available September 2014. That version of PDFUnit contains a Perl module `PDF::PDFUnit` with all necessary scripts. Together with other CPAN modules e.g. `TEST::More` or `Test::Unit` it is easy to write automated tests, which are 100% compatible with “PDFUnit-Java”.

It is intended to upload `PDF::PDFUnit` to the CPAN archive.

Here are two simple examples using `TEST::More` and `PDF::PDFUNIT`:

```
#
# Test hasFormat
#
ok(
  com::pdfunit::AssertThat
    ->document("documentInfo/documentInfo_allInfo.pdf")
    ->hasFormat($com::pdfunit::Constants::A4_PORTRAIT)
    , "Document does not have the expected format A4 portrait")
;
```

```
#
# Test hasAuthor_WrongValueIntended
#
throws_ok {
  com::pdfunit::AssertThat
    ->document("documentInfo/documentInfo_allInfo.pdf")
    ->hasAuthor()
    ->matchingComplete("wrong-author-intended")
} 'com::pdfunit::errors::PDFUnitValidationException'
, "Test should fail. Demo test with expected exception."
;
```

A separate documentation covers PDFUnit-Perl.

12.3. A quick Look at PDFUnit-NET

A “PDFUnit-NET” version for a .NET environment is intended be provided in Oktober 2014. A “proof of concept” is going well:

```
[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")
  ;
}
```

PDFUnit-NET is fully compatible to PDFUnit-Java because a DLL is generated from the Java version. However, this means that method names in C# begin with lowercase letters.

The development of PDFUnit-NET is not finished, so this code might change.

PDFUnit-NET will come with it's own special manual.

Chapter 13. Appendix

13.1. Running PDFUnit with TestNG

PDFUnit also runs with TestNG.

If you only use the annotation `@Test` no difference can be seen. Only when you expect exceptions can you see it's TestNG:

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

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

13.2. Instantiation of PDF Documents

The PDF documents must belong to one of the types shown in the following list.

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

Also a master PDF document can be read from a `String`, `File`, `URL`, `InputStream` or `byte[]`.

If documents are password protected, PDFUnit needs either the “user password” or the “owner password” to open the file.

13.3. Page Selection

Predefined Pages

Constants allow your tests to focus on specific pages in a PDF document. Their names clearly express their intent:

```
// Possibilities to focus tests to specific pages:

com.pdfunit.Constants.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` and `FIRST_PAGE` are functionally identical, as well as `ON_LAST_PAGE` and `LAST_PAGE`. The redundancy is due to linguistic reasons so that the API reads better depending on which function follows.

Here an example using one of the predefined page constants:

```
@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")
    ;
}
```

Individual Pages

The next example shows how individual pages can be addressed: Page numbers must be separated by commas.

```
@Test
public void hasText_OnMultiplePages() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";
    PagesToUse ON_SELECTED_PAGES = PagesToUse.getPages(1, 2, 3);

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

Two methods are available to select a single page or a set of pages:

```
// How to define individual pages:

PagesToUse.getPage(2);
PagesToUse.getPages(1, 2, 3);
```

Open Ranges

Various methods are available to define a ranges of pages at the beginning or at the end of a document:

```
// How to define open ranges:

OnAnyPage.after(2);
OnAnyPage.before(3);

OnEveryPage.after(2);
OnEveryPage.before(2);
```

Here is an example using a page range:

```
@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))
    ;
}
```

Inner Ranges

And finally attributes exist to set the scope of a test to a range of pages inside a document, `<hasText fromPage="1" toPage="2" >`.

```

@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)
    ;
}

```

In combination with `.spanningFrom().to()` only the methods `containing()`, `matchingRegex()`, `notContaining()` and `notMatchingRegex()` can be used.

Important Hints

- Page numbers begin with '1'.
- The page number in `before(int)` and `after(int)` are both exclusive.
- The page number in `from(int)` and `to(int)` are both inclusive.
- `OnEveryPage` means that the expected text has to exist on each page in the given range.
- When using `OnAnyPage`, a test is successful if the expected string exists on one or more pages in the given range.

