jadice 4.2.

jadice document platform

Michael Grossmann
Carolin Köhler

February 2009

Annotations

Load - Save - Edit

Version 4.2.x

Developer's manual

levigo solutions gmbh
Max-Eyth-Straße 30
D-71088 Holzgerlingen
Telefon: 07031 / 4161-0
Telefax: 07031 / 4161-50
eMail: info@levigo.de

Table of contents

1. General information

This guide presents the technical coherences between jadice® document platform technology and annotations (additional document information).

The documentation is basically limited to areas which are interesting for developers in order to be able to edit annotations with jadice® by programming and it should be understood as an extension to the jadice® document platform documentation.

For a better legibility package names are displayed fully qualified only in footnotes.

An API-reference and an integration documentation of the jadice® document platform are each available as separate documents.

2. Introduction

Annotations in the jadice® document platform technology are considered to be

- comments or
- notations or
- remarks or
- explanations or
- notes or
- pointers in form of arrows or highlighted areas

which the user may add to a document on a certain page.

Annotations are additional information to a document and do not modify the actual document.

These annotations may contain information in form of

- text or
- graphic objects for
  - clarification
  - highlighting or even for
  - masking

and are displayed in their own layer „above" the document.

At the time being jadice® document platform supports:

- IBM ContentManager 7.x and 8.x compatible annotations as MO:DCA structures.
- FileNet annotations as XML structures.
- FileNet P8 annotations as XML structures.
- Wang annotations for mere displaying.

A mixed use of annotation structures is only possible in a limited way.

3. Annotation types

Annotations are document or page information which are supported in different kinds of implementations in the jadice® document platform.

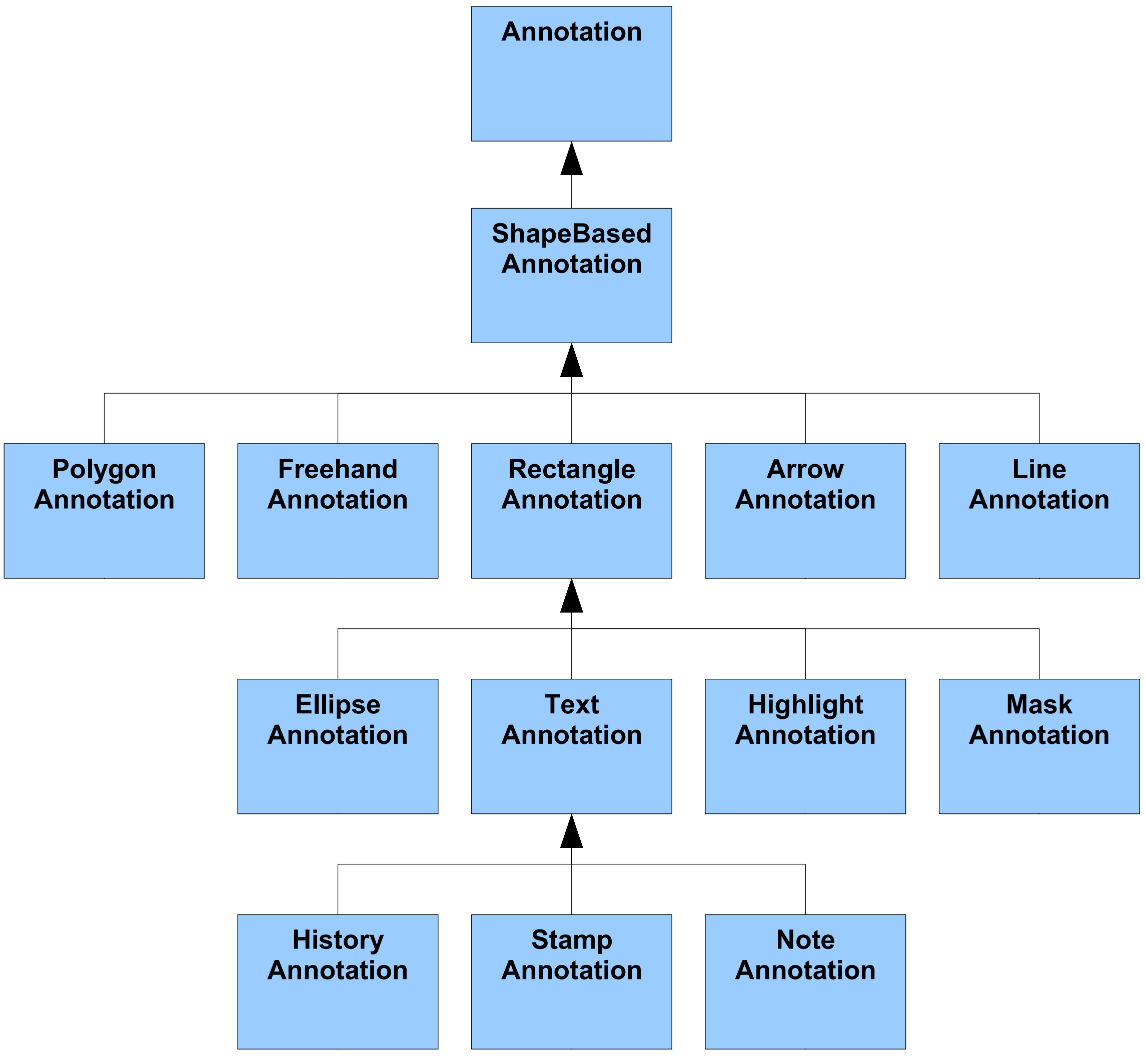

Chart 1 - Annotation architecture – rough survey

Provided types are:

† NOTE:	a post-it / sticky
† HIGHLIGHT:	a highlighting by a filled and transparent rectangle
† MASK:	a masking by a filled, not transparent rectangle

† ARROW:	an arrow
† ELLIPSE:	a not filled ellipse
† RECTANGLE:	a not filled rectangle
† LINE:	a line
† TEXT:	a textual remark with transparent background directly placed on the page
† STAMP:	a „stamp" with transparent background, frame, possible to rotate with text content
† FREEHAND:	a freehand draft
† HISTORY:	similar to note, audit proof, saved data cannot be deleted anymore.
† POLYGON:	a two-dimensional region of coordinate points which are joined by line segments.

In chart 1 you can see a rough survey of the annotations' class-architecture. ShapeBasedAnnotation[1] is an abstract basis class which provides all annotations with the property to have a geometric form (shape[2]). As a direct annotation this class is of no relevance for integrators. However, to anyone who would like to define his own annotations it presents a useful basis.

In contrast to the viewer generation 2.x a logical inheritence came to the fore, due to the displaying and editing properties of single annotation types. With that consistency and the avoiding of redundancies could be granted, further a flexible exchange of the annotation support for different archiving systems or completely self-defined annotations has become possible.

Developers who don't want to leave the annotations' administration to the jadice® document platform, but want to create or change annotations by programming, have to act with caution in order to grant a compatibility to the restrictions and annotation specifications of the underlying Content Management system.

3.1. VisualInfo / ImagePlus Annotations

Using VisualInfo / ImagePlus annotations the annotation type definition (jadice.viewer.annotation.type) in the Jadice.properties configuration file must be set to the value **vi**.

ImagePlus is compatible with the following annotations:

- Highlight
- Mask
- Note

VisualInfo is compatible with the following annotations:

- Arrow
- Ellipse

1 com.levigo.jadice.annotation.ShapeBasedAnnotation
2 java.awt.Shape

- Freehand
- Highlight
- Line
- Note
- Rectangle
- Stamp
- Text

Additional annotation types:

- History
  - This is an audit proof annotation. After the annotation's saving and reloading the contained text cannot be changed or deleted anymore.

    This specialised annotation type can be displayed in a VisualInfo Client, too, but the immutability of the text content is not granted here. The audit proof is only supported within the jadice® document platform.

3.2. FileNet Annotations

Using FileNet annotations, the annotation type definition (jadice.viewer.annotation.type) in the Jadice.properties configration file must be set to **fn**.

FileNet is compatible with the following annotations:

- Arrow
- Freehand
- Highlight
- Note
- Stamp
- Text

Incompatible annotations will not be saved, a warning is displayed in the log-output.

3.3. FileNet P8 Annotationen

Using FileNet P8 anntoations, the annotation type definition (jadice.viewer.annotation.type) in the Jadice.properties configuration file must be set to **fnp8.**

Additionally the parameter

jadice.viewer.default-page-resolution=100

should be set to the resolution value of 100 dpi.

The parameter defines a default resolution for documents that have no specified resolution information.

FileNet annotation sizes and positions are related to the resolution of the belonging document. Therefore a default resolution of 100 dpi has to be made to ensure a compatibility to the FileNet Viewer and other FileNet products.

Otherwise, if this parameter is not defined or set to any other value, annotations may be displayed incorrectly.

FileNet is compatible with the following annotations

- Highlight
- Note
- Arrow
- Ellipse
- Freehand
- Line
- Rectangle
- Stamp
- Text
- Polygon

Incompatible annotations will be not saved, a warning is displayed in the log-output.

4. Permission

Annotations may get defined access permissions which may affect the annotation's behaviour at the displaying in the viewer and at the saving/writing.

In order to set/check permissions the AnnotationPermission[3] class is used. Here are all permissions and methods for the searching of the annotation's permission defined.

The following permissions and combinations from these are possible:

- PERMISSION_NONE

  no permission, annotation is not displayed and can't be changed, deleted or saved.
- PERMISSION_READ

  Annotation is displayed in the viewer.
- PERMISSION_WRITE

  Annotation may be saved.
- PERMISSION_CHANGE

  Annotation may be changed (not deleted !),

  Precondition: PERMISSION_READ must be set.
- PERMISSION_DELETE

  Annotation may be deleted, but not changed.

  Precondition: PERMISSION_READ must be set.

