Automated PDF Tests with XML

PDFUnit-XML

Carsten Siedentop

Table of Contents

Preface

The Current Situation of Testing PDF in Projects

These days telephone bills, insurance policies, official notifications and many types of contract are delivered as PDF documents. They are the result of a process chain consisting of programs in various programming languages using numerous libraries. Depending on the complexity of the documents to be produced, 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-XML is tightly coupled to the API of PDFUnit-Java and follows the "Fluent Builder" principle. The names of the XML tags are similar to everyday speech to support general thought patterns. This results in XML that is easy to understand for a long time.

The next example shows the simplicity of the API:

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

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 to write successful tests.

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 members of the quality assurance staff who have to test PDF documents but who are not Java programmers.

We assume that you have some knowledge of and experience with XML. And having a basic understanding of test automation is helpful, but not required.

Code Examples

A demo project containing many examples is available here: http://www.pdfunit.com/en/download/index.html.

XML-Structur

An XML Schema exists to validate PDFUnit. It's documentation is available online: http://www.pdfunit.com/en/api/xml/index.html.

Other Programming Languages

PDFUnit is available not only for XML, but also for Java and Perl. 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 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.

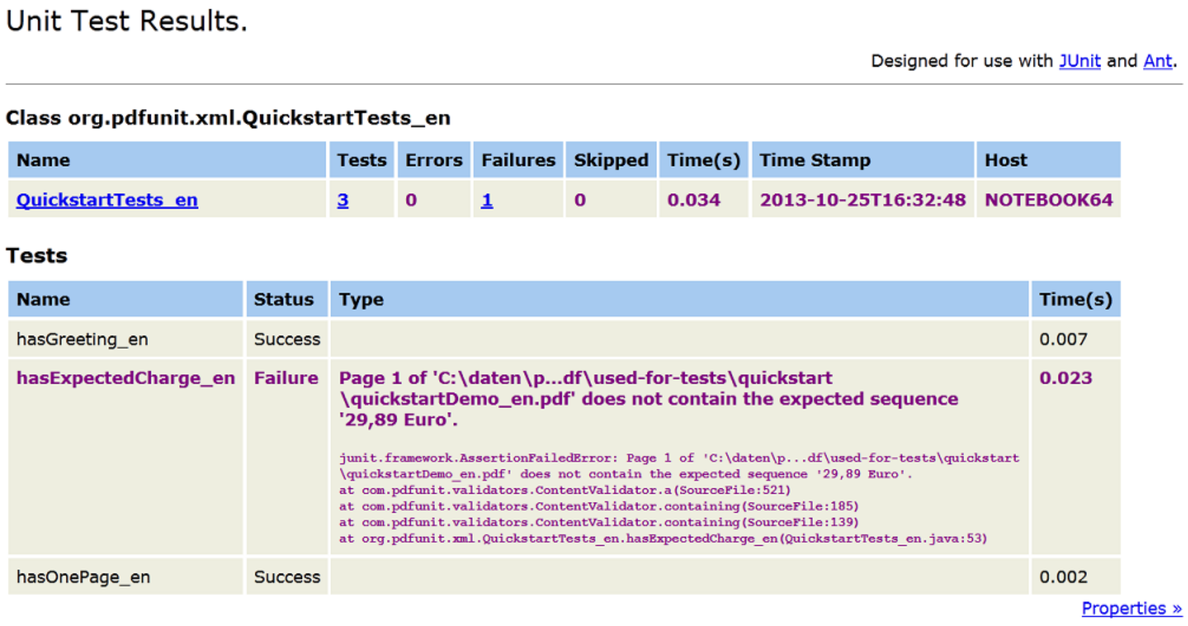
Chapter 2. Quickstart

Quickstart

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

```xml
<testcase name="hasOnePage_en">
  <assertThat testDocument="quickstart/quickstartDemo_en.pdf">
    <hasNumberOfPages>1</hasNumberOfPages>
  </assertThat>
</testcase>

<testcase name="hasGreeting_en">
  <assertThat testDocument="quickstart/quickstartDemo_en.pdf">
    <hasText on="LAST_PAGE">
      <containing>Thank you for using our services.</containing>
    </hasText>
  </assertThat>
</testcase>

<testcase name="hasExpectedCharge_en">
  <assertThat testDocument="quickstart/quickstartDemo_en.pdf">
    <hasText on="FIRST_PAGE">
      <inClippingArea upperLeftX="172" upperLeftY="178" width="20" height="9">
        <containing>29.89 Euro</containing> <!-- This value is intentionally false. -->
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

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



Unit Test Results.

Designed for use with JUnit and Ant.

Class org.pdfunit.xml.QuickstartTests_en

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
QuickstartTests_en	3	0	1	0	0.034	2013-10-25T16:32:48	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
hasGreeting_en	Success		0.007
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.a(SourceFile:521) at com.pdfunit.validators.ContentValidator.containing(SourceFile:185) at com.pdfunit.validators.ContentValidator.containing(SourceFile:139) at org.pdfunit.xml.QuickstartTests_en.hasExpectedCharge_en(QuickstartTests_en.java:53)	0.023
hasOnePage_en	Success		0.002

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 tag `<testcase name=".." />`, containing its name in the attribute `name=".."`. Then the tag `<assertThat testDocument=".." />` is nested, which defines which PDF file should be tested.

You can place other tags inside of `<assertThat>` for various test scopes, e.g. visible content, fonts, layout.

The following examples show the basic syntax for different types of tests. You will replace “XXX” with real terms which are listed at the end of this chapter.

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

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

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

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

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

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

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

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

It is possible to write a test for a given set of PDF documents. Such tests starts with the method `<assertThatEachDocument>`:

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

The chapter 5: “Tests with Multiple Documents” (p. 90) describes more about tests on multiple documents.

Exception

Tests that expect an exception have to declare that with the attribute `errorExpected="YES"`:

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

Test Scopes

The following list gives a complete overview of the test scopes of PDFUnit. The link behind each tag points to the relevant chapter. The list is sorted alphabetically.

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

<areBothForFastWebView />       4.20: “Further Comparisons” (p. 88)
<asRenderedPage />              3.17: “Layout - Entire PDF Pages” (p. 48)

<containsImage />               3.13: “Images in PDF Documents” (p. 41)
<containsOneOfTheseImages />    3.13: “Images in PDF Documents” (p. 41)

<hasAnyAction />                3.2: “Actions” (p. 11)
<hasAuthor />                   3.7: “Document Properties” (p. 24)
<hasBookmark />                 3.4: “Bookmarks and Named Destinations” (p. 19)
<hasBookmarks />                3.4: “Bookmarks and Named Destinations” (p. 19)
<hasChainedAction />            3.2: “Actions” (p. 11)
<hasCloseAction />              3.2: “Actions” (p. 11)
<hasCreationDate />             3.6: “Dates” (p. 23)
<hasCreationDateAfter />        3.6: “Dates” (p. 23)
<hasCreationDateBefore />       3.6: “Dates” (p. 23)
<hasCreator />                  3.6: “Dates” (p. 23)
<hasEmbeddedFile />             3.3: “Attachments” (p. 17)
<hasEmbeddedFileContent />      3.3: “Attachments” (p. 17)
<hasEncryptionLength />         3.21: “Passwords” (p. 53)
<hasField />                    3.10: “Form Fields” (p. 31)
<hasFields />                   3.10: “Form Fields” (p. 31)
<hasFont />                     3.9: “Fonts” (p. 28)
<hasFonts />                    3.9: “Fonts” (p. 28)
<hasFormat />                   3.12: “Format” (p. 39)
<hasImportDataAction />         3.2: “Actions” (p. 11)
<hasJavaScript />               3.14: “JavaScript” (p. 43)
<hasJavaScriptAction />         3.2: “Actions” (p. 11)
<hasKeywords />                 3.7: “Document Properties” (p. 24)
<hasLaunchAction />             3.2: “Actions” (p. 11)
<hasLayer />                    3.16: “Layers” (p. 46)
<hasLayers />                   3.16: “Layers” (p. 46)
<hasLessPages />                3.20: “Page Numbers as Objectives” (p. 52)
<hasLocale />                   3.15: “Language” (p. 45)
<hasLocalGotoAction />          3.2: “Actions” (p. 11)
<hasModificationDate />         3.6: “Dates” (p. 23)
<hasModificationDateAfter />    3.6: “Dates” (p. 23)
<hasModificationDateBefore />   3.6: “Dates” (p. 23)
<hasMorePages />                3.20: “Page Numbers as Objectives” (p. 52)
<hasNamedAction />              3.2: “Actions” (p. 11)
<hasNamedDestination />         3.4: “Bookmarks and Named Destinations” (p. 19)

...  continued
... continuation:

<hasNoAuthor />                 3.7: “Document Properties” (p. 24)
<hasNoCreationDate />           3.6: “Dates” (p. 23)
<hasNoCreator />                3.7: “Document Properties” (p. 24)
<hasNoKeywords />               3.7: “Document Properties” (p. 24)
<hasNoLocale />                 3.15: “Language” (p. 45)
<hasNoModificationDate />       3.6: “Dates” (p. 23)
<hasNoProducer />               3.7: “Document Properties” (p. 24)
<hasNoProperty />               3.7: “Document Properties” (p. 24)
<hasNoSubject />                3.7: “Document Properties” (p. 24)
<hasNoText />                   3.25: “Text” (p. 60)
<hasNoTitle />                  3.7: “Document Properties” (p. 24)
<hasNoXFAData />                3.30: “XFA Data” (p. 68)
<hasNoXMPData />                3.31: “XMP Data” (p. 70)

<hasNumberOfXXX />              3.19: “Number of PDF Elements” (p. 51)

<hasOCG />                      3.16: “Layers” (p. 46)
<hasOCGs />                     3.16: “Layers” (p. 46)
<hasOpenAction />               3.2: “Actions” (p. 11)
<hasOwnerPassword />            3.21: “Passwords” (p. 53)
<hasPermission />               3.22: “Permissions” (p. 54)
<hasPrintAction />              3.2: “Actions” (p. 11)
<hasProducer />                 3.7: “Document Properties” (p. 24)
<hasProperty />                 3.7: “Document Properties” (p. 24)
<hasRemoteGotoActionTo />       3.2: “Actions” (p. 11)
<hasResetFormAction />          3.2: “Actions” (p. 11)
<hasSaveAction />               3.2: “Actions” (p. 11)
<hasSignature />                3.23: “Signatures and Certificates” (p. 55)
<hasSignatures />               3.23: “Signatures and Certificates” (p. 55)
<hasSignedSignatureFields />    3.23: “Signatures and Certificates” (p. 55)
<hasSubject />                  3.7: “Document Properties” (p. 24)
<hasSubmitFormAction />         3.2: “Actions” (p. 11)
<hasText />                     3.25: “Text” (p. 60)
<hasTitle />                    3.7: “Document Properties” (p. 24)
<hasTrappingInfo />             3.28: “Trapping Info” (p. 66)
<hasUnsignedSignatureFields />  3.23: “Signatures and Certificates” (p. 55)
<hasURIAction />                3.2: “Actions” (p. 11)
<hasUserPassword />             3.21: “Passwords” (p. 53)
<hasVersion />                  3.29: “Version Info” (p. 67)
<hasXFAData />                  3.30: “XFA Data” (p. 68)
<hasXMPData />                  3.31: “XMP Data” (p. 70)

<haveSame... />                 4.1: “Overview” (p. 74)

<isCertified />                 3.5: “Certified PDF” (p. 22)
<isLinearizedForFastWebView />  3.8: “Fast Web View” (p. 27)
<isSigned />                    3.23: “Signatures and Certificates” (p. 55)
<isTagged />                    3.24: “Tagged Documents” (p. 59)

...  (end of list)
```

PDFUnit is continuously being improved and we keep the manul up to date. Wishes and requests for new functions are appreciated. Please send them to 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
```

PDFUnit provides tags for some of these actions:

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

<hasNumberOfActions />
<hasNumberOfJavaScriptActions />

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

...  continued
```

```
... continuation:

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

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

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

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

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

...  continue
... continuation

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

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

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

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

...  continued
```

```
... continuation:

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

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

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

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

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

...  (end of list)
```

The following sections show examples for different types of actions.

Close-Actions

Close-Actions are executed when the document is being closed:

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

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

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

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

The attribute `whitespaces=".."` is optional. The default whitespace processing is `NORMALIZE`.

ImportData-Actions

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

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

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:

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

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:

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

PDFUnit compares the content of the attributes `application=".."` and `operation=".."` 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:

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

Goto-Actions

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

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

The test is successful, when the current test PDF contains the expecte destination “destination2.1”.

GotoRemote-Actions

GotoRemote-Actions need a destination in another PDF file.

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

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

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.

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

In addition to the tags for comparing text, Open-Actions can be tested using the tag <withDestinationTo />:

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

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.

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

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:

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

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`.

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

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:

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

PDFUnit 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:

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

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

Any Action

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

```
<testcase name="hasAnyAction_DifferentKindOfActions">
  <assertThat testDocument="actions/chainedActions.pdf">
    <hasAnyAction>
      <matchingComplete>app.alert('Demo: the first action of five.');</matchingComplete>
      <matchingComplete>http://www.google.de</matchingComplete>
      <matchingComplete>c:/windows/notepad.exe</matchingComplete>
      <matchingComplete>this.print(true);</matchingComplete>
    </hasAnyAction>
  </assertThat>
</testcase>
```

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:

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

The chapter 13.4: “Whitespace Processing” (p. 135) explains the flexible handling of whitespaces.

3.3. Attachments

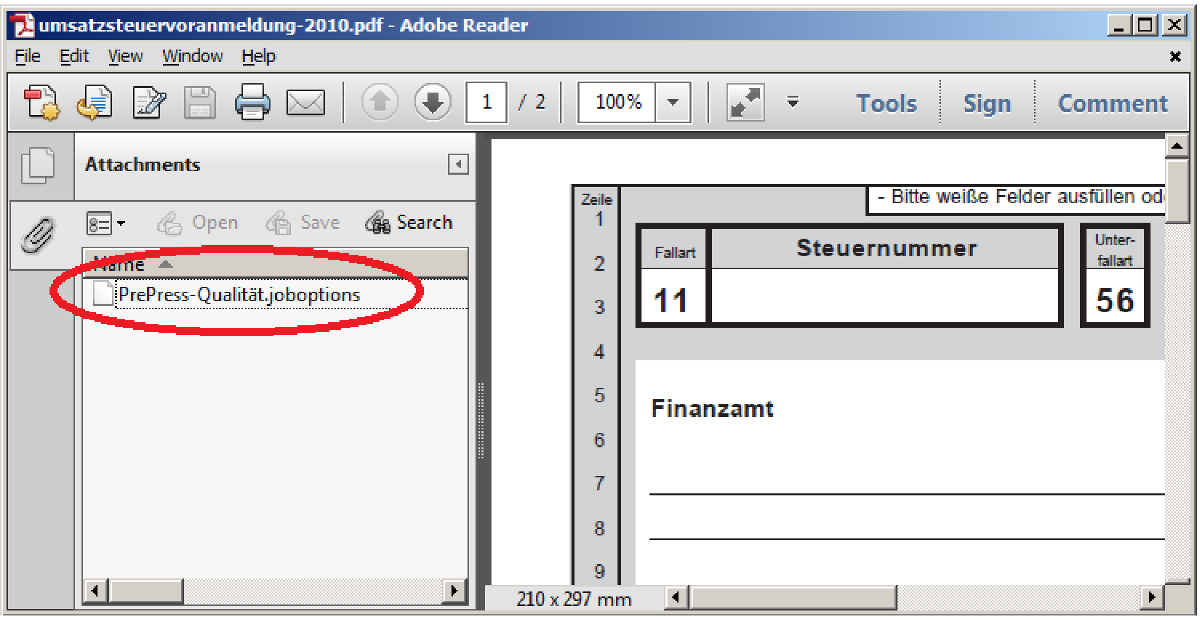
Overview

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

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

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

Existence of Attachments

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”.


umsatzsteuervoranmeldung-2010.pdf - Adobe Reader
File Edit View Window Help
1 / 2
100%
Tools
Sign
Comment
Attachments
Open
Save
Search
Name
PrePress-Qualität.joboptions
Zeile
1
2
3
4
5
6
7
8
9
- Bitte weiße Felder ausfüllen od
Fallart
11
Steuernummer
Unter-
fallart
56
Finanzamt
210 x 297 mm

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

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

Number of Attachments

The next test verifies the expected number of embedded files:

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

Filename

Also the names of embedded files can be tested:

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

Content

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

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

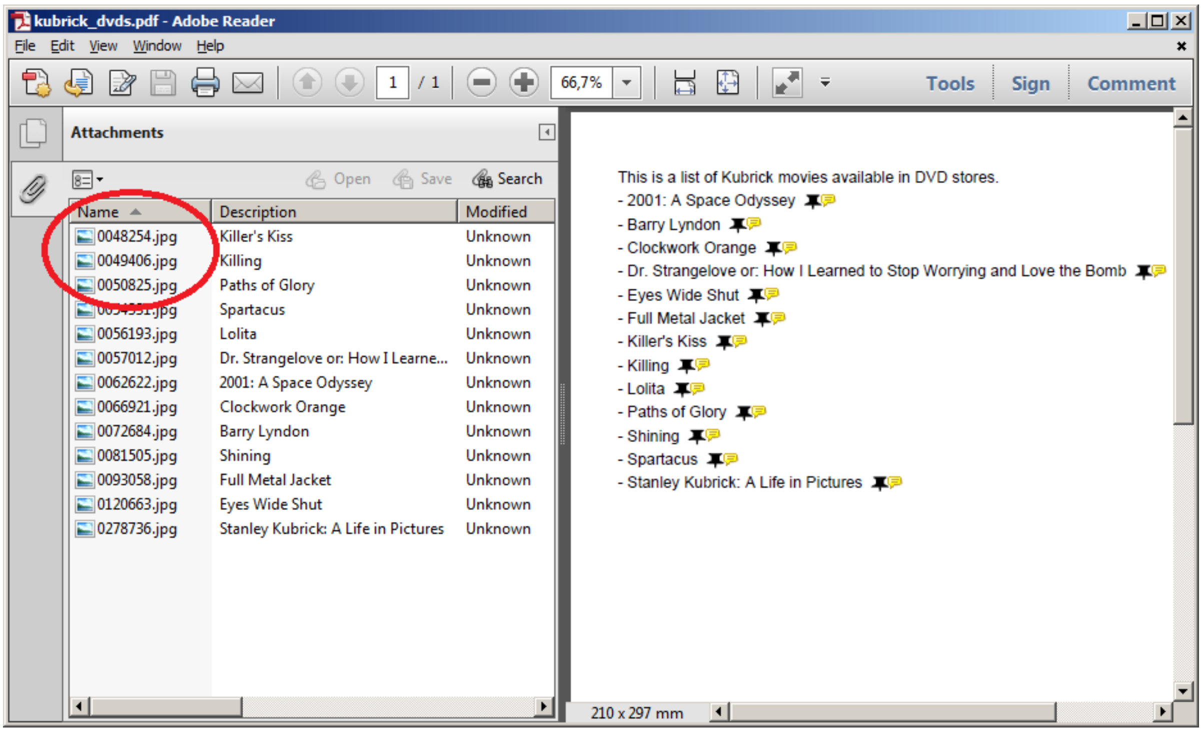
The comparison is carried out byte-wise.

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. 104):

Multiple filenames can be tested with one test.

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

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



Name	Description	Modified
0048254.jpg	Killer's Kiss	Unknown
0049406.jpg	Killing	Unknown
0050825.jpg	Paths of Glory	Unknown
0054331.jpg	Spartacus	Unknown
0056193.jpg	Lolita	Unknown
0057012.jpg	Dr. Strangelove or: How I Learne...	Unknown
0062622.jpg	2001: A Space Odyssey	Unknown
0066921.jpg	Clockwork Orange	Unknown
0072684.jpg	Barry Lyndon	Unknown
0081505.jpg	Shining	Unknown
0093058.jpg	Full Metal Jacket	Unknown
0120663.jpg	Eyes Wide Shut	Unknown
0278736.jpg	Stanley Kubrick: A Life in Pictures	Unknown

This is a list of Kubrick movies available in DVD stores.

- 2001: A Space Odyssey
- Barry Lyndon
- Clockwork Orange
- Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb
- Eyes Wide Shut
- Full Metal Jacket
- Killer's Kiss
- Killing
- Lolita
- Paths of Glory
- Shining
- Spartacus
- Stanley Kubrick: A Life in Pictures

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.

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

<hasNumberOfBookmarks />
<hasBookmarks />

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

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

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

We can see bookmarks as **starting points** and “named destinations” as the **landing points**. Named destinations 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 tags are available:

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

<hasNamedDestination />

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

Named Destinations

The names of named destinations can be tested easily:

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

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:

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

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

Existence of Bookmarks

It is easy test to verify the existence of bookmarks:

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

Number of Bookmarks

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

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

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:

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

Destinations of Bookmarks

Bookmarks can have different kinds of destinations. A suitable attribute is provided for each destination inside the tag `<hasBookmark />`.

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

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

The attribute `linkingToPage=".."` can only be used together with the attribute `withLabel=".."`. In such a test the given label has to point to the expected page number.

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

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

Does a bookmark exist which points to an expected destination:

```

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

Is there a bookmark pointing to a URI:

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

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.
-->
<testcase name="hasBookmark_WithoutDeadLink">
  <assertThat testDocument="bookmarks/diverseContentOnMultiplePages.pdf">
    <hasBookmark withoutDeadLink="YES" />
  </assertThat>
</testcase>
```

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.

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

❶ When comparing PDF parts against any XML, whitespaces and comments are ignored.

Bookmark information can also be verified using individual XPath expressions:

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

3.5. Certified PDF

Overview

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

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.

PDFUnit provides this tag:

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

<isCertified for=".."   (optional)
/>

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

PDFUnit's constants correspond to iText's `PdfSignatureAppearance.CERTIFIED_*`.

Important hint, don't confuse “certified PDF” with the “certificate” of a signature.

Examples

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

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

Next you can check the level of certification:

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

3.6. Dates

Overview

Date values in PDF documents are normally not a matter to be tested. But when you want to test them anyway, it is not easy because date values are formatted in many different ways.

Tests in the following listings only handle the creation date because tests for the modification date have exactly the same syntax.