13.4. Comparing Text

Expected text and actual text on a PDF page can be compared using the following methods:

```

// 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 Methods **without** the second parameter normalize the whitespaces. That means whitespaces at the beginning and the end are removed and all sequences of any whitespace within a text are reduced to one space.
- ❷❹❻ The processing of whitespaces in these methods is controlled by the second parameter. For this parameter, the constants `KEEP_WHITESPACES`, `NORMALIZE_WHITESPACES`, and `IGNORE_WHITESPACES` exist. These constants are explained separately in section 13.5: "Whitespace Processing" (p. 150).

Comparisons with regular expressions follow the rules and possibilities of the class `java.util.regex.Pattern`:

```
// Using regular expression to compare page content
@Test
public void hasText_MatchingRegex() throws Exception {
    String filename = PATH + "content/diverseContentOnMultiplePages.pdf";

    AssertThat.document(filename)
        .hasText(ON_FIRST_PAGE)
        .matchingRegex(".*[Cc]ontent.*")
    ;
}
```

The methods `containing(String[])` and `notContaining(String[])` can be called with multiple search terms. A test with `containing(String[])` is considered successful if each expected term appears on every selected page. A test with `notContaining(String[])` is considered successful if none of the terms exist on any of the selected pages:

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

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

13.5. Whitespace Processing

Almost all tests compare strings. Many comparisons would fail if whitespaces remained as they are. So you can control the way whitespaces are handled using one of the three predefined constants. `NORMALIZE_WHITESPACES` is the default if nothing is declared:

```
// Constants for whitespace processing:

com.pdfunit.Constants.IGNORE_WHITESPACES           ❶
com.pdfunit.Constants.KEEP_WHITESPACES             ❷
com.pdfunit.Constants.NORMALIZE_WHITESPACES // default ❸
```

- ❶ All whitespaces are deleted before comparing two strings.
- ❷ Existing whitespaces are not changed.
- ❸ Whitespaces at the beginning and at the end of a string are deleted. Any sequences of whitespaces within a text are reduced to one space.

The constants can be used in the following methods:

```
// defining whitespace processing:
.hasXXXAction().containing(.., WhitespaceProcessing)
.hasXXXAction().matchingComplete(.., WhitespaceProcessing)
.hasText(..).containing(.., WhitespaceProcessing)
.hasText(..).matchingComplete(.., WhitespaceProcessing)
.hasText(..).notContaining(.., WhitespaceProcessing)
```

An example:

```
@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)
    ;
}
```

The expected string in this example is written completely without linebreaks, although the PDF page contains many of them. However when using `IGNORE_WHITESPACES` the test runs successfully.

`NORMALIZE_WHITESPACES` is the default when nothing else is set explicitly. Test methods in which a flexible treatment of whitespaces does not make sense do not have a second parameter.

```
// Examples for default whitespace processing,  
// The list is not complete:  
  
.hasText(..).startsWith(..)  
.hasText(..).endsWith(..)  
.hasText(..).matchingComplete(..)  
.hasBookmark().withLinkToName(..)  
.hasAuthor().containing(..)
```

As an exception to this rule, no method involving regular expressions changes whitespaces automatically. It is up to you to integrate the whitespace processing into the regular expression, for example like this:


```
(?ms).*print(.*)
```

The term `(?ms)` means that the search extends over multiple lines. Line breaks are interpreted as characters.

13.6. Single and Double Quotation Marks inside Strings

Different Types of Quotes

Important Notice: The term “Quote” has different meanings as the following picture shows:



Example 1: 'single quotes'

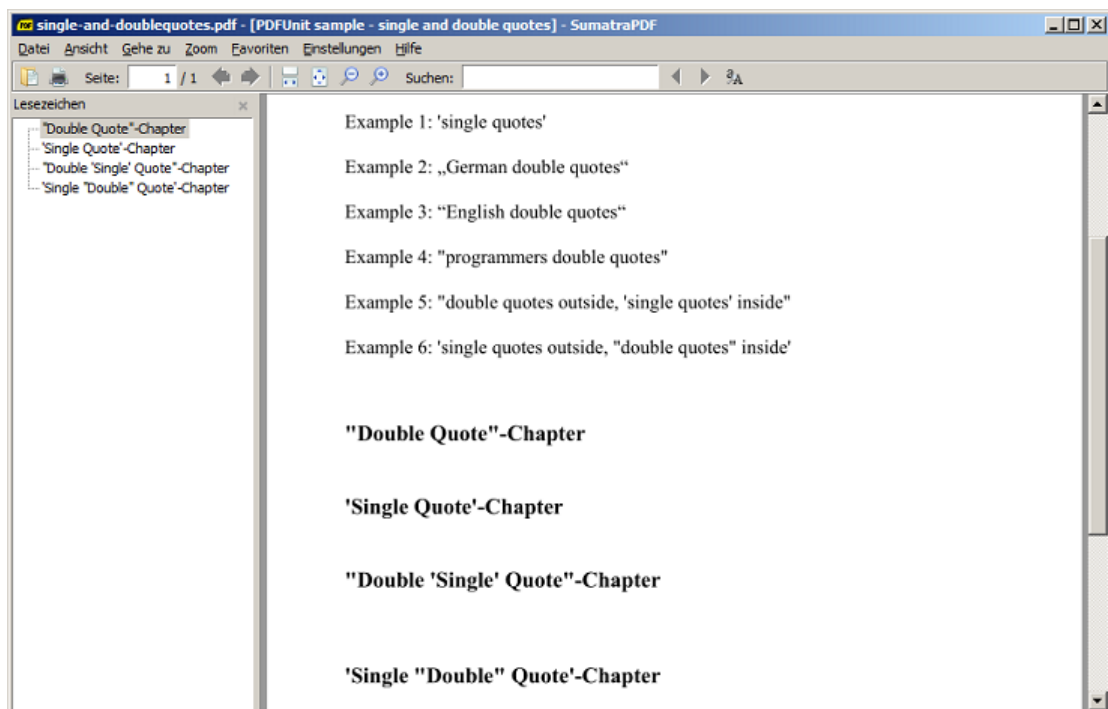
Example 2: „German double quotes“

Example 3: “English double quotes”

Example 4: "programmers double quotes"

“English” and „German” style quotation marks do not disturb the execution of tests. But during the creation of a test you could have problems typing them with your editor. Hint: copy the required quotation marks from a word-processor or an existing PDF document and paste them into your file.

The "programmers double quotes" need special attention because they are used as string delimiters in Java. The following paragraphs and examples go into detail about this. They are all based on the following document:



Valid Examples without XPath

'Single-Quotes', "English", and „German“ quotation marks within strings cause no problems. But "Double-Quotes" have to be escaped with a backslash:

```
@Test
public void hasText_SingleQuotes() throws Exception {
    String filename = 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.*\".*")
    ;
}
```

Valid Examples with XPath

Strings which are used as XPath expressions must **not contain** both single and double quotes in any combination. This condition is fulfilled in the following examples, so these tests run successfully:

```
@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)
    ;
}
```

Invalid Examples with XPath

Strings in the next examples contain both single and double quotes. Therefore, this error message appears at runtime: "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)
    ;
}
```

The error can only be avoided when you improve the XPath-expression like this:

```
@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. Defining Page Areas

Comparing text or rendered pages can be restricted to regions of one or more pages. Such an area is defined by four values: the x/y values of the **upper left** corner, the width and the height:

```
// Instantiating a clipping area
public ClippingArea(double upperLeftX, double upperLeftY, double width, double height)
public ClippingArea(double ulX, double ulY, double width, double height, FormatUnit init)
```

The units are described in chapter 13.8: “Format Units” (p. 155). If no unit is set, PDFUnit uses the unit `MILLIMETER`.

Here's an example:

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

It's easy to use a region of a page in a test. But it might be more difficult to find the right values for the region you need. PDFUnit provides the utility `RenderPdfClippingAreaToImage` to extract a page section into an image file (PNG). You can use that program using `mm` or `points`:

```
::
:: 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
```

The generated image needs to be checked. Does it contain the section you want? If not, change the parameters until they are right. Then you can copy the four values into your test.

13.8. Format Units

Some tests need width and height. You can use your favourite unit using one of the predefined constants:

```
// 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
```

If you omit the unit, the default unit `MILLIMETER` is used.

Format Units

```
@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)
    ;
}
```

Example - Sections of a Page

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

Example - Page Format

```
@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)
    ;
}
```

Example - Error messages

Error messages print both the original units and millimeters. For example, when you expected 111 POINTS for the width in the last example, you see the following error message:

```
Wrong page format in 'physical-map-of-the-world-1999_1117x863mm.pdf' on page 1.
Expected: 'height=1117.60, width=39.16 (as 'mm', converted from unit 'points')',
but was: 'height=1117.60, width=863.60 (as 'mm')'.
```

13.9. Error Messages

Error messages of PDFUnit provide detailed information to support bug fixing. And they are as clear and expressive as possible. A message for an incorrect page size demonstrates this intention:

```
Wrong page format in 'multiple-formats-on-individual-pages.pdf' on page 1.
Expected: 'height=297.00, width=210.00 (as 'mm')',
but was: 'height=209.90, width=297.04 (as 'mm')'.
```

The position of an error is marked by the double brackets <[and]>. To keep error messages readable, long values are shortened. The number of dropped characters is indicated as a number in the text surrounded by '...', e.g.:

```
The expected content does not match the JavaScript in 'javaScriptClock.pdf'.
Expected: '///<[Thisfileco...41...dbyPDFUnit]>',
but was: '///<[Constantsu...4969...];break;}}>'.
```

Two XML structures are compared internally using XMLUnit (<http://xmlunit.sourceforge.net>). XMLUnit's original error message is also shown by 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. Date Resolution

PDFUnit is able to compare dates (creation and modification dates) as year-month-day or additionally hour-minute-second. Two constants are available to choose between these options:

```
// Constants for date resolutions:

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

You can set a date resolution in the following methods:

```
// Date resolution in test methods:

.hasCreationDate().after(..., 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()
```

When comparing two PDF documents, date values are always compared using `DateResolution.DATE`.

13.11. Using the Default-Namespace

When running tests which are based on XML or XPath, namespaces which are declared with a prefix are detected automatically. Because the XML standard allows to declare namespaces multiple times, PDFUnit does not detect the default namespace. It has to be set using the class `com.pdfunit.DefaultNamespace`:

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

Note that there are two prefixes used for the same namespace, first `foo` and then `bar`. That seems strange, but the Java Standard requires an arbitrary prefix, which must not be omitted.

The next example shows the usage of a default namespace for an `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. Verify Configuration

As described in section 11.7: “Verifying the Configuration” (p. 142) PDFUnit provides a test method to check that all required libraries and files can be found in the classpath.

```

/*
 * 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");
}

```

The result is written into an HTML file whose name is part of the error message:

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
_InstallationTest	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'. 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)	0.078

[Properties >](#)

13.13. JAXP-Configuration

The standard configuration of JAXP can be changed in several ways. Because they are not all well known, they will be explained in the following sections using Xerces and Xalan.

The Java runtime reads the following environment variables and tries to load their values as classes:

```

"javax.xml.parsers.DocumentBuilderFactory"
"javax.xml.parsers.SAXParserFactory"
"javax.xml.transform.TransformerFactory"

```

These properties can be set in different ways, which are shown by the following numbered sections. If a configuration option can override another, the one with the higher is described first.

1. The JAXP properties can be set directly in a program:

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
    "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
System.setProperty("javax.xml.parsers.SAXParserFactory",
    "org.apache.xerces.jaxp.SAXParserFactoryImpl");
System.setProperty("javax.xml.transform.TransformerFactory",
    "org.apache.xalan.processor.TransformerFactoryImpl");
```

2. A configuration parameter for the Java process to be started can be set using the flag for system properties `-D` in combination with the special environment variable `_JAVA_OPTIONS`. The next example shows one of the three JAXP properties, but of course all three are possible:

```
set _JAVA_OPTIONS=-Djavax.xml.transform.TransformerFactory=
    org.apache.xalan.processor.TransformerFactoryImpl
```

3. The configuration with the start option `-D` can also be placed in the special environment variable `JAVA_TOOL_OPTIONS`. This variable is evaluated by some JDK implementations. The next example shows this option again with only one property:

```
set JAVA_TOOL_OPTIONS=-Djavax.xml.transform.TransformerFactory=
    org.apache.xalan.processor.TransformerFactoryImpl
```

4. The JAXP configuration can also be placed in the file `jaxp.properties` in the folder `JAVA_HOME/jre/lib`:

```
#
# Sample configuration, file %JAVA_HOME%\jre\lib\jaxp.properties.
#
# Defaults in Java 1.7.0, Windows:
#
#javax.xml.parsers.DocumentBuilderFactory = \
#    com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl
#javax.xml.parsers.SAXParserFactory = \
#    com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl
#javax.xml.transform.TransformerFactory = \
#    com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl
#
# Values for Xerces and Xalan:
#
javax.xml.parsers.DocumentBuilderFactory = \
    org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
javax.xml.parsers.SAXParserFactory = \
    org.apache.xerces.jaxp.SAXParserFactoryImpl
javax.xml.transform.TransformerFactory = \
    org.apache.xalan.processor.TransformerFactoryImpl
```

5. And a special technique for ANT is to use the environment variable `ANT_OPTS`:

```
set ANT_OPTS=-Djavax.xml.transform.TransformerFactory=
    org.apache.xalan.processor.TransformerFactoryImpl
set ANT_OPTS=-Djavax.xml.parsers.DocumentBuilderFactory=
    org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
set ANT_OPTS=-Djavax.xml.parsers.SAXParserFactory=
    org.apache.xerces.jaxp.SAXParserFactoryImpl
```

The declared XML and XSLT classes have to be found in the classpath or in the folder `JAVA_HOME/jre/lib/ext`. This last option is not recommended because only a few developers know about it. In case you are debugging a strange problem, you would not think to look at the version of the JAR's in that folder.

Note that Eclipse reads all existing environment variables only at startup and does not change anything later.

13.14. Version History

How it Starts...

During a custom seminar in August 2010 the conversation comes to the subject "testing of PDF documents". At that time I presented jPdfUnit 1.1 (<http://jpdfunit.sourceforge.net>), but the customer require-

ments could not be covered by this release. So I sent a wishlist with new functions to the project leader and a short time later I found myself in the role of an OpenSource developer. After analyzing the existing sources (December 2010) it was clear that the internally used Apache project PDFBox <http://pdfbox.apache.org> could not support the intended new functions. But it seems that iText (<http://itextpdf.com>) could.

2011

During 2011 I collected ideas and learnt more about PDF and iText. When I found some spare time I continued with the project.

At the end of 2011 I decided - after an intensive period of thinking and after having studied several license terms - to start a completely new project based on iText. The project runs parallel to my workshops. Now I could demonstrate that good concepts and principles can also be implemented in bigger projects than in the typical "hello world" projects of a workshop.

Release 2012.07

With time, the project developed into a powerful product, but the development stopped in the second half of the year due to pressures of work.

Release 2013.01

In the beginning of 2013 I had less workshops to give. During that time I created a developer manual and polished the existing functions. Perhaps surprisingly, I also fixed bugs.

Release 2014.06

Many small improvements led to a new version of PDFUnit-Java.

Release 2015.10

The new release contains some new functions. The main new feature is a graphical user interface called PDFUnit-Monitor. With the monitor non-developers can use PDFUnit. The monitor read the test information from one or more Excel files.

13.15. Unimplemented Features, Known Bugs

Problems with RTL Text Direction

Currently, text with the writing direction "right-to-left" (RTL), e.g. PDF documents with text in Hebrew or Arabic are not processed correctly. (Rotated text with the writing direction left-to-right can be tested without any problem.)

Text Overflow in Fields

In the current release 2054.10 text overflow in fields can not be detected when the last line of the text **starts inside** the field but extends outside.

Fields with the attribute `hidden`

The property `hidden` of a form field is evaluated incorrectly in some situations. This problem is currently under investigation using appropriate test documents.

Extraction of Field Information

Some properties of form fields are not extracted, for example "background color", "border color" and "border styles".

Extraction of Signature Information

Currently not all signature data is exported into XML. It is possible that the XML structure will change in future releases.

Color

In the current release 2054.10 colors cannot be tested directly. If colors have to be tested, the test has to be carried out using rendered pages. The sections 3.17: "Layout - Entire PDF Pages" (p. 52) and 3.18: "Layout - in Clipping Areas" (p. 54) describe such tests.

Content of Layers

Tests related to text and images are not restricted to layers.

Verifying PDF/A compliance

In one of the next releases PDFUnit will check the PDF/A compliance of a PDF document.

Complete XMP data