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

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

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

<hasNoCreationDate         />
<hasNoModificationDate     />
```

Existence of a Date

The first test checks that a creation date exists.

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

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

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

Date Resolution

You have to use the attribute `resolution` to declare which parts of a date should be used for the test. The attribute can have the values `resolution="DATE"` or `resolution="DATETIME"`. Using the constant `DATE` date values were compared using year, month and day. And using `DATETIME` hour, minute and second are alos considered.

Here an example:

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

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

The format of the expected date is defined by XML Schema (xs:date). But the PDF internal date format may be different, so a suitable format string for the internal date has to be declared in the file “config.properties”.

When you omit the attribute resolution=".." the default resolution="DATE" is used.

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 “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 the tag <hasProperty />:

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

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:

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

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

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. 55) 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 tags are available:

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

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

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

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

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. 79).

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”:

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

Use the tag `<hasNoAuthor />` to verify that the property “author” does not exist:

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

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

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

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

```
<!-- Comparing text for author, creator, keywords, producer, subject, title: -->

<containing       />
<endingWith       />
<matchingComplete />
<matchingRegex    />
<notContaining    />
<notMatchingRegex />
<startingWith     />
```

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

Each comparison is case sensitive.

The implementation of the tag <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.

Each property has it's own tag <hasXXX /> and <hasNoXXX />.

You can combine multiple tags in one test:

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

But such a test is not recommended because the name of the test is not specific enough.

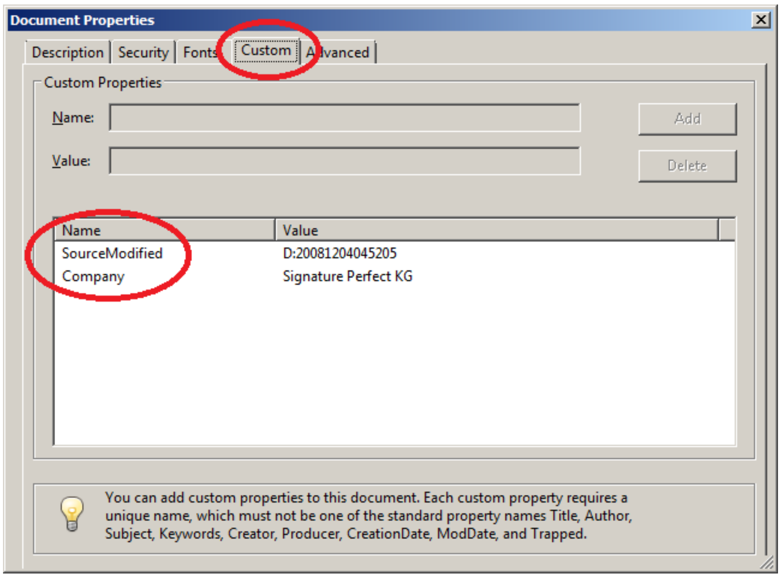
Common Validation as a Key-Value Pair

All tests for document properties shown in the previous sections can also be implemented with the general tag <hasProperty />:

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

`<hasProperty />` validates any document property as a key-value pair:

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



Name	Value
SourceModified	D:20081204045205
Company	Signature Perfect KG

You can add custom properties to this document. Each custom property requires a unique name, which must not be one of the standard property names Title, Author, Subject, Keywords, Creator, Producer, CreationDate, ModDate, and Trapped.

And this is the test for custom properties:

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

To ensure that a property does **not exist**, see the following test:

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

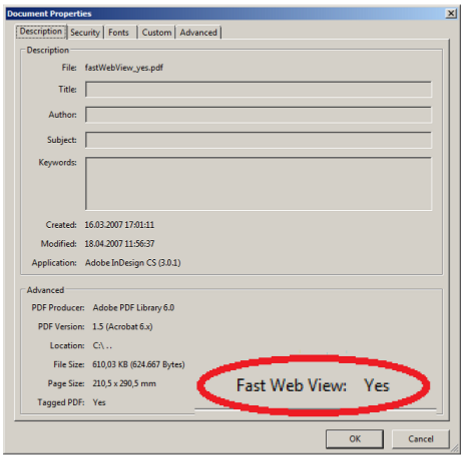
PDF documents of version PDF-1.4 or higher can have metadata as XML (Extensible Metadata Platform, XMP). Chapter 3.31: “XMP Data” (p. 70) explains that in detail.

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®.



Description

File: fastWebView_yes.pdf
Title:
Author:
Subject:
Keywords:
Created: 16.03.2007 17:01:11
Modified: 18.04.2007 11:56:37
Application: Adobe InDesign CS (3.0.1)

Advanced

PDF Producer: Adobe PDF Library 6.0
PDF Version: 1.5 (Acrobat 6.x)
Location: C:\ ..
File Size: 610,03 KB (624.667 Bytes)
Page Size: 210,5 x 290,5 mm
Tagged PDF: Yes
Fast Web View: Yes

PDFUnit looks for a PDF-object (dictonary) 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 results in a “green bar”.

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

<isLinearizedForFastWebView />
```

Example

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

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 tags for different requirements.

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

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

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

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

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: -->

identifiedBy="ALLPROPERTIES"
identifiedBy="BASENAME"
identifiedBy="BASENAME_ENCODING"
identifiedBy="BASENAME_ENCODING_ENCODINGDIFF"
identifiedBy="CONVERTIBLE2UNICODE"
identifiedBy="EMBEDDED"
identifiedBy="EMBEDDED_CONVERTIBLE2UNICODE"
identifiedBy="NAME"
identifiedBy="NAME_TYPE"
identifiedBy="TYPE"
```

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 `/Differences`.
`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.
`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:

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

Font Names

Testing the names of fonts are easy:

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

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.

You can test multiple font names in one test:

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

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

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

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:

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

Predefined font types are:

```
<!-- constants for font types -->

ofThisTypeOnly="CID"
ofThisTypeOnly="CID_TYPE0"
ofThisTypeOnly="CID_TYPE2"
ofThisTypeOnly="CJK"
ofThisTypeOnly="MMTYPE1"
ofThisTypeOnly="OPENTYPE"
ofThisTypeOnly="TRUETYPE"
ofThisTypeOnly="TYPE0"
ofThisTypeOnly="TYPE1"
ofThisTypeOnly="TYPE3"
```

XML 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="FGNNPL+ArialMT"       baseFontName="ArialMT"
        type="TrueType"             embedded="true"
        encoding="WinAnsiEncoding"  convertibleToUnicode="false"
  />
  ...
</fontlist>
```

Here is a test based on that XML file:

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

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:

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

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. 100).

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:

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

<hasField   withName                (required)

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

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

            withType".."            (optional)

/>

...  continued
```

```
... continuation

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

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

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

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

❶ This test is described separately in chapter 3.11: “Form Fields - Text Overflow” (p. 37):

Existence of Fields

The following test verifies whether or not fields exist:

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

Number of Fields

If you only need to verify the number of fields, you can use the tag `<hasNumberOfFields />`:

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

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

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

Name of Fields

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

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

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 test to check the absence of duplicate names.

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

Content of Fields

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

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

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

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

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

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

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

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

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

Type of Fields

Each field has a type. Although a field type is not as important as the name, it can be tested with the attribute `withType=".."`:

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

Available field types are defined as constants for the attribute `withType`. 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"
```

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:

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

Datailed tests for signatures and certificates are described in the chapter 3.23: “Signatures and Certificates” (p. 55):

Field Size

If the size of form fields is important, check it using the attributes `width=".."` and `height=".."`:

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

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

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

❶❷ 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. 37) 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 tests have to be part of every PDF testing tool. The following example shows the principle.

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

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

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

hasMultipleLines="YES"
hasSingleLine="YES"

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

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

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

JavaScript 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.

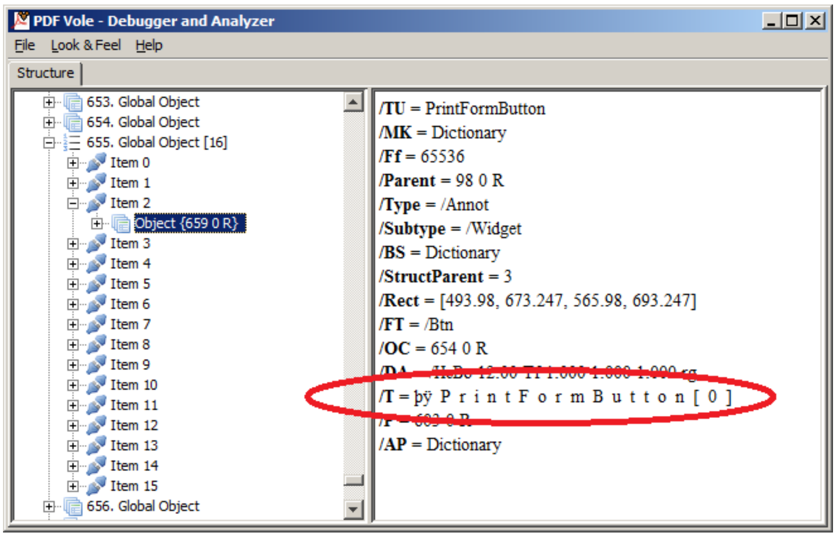
PDFUnit can verify that a field is linked with an action:

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

Tags to validate the JavaScript itself are described in chapter: 3.14: “JavaScript” (p. 43).

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:



/TU = PrintFormButton
/MK = Dictionary
/Ff = 65536
/Parent = 98 0 R
/Type = /Annot
/Subtype = /Widget
/BS = Dictionary
/StructParent = 3
/Rect = [493.98, 673.247, 565.98, 693.247]
/FT = /Btn
/OC = 654 0 R
/DA = /HeBo 12.00 Tf 1.000 1.000 1.000 rg
/T = þÿ P r i n t F o r m B u t t o n [0]
/P = 603 0 R
/AP = Dictionary

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.
 -->

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

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

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. 103) describes how to extract information about all fields of a PDF document. You can compare the field properties in a PDF document with the extracted XML file:

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

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:

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

The tag `<matchingXPath />` can be used multiple times in one test.

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

The following two example check whether unsigned signature fields exist:

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

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

PDFUnit 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.

3.11. Form Fields - Text Overflow

Overview

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

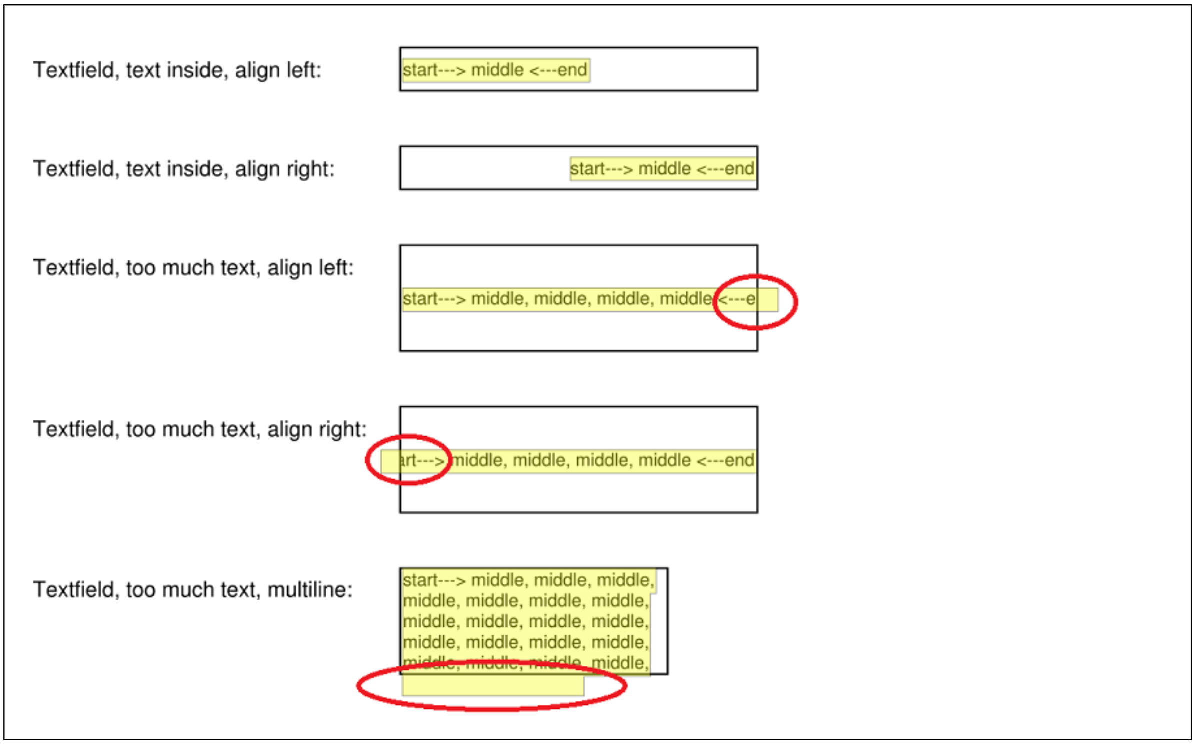
Verifying if text fits in a field is not easy because the required space depends not only on the font size, but also on the length of words at the end of each line and whether hyphenation is used. Following requests by various users, PDFUnit now provides these two tests:

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

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

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

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

Correct Size of a Field

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



Textfield, text inside, align left: start---> middle <---end

Textfield, text inside, align right: start---> middle <---end

Textfield, too much text, align left: start---> middle, middle, middle, middle <---e

Textfield, too much text, align right: art---> middle, middle, middle, middle <---end

Textfield, too much text, multiline: start---> middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle, middle,

And this is the test for the last field:

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

If you didn't expect an exception (by including errorExpected="YES"), 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 tag:

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

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 tag <containing>..</containing>. If you anyhow want to verify it, the PDF document must first be flattened.

Therfore, the following test fails:

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

3.12. 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 ... ;-(

That is the reason why PDFUnit provides the tag <hasFormat /> to verify the format:

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

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

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

Documents with one Page Format

A page format can be checked with predefined constants:

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

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

❶❷ Constants exist for many popular formats.

You can also verify individual formats:

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

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

❸❹ You can use the measurement units `POINTS`, `MILLIMETER`, `CENTIMETER`, `INCH` and `DPI72`. The units `DPI72` and `POINTS` are equivalent. All units must be declared in the attribute `unit=".."`.

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:

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

The format tests can be restricted to individual pages or page ranges as described in chapter 13.2: “Page Selection” (p. 133):

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

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

3.13. 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 tags:

```
<!-- Tags for image tests: -->

<hasNumberOfDifferentImages />
<hasNumberOfVisibleImages />
<containsImage file=".."   (required)

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

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 tags. The tag `<hasNumberOfDifferentImages />` validates the number of images **stored internally** and the tag `<hasNumberOfVisibleImages />` validates the number of **visible** images.

Number of different Images inside PDF

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

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

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. 108) describes this topic in detail.

Number of visible Images inside a PDF

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

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

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:

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

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

The possibilities for limitting tests to specified pages are described in chapter 13.2: “Page Selection” (p. 133).

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:

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

The result of a comparison of two images depends on their file formats. PDFUnit can handle `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.

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 stored as PNG.
- Verify the picture you want to use.
- Use PDFUnit as demonstrated in the listing above.

Use Multiple 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 tag `<containsOneOfTheseImages />` to test such a situation:

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

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

Validate Images on Specified Pages

The tests for images can be restricted to single pages, multiple individual or multiple contiguous pages. All possibilities are described in chapter 13.2: “Page Selection” (p. 133).

Here are some examples:

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

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

Tags can be used multiple times. But it might be better to write two separate tests:

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

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. 81).

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.

You can test JavaScript using the following tag:

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

<hasJavaScript>
  <containing       /> (optional)
  <equals           /> (optional)
  <matchingComplete /> (optional)
</hasJavaScript>
```

Existence of JavaScript

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

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

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:

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

The expected JavaScript need not necessarily be read from a file. You can type it directly inside the tag:

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


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

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

        ... (code shortened for presentation)

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

Comparing Substrings

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

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

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

Whitespaces 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.

The language of a PDF document can easily be checked using the following tag:

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

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

<hasNoLocale />
```

Examples

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

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

❶ Notation typical for PDF
❷ Notation typical for Java

The string for the language is not case sensitive. Underscore and hyphen are equivalent.

You can also use a value of `java.util.Locale`. But that is case sensitivity:

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

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

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:

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

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 tags to test layers:

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

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

<hasNumberOfLayers />, <hasNumberOfOCGs />

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

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

The tag `<endingWith />` is not provided because duplicate layer names are expanded internally with a suffix and thus the end of a name is not predictable.

A tag `matchingRegex />` is also not provided because layer names are usually short.

Number of Layers

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

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

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:

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

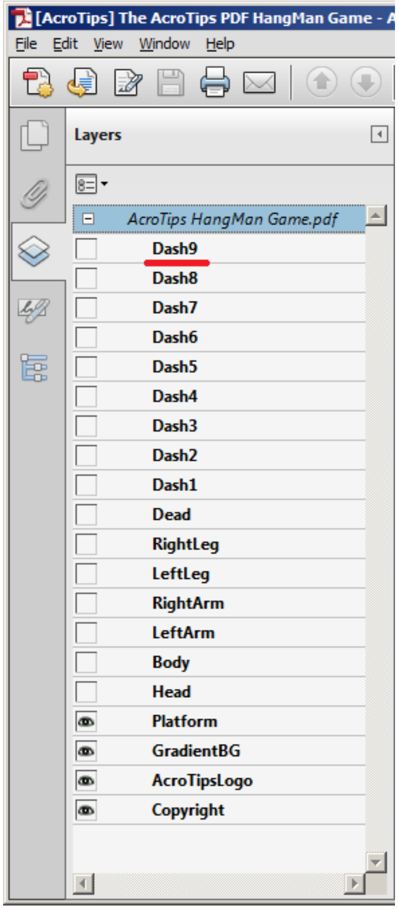
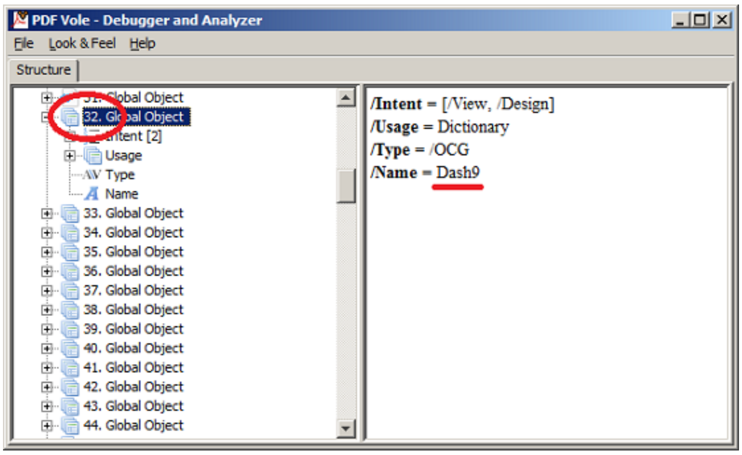
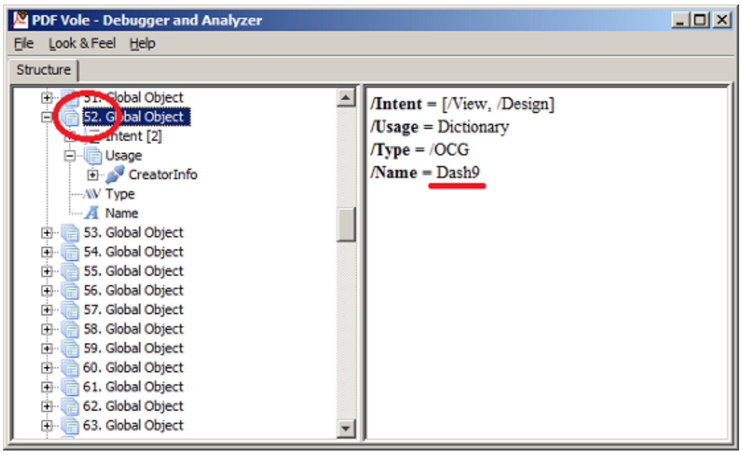
The tags `endingWith />` and `matchingRegex />` are intentionally not provided as explained at the beginning of this chapter.

String comparisons are case-insensitive. Whitespaces remain unchanged.

```
<!-- Layer names are treated case-insensitive. -->

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

Duplicate Layer Names

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

Layer Names in Adobe Reader®	Layer Objects in PDFVole
	 

/Intent = [/View, /Design]
/Usage = Dictionary
/Type = /OCG
/Name = Dash9

/Intent = [/View, /Design]
/Usage = Dictionary
/Type = /OCG
/Name = Dash9

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

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

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

In the current release 2015.10 PDFUnit does not provide 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. 115) explains how to use it.

- ... with a rendered page of a master document. The chapter 4.12: “Comparing Layout as Rendered Pages” (p. 82) describes tests with master documents.

Tests with rendered PDF pages can be carried out with the tag `<asRenderedPage />`:

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

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

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

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

You can specify individual pages to be compared. This is described in the chapter 13.2: “Page Selection” (p. 133).

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. 49) 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:

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

Image Format of Rendered Pages

Supported image types are `GIF`, `PNG`, `JPEG`, `BMP` and `WBMP`.

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 similiar to the syntax for comparing entire pages. The tag `<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.

```
<!-- Testing a section of a rendered page: -->

<testcase name="..">
  <assertThat testDocument="..">
    <asRenderedPage on="..">              (page selection)
      <isEqualTo positionUpperLeftX=".."  (optional)
                 positionUpperLeftY=".."  (optional)
                 unit=".."                (optional)
                 image=".."               (required)
      />
    </asRenderedPage>
  </assertThat>
</testcase>
```

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:

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

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:

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

Multiple Comparisons

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

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

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:

```
<!-- Tags to count parts of a PDF: -->

<hasNumberOfActions                 />
<hasNumberOfBookmarks               />
<hasNumberOfDifferentImages         />    ❶
<hasNumberOfEmbeddedFiles           />
<hasNumberOfFields                  />
<hasNumberOfFonts identifiedBy=".."      (required)
/>
<hasNumberOfJavaScriptActions       />
<hasNumberOfLayers                  />
<hasNumberOfOCGs                    />
<hasNumberOfPages                   />    ❷
<hasNumberOfSignatures              />
<hasNumberOfVisibleImages           />    ❸
```

❶❸ Tests for the number of images are described in chapter 3.13: “Images in PDF Documents” (p. 41).

❷ Tests for the number of pages are described in chapter 3.20: “Page Numbers as Objectives” (p. 52).

Examples

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

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

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

All tests can be combined:

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

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 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 tags:

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

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

Examples

You can check the number of pages like this:

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

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

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

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

The values for upper- and lower limits are exclusive.

Both limits can be checked with one test:

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

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. Passwords have to be declared with the tag `<assertThat />`. The general syntax looks like this:

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

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

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

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

Verifying Both Passwords

Opening a document with **one** password is already a test. But you can verify the **other** password using the tags `<hasOwnerPassword />` or `<hasUserPassword />`:

```
<!-- Verify the owner-password of the document: -->

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

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

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

❶❸ 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

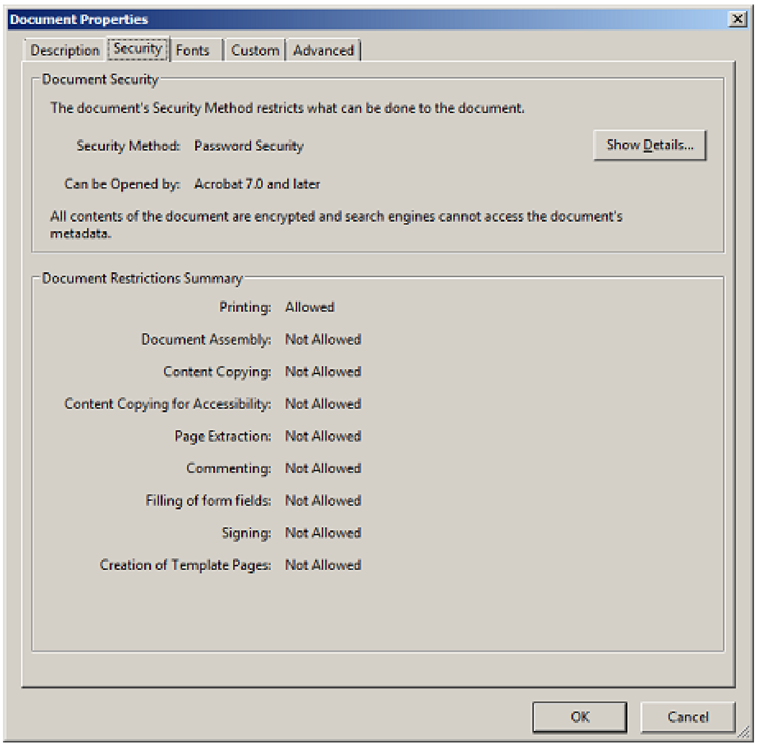
This example shows how to verify the encryption length:

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

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”:



Document Security

The document's Security Method restricts what can be done to the document.

Security Method: Password Security

Can be Opened by: Acrobat 7.0 and later

All contents of the document are encrypted and search engines cannot access the document's metadata.

Document Restrictions Summary

Printing:	Allowed
Document Assembly:	Not Allowed
Content Copying:	Not Allowed
Content Copying for Accessibility:	Not Allowed
Page Extraction:	Not Allowed
Commenting:	Not Allowed
Filling of form fields:	Not Allowed
Signing:	Not Allowed
Creation of Template Pages:	Not Allowed

You can test permissions using the tag `<hasPermission />`. That tag has a suitable attribute for every possible permission:

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

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

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

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

Example

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

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 tags for signatures and certificates:

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

<hasNumberOfSignatures />
<isSigned              />

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

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

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

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. 22).

Existence of Signatures

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

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

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:

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

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

Validity Date

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

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

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

❶ The attribute `pattern=".."` declares that dates are compared using year-month-day.

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

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

Revision, Reason, Sign-Name

The next examples show how various details of a signature can be tested using specific tags:

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

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

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

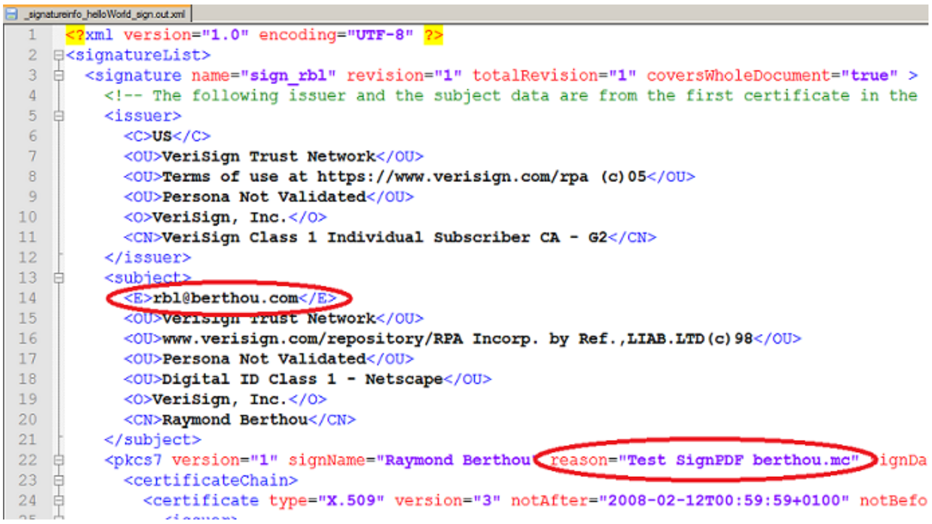
```
<testcase name="hasSignature_WithNumberOfRevisions">
  <assertThat testDocument="signed/sampleSignedPDFDocument.pdf">
    <hasSignature name="Signature2">
      <withNumberOfRevisions nr="1" />
    </hasSignature>
  </assertThat>
</testcase>
```

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

You can check other signature and certificate data using XPath.

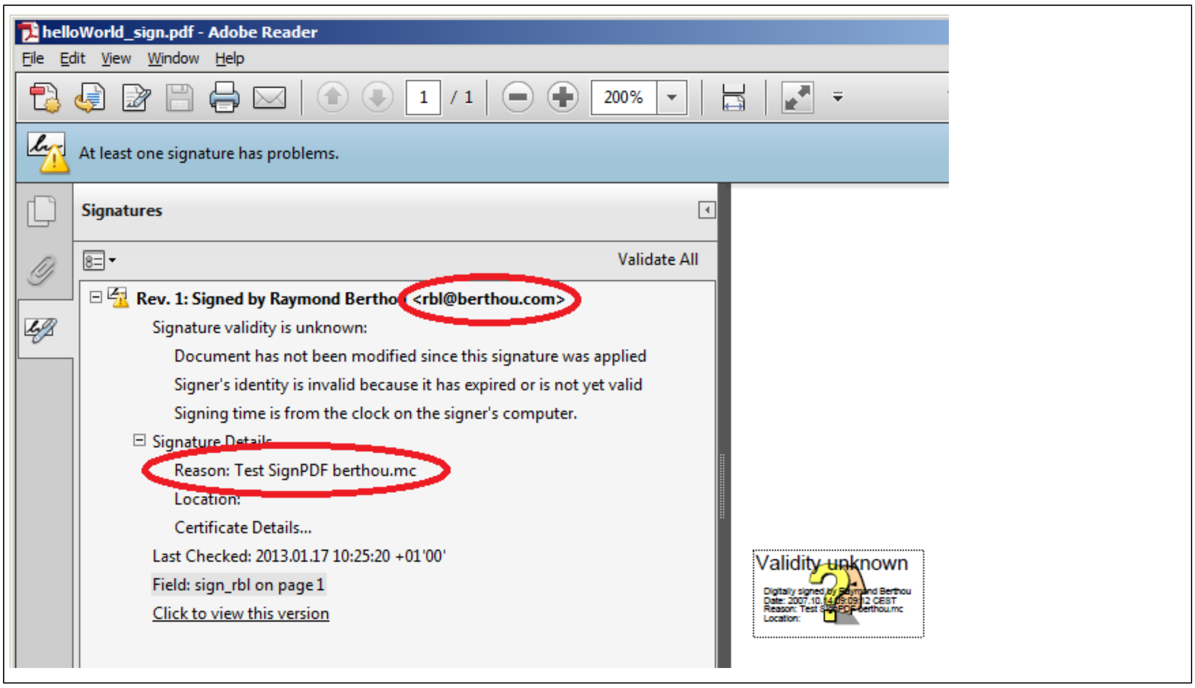
Compare Signatures using XML/XPath

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



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

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



helloWorld_sign.pdf - Adobe Reader
File Edit View Window Help
1 / 1
200%
Signatures
Validate All

At least one signature has problems.

Rev. 1: Signed by Raymond Bertho <rbl@berthou.com>

Signature validity is unknown:

Document has not been modified since this signature was applied

Signer's identity is invalid because it has expired or is not yet valid

Signing time is from the clock on the signer's computer.

Signature Details

Reason: Test SignPDF berthou.mc

Location:

Certificate Details...

Last Checked: 2013.01.17 10:25:20 +01'00'

Field: sign_rbl on page 1

Click to view this version

Validity unknown

Digitally signed by Raymond Berthou
Date: 2007.10.14 09:09:12 CEST
Reason: Test SignPDF berthou.mc
Location:

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

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

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:

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

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

Multiple Invocation

Of course, a single signature can be checked for multiple properties:

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

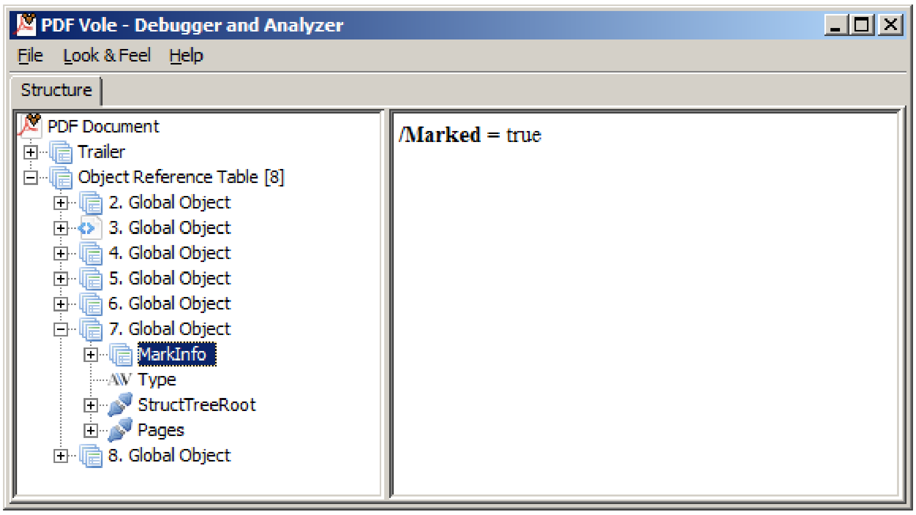
But think of a better name for this test. It would be better to split it into several tests with specific names.

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”.



/Marked = true

Following tags are available:

```
<!-- Tag to verify tagging information: -->

<isTagged />

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

Examples

The simplest test checks whether a document is tagged.

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

Further tests verify the existence of a particular tag.

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

And finally you can verify values of tags:

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

3.25. Text

Overview

The most common test case for PDF documents is probably to check the presence of expected text. That can be done with the tag `<hasText />` which can contain other tags and attributes.

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

<hasText />

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

<hasText >
  <inClippingArea />     (optional)

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

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

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

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

<!--
Whitespace processing, see: 13.4: “Whitespace Processing” (p. 135)
-->
```

Text on Individual Pages

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

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

You can declare specific pages using the attribute `on=".."` which provides several constants, e.g. `on="FIRST_PAGE"`, `on="EVERY_PAGE"`, `on="ODD_PAGES"` etc. Chapter 13.2: “Page Selection” (p. 133) describes more constants and how to use them.

The next example searches a text on the last page:

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

Also, you can test individual pages using the attribute `onPage=".."`.

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

Page numbers in the attribute `onPage=".."` must be separated by commas.

The chapter 13.2: “Page Selection” (p. 133) describes page selection in detail.

Text on All Pages

There are two constants for the attribute `on=".."` to search text on multiple pages: `on="EACH_PAGE"`, `on="EVERY_PAGE"` and `on="ANY_PAGE"`.

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

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

The constants `on="EVERY_PAGE"` and `on="EACH_PAGE"` require that the text really exists **on every 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:

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

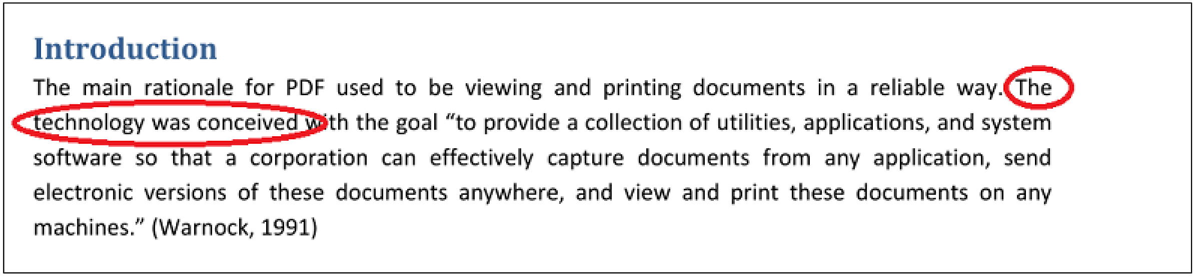
The error message is:

```
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 as well as in the text being tested. In the following example the text to be searched for 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:

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

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

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

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

Text in Parts of a Page

Text can be searched not only on whole pages, but also on a section of a page. The chapter 13.6: “Defining Page Areas” (p. 139) describes that topic.

Empty Pages

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

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

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

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

Multiple Search Tokens

It is annoying to write a separate test for every expected text on a page. So you can invoke some tags more than once:

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

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

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**.

You can only use the tags `<startingWith />` and `<endingWith />`.

Multiple text comparisons are all related to the specificied page numbers declared in the outer tag `<hasText />`:

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

The tag `<hasText />` must be used multiple times if multiple validations are used pointing to different pages:

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

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:

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

Page numbers start from “1”.

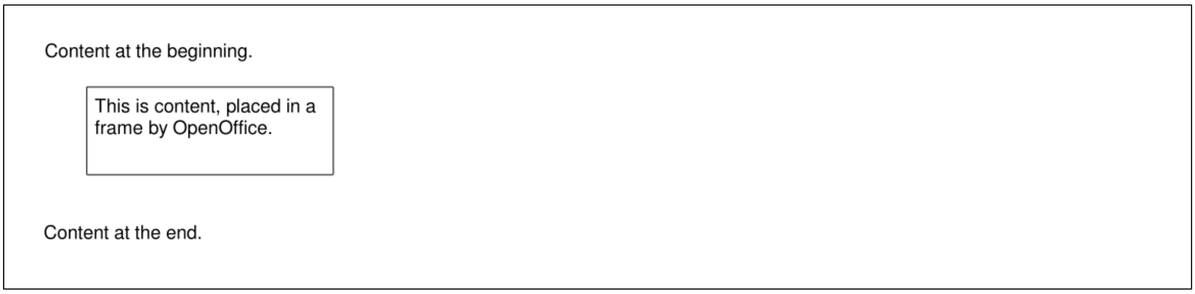
Invalid page limits are not necessarily an error. In the following example, the text is searched for on all pages between 1 and 99 (exclusive). Although the document has only 4 pages, the test ends successfully because the expected string is found on page 1:

```
<!--
  Attention: the document has the search token on page 1.
  And '1' is before '99'. So this test ends successfully.
-->
<testcase name="hasText_OnAllPagesBefore_WrongPageNumber">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onAnyPageBefore="99">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

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:



Content at the beginning.

This is content, placed in a frame by OpenOffice.

Content at the end.

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

3.26. 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:

```
<!-- Compare text inside a clipping area: -->

<testcase name="..">
  <assertThat testDocument="..">
    <hasText on=".." >
      <inClippingArea upperLeftX=".." upperLeftY=".." width=".." height=".." >
        ... <!-- compare text here -->
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

Example

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

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

❶ This defines a clipping area. More detailed information is available in chapter 13.6: “Defining Page Areas” (p. 139). The possibility to use measuring units such as `MILLIMETER` is described in chapter 13.7: “Format Units” (p. 140).

When comparing text in page areas, all the test are available which are available when comparing text on entire pages. These tags are described in chapter 13.3: “Comparing Text” (p. 135).

A clipping area can also be used when testing images.

3.27. Text - Rotated and Overhead

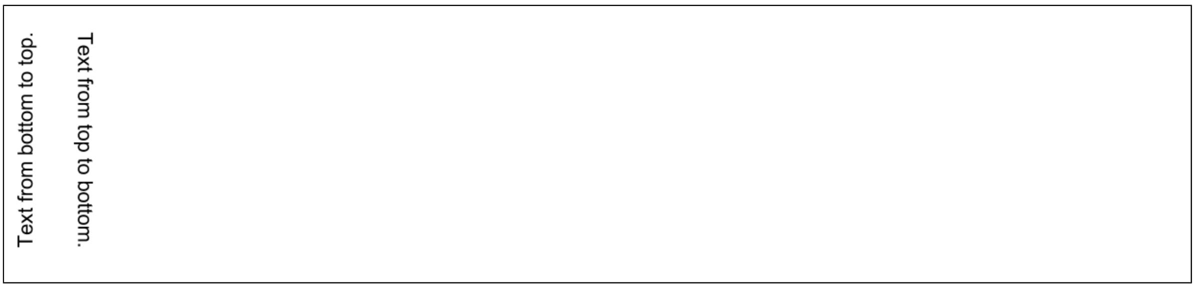
Overview

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, you can use the same tags 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:

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

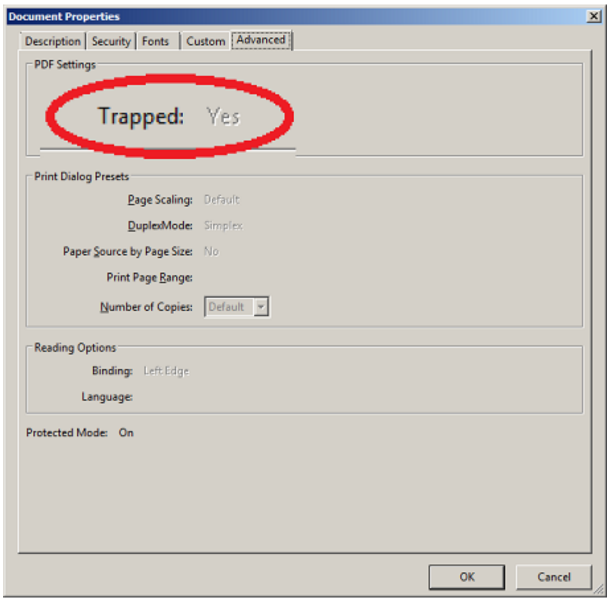
Note that the vertical text is still text with the writing direction LTR (left-to-right). Text with the direction RTL (right-to-left) will be supported by PDFUnit in a future release.

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:



Document Properties

PDF Settings

Trapped: Yes

Print Dialog Presets

Page Scaling: Default

DuplexMode: Simplex

Paper Source by Page Size: No

Print Page Range:

Number of Copies: Default

Reading Options

Binding: Left Edge

Language:

Protected Mode: On

“Trapped” can have the values `Yes`, `No` and `Unknown`. Only one tag is provided to verify this property:

```
<!-- Tag to verify trapping information: -->

<hasTrappingInfo value=".."  (required)
/>

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

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

Examples

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

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

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

3.29. 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 using the following tag:

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

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

One Distinct Version

The format of a PDF version number is defined by XML Schema as “number point number”. The next example checks for the version “1.4”:

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

Ranges of Versions

With the attributes `greaterThan` and `lessThan` you can verifiy that the version of a PDF document lies in a given range:

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

❶ Upper and lower limit values are exclusive.

Also upcoming versions can be tested:

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

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.

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

<hasXFAData   />
<hasNoXFAData />

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

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

Existence and Absence of XFA

The first test focuses on the existence of XFA data:

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

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

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

Comparing XFA with an XML File

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

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

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

The value of a certain XML node can be tested with the Tag `<withNode />`:

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

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:

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

XPath based XFA Tests

XPath can do more than just identify individual nodes. With the tag `<matchingXPath />` you can use all the power of XPath.

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

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

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

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 your JDK. So the XPath compatibility also depends on the version of your JDK.

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:

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

For the same reason, the default namespace must be given when using the tag `<withNode />`:

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

❶ It is not important which alias you choose for the default namespace.

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 tags for XMP data as for XFA data:

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

<hasXMPData   />
<hasNoXMPData />

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

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

Existence and Absence of XMP

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

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

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

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:

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

❶ Whitespaces are ignored when comparing XML 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>
```

If you want to check that a node exists in the structure of the XMP data you can use the tag `<withNode />`. The next example checks the existence of two nodes:

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

When you want to verify the value of a node, you also have to declare the expected value using the attribute `value=".."`:

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

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 tag `<matchingXPath />` you can use the full power of XPath:

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

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

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.

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 the tag `<matchingXPath />`:

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

Default namespace for the tag `<withNode />` with an expected value:

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

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.

Many properties of a PDF document that can be tested individually can also be tested as a comparison with a master PDF.

Initialization

A master document is declared using the attribute `masterDocument`:

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

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 be given with the attribute `masterPassword`.

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: “Further Comparisons” (p. 88) collects all the functions which are not described in other chapters.

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

<areBothForFastWebView />         4.20: “Further Comparisons” (p. 88)
<haveSameActions />               4.3: “Comparing Actions” (p. 76)
<haveSameAppearance />            4.12: “Comparing Layout as Rendered Pages” (p. 82)
<haveSameAuthor />                4.7: “Comparing Document Properties” (p. 79)
<haveSameBookmarks />             4.5: “Comparing Bookmarks” (p. 78)
<haveSameCreationDate />          4.6: “Comparing Date Values” (p. 79)
<haveSameCreator />               4.7: “Comparing Document Properties” (p. 79)
<haveSameEmbeddedFiles />         4.4: “Comparing Attachments” (p. 77)
<haveSameFields />                4.2: “Comparing Form Fields” (p. 75)
<haveSameFonts />                 4.8: “Comparing Fonts” (p. 80)
<haveSameFormat />                4.9: “Comparing Format” (p. 80)
<haveSameImages />                4.10: “Comparing Images” (p. 81)
<haveSameJavaScript />            4.20: “Further Comparisons” (p. 88)
<haveSameKeywords />              4.20: “Further Comparisons” (p. 88)
<haveSameLanguage />              4.20: “Further Comparisons” (p. 88)
<haveSameLayerNames />            4.20: “Further Comparisons” (p. 88)
<haveSameModificationDate />      4.6: “Comparing Date Values” (p. 79)
<haveSameText />               4.17: “Comparing Text” (p. 85)

...  continued
... continuation:

<haveSameNumberOfActions />       4.3: “Comparing Actions” (p. 76)
<haveSameNumberOfBookmarks />     4.5: “Comparing Bookmarks” (p. 78)
<haveSameNumberOfEmbeddedFiles /> 4.4: “Comparing Attachments” (p. 77)
<haveSameNumberOfFields />        4.2: “Comparing Form Fields” (p. 75)
<haveSameNumberOfFonts />         4.8: “Comparing Fonts” (p. 80)
<haveSameNumberOfImages />        4.10: “Comparing Images” (p. 81)
<haveSameNumberOfLayers />        4.15: “Comparing Quantities of PDF Elements” (p. 84)
<haveSameNumberOfPages />         4.15: “Comparing Quantities of PDF Elements” (p. 84)
<haveSameNumberOfTaggingInfo />   4.15: “Comparing Quantities of PDF Elements” (p. 84)
<haveSamePermission />            4.14: “Comparing Permissions” (p. 84)
<haveSamePermissions />           4.14: “Comparing Permissions” (p. 84)
<haveSameProducer />              4.7: “Comparing Document Properties” (p. 79)
<haveSameProperties />            4.7: “Comparing Document Properties” (p. 79)
<haveSameProperty />              4.7: “Comparing Document Properties” (p. 79)
<haveSameSignatureNames />        4.16: “Comparing Signature Names” (p. 85)
<haveSameSubject />               4.7: “Comparing Document Properties” (p. 79)
<haveSameTaggingInfo />           4.20: “Further Comparisons” (p. 88)
<haveSameTitle />                 4.7: “Comparing Document Properties” (p. 79)
<haveSameTrappingInfo />          4.20: “Further Comparisons” (p. 88)
<haveSameXFAData />               4.18: “Comparing XFA Data” (p. 86)
<haveSameXMPData />               4.19: “Comparing XMP Data” (p. 88)

...  (end of list)
```

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:

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

Field Names

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

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

Field Properties

If you want to compare the fields of two documents including all properties you have to use the tag `<haveSameFields />` with the attribute `by="PROPERTIES"`:

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

All field properties can be extracted into an XML file using the utility program `ExtractFieldsInfo`, see chapter 9.3: “Extract Field Information to XML” (p. 103). This file can be analyzed later.

Field Content

And finally you can compare the content of form fields of two documents using the tag `<haveSameFields />` and the attribute `by="VALUE"`. The test fails when a field have different contents in the two files:

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

❶ Whitespaces are “normalized”, see chapter 13.4: “Whitespace Processing” (p. 135).

Concatenated Tests

You can compare fields and other parts of two PDF documents in one test, but such a test is not recommended because it's hard to find a good name:

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

4.3. Comparing Actions

Quantity

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

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

Properties

To compare all actions of two PDF documents, you can use the tag `<haveSameActions />`:

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

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”.

Type	Relevant Property for equals()	
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 13.4: "Whitespace Processing" (p. 135).
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.

4.4. Comparing Attachments

Quantity

Compare the number of attachments in two documents:

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

Name and Content

To compare the attachments by name or by content you can use the tag `<haveSameEmbeddedFiles />`:

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

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

The attribute `comparedBy=".."` provides the two constants `NAME` and `CONTENT`.