In the current release only the document-level XMP data are extracted and evaluated.

Index

A

- actions, 12
 - any action, 17
 - close, 13
 - compare, 83
 - equality of actions, 84
 - goto, 15
 - goto remote, 15
 - import data, 14
 - JavaScript, 14
 - launch data, 14
 - named actions, 15
 - open, 15
 - print, 16
 - reset form, 16
 - save, 16
 - submit form, 17
 - URI, 17
 - whitespace processing, 18
- attachments, 18
 - compare, 85
 - content, 19
 - existence, 19
 - extract to XML, 114
 - file name, 19
 - number, 19

B

- bookmarks, 21
 - compare, 86
 - destination, 22
 - destination pagenumber, 22
 - destination URI, 23
 - existence, 21
 - extract to XML, 116
 - label, 22
 - named destination, 22
 - no destination, 23
 - number, 22
 - verify with XML file, 23
 - verify with XPath, 23

C

- certificate (see 'signature/certificate')
- certified PDF, 24
- classpath, 137
 - in ANT, 139
 - in Eclipse, 138
 - in Maven, 139
- comparing attachments, 85
- comparing date values
 - creation date, 86
 - modification date, 86

- comparing form fields
 - content, 83
 - field names, 82
 - field properties, 82
 - quantity, 82
- comparing quantities of PDF elements, 93
- comparing text, 94, 149
 - in page sections, 94
 - whitespace, 95
- comparing with a master PDF, 81
 - actions, 83
 - attachments, 85
 - bookmarks, 86
 - creation date, 86
 - diff image, 91
 - fast web view, 98
 - fonts, 87
 - format, 88
 - form fields, 82
 - images, 89
 - images on individual pages, 89, 89
 - JavaScript, 90
 - modification date, 86
 - named destinations, 92
 - permissions, 92
 - quantities of countable PDF parts, 93
 - rendered pages, 90
 - rendered page section, 90
 - signature names, 94
 - tagging, 98
 - text, 94
 - text in page sections, 94
 - whitespace, 95
 - XFA data, 95
 - XMP data, 97
- comparing with a master-PDF
 - document properties, 87
- configuration
 - _JAVA_OPTIONS, 159
 - ANT_OPTS, 159
 - JAVA_TOOL_OPTIONS, 159
 - JAXP, 158
 - jaxp.properties, 159
 - locale, 141
 - output directory for diff images, 141
 - PDF internal date format, 140
 - verify, 142
 - verify as a unit test, 143
 - verify with script, 142, 158
- configuring ANT, 139
- configuring classpath, 136
- configuring Eclipse, 138
- configuring Maven, 139
- creation date, 24

D

date

- creation date, 24
- creation date of a certificate, 26
- existence, 25
- lower and upper limit, 26
- modification date, 24

date format

- configuration, 26

date resolution, 25, 157

default namespace, 79, 96, 111, 157

default namespaces, 76

define page area, 154

diff image, 91

document properties, 27

- compare, 87
- comparisons, 28
- custom property, 29
- test as key/value pair, 28

double quotes in strings, 151

E

encryption length, 58

equality

- of actions, 84
- of bookmarks, 86
- of document properties, 87
- of fonts, 31, 88
- of images, 89

error messages, 156

evaluation version, 136

even pages, 147

every page, 147

example

- caching of test documents, 133
- does a text fits into a form field, 128
- name of the former CEO, 129
- nesting depth of bookmarks, 134
- new logo on every page, 128
- Selenium and PDFUnit in combination, 130
- sign of the new CEO, 129
- validate HTML2PDF, 131

examples, 128

exception

- expected, 10

F

fast web view, 30

feedback, 8

field properties

- extract to XML, 113

first page, 147

fluent builder, 6

font properties

- extract to XML, 117

fonts, 30

- compare, 87
- names, 32
- number, 31
- properties to compare, 31
- types, 33
- verify with XML, 33
- verify with XPath, 33

font types, 33

format, 43

- compare, 88
- individual size, 44
- measuring units, 155
- multiple formats in one document, 44

format units

- Centimeter, 155
- DPI72, 155
- Inch, 155
- Millimeter, 155
- Points, 155

form field

- text overflow, 42

form fields, 34

- attribute hidden, 160
- compare, 82
- compare with XML file, 40
- content, 36
- existence, 35
- JavaScript actions, 39
- name, 35
- number, 35
- properties, 38
- size, 38
- text overflow, 160
- type, 37
- Unicode, 39
- verify with XPath, 41

I

images, 45

- compare, 89
- compare with an array, 47
- compare with file, 46
- extract from PDF, 118
- number of different images, 45