You can use the utility `ExtractEmbeddedFiles` to extract the attachments. See chapter 9.4: “Extract Attachments” (p. 104).

4.5. Comparing Bookmarks

Quantity

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

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

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

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

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. 106).

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 tags:

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

<haveSameCreationDate     />
<haveSameModificationDate />
```

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

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

Comparing two dates is always carried out with the time resolution `DATE (yyyy-MM-dd)`.

4.7. Comparing Document Properties

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

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

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

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

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

Custom properties can be compared using the tag `<haveSameProperty />`:

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

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

If you want to compare **all** properties of two documents, you can use the tag `<haveSameProperties />`:

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

4.8. Comparing Fonts

Quantity

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

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

Font Properties

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

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

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. 107).

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.

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

The comparison can be restricted to selected pages:

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

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

All possibilities to select pages are explained in chapter 13.2: “Page Selection” (p. 133).

4.10. Comparing Images

Quantity

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

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

The comparison can be limited to selected pages:

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

All possibilities to select pages are described in chapter 13.2: “Page Selection” (p. 133).

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

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

This test 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:

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

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

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. 108).

4.11. Comparing JavaScript

Two PDF documents can have the “same” JavaScript. The comparison is done byte-wise using the tag `<haveSameXMPData />`. Whitespaces are ignored.

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

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. 109).

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.

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

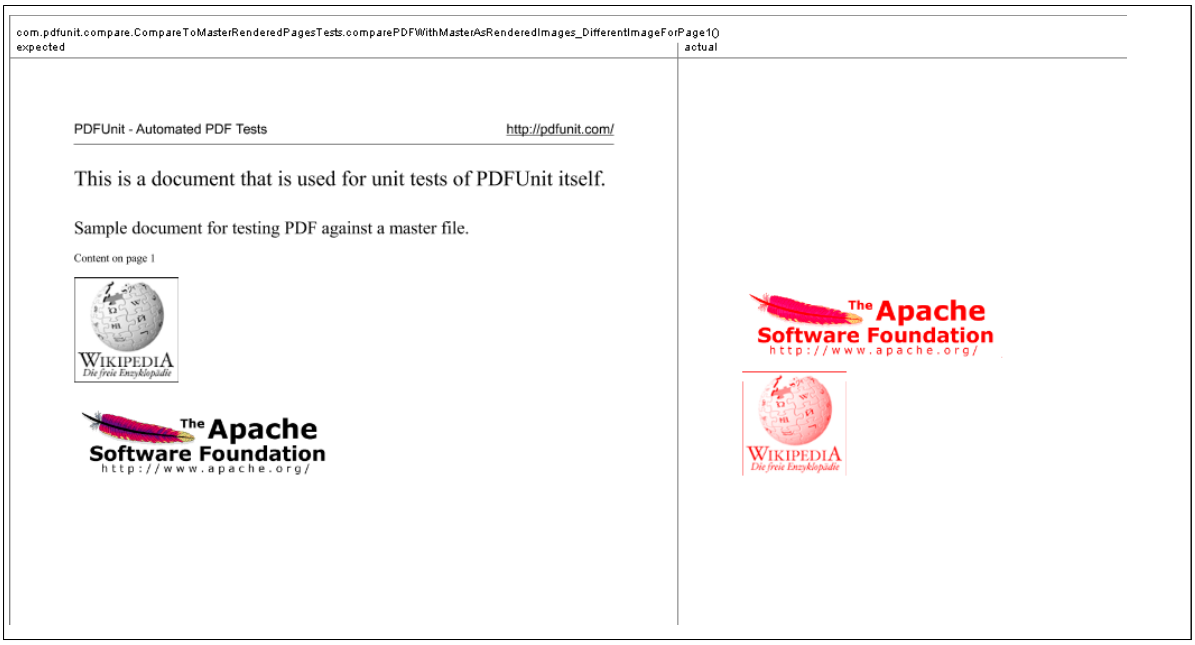
You can select individual pages in many ways. All possibilities are described in chapter 13.2: “Page Selection” (p. 133).

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

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

A clipping area can be defined using the measuring units `POINTS`, `MILLIMETER`, `CENTIMETER`, `INCH` and `DPI72`. All constants are declared with XML Schema for the attribute `unit=".."`. If you omit a measuring unit, the values are taken as `MILLIMETER`.

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



com.pdfunit.compare.CompareToMasterRenderedPagesTests.comparePDFWithMasterAsRenderedImages_DifferentImageForPage1()
expected actual

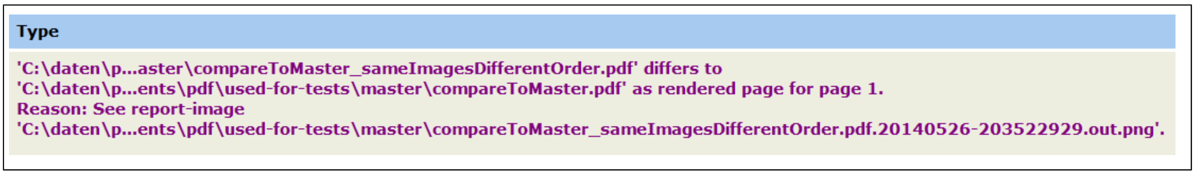
PDFUnit - Automated PDF Tests http://pdfunit.com/

This is a document that is used for unit tests of PDFUnit itself.

Sample document for testing PDF against a master file.

Content on page 1

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



```
Type
'C:\daten\p...aster\compareToMaster_sameImagesDifferentOrder.pdf' differs to
'C:\daten\p...ents\pdf\used-for-tests\master\compareToMaster.pdf' as rendered page for page 1.
Reason: See report-image
'C:\daten\p...ents\pdf\used-for-tests\master\compareToMaster_sameImagesDifferentOrder.pdf.20140526-203522929.out.png'.
```

The filename of a diff-image file is built as follows:

- The first part is the full name of the test file.
- If the tested file is a stream, the name starts with “_pdfunit_stream_”. If it is a byte array, it starts with “_pdfunit_bytearray_”. Both strings are extended by a random number.
- The second part is a formatted date in the format “yyyyMMdd-HHmmssSSS”.
- The last part of the file name is the string “.out.png”.

Due to the default configuration a diff image for test files is stored in the same directory in which 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 settings in the config.properties file.

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:

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

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:

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

4.14. Comparing Permissions

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

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

If you want to compare a **single permission** you can use the attribute `permission=".."` with predefined constants:

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

The following constants are available:

```
<!-- Available permissions: -->

permission="ALLOW_ASSEMBLE_DOCUMENTS"
permission="ALLOW_COPY"
permission="ALLOW_DEGRADED_PRINTING"
permission="ALLOW_EXTRACT_CONTENT"      ❶
permission="ALLOW_FILL_IN"
permission="ALLOW_MODIFY_ANNOTATIONS"
permission="ALLOW_MODIFY_CONTENT"
permission="ALLOW_PRINTING"
permission="ALLOW_SCREENREADERS"        ❷
```

❶❷ 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 tags comparing countable components:

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

<haveSameNumberOfActions       />
<haveSameNumberOfBookmarks     />
<haveSameNumberOfEmbeddedFiles />
<haveSameNumberOfFields        />
<haveSameNumberOfFonts         />
<haveSameNumberOfImages        />
<haveSameNumberOfLayers        />
<haveSameNumberOfPages         />
<haveSameNumberOfTaggingInfo   />
```

Here is a small sample which is not shown in other chapters:

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

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

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

4.16. Comparing Signature Names

When comparing signatures of two PDF documents using the PDFUnit release 2015.10, only the names of the signatures are compared:

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

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. 111) 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):

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

You can restrict the test to selected pages which is explained in chapter 13.2: “Page Selection” (p. 133):

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

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

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

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

4.18. Comparing XFA Data

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

Overview

There is only one powerful tag to test XFA data:

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

<haveSameXFAData />
```

Example - Resulttype Node

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

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

As you can see in the example the type of the expected result has to be specified. Possible result types for the attribute `withResultType`:

```
<!-- Result types for XPath-processing:  -->

withResultType="BOOLEAN"
withResultType="NUMBER"
withResultType="NODE"
withResultType="NODESET"
withResultType="STRING"
```

Example - Resulttype Boolean

The XPath-expressions may contain XPath functions:

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

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. 100).

Example - Default Namespace

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

You can use the tag `<matchingXPath />` more than once in a test. But it would be better to split the following test:

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

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

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. 68) 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. 113).

Overview

PDFUnit provides the following tag:

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

<haveSameXMPData />
```

Example

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

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. Further Comparisons

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

```
<!-- Various tags to compare 2 PDF documents, not described before: -->

<areBothForFastWebView />
<haveSameKeywords      />
<haveSameLanguage      />
<haveSameLayerNames    />
<haveSameTrappingInfo  />
<haveSameTaggingInfo   />
```

Here are two examples:

Fast WebView, Tagging

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

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

Combination of tests

All tests can be combined into one test:

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

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.

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

The test documents can als be of type URL:

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

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

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

<containsImage />               3.13: “Images in PDF Documents” (p. 41)

<hasAuthor />                   3.7: “Document Properties” (p. 24)
<hasBookmark />                 3.4: “Bookmarks and Named Destinations” (p. 19)
<hasBookmarks />                3.4: “Bookmarks and Named Destinations” (p. 19)
<hasCreationDate />             3.6: “Dates” (p. 23)
<hasCreationDateAfter />        3.6: “Dates” (p. 23)
<hasCreationDateBefore />       3.6: “Dates” (p. 23)
<hasCreator />                  3.6: “Dates” (p. 23)

<hasEmbeddedFile />             3.3: “Attachments” (p. 17)
<hasEmbeddedFileContent />      3.3: “Attachments” (p. 17)
<hasEncryptionLength />         3.21: “Passwords” (p. 53)
<hasField />                    3.10: “Form Fields” (p. 31)
<hasFields />                   3.10: “Form Fields” (p. 31)
<hasFont />                     3.9: “Fonts” (p. 28)
<hasFonts />                    3.9: “Fonts” (p. 28)
<hasFormat />                   3.12: “Format” (p. 39)
<hasJavaScript />               3.14: “JavaScript” (p. 43)
<hasKeywords />                 3.7: “Document Properties” (p. 24)
<hasLayer />                    3.16: “Layers” (p. 46)
<hasLayers />                   3.16: “Layers” (p. 46)
<hasLessPages />                3.20: “Page Numbers as Objectives” (p. 52)
<hasLocale />                   3.15: “Language” (p. 45)
<hasModificationDate />         3.6: “Dates” (p. 23)
<hasModificationDateAfter />    3.6: “Dates” (p. 23)
<hasModificationDateBefore />   3.6: “Dates” (p. 23)
<hasMorePages />                3.20: “Page Numbers as Objectives” (p. 52)

...  continued
... continuation:

<hasNoAuthor />                 3.7: “Document Properties” (p. 24)
<hasNoCreationDate />           3.6: “Dates” (p. 23)
<hasNoCreator />                3.7: “Document Properties” (p. 24)
<hasNoKeywords />               3.7: “Document Properties” (p. 24)
<hasNoLocale />                 3.15: “Language” (p. 45)
<hasNoModificationDate />       3.6: “Dates” (p. 23)
<hasNoProducer />               3.7: “Document Properties” (p. 24)
<hasNoProperty />               3.7: “Document Properties” (p. 24)
<hasNoSubject />                3.7: “Document Properties” (p. 24)
<hasNoText />                   3.25: “Text” (p. 60)
<hasNoTitle />                  3.7: “Document Properties” (p. 24)
<hasNoXFAData />                3.30: “XFA Data” (p. 68)
<hasNoXMPData />                3.31: “XMP Data” (p. 70)

<hasOwnerPassword />            3.21: “Passwords” (p. 53)
<hasPermission />               3.22: “Permissions” (p. 54)
<hasProducer />                 3.7: “Document Properties” (p. 24)
<hasProperty />                 3.7: “Document Properties” (p. 24)
<hasSignature />                3.23: “Signatures and Certificates” (p. 55)
<hasSignatures />               3.23: “Signatures and Certificates” (p. 55)
<hasSignedSignatureFields />    3.23: “Signatures and Certificates” (p. 55)
<hasSubject />                  3.7: “Document Properties” (p. 24)
<hasText />                     3.25: “Text” (p. 60)
<hasTitle />                    3.7: “Document Properties” (p. 24)
<hasTrappingInfo />             3.28: “Trapping Info” (p. 66)
<hasUnsignedSignatureFields />  3.23: “Signatures and Certificates” (p. 55)
<hasVersion />                  3.29: “Version Info” (p. 67)
<hasXFAData />                  3.30: “XFA Data” (p. 68)
<hasXMPData />                  3.31: “XMP Data” (p. 70)

<isCertified />                 3.5: “Certified PDF” (p. 22)
<isLinearizedForFastWebView />  3.8: “Fast Web View” (p. 27)
<isSigned />                    3.23: “Signatures and Certificates” (p. 55)
<isTagged />                    3.24: “Tagged Documents” (p. 59)

...  (end of list)
```

If you are looking for more tests please send your request to info[at]pdfunit.com.

Chapter 6. PDFUnit-Monitor

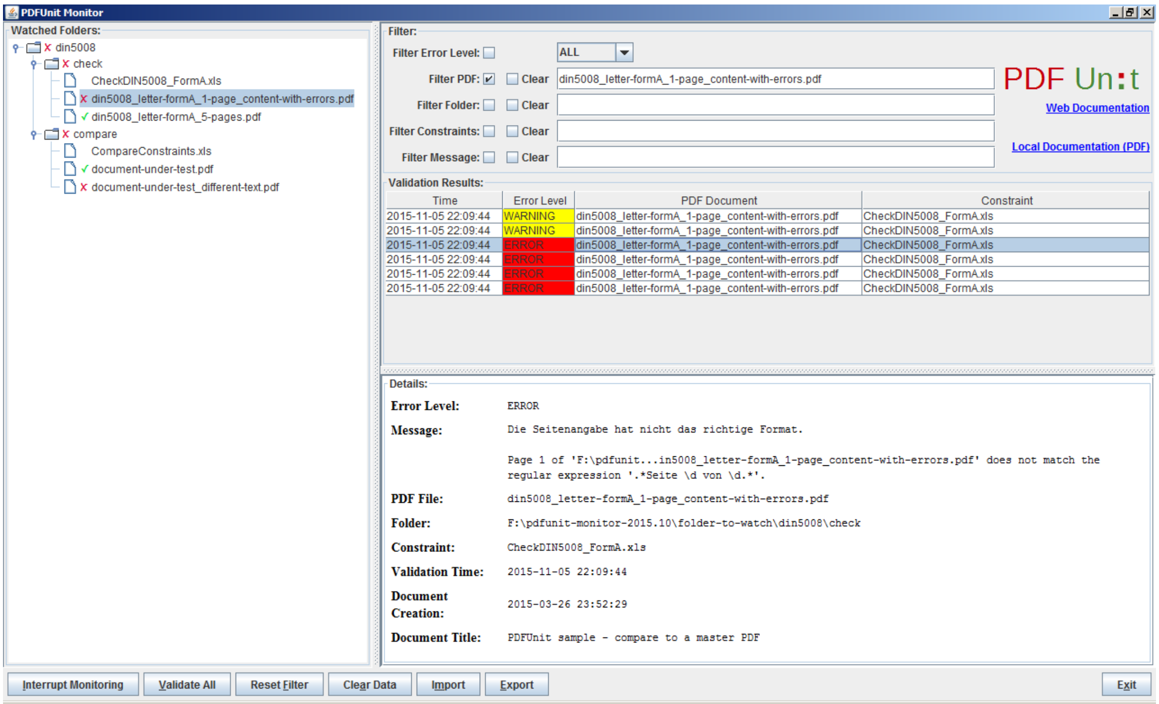
The PDFUnit-Monitor is an application that shows the result of PDFUnit tests. Tests are written in Excel files, so non-developers can create and run the tests.

The functional scope of the PDFUnit-Monitor is large. Therefor 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.



Filter:

Filter Error Level: □ ALL

Filter PDF: ☑ □ Clear din5008_letter-formA_1-page_content-with-errors.pdf

Filter Folder: □ □ Clear

Filter Constraints: □ □ Clear

Filter Message: □ □ Clear

Validation Results:

Time	Error Level	PDF Document	Constraint
2015-11-05 22:09:44	WARNING	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA.xls
2015-11-05 22:09:44	WARNING	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA.xls
2015-11-05 22:09:44	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA.xls
2015-11-05 22:09:44	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA.xls
2015-11-05 22:09:44	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA.xls
2015-11-05 22:09:44	ERROR	din5008_letter-formA_1-page_content-with-errors.pdf	CheckDIN5008_FormA.xls

Details:

Error Level: ERROR

Message: Die Seitenangabe hat nicht das richtige Format.

Page 1 of 'F:\pdfunit...in5008_letter-formA_1-page_content-with-errors.pdf' does not match the regular expression '.*Seite \d von \d.*'.

PDF File: din5008_letter-formA_1-page_content-with-errors.pdf

Folder: F:\pdfunit-monitor-2015.10\folder-to-watch\din5008\check

Constraint: CheckDIN5008_FormA.xls

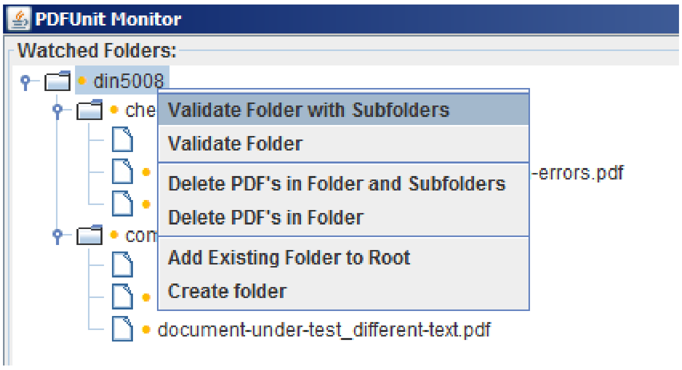
Validation Time: 2015-11-05 22:09:44

Document Creation: 2015-03-26 23:52:29

Document Title: PDFUnit sample - compare to a master PDF

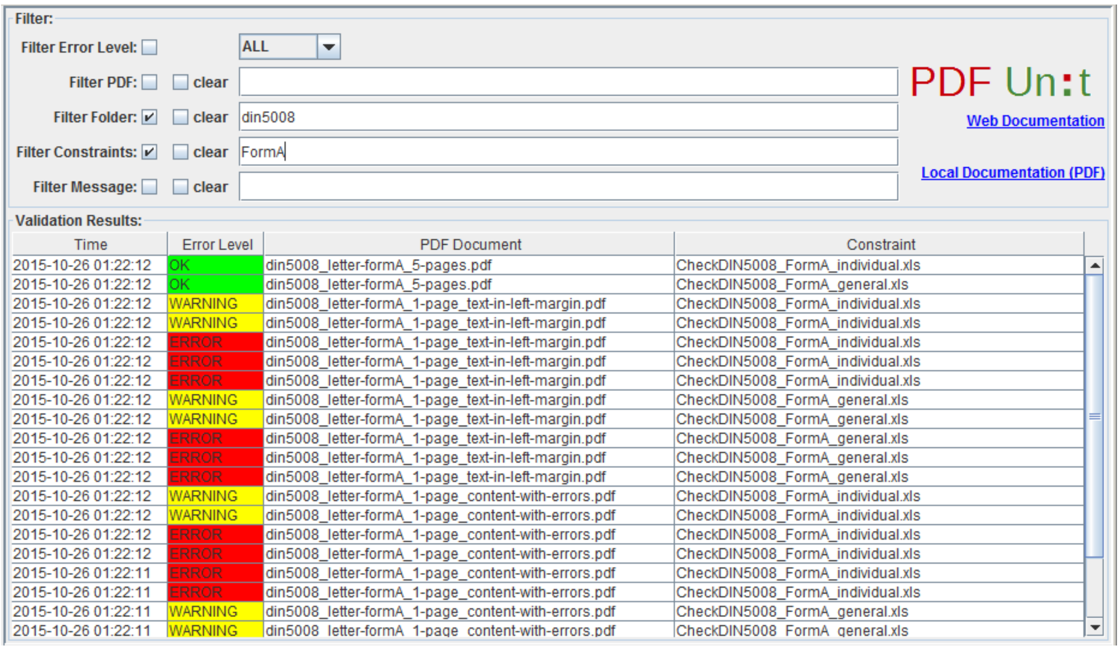
A double click on a PDF or Excel document in the folder structure or in the error list opens the standard application of the operating system for the document.

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


PDFUnit Monitor
Watched Folders:
din5008
che
con
-errors.pdf
document-under-test_different-text.pdf
Validate Folder with Subfolders
Validate Folder
Delete PDF's in Folder and Subfolders
Delete PDF's in Folder
Add Existing Folder to Root
Create folder

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:

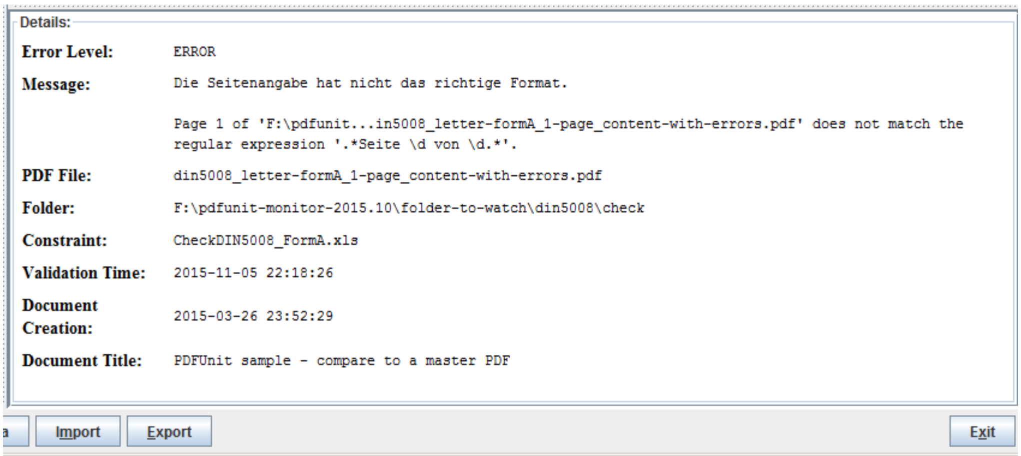
Time	Error Level	PDF Document	Constraint
2015-10-26 01:22:12	OK	din5008_letter-formA_5-pages.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	OK	din5008_letter-formA_5-pages.pdf	CheckDIN5008_FormA_general.xls
2015-10-26 01:22:12	WARNING	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_individual.xls
2015-10-26 01:22:12	WARNING	din5008_letter-formA_1-page_text-in-left-margin.pdf	CheckDIN5008_FormA_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	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.



Details:

Error Level: ERROR

Message: Die Seitenangabe hat nicht das richtige Format.

Page 1 of 'F:\pdfunit...in5008_letter-formA_1-page_content-with-errors.pdf' does not match the regular expression '.*Seite \d von \d.*'.

PDF File: din5008_letter-formA_1-page_content-with-errors.pdf

Folder: F:\pdfunit-monitor-2015.10\folder-to-watch\din5008\check

Constraint: CheckDIN5008_FormA.xls

Validation Time: 2015-11-05 22:18:26

Document Creation: 2015-03-26 23:52:29

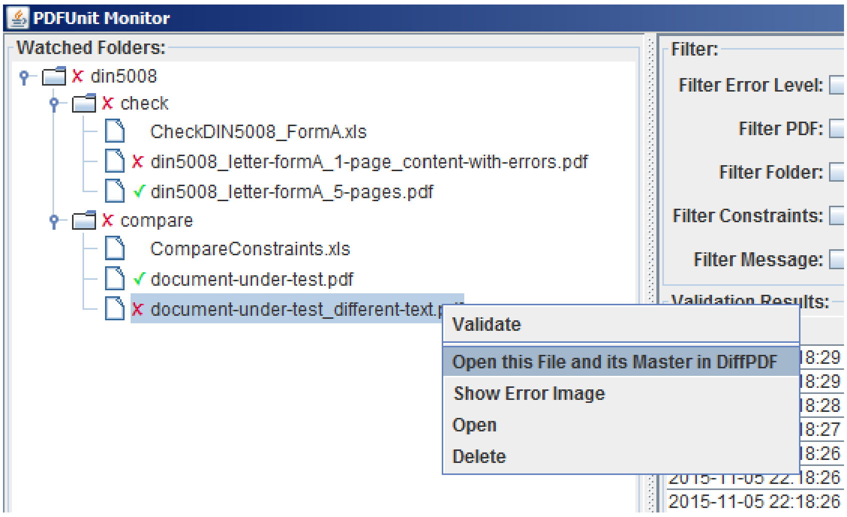
Document Title: PDFUnit sample - compare to a master PDF

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.

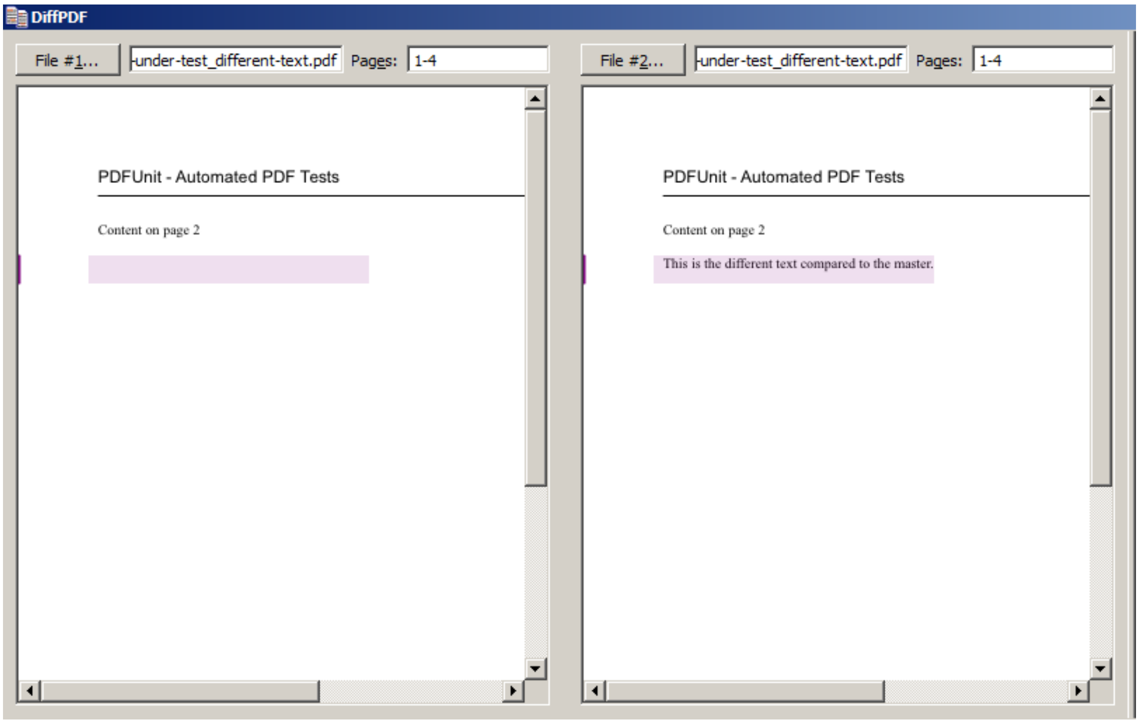
Comparing a PDF with a Master-PDF

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.



PDFUnit - Automated PDF Tests

Content on page 2

PDFUnit - Automated PDF Tests

Content on page 2

This is the different text compared to the master.

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`:

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

Longer Unicode Text

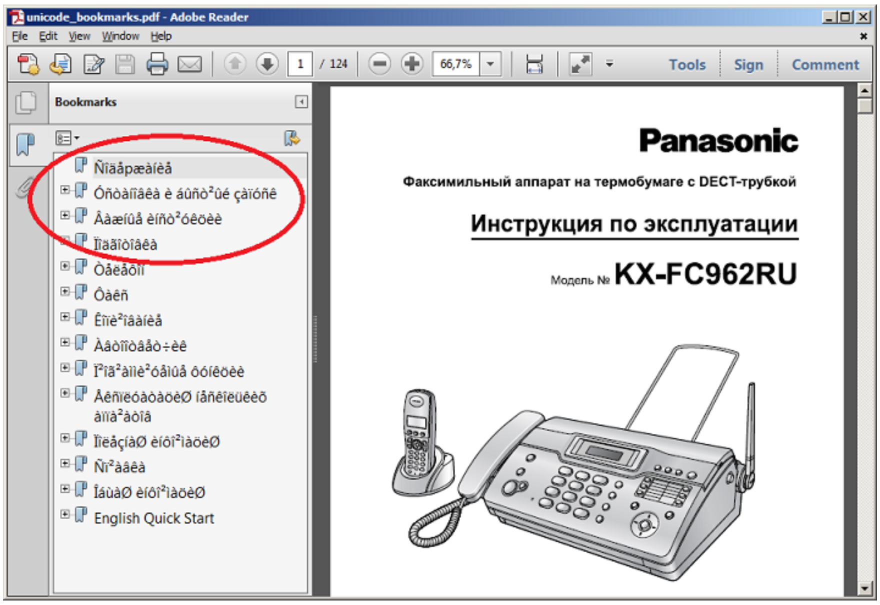
It would be too difficult to figure out the hex code for all characters of a longer text. Therefore PDFUnit provides the small utility `ConvertUnicodeToHex`. Pass the foreign text as a string to the tool, run the program and place the generated hex code into your test. Detailed information can be found in chapter 9.2: “Convert Unicode Text into Hex Code” (p. 102). A test with a longer sequence may look like this:

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

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:


unicode_bookmarks.pdf - Adobe Reader
File Edit View Window Help
1 / 124
66,7%
Tools Sign Comment
Bookmarks

Ñîäåpæàíèå
Óñòàíîâêà è áûñò²ûé çàïóñê
Âàæíûå èíñò²óêöèè
Ïîäãîòîâêà
Òåëåôîí
Ôàêñ
Êîïè²îâàíèå
Àâòîîòâåò÷èê
Ï²îã²àììè²óåìûå ôóíêöèè
ÀêñïëóàòàöèØ íåñêîëüêèõ àïïà²àòîâ
ÏîëåçíàØ èíôî²ìàöèØ
Ñï²àâêà
ÎáùàØ èíôî²ìàöèØ
English Quick Start

Panasonic

Факсимильный аппарат на термобумаге с DECT-трубкой

Инструкция по эксплуатации

Модель № KX-FC962RU

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

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

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. 100) describes how to use XPath in PDFUnit tests. You can also use Unicode sequences in XPath expressions:

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

    </hasBookmarks>
  </assertThat>
</testcase>
```

File Encoding UTF-8 for Shell Scripts

Just like any Java program that processes files, PDFUnit depends on the environment variable `file.encoding` which can be set in the following ways:

```
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:

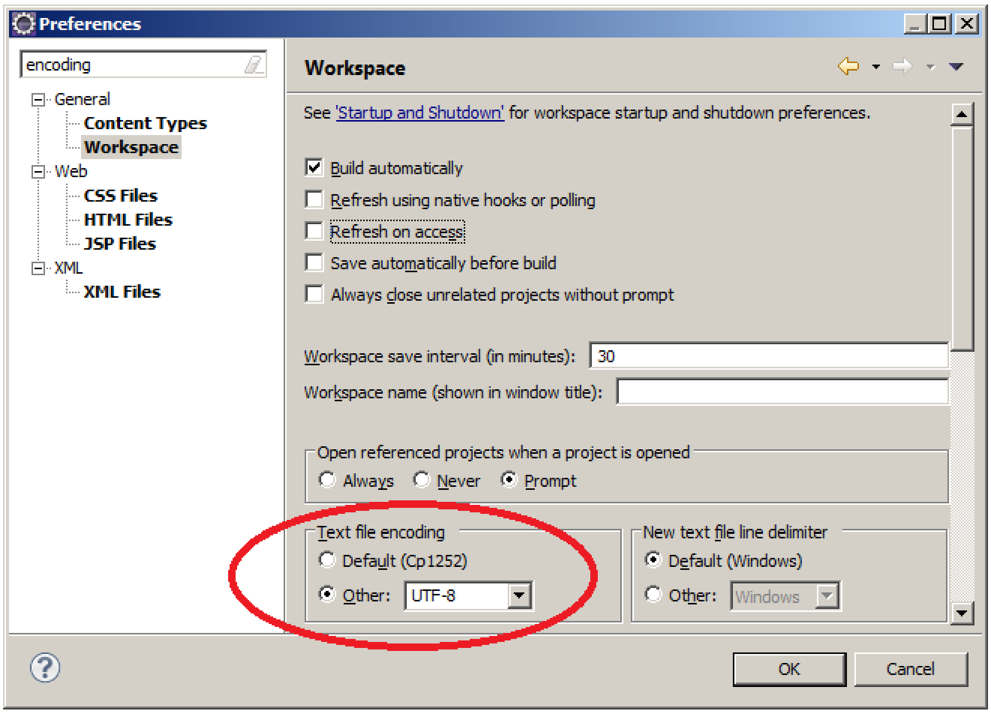
```
// does not work for ANT:
ant -Dfile.encoding=UTF-8
```

Instead, the property had to be set using `JAVA_TOOLS_OPTIONS`:

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

Configure Eclipse to UTF-8

When working with XML files in Eclipse, you do not need to configure Eclipse for UTF-8, because that 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:



Workspace

See 'Startup and Shutdown' for workspace startup and shutdown preferences.

☑ Build automatically
□ Refresh using native hooks or polling
□ Refresh on access
□ Save automatically before build
□ Always close unrelated projects without prompt

Workspace save interval (in minutes): 30
Workspace name (shown in window title):

Open referenced projects when a project is opened
○ Always ○ Never ● Prompt

Text file encoding
○ Default (Cp1252)
● Other: UTF-8

New text file line delimiter
● Default (Windows)
○ Other: Windows

This default can be changed for each file.

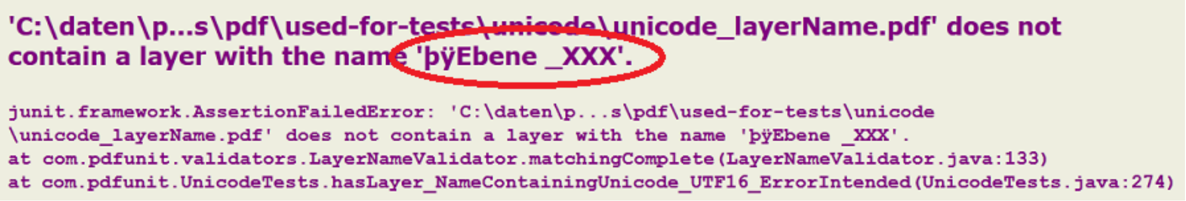
Unicode in Error Messages

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.

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

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

When the tests were 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)
```

Unicode for invisible Characters - &nbsp;

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:

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

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

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 related test facilities: -->

<hasBookmarks><matchingXPath />...     3.4: "Bookmarks and Named Destinations" (p. 19)
<hasFields><matchingXPath />...        3.10: "Form Fields" (p. 31)
<hasFonts><matchingXPath />...         3.9: "Fonts" (p. 28)
<hasSignatures><matchingXPath />...    3.23: "Signatures and Certificates" (p. 55)
<hasXFAData><matchingXPath />...       3.30: "XFA Data" (p. 68)
<hasXMPData><matchingXPath />...       3.31: "XMP Data" (p. 70)

<!-- Comparing two documents using XPath: -->
<haveSameXFAData><matchingXPath />... 4.18: "Comparing XFA Data" (p. 86)
<haveSameXMPData><matchingXPath />... 4.19: "Comparing XMP Data" (p. 88)
```

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.ExtractXFAData
com.pdfunit.tools.ExtractXMPData
```

The utilities are described in the chapter 9.1: "Common Remarks for all Utilities" (p. 102):

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.
-->
<testcase name="hasXFAData_UsingDefaultNamespace">
  <assertThat testDocument="xfa/xfa-enabled.pdf">
    <hasXFAData>
      <withNode tag="foo:log/foo:to"
                value="memory"
                defaultNamespace="http://www.xfa.org/schema/xci/2.6/"
      />
    </hasXFAData>
  </assertThat>
</testcase>
```

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. The available result types are defined as constants for the attribute `withResultType`.

```
<!-- Result types for XPath-processing:  -->

withResultType="BOOLEAN"
withResultType="NUMBER"
withResultType="NODE"
withResultType="NODESET"
withResultType="STRING"
```

Tests with the expected node type `BOOLEAN` are a problem because XPath can not distinguish between “not found” and “false”. Try to use another XPath expression with a different result type.

XPath Compatibility

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.

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. 102)
ExtractBookmarks             9.5: “Extract Bookmarks to XML” (p. 106)
ExtractEmbeddedFiles         9.4: “Extract Attachments” (p. 104)
ExtractFieldsInfo            9.3: “Extract Field Information to XML” (p. 103)
ExtractFontsInfo             9.6: “Extract Font Information to XML” (p. 107)
ExtractImages                9.7: “Extract Images from PDF” (p. 108)
ExtractJavaScript            9.8: “Extract JavaScript to a Text File” (p. 109)
ExtractNamedDestinations     9.9: “Extract Named Destinations to XML” (p. 110)
ExtractSignaturesInfo        9.10: “Extract Signature Information to XML” (p. 111)
ExtractXFAData               9.11: “Extract XFA Data to XML” (p. 112)
ExtractXMPData               9.12: “Extract XMP Data to XML” (p. 113)
RenderPdfClippingAreaToImage 9.13: “Render Page Sections to PNG” (p. 114)
RenderPdfToImages            9.14: “Render Pages to PNG” (p. 115)
```

The utility programs generate files. Their names are derived from those of the input files. The following rules are used to avoid naming conflicts with existing files:

- Generated file names start with an underscore.
- The names have two suffices. The penultimate is `.out` and the last one is the typical suffix for the kind of file type.

For example, when you extract bookmarks from `foo.pdf`, the file `_bookmarks_foo.out.xml` is created. Rename it before using it in a test, because then it is no longer an output file.

The Windows batch scripts in the following chapters demonstrate how to start the programs. These scripts are part of the PDFUnit release, but you have to adapt most of their content to your environment anyway: you need to set the classpath, input file and output directory.

When you start a program without parameters or with incorrect parameters, PDFUnit shows a message detailing the corect command line parameters.

The utilities also run on Unix. Unix developers should easily translate the Windows scripts into shell scripts. If you need assistance, please contact us at: 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. 96) 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!

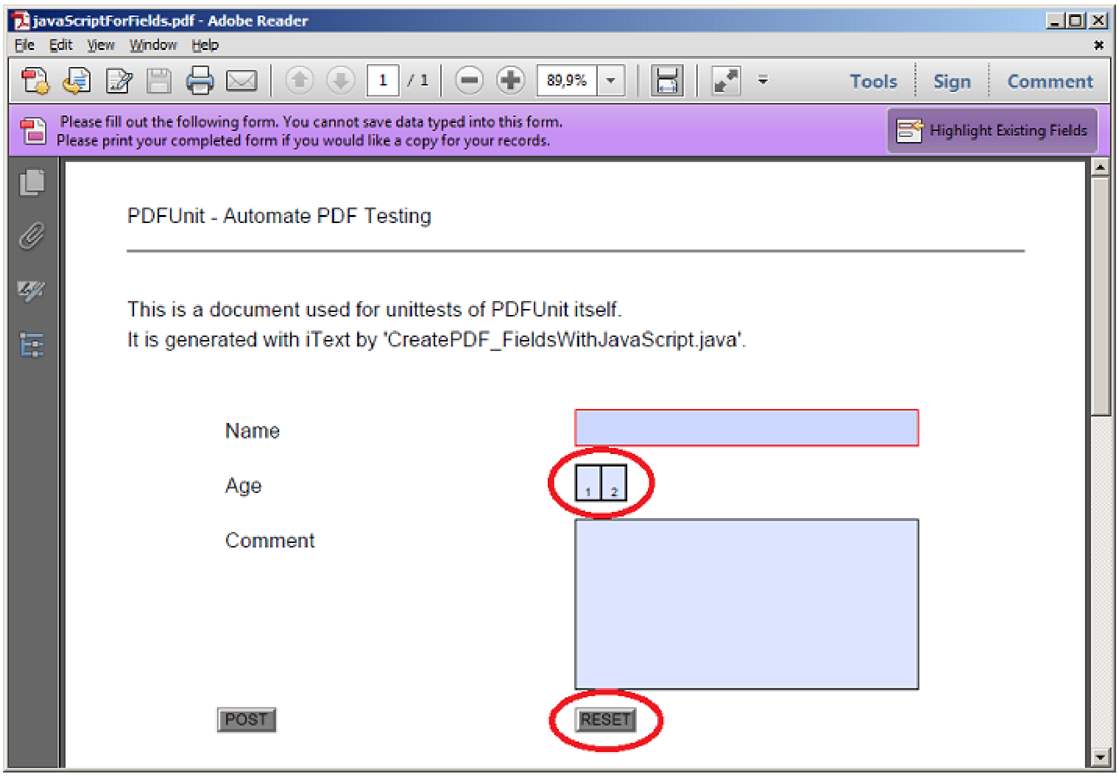
Testing field properties is described in chapter 3.10: “Form Fields” (p. 31).

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:


javaScriptForFields.pdf - Adobe Reader
File Edit View Window Help
1 / 1
89,9%
Tools
Sign
Comment
Highlight Existing Fields

Please fill out the following form. You cannot save data typed into this form.
Please print your completed form if you would like a copy for your records.

PDFUnit - Automate PDF Testing

This is a document used for unittests of PDFUnit itself.
It is generated with iText by 'CreatePDF_FieldsWithJavaScript.java'.

Name

Age

Comment

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"                  ❶
         isEditable="true"                  isRequired="false"
         isPrintable="false"                isVisible="false"
         isHidden="false"                   isHiddenButPrintable="false"
         isVisibleButNotPrintable="false"   isExportable="true"
         isPasswordField="false"            isMultiLineField="false"     ❷
  />
  <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"          ❸
  />
  <!-- 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.

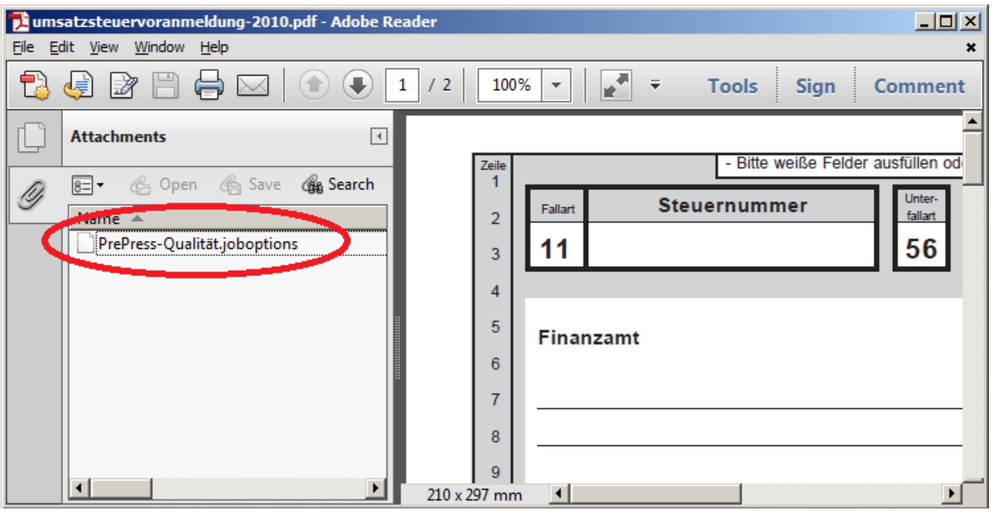
The attachments are exported byte for byte, so all file formats are supported.