- number of visible images, 46
- on specified pages, 47

installation, 135

- configuring classpath, 136
- iText, 135
- license key, 136
- new release, 143
- order a license key, 136
- PDFUnit-Java, 135
- with different iText version, 137
- with existing iText, 137

- without existing iText, 135
- instantiation of PDF documents, 147
- iText installation, 135

J

- JavaScript, 48
 - compare, 90
 - compare substrings, 49
 - compare with a text file, 48
 - existence, 48
 - extracting, 119

L

- language, 49
- last page, 147
- layer
 - duplicate names, 52
 - name, 51
 - number, 51
- layers, 50
- layout
 - compare, 90
 - entire pages, 52
 - page section, 54
- license key
 - classpath, 140
 - installation, 136
 - order, 136
- line breaks in text, 18, 67, 150

M

- measuring units, 155
- meta data (see 'document properties')
- modification date, 24
- multiple documents, 99

N

- named destination, 21, 21
 - compare, 92
 - extract to XML, 120
- number of PDF parts, 55

O

- odd pages, 147
- overview
 - comparing with a master PDF, 81
 - test scope, 11
 - utilities, 112
- owner password, 57

P

- page area
 - define, 154
- page numbers as objectives, 56
- page numbers with lower and upper limit, 69

- pages
 - comparing as rendered images, 90
 - render to PNG, 125
- page section
 - example, 70
 - layout, 54
 - measuring unit, 156
 - render to PNG, 124
 - validate layout, 54
 - validate text, 67
- page selection, 147
 - individual pages, 148
 - open range, 148
 - range, 148
- password as a test goal, 58
- PdfDIFF, 103
- PDFUnit-Monitor, 101
 - compare with master, 103
 - export, 104
 - filtering, 102
 - import, 104
 - message details, 102
- PDFUnit-NET, 146
- PDFUnit-Perl, 145
- PDFUnit-XML, 145
- permission, 58
- permissions
 - compare, 92

Q

- quickstart, 9
- quotes in strings, 151

R

- regular expressions, 149

S

- signature/certificate, 59
 - compare names, 94
 - compare with XML file, 62
 - existence, 60
 - extract to XML, 121
 - name, 60
 - number, 60
 - reason, 61
 - revision, 61
 - signer name, 61
 - validity date, 60
 - verify with XPath, 62
- spaces in text, 67
- syntax
 - introduction, 10
- system properties, 140

T

- tagging, 64

technical requirements, 135
 TestNG, 147
 text in page sections, 70
 text overflow, 42

- all fields, 43
- one field, 42
- technical constraint, 43

 text - vertical, angular, overhead, 71
 trapping, 71

U

Unicode, 105

- compare with XML file, 105
- convert to hex code, 112
- in error messages, 108
- invisible characters, 109
- long text, 105
- single characters, 105
- UTF-8 (ANT), 107
- UTF-8 (console), 106
- UTF-8 (Eclipse), 107
- UTF-8 (Maven), 107
- verify with XPath, 106

 uninstall, 144
 update, 143
 user password, 57
 utilities, 112

- convert Unicode to hex code, 112
- extract attachments, 114
- extract bookmarks, 116
- extract field properties, 113
- extract font properties, 117
- extract images from PDF, 118
- extract JavaScript, 119
- extract named destinations, 120
- extract signature data, 121
- extract XFA data, 122
- extract XMP data, 123
- render PDF pages, 125
- render PDF sections, 124

V

validate text, 65

- absence of text, 66
- empty pages, 67
- in page sections, 67
- line break, blanks, 67
- multiple search items, 68
- on all pages, 66
- on individual pages, 65
- page numbers with lower and upper limit, 69

 validity date of a certificate, 60
 verify installation, 136
 version info, 73

- upcoming versions, 73
- version ranges, 73

W

whitespace in text, 18, 150
 whitespace processing, 18, 67, 150

- IGNORE, KEEP, NORMALIZE, 150, 150

 writing direction (right-to-left), 160

X

XFA data, 74

- compare, 95
- compare with an XML file, 74
- default namespaces, 76
- existence, 74
- extract to XML, 122
- verify single nodes, 74
- verify with XPath, 75

 XML

- default namespace, 111
- extract data, 110
- namespace, 110

 XMLUnit, 110, 156
 XMP data, 77

- compare, 97
- compare with an XML file, 77
- default namespace, 79
- existence, 77
- extract to XML, 123
- verify single nodes, 78
- verify with XPath, 78

 XPath

- compatibility, 111
- general annotations, 110
- result type, 111

 XPath result type, 95, 97

- boolean, 96