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
```

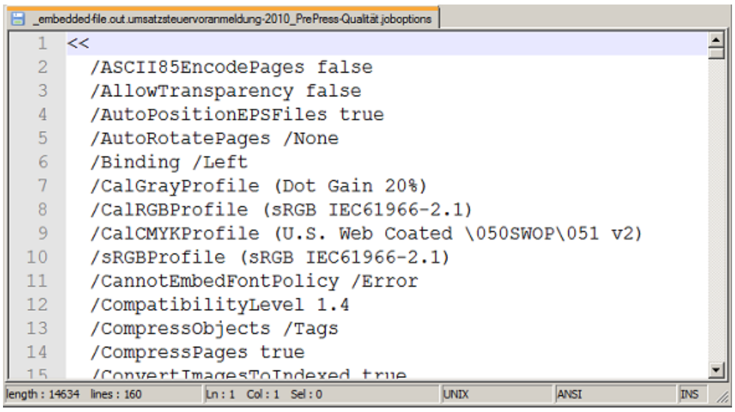
Input

The PDF document `umsatzsteuervoranmeldung-2010.pdf` contains the file `PrePress-Qualität.joboptions`.


umsatzsteuervoranmeldung-2010.pdf - Adobe Reader
File Edit View Window Help
1 / 2
100%
Tools
Sign
Comment
Attachments
Open
Save
Search
Name
PrePress-Qualität.joboptions
Zeile
1
2
3
4
5
6
7
8
9
- Bitte weiße Felder ausfüllen od
Fallart
11
Steuernummer
Unter-fallart
56
Finanzamt
210 x 297 mm

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`.


_embedded-file.out.umsatzsteuervoranmeldung-2010_PrePress-Qualität.joboptions
length : 14634 lines : 160
Ln : 1 Col : 1 Sel : 0
UNIX
ANSI
INS

```
<<
  /ASCII85EncodePages false
  /AllowTransparency false
  /AutoPositionEPSFiles true
  /AutoRotatePages /None
  /Binding /Left
  /CalGrayProfile (Dot Gain 20%)
  /CalRGBProfile (sRGB IEC61966-2.1)
  /CalCMYKProfile (U.S. Web Coated \050SWOP\051 v2)
  /sRGBProfile (sRGB IEC61966-2.1)
  /CannotEmbedFontPolicy /Error
  /CompatibilityLevel 1.4
  /CompressObjects /Tags
  /CompressPages true
  /ConvertImagesToIndexed true
```

9.5. Extract Bookmarks to XML

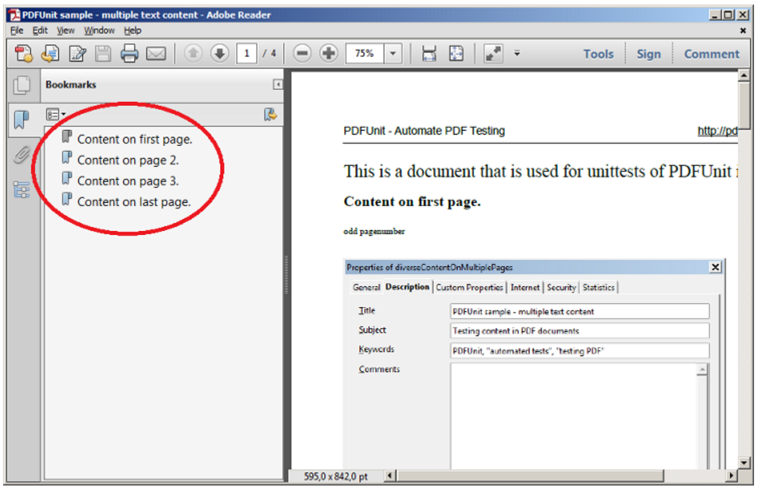
PDFUnit comes with the utility `ExtractBookmarks` which exports bookmarks from PDF documents into an XML file. Chapter 3.4: “Bookmarks and Named Destinations” (p. 19) describes how to use this created XML file for bookmark tests.

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:


PDFUnit sample - multiple text content - Adobe Reader
File Edit View Window Help
1 / 4
75%
Tools
Sign
Comment
Bookmarks
595,0 x 842,0 pt

Content on first page.
Content on page 2.
Content on page 3.
Content on last page.

PDFUnit - Automate PDF Testing http://pd

This is a document that is used for unittests of PDFUnit i

Content on first page.

odd pagenumber

Properties of diverseContentOnMultiplePages

General | **Description** | Custom Properties | Internet | Security | Statistics

Title: PDFUnit sample - multiple text content
Subject: Testing content in PDF documents
Keywords: PDFUnit, "automated tests", "testing PDF"
Comments:

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. 28) 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.

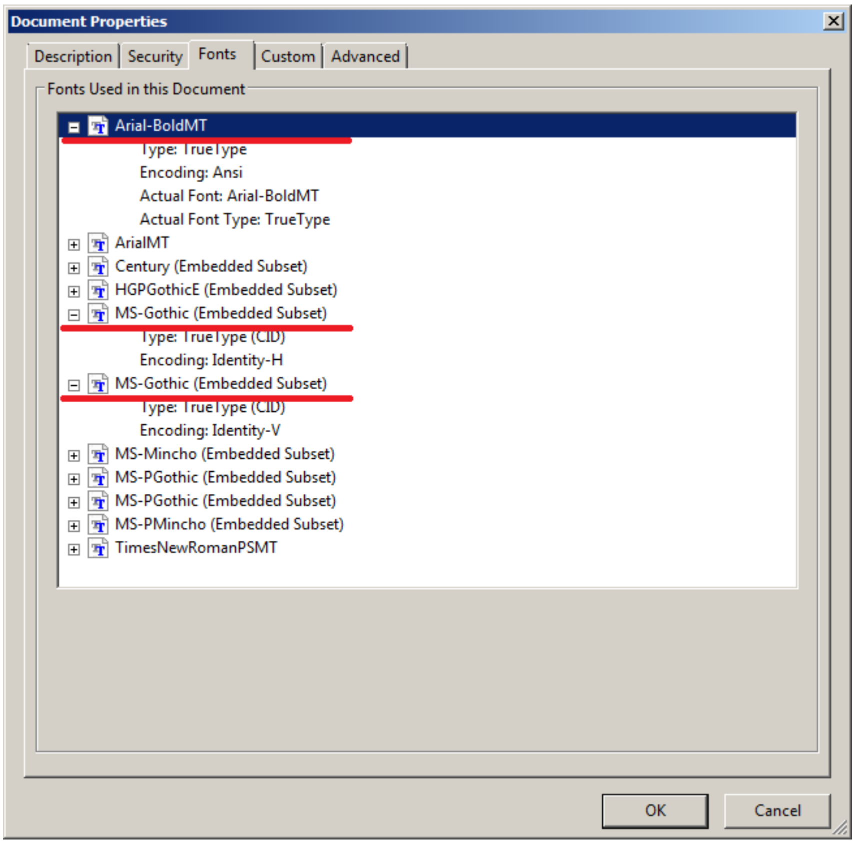
The algorithm that generates the XML file is the same as the one used by the PDFUnit tests.

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`:



Fonts Used in this Document

- Arial-BoldMT
  - Type: TrueType
  - Encoding: Ansi
  - Actual Font: Arial-BoldMT
  - Actual Font Type: TrueType
- ArialMT
- Century (Embedded Subset)
- HGPGothicE (Embedded Subset)
- MS-Gothic (Embedded Subset)
  - Type: TrueType (CID)
  - Encoding: Identity-H
- MS-Gothic (Embedded Subset)
  - Type: TrueType (CID)
  - Encoding: Identity-V
- MS-Mincho (Embedded Subset)
- MS-PGothic (Embedded Subset)
- MS-PGothic (Embedded Subset)
- MS-PMincho (Embedded Subset)
- TimesNewRomanPSMT

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

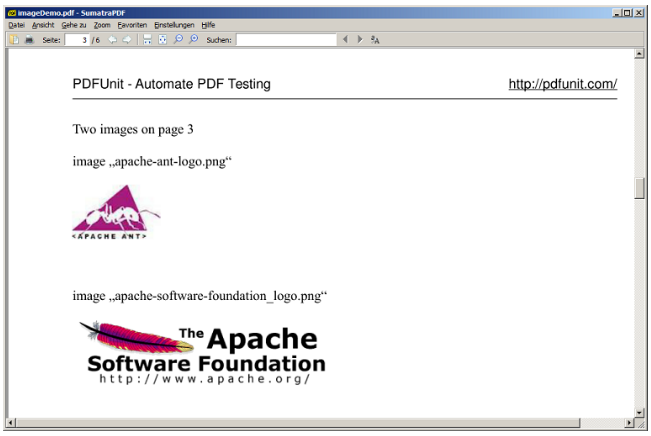
This utility extracts images imbedded in PDF document to PNG images. Each image is written to a separate file. Tests with those images are described in section 3.13: “Images in PDF Documents” (p. 41).

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:


imageDemo.pdf - SumatraPDF
Datei Ansicht Gehe zu Zoom Favoriten Einstellungen Hilfe
Seite: 3 / 6
Suchen:

PDFUnit - Automate PDF Testing http://pdfunit.com/

Two images on page 3

image „apache-ant-logo.png“

image „apache-software-foundation_logo.png“

Output

After running the utility, two files are created:

The Apache Software Foundation
http://www.apache.org/

```
# 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. 43).

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. 103) 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. 19).

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
<?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 containe 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. 55) describes tests with signatures and certificates.

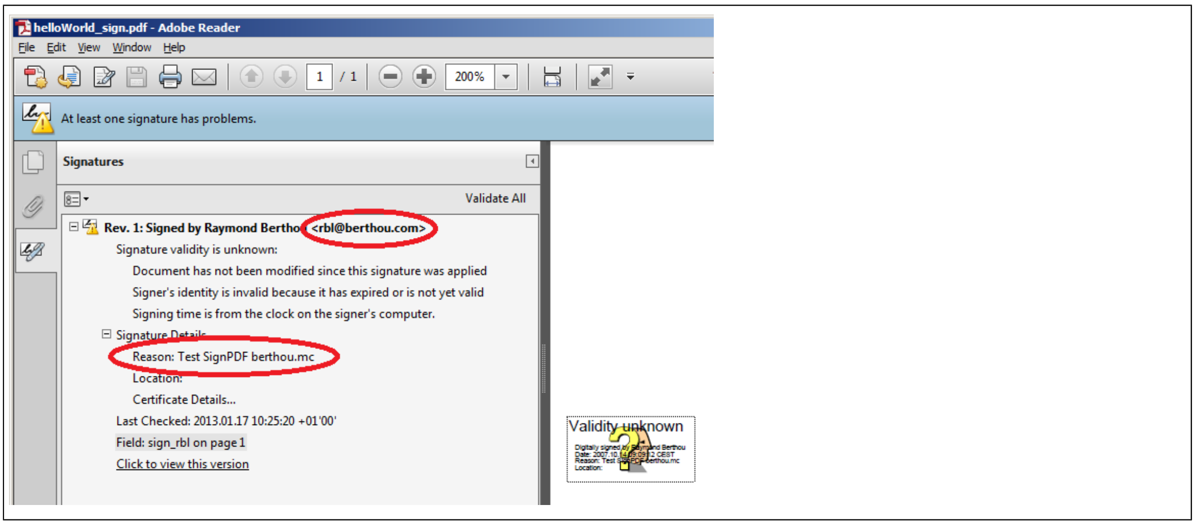
The following script will start the extraction:

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`:



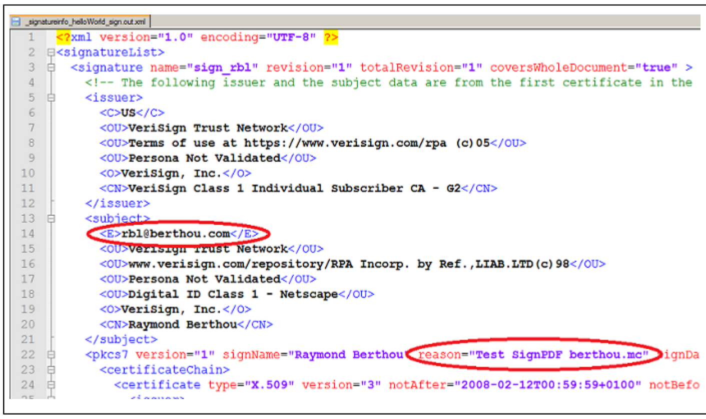
At least one signature has problems.

Rev. 1: Signed by Raymond Bertho <rbl@berthou.com>
Signature validity is unknown:
Document has not been modified since this signature was applied
Signer's identity is invalid because it has expired or is not yet valid
Signing time is from the clock on the signer's computer.
Signature Details
Reason: Test SignPDF berthou.mc
Location:
Certificate Details...
Last Checked: 2013.01.17 10:25:20 +01'00'
Field: sign_rbl on page 1
Click to view this version

Validity unknown
Digitally signed by Raymond Berthou
Date: 2007.10.14 09:09:12 CEST
Reason: Test SignPDF berthou.mc
Location:

Output

Here is a snippet of the output file `_signatureinfo_helloWorld_sign.out.xml`:



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

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 `ExtractXFAData` 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. 68).

Program Start

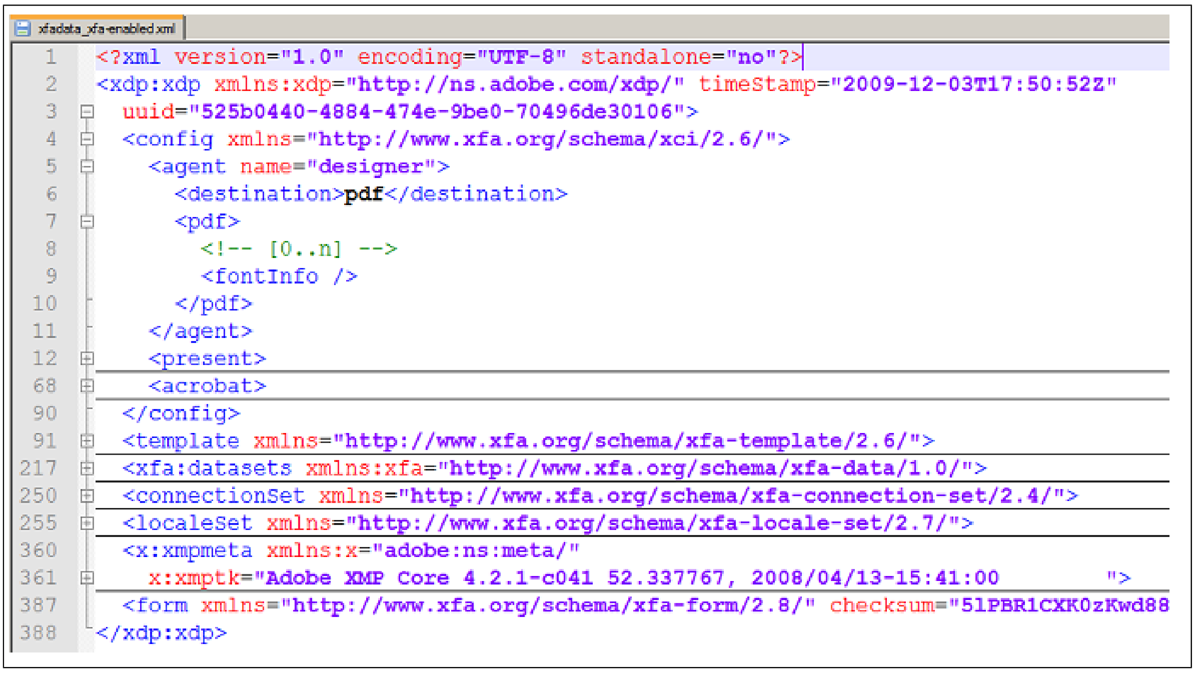
```
::
:: Extract XFA data of a PDF document as XML
::

@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.ExtractXFAData
set OUT_DIR=./tmp
set IN_FILE=xfa-enabled.pdf
set PASSWD=

java  %TOOL%  %IN_FILE%  %OUT_DIR%  %PASSWD%
endlocal
```

Input

The input file for the script is `xfa-enabled.pdf`, a sample document from iText.

Output

The output XML file `_xfadata_xfa-enabled.out.xml` is quite long. To get a better impression of the generated code, some of the XML-Tags in the next picture are folded:



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/" timeStamp="2009-12-03T17:50:52Z"
  uuid="525b0440-4884-474e-9be0-70496de30106">
  <config xmlns="http://www.xfa.org/schema/xci/2.6/">
    <agent name="designer">
      <destination>pdf</destination>
      <pdf>
        <!-- [0..n] -->
        <fontInfo />
      </pdf>
    </agent>
    <present>
    <acrobat>
  </config>
  <template xmlns="http://www.xfa.org/schema/xfa-template/2.6/">
  <xfa:datasets xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
  <connectionSet xmlns="http://www.xfa.org/schema/xfa-connection-set/2.4/">
  <localeSet xmlns="http://www.xfa.org/schema/xfa-locale-set/2.7/">
  <x:xmpmeta xmlns:x="adobe:ns:meta/"
    x:xmptk="Adobe XMP Core 4.2.1-c041 52.337767, 2008/04/13-15:41:00        ">
  <form xmlns="http://www.xfa.org/schema/xfa-form/2.8/" checksum="51PBR1CXK0zKwd88
</xdp:xdp>
```

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. 70).

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-f1ac-4b71-a555-34b7622eaa94'
                 xmlns:pdf='http://ns.adobe.com/pdf/1.3/'
                 pdf:Producer='Acrobat Distiller 6.0.1 (Windows)'
                 pdf:Keywords='LXX, Septuagint, vocabulary, frequency'>
</rdf:Description>
<rdf:Description rdf:about='uuid:f6a30687-f1ac-4b71-a555-34b7622eaa94'
                 xmlns:xap='http://ns.adobe.com/xap/1.0/'
                 xap:CreateDate='2006-05-02T11:35:38-04:00'
                 xap:CreatorTool='PScript5.dll Version 5.2.2'
                 xap:ModifyDate='2006-05-02T11:37:57-04:00'
                 xap:MetadataDate='2006-05-02T11:37:57-04:00'>
</rdf:Description>
...
</rdf:RDF>
</x:xmpmeta>
```

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. 49). 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

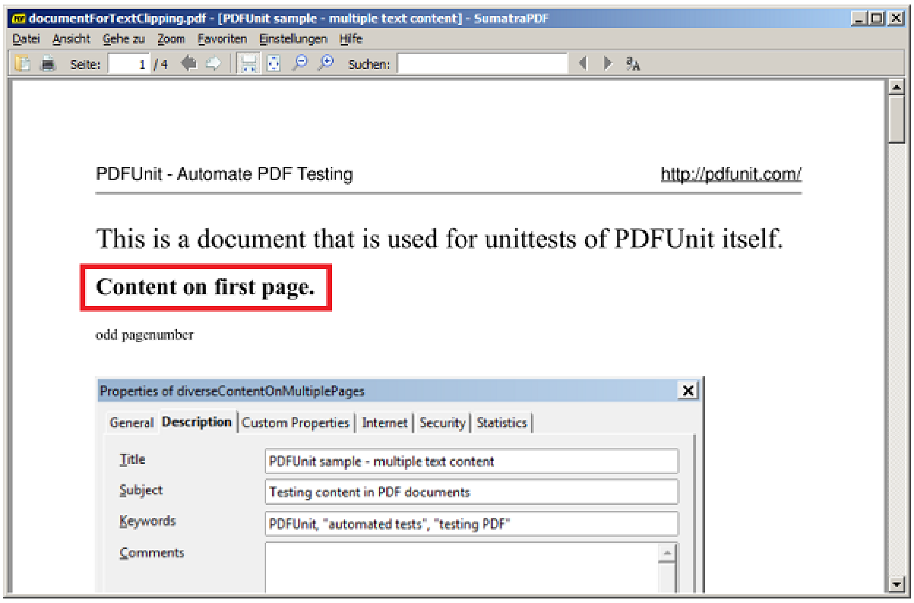
```
::
:: Render a part of a PDF page into an image file
::

@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.` ”


documentForTextClipping.pdf - [PDFUnit sample - multiple text content] - SumatraPDF
Datei Ansicht Gehe zu Zoom Favoriten Einstellungen Hilfe
Seite: 1 / 4
Suchen:

PDFUnit - Automate PDF Testing http://pdfunit.com/

This is a document that is used for unittests of PDFUnit itself.

Content on first page.

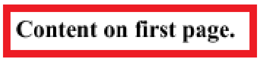
odd pagenumber

Properties of diverseContentOnMultiplePages

General Description Custom Properties Internet Security Statistics

Title	PDFUnit sample - multiple text content
Subject	Testing content in PDF documents
Keywords	PDFUnit, "automated tests", "testing PDF"
Comments	

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

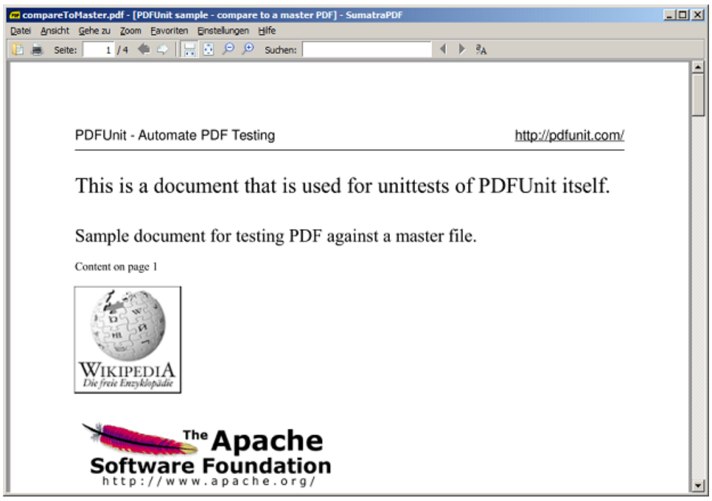
If you want to test formatted text, the only way to do it is to render a PDF page and compare the result with an image of the correctly formatted content. Section 3.17: “Layout - Entire PDF Pages” (p. 48) describes layout-tests using rendered pages. And the utility `RenderPdfToImages` renders a PDF document page by page into PNG files.

Program Start

```
::
:: Render PDF into image files. Each page as a file.
::

@echo off
setlocal
set CLASSPATH=./lib/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:



PDFUnit - Automate PDF Testing http://pdfunit.com/

This is a document that is used for unittests of PDFUnit itself.

Sample document for testing PDF against a master file.

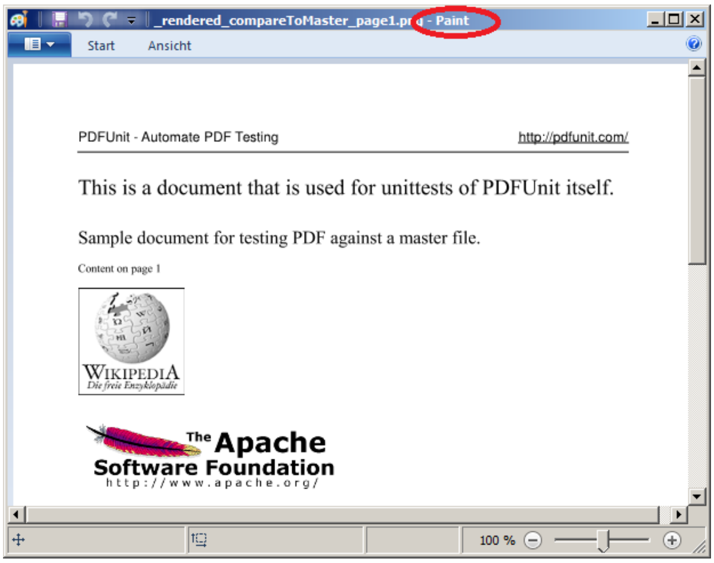
Content on page 1

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.


_rendered_compareToMaster_page1.png - Paint
Start
Ansicht
100 %
WIKIPEDIA
Die freie Enzyklopädie
The Apache Software Foundation
http://www.apache.org/

PDFUnit - Automate PDF Testing http://pdfunit.com/

This is a document that is used for unittests of PDFUnit itself.

Sample document for testing PDF against a master file.

Content on page 1

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.jpedal.PdfDecoder` from the project “jPedal” (http://www.idrsolutions.com/). Thanks to the developers.

Chapter 10. Best Practices

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 a tag to detect **text overflow**.

Solution

```
<testcase name="noTextOverflow_AllFields">
  <assertThat testDocument="acrofields/fieldsWithAttributes.pdf">
    <hasFields>
      <allWithoutTextOverflow />
    </hasFields>
  </assertThat>
</testcase>
```

A similar test can be done with one field:

```
<testcase name="noTextOverflow_Field_AlignLeft">
  <assertThat testDocument="acrofields/fieldSizeAndText.pdf">
    <hasField withName="Textfield, text inside, align left:" >
      <withoutTextOverflow />
    </hasField>
  </assertThat>
</testcase>
```

Chapter 3.11: “Form Fields - Text Overflow” (p. 37) 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

```
<!--
  This sample shows how to verify that a logo is visible on each page.
-->
<testcase name="verifyNewLogoOnEveryPage">
  <assertThat testDocument="images/imagesWithSameImagesOnOnePage.pdf">
    <containsImage file="images/exported-image_ant-logo-PNG.png"
                   on="EVERY_PAGE"
    />
  </assertThat>
</testcase>
```

10.3. 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

```
<!--
  Situation: two companies combine.
  This sample shows how to verify that the new signature is used.
-->
<testcase name="verifyNewSignatureOnLastPage">
  <assertThat testDocument="images/imagesWithSameImagesOnOnePage.pdf">
    <containsImage file="images/CEO-signature.png"
                   on="LAST_PAGE"
    />
  </assertThat>
</testcase>
```

10.4. Name 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 a tag to check documents for the **non-existence** of text.

Solution

```
<!--
  This example shows how to verify that an expected text
  is not present in the complete document.
-->
<testcase name="verifyOldCEONotPresent">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="EVERY_PAGE">
      <notContaining>NameOfOldCEO</notContaining>
    </hasText>
  </assertThat>
</testcase>
```

10.5. 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.
-->
<testcase name="hasBookmarks_LimitedNestedDepthTo1">
  <assertThat testDocument="bookmarks/bookmarksWithPdfOutline.pdf">
    <hasBookmarks>
      <matchingXPath expr="count(//Title[count(ancestor::Title) > 1]) = 0" />
    </hasBookmarks>
  </assertThat>
</testcase>

<!--
  Important hint:
  The value of the attribute 'expr' is used as a parameter of type String.
  So you have to use double quotes as outer quotes for the attribute.
  Otherwise you get a compile error after the transformation from XML to Java.
-->
```

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

- Windows-XP, 32 Bit
- Windows-7, 64 Bit
- Kubuntu Linux 12/04, 32 Bit
- Kubuntu Linux 12/04, 64 Bit
- Mac OS X, 64 Bit

Java Version

- Oracle JDK-1.7.0, 32 + 64 Bit
- Oracle JDK-1.8.0, 64 Bit
- Oracle JDK-1.8.0 b100, Windows, 32 + 64 Bit
- IBM J9, R26_Java726_SR4, Windows 7, 64 Bit
- OpenJDK-1.6.0, 64 Bit
- Apple Inc. 1.6.0, 64 Bit

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.

11.2. Installation

The following sections describe installing PDFUnit-XML, PDFUnit-Java and iText. All of these components are needed to run PDFUnit-XML.

Installation of PDFUnit-XML

Download the file `pdfunit-xml-VERSION.zip` from the project site: http://www.pdfunit.com/en/download/index.html. If you have purchased a license, you get the ZIP file by mail.

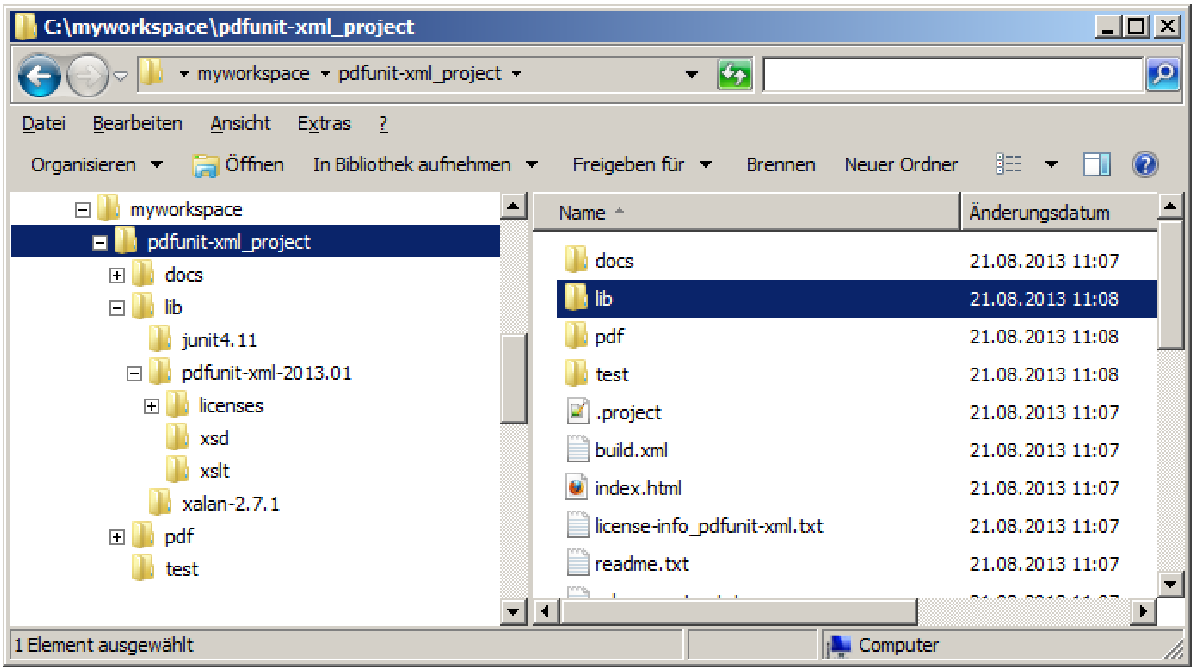
Unzip the file in a folder which is refered to in the following text as `PROJECT_HOME`.

The following table describes the directory structure.

Directory	Description
`PROJECT_HOME`	ANT build script: `build.xml`. Scripts to run the utilities: `*.bat`. Project file for Eclipse: `.project`. A starting point for useful links: `index.html` and `readme.txt`. And `release-notes.txt`, `license-info_pdfunit-xml.txt`.

Directory	Description
`PROJECT_HOME/lib`	Place the iText and PDFUnit-Java runtime libraries into this folder. They are necessary to run PDFUnit-XML.
`PROJECT_HOME/lib/pdfunit-xml-VERSION/xslt`	Style sheets that are needed to process PDFUnit-XML
`PROJECT_HOME/lib/pdfunit-xml-VERSION/xsd`	XML schema file to validate the PDFUnit tests
`PROJECT_HOME/lib/pdfunit-xml-VERSION/licenses`	License information for all libraries used by PDFUnit
`PROJECT_HOME/test`	Examples. You should place your own tests here.
`PROJECT_HOME/pdf`	PDF documents for the test examples. You can also store your own test documents here.

The following figure illustrates this structure.



Name	Änderungsdatum
docs	21.08.2013 11:07
lib	21.08.2013 11:08
pdf	21.08.2013 11:08
test	21.08.2013 11:08
.project	21.08.2013 11:07
build.xml	21.08.2013 11:07
index.html	21.08.2013 11:07
license-info_pdfunit-xml.txt	21.08.2013 11:07
readme.txt	21.08.2013 11:07

The file `PROJECT_HOME/index.xml` contains links to useful files in the project.

The installation comes with the start-script `PROJECT_HOME/build.xml`. This ANT script can run on Windows and Linux. You can start the script as described in the next chapter. The test results can be reached from the page `PROJECT_HOME/index.html`.

It is possible to start the project with other scripting languages. Please write a mail to support[at]pdfunit.com if you need help.

Installation of PDFUnit-Java

PDFUnit-XML requires PDFUnit-Java at runtime. It can be downloaded here: PDFUnit-Java. Make sure that the PDFUnit-Java version matches the PDFUnit-XML version. Which versions work together are documented in the file 'readme.txt'.

Unzip the downloaded ZIP file and copy the resulting folder so it is subfolder of `PROJECT_HOME/lib`. This folder is called `PDFUNITJAVA_HOME` in the following text.

Installation of iText

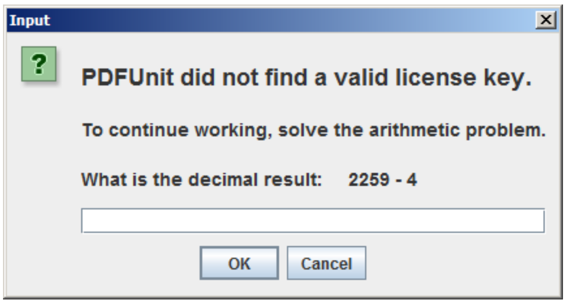
You need iText version 5.3.3 or higher. Download it from here (http://sourceforge.net/projects/itext/files/iText/).

Unzip the iText ZIP file and also copy the resulting folder so it is a subfolder of `PROJECT_HOME/lib`.

Note that iText requires a license for commercial use.

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.



PDFUnit did not find a valid license key.

To continue working, solve the arithmetic problem.

What is the decimal result: 2259 - 4

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 to extract information from PDF documents. These tools do not need a license key.

Ordering a License Key

If you use PDFUnit in a commercial context you need a license. Write a mail to 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`. Copy this file to `PDFUNITJAVA_HOME`.

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 11.5: “Verifying the Configuration” (p. 127).

11.3. Running PDFUnit-XML

The fundamentals of Running PDFUnit-XML

The ANT script PDFUNITXML_HOME/build.xml starts your PDFUnit-XML tests. The script requires that Java and ANT are installed and can be found. All configurations are done in this script.

If you want to change the file build.xml, make sure that the classpath contains the directory PDFUNITJAVA_HOME. The file config.properties is searched on the classpath.

Running PDFUnit-XML from a Shell

Open a “shell”. (Depending on your operating system you might call it “command prompt”, “DOS box” or “command line”.) Go to your project directory PROJECT_HOME and enter the command:

```
ant all
```

or:

```
runPDFUnit.bat
```

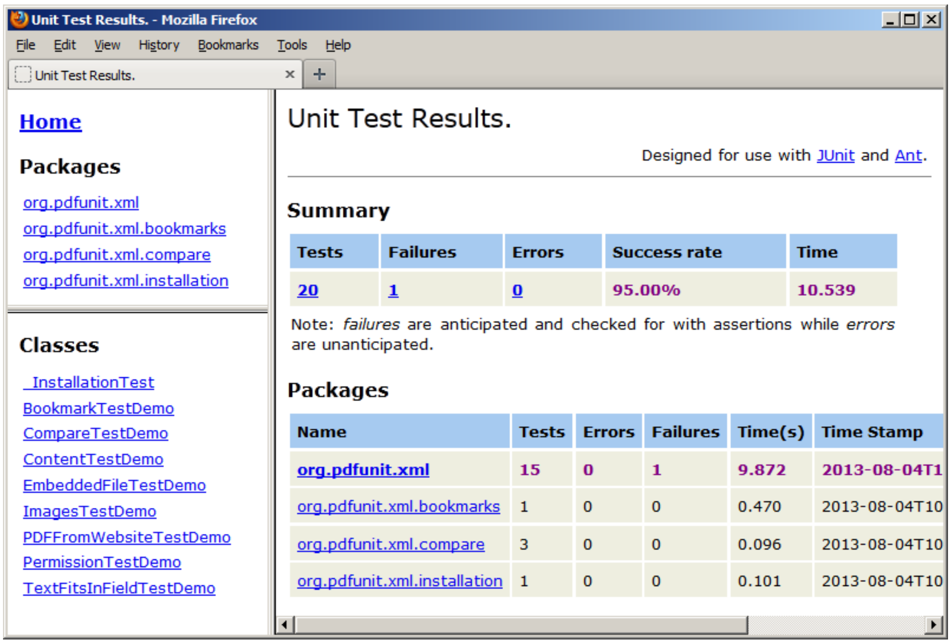
The scripts runPDFUnit.bat and runPDFUnit.sh call ANT and allow you to configure the pathes to your Java and ANT installations.

Assuming Java and ANT are installed and registered in the operating system's path, a detailed log can be seen during execution of the ANT script:

```

Buildfile: C:\pdfunit-xml_demo\build.xml

clean:
     [delete] Deleting directory C:\pdfunit-xml_demo\build_ant

00_resolveDTD:
     [echo] start resolving DTD entities ...
     [mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\xml
     [xslt] Loading stylesheet C:\pdfunit-xml_demo\lib\pdfunit-xml-2015.10\xslt\...
     [xslt] Processing C:\pdfunit-xml_demo\src\test\xml\CompareTestDemo.xml
      ...
     [echo] ... finished

     ...
```

```
     ...

01_verifyXML:
     [echo] start validating PDFUnit test files (xml) ...
     [copy] Copying 1 file to C:\pdfunit-xml_demo\build_ant\xsd
     [echo] ... finished

02_generateJavaSourcesFromXML:
     [echo] start transforming PDFUnit test files (xml) into Java code ...
     [mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\java\org\pdfunit\xml
     [xslt] Loading stylesheet C:\pdfunit-xml_demo\lib\pdfunit-xml-2015.10\xslt\...
     [xslt] Processing C:\pdfunit-xml_demo\src\test\xml\CompareTestDemo.xml
      ...
     [echo] ... finished

     ...
     ...

03_compileGeneratedSources:
     [echo] start compiling generated sources ...
     [mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\classes
     [javac] Compiling 9 source files to C:\pdfunit-xml_demo\build_ant\classes
     [echo] ... finished

04_runUnittest:
     [echo] start running unit tests from compiled sources ...
     [mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\junit\data
     [junit] Running org.pdfunit.xml.ContentTestDemo
     [junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 1.054 sec
      ...
     [echo] ... finished

-testreport:
     [echo] start creating HTML report from unit test result ...
     [mkdir] Created dir: C:\pdfunit-xml_demo\build_ant\junit\html
     [junitreport] Processing C:\pdfunit-xml_demo\build_ant\junit\html\...
     [junitreport] Loading stylesheet JAR:file:/C:/environment/share32/tools/...
     [junitreport] Transform time: 890ms
     [junitreport] Deleting: c:\Temp\null1904905169
     [echo] ... finished
     [echo] Please look for index.html in subfolder build_ant/junit/html

all:

BUILD SUCCESSFUL
```

The script creates a detailed HTML report for all tests. It is be found in the folder `PROJECT_HOME/build_ant/junit/html/index.html`. Here's a look at the report of the demo project included in the installation ZIP file you downloaded:



Home

Packages

org.pdfunit.xml
org.pdfunit.xml.bookmarks
org.pdfunit.xml.compare
org.pdfunit.xml.installation

Classes

_InstallationTest
BookmarkTestDemo
CompareTestDemo
ContentTestDemo
EmbeddedFileTestDemo
ImagesTestDemo
PDFFromWebsiteTestDemo
PermissionTestDemo
TextFitsInFieldTestDemo

Unit Test Results.

Designed for use with JUnit and Ant.

Summary

Tests	Failures	Errors	Success rate	Time
20	1	0	95.00%	10.539

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)	Time Stamp
org.pdfunit.xml	15	0	1	9.872	2013-08-04T1
org.pdfunit.xml.bookmarks	1	0	0	0.470	2013-08-04T10
org.pdfunit.xml.compare	3	0	0	0.096	2013-08-04T10
org.pdfunit.xml.installation	1	0	0	0.101	2013-08-04T10

If you forget to copy the PDFUnit-Java libraries into the directory `PROJECT_HOME/lib/pdfunit-java_VERSION`, the following error message appears:

```
[javac]   ... error: cannot find symbol
[javac]     AssertThat.document(filename)
[javac]     ^
[javac]   symbol:   variable AssertThat
```

Using PDFUnit-XML in IDE's

You prefere to start PDFUnit-XML from an IDE like Eclipse? You can, simply start the ANT script `PROJECT_HOME/build.xml` from within the IDE. If you need help, write to us at support[at]pdfunit.com.

11.4. 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 = ./

```

11.5. 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
```

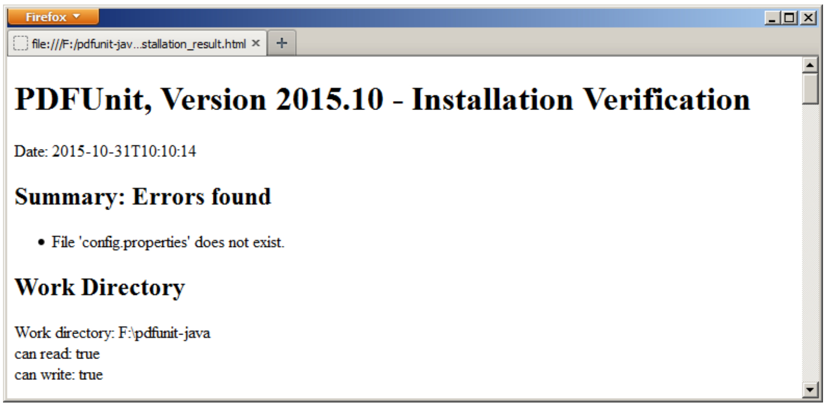
You have do edit the paths depending on your installation.

The stylesheet option is provided to use individual stylesheets. If you don't use a stylesheet, a simple one is used automatically.

The script produces the following output on the console:

```

Checking installation ...
... finished. Report created, see 'verifyInstallation_result.html'.
```

The resulting report file shows errors and information about the classpath, environment variables and other runtime related data:



PDFUnit, Version 2015.10 - Installation Verification

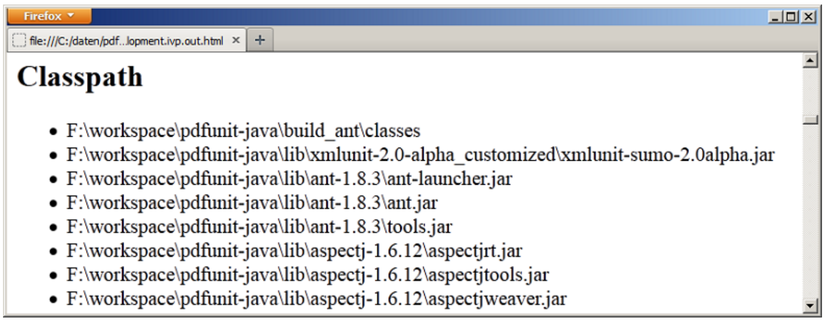
Date: 2015-10-31T10:10:14

Summary: Errors found

- File 'config.properties' does not exist.

Work Directory

Work directory: F:\pdfunit-java
can read: true
can write: true



Classpath

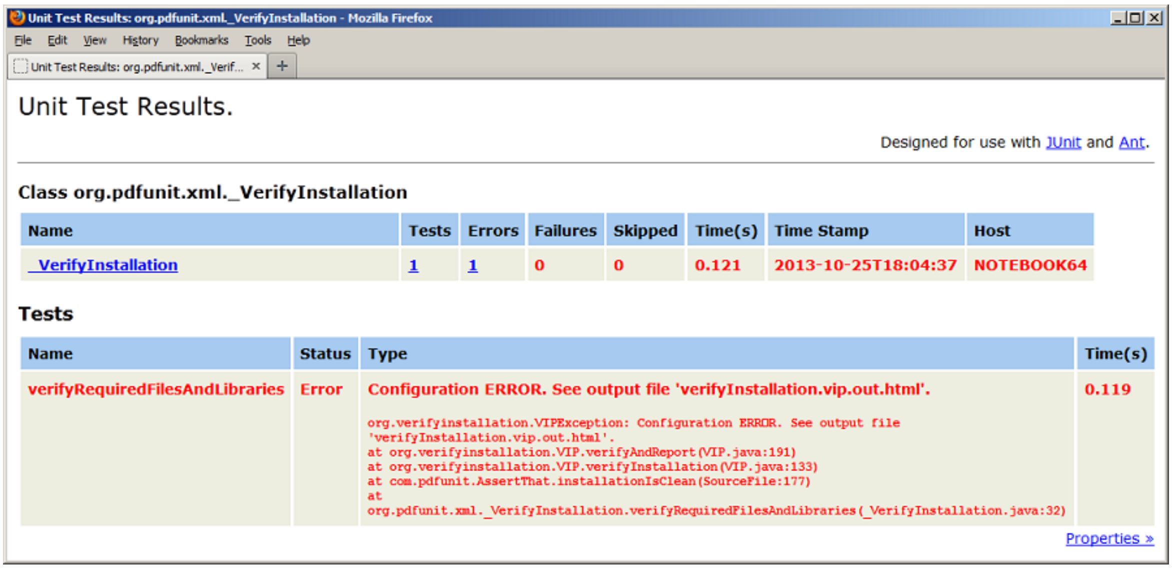
- F:\workspace\pdfunit-java\build_ant\classes
- F:\workspace\pdfunit-java\lib\xmlunit-2.0-alpha_customized\xmlunit-sumo-2.0alpha.jar
- F:\workspace\pdfunit-java\lib\ant-1.8.3\ant-launcher.jar
- F:\workspace\pdfunit-java\lib\ant-1.8.3\ant.jar
- F:\workspace\pdfunit-java\lib\ant-1.8.3\tools.jar
- F:\workspace\pdfunit-java\lib\aspectj-1.6.12\aspectjrt.jar
- F:\workspace\pdfunit-java\lib\aspectj-1.6.12\aspectjtools.jar
- F:\workspace\pdfunit-java\lib\aspectj-1.6.12\aspectjweaver.jar

Verifying as an XML Test

You can also verify the installation using an XML test. That makes it possible to check the environment of the currently running tests:

```
<comment>
  This tests verifies that all required libraries and files are found.
  Additionally it logs some system properties and writes
  all of them into both an XML file and an HTML formatted file.
</comment>

<testcase name="verifyRequiredFilesAndLibraries">
  <verifyInstallation verificationFile="verifyInstallation.vip"/>
</testcase>
```

This XML test performs the same tests as the script described above. In case of a configuration error, the test is “red” and the error message refers to the report file:


Unit Test Results: org.pdfunit.xml._VerifyInstallation - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
Unit Test Results: org.pdfunit.xml._Verif...

Unit Test Results.

Designed for use with JUnit and Ant.

Class org.pdfunit.xml._VerifyInstallation

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
_VerifyInstallation	1	1	0	0	0.121	2013-10-25T18:04:37	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
verifyRequiredFilesAndLibraries	Error	Configuration ERROR. See output file 'verifyInstallation.vip.out.html'. org.verifyinstallation.VIPException: Configuration ERROR. See output file 'verifyInstallation.vip.out.html'. at org.verifyinstallation.VIP.verifyAndReport(VIP.java:191) at org.verifyinstallation.VIP.verifyInstallation(VIP.java:133) at com.pdfunit.AssertThat.installationIsClean(SourceFile:177) at org.pdfunit.xml._VerifyInstallation.verifyRequiredFilesAndLibraries(_VerifyInstallation.java:32)	0.119

Properties »

The report file contains the same data as when it was created by a shell script.

11.6. Update PDFUnit-XML

The installation of a new release of PDFUnit-XML runs just like the initial installation, because PDFUnit-XML is always delivered as a full release, never incrementally.

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 and a new license file by mail.

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 into a new folder.

Make sure that the new version of PDFUnit-XML is compatible with the existing version of PDFUnit-Java. You may also install a new version of PDFUnit-Java as described in the next chapter.

Copy all your test files, PDF documents and individual configuration files from the old project into the new release.

Last Step

Run your existing tests with the new release. If there are no documented incompatibilities between the old and new release, your tests should run. Otherwise read the release information.

11.7. Update PDFUnit-Java

PDFUnit-XML needs PDFUnit-Java to be installed before it can run. The version of PDFUnit-XML must be compatible with your version of PDFUnit-Java.

If the version numbers of PDFUnit-XML and PDFUnit-Java are the same, the versions will always match. If they not equal, you can find more information about compatibility here: http://www.pdfunit.com/en/download/index.html.

Getting the New Release

If you use PDFUnit with a license, you will receive a new release and a license file by mail.

Otherwise, download the new ZIP file from: http://www.pdfunit.com/en/download/index.html.

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 into a new folder which is refered to in the following text as `PDFUNITJAVA_NEW`. The project folder with the old release is refered to as `PROJECT_HOME`.

Delete the folder `PROJECT_HOME/lib/pdfunit-OLD-VERSION`.

Copy the folder `PDFUNITJAVA_NEW` to `PROJECT_HOME/lib/pdfunit-NEW-VERSION`.

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's `*.lic` file. Remove the old license file.

Make sure that the new version of PDFUnit-Java is compatible with the existing version of PDFUnit-XML. You may also install a new version of PDFUnit-XML as described in the previous chapter.

Last Step

Run your existing tests with the new release. If there are no documented incompatibilities between the old and new releases, your tests should be run. Otherwise read the release information.

11.8. Uninstall

PDFUnit can be uninstalled cleanly by deleting the installation directories. PDFUnit doesn't create any entries in the registry or system directories, so none need to be removed. Don't forget to remove the references to JAR files and PDFUnit directories from your own scripts.

Chapter 12. PDFUnit for non-XML Systems

12.1. A quick Look at PDFUnit-Java

The first implementation of PDFUnit was “PDFUnit-Java”. It is the reference for implementations in other programming languages. Whenever it is possible, the keywords in all implementations are chosen to be the same as in PDFUnit-Java.

The following examples shows that the API follows the “Fluent Interface” (http://de.wikipedia.org/wiki/Fluent_Interface):

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

  int upperLeftX  =  50;
  int upperLeftY  = 130;
  int width       = 170;
  int height      =  25;
  ClippingArea inClippingArea = new ClippingArea(upperLeftX, upperLeftY, width, height);

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

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

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

```
@Test
public void hasSignature() throws Exception {
  String filename = PATH + "signed/sampleSignedPDFDocument.pdf";
  String xpath = "//signature[@name='Signature2']";

  XPathExpression expression = new XPathExpression(xpath);
  AssertThat.document(filename)
            .hasSignatures()
            .matchingXPath(expression)
  ;
}
```

A detailed documentation of PDFUnit-Java is available from http://www.pdfunit.com/en/documentation/java/index.html.

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. Instantiation of PDF Documents

PDFUnit-XML reads a PDF documents using a **file name** or a **URL**. To distinguish between these two formats the attribute `testIsURL` or `masterIsURL` must be used. Here an example:

```
<comment>
  This test shows the usage of PDFUnit-XML when comparing two PDF
  documents loaded as a URL.
</comment>

<testcase name="compareDocuments_bothLoadedFromURL">
  <assertThat testDocument="./testdocuments/test.pdf"
              masterDocument="http://localhost/.../master/master.pdf"
              masterIsURL="YES"
  >
    ...
  </assertThat>
</testcase>
```

If documents are password protected, PDFUnit needs either the “user password” or the “owner password” to open the file. Both have to be defined in a separate attribute:

```
<testcase name="compareEncryptedTestAgainstEncryptedMaster">
  <assertThat testDocument="master/test.pdf"
              testPassword="secret-test"
              masterDocument="master/master.pdf"
              masterPassword="secret-master"
  >
    ...
  </assertThat>
</testcase>
```

13.2. 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:  -->

<!-- Predefined constants for pages: -->
<xxx on="ANY_PAGE"   />
<xxx on="EVEN_PAGES" />
<xxx on="EACH_PAGE"  />
<xxx on="EVERY_PAGE" />
<xxx on="FIRST_PAGE" />
<xxx on="LAST_PAGE"  />
<xxx on="ODD_PAGES"  />

<!-- Attributes for individual pages: -->
<xxx onPage=".."            />
<xxx onEveryPageAfter=".."  />
<xxx onEveryPageBefore=".." />
<xxx onAnyPageAfter=".."    />
<xxx onAnyPageBefore=".."   />
```

`on="EACH_PAGE"` and `on="EVERY_PAGE"` are functionally identical. The redundancy is intended due to linguistic reasons.

Here an example using the attribute `on=".."` and a predefined page constant:

```
<testcase name="hasText_MultipleSearchTokens_EvenPages">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="EVEN_PAGES">
      <containing>Content</containing>
      <containing>even pagenumber</containing>
    </hasText>
  </assertThat>
</testcase>
```

Individual Pages

The next example shows how individual pages can be addressed. Page numbers must be separated by commas:

```
<testcase name="hasText_OnMultiplePages_SelectedPages">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onPage="1, 2, 3">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

Open Ranges

Constants are available for contiguous ranges of pages at the beginning and at the end of a document:

```
<testcase name="hasText_OnAnyPageAfter1">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onAnyPageAfter="1">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="hasText_OnEveryPageBefore3">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText onEveryPageBefore="3">
      <containing>Content on</containing>
    </hasText>
  </assertThat>
</testcase>
```

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" >`.

```
<testcase name="hasText_SpanningOver2Pages_matchingRegex">
  <assertThat testDocument="&pdfdir;/content/text-starts-on-page1-continues-on-page2.pdf">
    <hasText fromPage="1" toPage="2" >
      <inClippingArea upperLeftX="18" upperLeftY="30"
                      width="182" height="238"
                      unit="MILLIMETER"
      >
        <matchingRegex>Text starts  on page 1.*continues on page 2</matchingRegex>
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

In combination with `<hasText fromPage="1" toPage="2" >` only the tags `<containing>`, `<matchingRegex>`, `<notContaining>` and `<notMatchingRegex>` can be used.

Important Hints

- Page numbers begin with '1'.
- The page number in `onXxxPageBefore` and `onXxxPageAfter` are both exclusive.

- The page number in the attributes `from` and `to` are both inclusive.
- The attribute `onEveryPageXxx` means that the expected text has to exist on each page in the given range.
- When using `onAnyPageXXX`, a test is successful if the expected string exists on one or more pages in the given range.

13.3. Comparing Text

An expected text and actual text on a PDF page can be compared using the following tags:

```
<!-- Comparing text: -->

<startingWith />
<containing />                           ❶
<matchingComplete />                     ❷
<matchingRegex />
<endingWith />

<notStartingWith />
<notContaining />                        ❸
<notMatchingRegex />
<notEndingWith />

<containing       whitespaces=".." />    ❹
<matchingComplete whitespaces=".." />    ❺
<notContaining    whitespaces=".." />    ❻
```

❶❷❸ Tests **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.

❹❺❻ Use the optional attribute `whitespaces=".."` to treat whitespaces in a specail way. For this attribute, the constants `KEEP`, `NORMALIZE`, and `IGNORE` exist. These constants are explained separately in section 13.4: “Whitespace Processing” (p. 135).

Comparisons with regular expressions follow the rules and possibilities of the class `java.util.regex.Pattern`:

```
<!-- Using regular expression to compare page content: -->
<testcase name="hasText_MatchingRegex">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="FIRST_PAGE">
      <matchingRegex>.*[Cc]ontent.*</matchingRegex>
    </hasText>
  </assertThat>
</testcase>
```

13.4. 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 the attribute `whitespaces=".."` with one of the three predefined constants. `NORMALIZE` is the default if nothing is declared:

```
<!-- Constants for whitespace processing: -->

<xxx whitespaces="IGNORE"    />  ❶
<xxx whitespaces="KEEP"      />  ❷
<xxx whitespaces="NORMALIZE" />  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 tags:

```
<!-- Tags which allow to define the whitespace processing -->

<hasXXXAction>
  <containing       whitespaces=".." />
  <matchingComplete whitespaces=".." />
</hasXXXAction>

<hasText>
  <containing       whitespaces=".." />
  <notContaining    whitespaces=".." />
  <matchingComplete whitespaces=".." />
</hasText>
```

An example:

```
<testcase name="hasText_WithLineBreaks_UsingIGNORE">
  <assertThat testDocument="content/diverseContentOnMultiplePages.pdf">
    <hasText on="FIRST_PAGE">
      <matchingComplete whitespaces="IGNORE">
        PDFUnit - Automated PDF Tests http://pdfunit.com/
        This is a document that is used for unit tests of PDFUnit itself.
        Content on first page.
        odd pagenumber
        Page # 1 of 4
      </matchingComplete>
    </hasText>
  </assertThat>
</testcase>
```

The expected string in this example is written with many linebreaks, which are different from the linebreaks inside the PDF page. However when using `whitespaces="IGNORE"` the test runs successfully.

As an exception to this rule, no tag 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.

13.5. Single and Double Quotation Marks inside Strings

The meaning of double quotes in XML, XPath and Java

Strings inside XML Tags are enclosed by a start tag and an end tag. So the content may have any combination of single and double quotes.

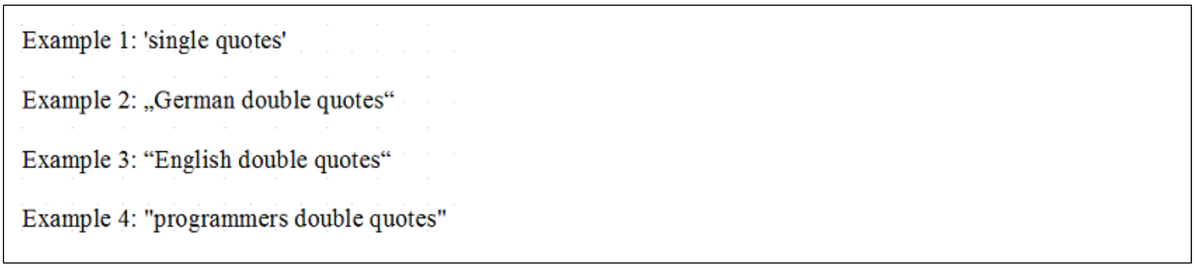
The values of XML attributes are delimited by either single or double quotes. This means that you can use double quotes as content when you use single quotes as the delimter and vice versa. You can also use the entities `&apos;` and `&quot;` as a content.

An arbitrary use of single and double quotes in XPath expressions is not possible, even if you use entities. It is recommended to use single quotes.

In Java strings will be enclosed in double quotes. And because PDFUnit-XML tests will first be converted into Java code and then executed as a Java program, double quotes must be escaped by a backslash when they are part of the content.

Different types of quotes

Important Notice: The term “Quote” has different meanings as the following picture shows:



Example 1: 'single quotes'

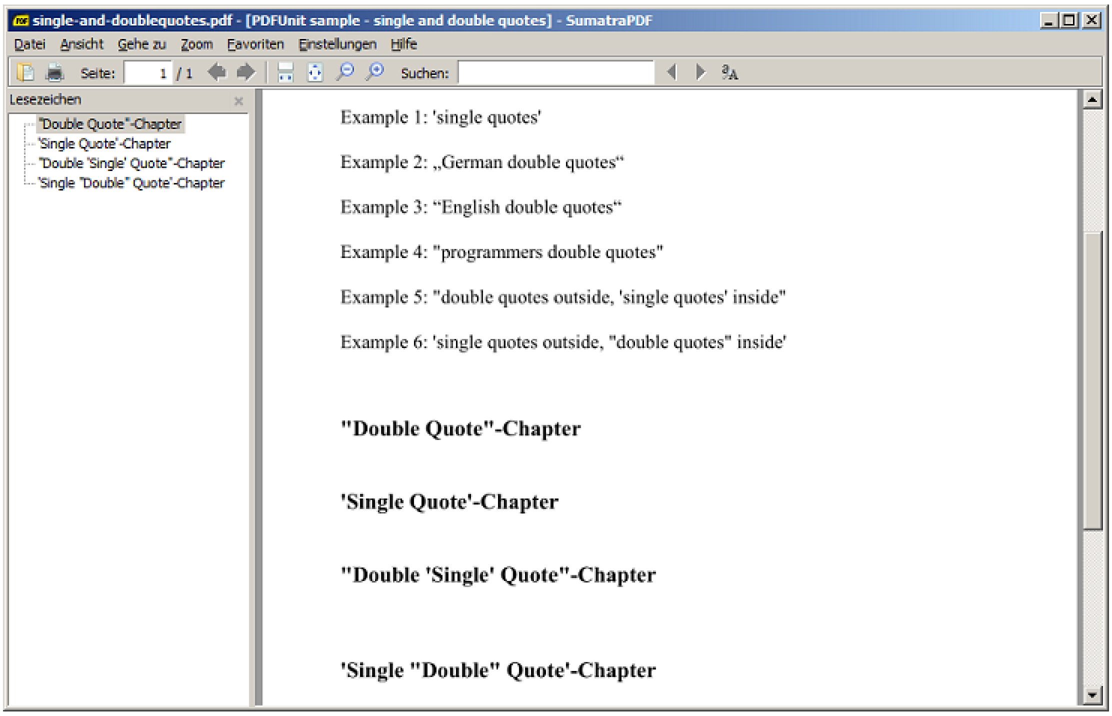
Example 2: „German double quotes“

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 exisiting PDF document and paste them into your XML editor.

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:



single-and-doublequotes.pdf - [PDFUnit sample - single and double quotes] - SumatraPDF
Datei Ansicht Gehe zu Zoom Favoriten Einstellungen Hilfe
Seite: 1 / 1
Suchen:
Lesezeichen
"Double Quote"-Chapter
'Single Quote'-Chapter
"Double 'Single' Quote"-Chapter
'Single "Double" Quote'-Chapter

Example 1: 'single quotes'

Example 2: „German double quotes“

Example 3: “English double quotes“

Example 4: "programmers double quotes"

Example 5: "double quotes outside, 'single quotes' inside"

Example 6: 'single quotes outside, "double quotes" inside'

"Double Quote"-Chapter

'Single Quote'-Chapter

"Double 'Single' Quote"-Chapter

'Single "Double" Quote'-Chapter

Quotes in Tags

Single quotes in PDFUnit tags cause no problems. But double quotes must be escaped with a backslash:

```
<testcase name="tagWith_SingleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 1: 'single quotes'</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_GermanDoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 2: „German double quotes“</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_EnglishDoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 3: “English double quotes“</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_ProgrammersDoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 4: \"programmers double quotes\"</containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_DoubleAndSingleQuotes_1">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>
        Example 5: \"double quotes outside, 'single quotes' inside\"
      </containing>
    </hasText>
  </assertThat>
</testcase>
```

```
<testcase name="tagWith_DoubleAndSingleQuotes_2">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <containing>Example 6: 'single quotes outside, \"double quotes\" inside'</containing>
    </hasText>
  </assertThat>
</testcase>
```

Quotes in Attributes

Single quotes in attributes can be used without restriction, when the attribute is enclosed by double-quotes. And vice versa.

The following examples show a valid usage of single and double quotes in attributes.

```
<testcase name="attributeWith_DoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmark withLabel='"Double Quote"-Chapter' />
  </assertThat>
</testcase>
```

```
<testcase name="attributeWith_SingleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmark withLabel="'Single Quote'-Chapter" />
  </assertThat>
</testcase>
```

But you may not use **not** single and double quotes **at the same time** within a string, not even as entities. The next examples shows an **invalid** usage of quotes:

```
<!--
  This example fails, because single- and double-quotes were used
  inside the expected label.
-->

<testcase name="attributeWith_SingleDoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmark withLabel="'Single &quot;Double&quot; Quote'-Chapter" />
  </assertThat>
</testcase>
```

Quotes in XPath-Expressions

The XML code which defines your test is first converted into Java code. This then evaluates the XPath expressions. So the same restrictions for using quotes in XPath expressions exists as for strings described above.

The following example can be converted into Java code but results in this runtime error: `One of the parameter contains single and double quote. You may use only one kind of quote.`

```
<testcase name="attributeWithXPath_DoubleQuotes_1">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmarks>
      <matchingXPath expr="count(//Title[.='\&quot;Double Quote\&quot;-Chapter']) = 1" />
    </hasBookmarks>
  </assertThat>
</testcase>
```

You can avoid this error by simply writing the XPath expression without double quotes:

```
<testcase name="attributeWithXPath_DoubleQuotes_2">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasBookmarks>
      <matchingXPath expr="count(//Title[contains(., 'Double Quote')]) = 1" />
      <matchingXPath expr="count(//Title[contains(., 'Chapter')]) = 4" />
    </hasBookmarks>
  </assertThat>
</testcase>
```

Quotes in Regular Expressions

Due to the evaluation of regular expressions via the intermediate step of generated Java code, you must escape double quotes:

```
<testcase name="regex_tag_DoubleQuotes">
  <assertThat testDocument="quotes/single-and-doublequotes.pdf">
    <hasText on="FIRST_PAGE">
      <matchingRegex>.*\"double.*</matchingRegex>
    </hasText>
  </assertThat>
</testcase>
```

13.6. 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, thie width and the height:

```
<!-- Defining a clipping area: -->

<inClippingArea upperLeftX=".." upperLeftY=".." width=".." height=".." />
<inClippingArea upperLeftY=".." upperLeftX=".." width=".." height=".." unit=".." />
```

The units are described in chapter 13.7: “Format Units” (p. 140). If no unit is set, PDFUnit uses the unit `MILLIMETER`.

Here's an example:

```
<testcase name="hasText_InClippingArea">
  <assertThat testDocument="content/documentForTextClipping.pdf">
    <hasText on="FIRST_PAGE" >
      <inClippingArea upperLeftX="17.6" upperLeftY="45.8"
                      width="60.0" height="8.8"
                      unit="MILLIMETER"
       >
        <containing>Content on first page.</containing>
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

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

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.

13.7. Format Units

Some tests need width and height. You can use your favourite unit which are predefined constants for the attribute `unit=".."`:

```
<!-- Predefined constants for units: -->

<xxx unit="CENTIMETER"  />
<xxx unit="DPI72"       />
<xxx unit="INCH"        />
<xxx unit="MILLIMETER"  />
<xxx unit="POINTS"      />
```

If you omit the attribute `unit=".."`, the default unit `MILLIMETER` is used.

Example - Field Size

```
<testcase name="hasField_WidthAndHeight">
  <assertThat testDocument="acrofields/notExportableAcrofield.pdf">
    <hasField withName="Title of 'someField'"
              width="450" height="30"
              unit="POINTS"
    />
  </assertThat>
</testcase>
```

Example - Sections of a Page

```
<testcase name="hasTextOnFirstPage_RectangleInInch">
  <assertThat testDocument="content/documentForTextClipping.pdf">
    <hasText on="FIRST_PAGE" >
      <inClippingArea upperLeftX="0.7" upperLeftY="1.8"
                      width="2.4" height="0.4"
                      unit="INCH"
      >
        <containing>Content on first page.</containing>
      </inClippingArea>
    </hasText>
  </assertThat>
</testcase>
```

Example - Page Format

```
<testcase name="hasFormat_HugeFormat">
  <assertThat testDocument="format/physical-map-of-the-world-1999_1117x863mm.pdf">
    <hasFormat width="2448" height="3168" unit="POINTS" />
  </assertThat>
</testcase>
```

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')'.
```

13.8. 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.
```

13.9. 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 to define the date resolution: -->

resolution="DATE"
resolution="DATETIME"
```

You can set a date resolution in the following tags:

```
<!-- Date resolution in tests:  -->

<hasCreationDate           withDate="2013-05-05T09:33:47" resolution="DATETIME" />
<hasCreationDateBefore     withDate="2099-01-01"          resolution="DATE" />
<hasCreationDateAfter      withDate="2013-01-01"          resolution="DATE" />

<hasModificationDate       withDate="2013-05-05T09:33:47" resolution="DATETIME" />
<hasModificationDateBefore withDate="2099-01-01"          resolution="DATE" />
<hasModificationDateAfter  withDate="2013-01-01"          resolution="DATE" />
```

When comparing two PDF documents, date values are always compared using `resolution="DATE"`.

13.10. 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 attribute `defaultNamespace=".."`:

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

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 the tag `<matchingXPath />`:

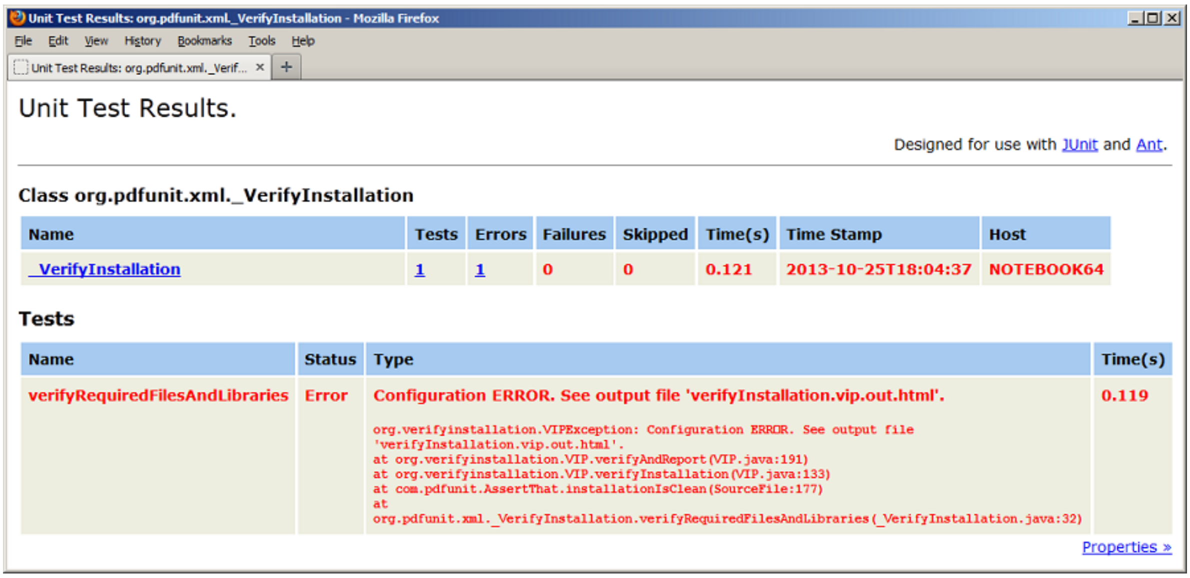
```
<testcase name="hasXMPData_MatchingXPath_WithDefaultNamespace">
  <assertThat testDocument="xmp/metadata-added.pdf">
    <hasXMPData>
      <matchingXPath expr="//foo:format = 'application/pdf'"
                     defaultNamespace="http://purl.org/dc/elements/1.1/"
      />
    </hasXMPData>
 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

13.11. Verify Configuration

As described in section 11.5: “Verifying the Configuration” (p. 127) PDFUnit provides a test to check that all required libraries and files can be found in the classpath:

```
<comment>
  This tests verifies that all required libraries and files are found.
  Additionally it logs some system properties and writes
  everything into both an XML file and an HTML formatted file.
</comment>

<testcase name="verifyRequiredFilesAndLibraries">
  <verifyInstallation verificationFile="verifyInstallation.vip"/>
</testcase>
```

The result is written into an HTML file whose name is part of the error message:


Unit Test Results: org.pdfunit.xml._VerifyInstallation - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Unit Test Results: org.pdfunit.xml._Verif... ×
+

Unit Test Results.

Designed for use with JUnit and Ant.

Class org.pdfunit.xml._VerifyInstallation

Name	Tests	Errors	Failures	Skipped	Time(s)	Time Stamp	Host
_VerifyInstallation	1	1	0	0	0.121	2013-10-25T18:04:37	NOTEBOOK64

Tests

Name	Status	Type	Time(s)
verifyRequiredFilesAndLibraries	Error	Configuration ERROR. See output file 'verifyInstallation.vip.out.html'. org.verifyinstallation.VIPException: Configuration ERROR. See output file 'verifyInstallation.vip.out.html'. at org.verifyinstallation.VIP.verifyAndReport(VIP.java:191) at org.verifyinstallation.VIP.verifyInstallation(VIP.java:133) at com.pdfunit.AssertThat.installationIsClean(SourceFile:177) at org.pdfunit.xml._VerifyInstallation.verifyRequiredFilesAndLibraries(_VerifyInstallation.java:32)	0.119

Properties »

13.12. Version History

August 2011

In November 2011 I presented the existing version of PDFUnit-Java to Jürgen Sieben. His question whether an XML-Dialect is scheduled is the reason why I started with PDFUnit-XML. First I did a proof of concept but I had no time to make substantial progress with this new idea.

2012

The XML API evolved mostly in the second quarter. Then my job and the enhancements to PDFUnit-Java took up all my time.

Release 2013.01

The release name is identical to the name of PDFUnit-Java because both cover the same functionality. In the second quarter of 2013 the german version of the manual was created.

Release 2014.06

Many small improvements led to a new version of PDFUnit-Java. They were also implemented in PDFUnit-XML. And the english version of the manual is available.

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.

13.13. 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. 48) and 3.18: “Layout - in Clipping Areas” (p. 49) 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.

Cross Constraints in XML

The content of some XML tags depend on each other. Such dependencies can not be specified by XML Schema. So a parser can not validate it. It is intended, to provide a Schematron ruleset for cross constraints. Good information about Schematron is available in Wikipedia (http://en.wikipedia.org/wiki/Schematron).

Index