The class AnnotationPermission offers the following methods for permission check:

- canWrite(Annotation anno)

  true = annotation can be saved

  false = saving impossible

- canRead(Annotation anno)

  true = annotation is displayed in the viewer.

  false = is not displayed in the viewer.

- canChange(Annotation anno)

  true = changes on annotation possible (position, size, properties)

  false = no changes on annotation possible, selection and changing of properties impossible.

- canDelete(Annotation anno)

3 com.levigo.jadice.annotation.AnnotationPermission

true = annotation can be deleted

false = annotation cannot be deleted.

Permissions may also be combined, e.g.:

PERMISSION_READ + PERMISSION_WRITE + PERMISSION_CHANGE + PERMISSION_DELETE

The permission above corresponds to the default permission of a newly created annotation.

4.1. Set permission of an annotation

After the complete annotations' loading changes on the permission may be done. For this each annotation to be changed has to be retrieved out of the belonging AnnotationPageSegment instance (see also chapter 7):

```
Document document = myDocument;
// for each page...
for  (int i = 0; i < document.getPageCount(); i++) {
  //...search the corresponding AnnotationPageSegment.
  AnnotationPageSegment aps =
    (AnnotationPageSegment) document.getPage(i)
    .getPageSegment(document.getLayer(
    AnnotationPageSegment.DEFAULT_LAYER_NAME));
  //If it exists, ...
  if (aps != null) {
    //...iterate by each annotation...
    for (Iterator it = aps.getAnnotations().iterator();
      it.hasNext();) {
      Annotation annotation = (Annotation)it.next();
      //... and change the permission.
      int  permission =
        AnnotationPermission.PERMISSION_NONE;
       annotation.setPermission(permission);
    }
  }
}
```

Code example 1 - Retrieve annotations of an AnnotationPageSement and change the permission.

4.2. Set annotation permission during loading process

Annotation permissions may also be set during the loading process. Therefore an approbate instance of AnnotationPermissionEstablisher[4] which performs the permissions' setting has to be defined and must be accessible within the annotation's loading process.

For this the AnnotationPermissionEstablisher to be applied has to be defined in the annotation configuration file AnnotationInit.properties.

4 com.levigo.jadice.annotation.AnnotationPermissionEstablisher

For VisualInfo / ImagePlus annotations use the parameter:

`vi.annotation.permission-apply-class=MyClass` `or`

`vi.annotation.permission-apply-class=packageName.MyClass`

For FileNet annotations:

`fn.annotation.permission-apply-class=MyClass` `or`

`fn.annotation.permission-apply-class=packageName.MyClass`

For FileNet P8 annotations:

`fnp8.annotation.permission-apply-class=MyClass` `or`

`fnp8.annotation.permission-apply-class=packageName.MyClass`

The AnnotationPermissionEstablisher class must be realised and provided by the integrator. For this the class must implement the interface AnnotationPermissionEstablisher[5].

After the annotations' loading the method **`applyPermission(Collection)`** is called. The collection contains all loaded annotations, the permission may then be set accordingly.

```
public class MyClass implements
  AnnotationPermissionEstablisher {
  /**
   * @see com.levigo.jadice.annotation
     .AnnotationPermissionApply#setPermission
     (java.util.Collection)
   */
  public void applyPermission(Collection annotations) {
    for (Iterator iter = annotations.iterator();
      iter.hasNext(); ) {
      Annotation anno = (Annotation) iter.next();
      // set permission read, change and write
      int permission =
        AnnotationPermission.PERMISSION_CHANGE
        + AnnotationPermission.PERMISSION_READ
        + AnnotationPermission.PERMISSION_WRITE;
      anno.setPermission(permission);
    }
  }
}
```

Code example 2 / Implementation of the interface AnnotationPermissionEstablisher

5 com.levigo.jadice.annotation.AnnotationPermissionEstablisher

5. Loading

In the following paragraph it is demonstrated by example how annotations may be loaded. To keep it concise at this point, it will not be dwelled on the loading process of the actual image document, but only on the loading of annotations.

The Loader[6] is jadice®'s central class for all loading processes, in this very example for the loading of annotations. A more detailed description of the loader and its use can be looked up in the jadice® integration documentation or in the jadice® API-reference.

At first an instance of the loader is created. If no already existing document[7] is passed to the loader, the loader creates a new document which is „filled" during the loading process and may be passed for displaying to the viewer.

In the next step it is tried to find a corresponding annotation file to the provided image file „file2Load". Annotation files use to have the same name like the image document and end with the suffix „.T_L" for VisualInfo / ImagePlus annotations or „xml" for FileNet annotations. If such a file exists, it is used for the loading of annotations. Here it should be pointed out that annotation data normally are not provided as a file but as a stream from an archive or similar.

Format information describe the format in which a document is available. If no format information is given to the loader in the „loadDocument" method, the loader tries to identify the format itself. Since annotation data has no specified „magic words" in its header data, it is advised to load annotation solely with its annotation format information. For example, since ImagePlus annotations are MO:DCA structures, the loading process should be accompanied by an ImagePlusAnnotationFormatInfo[8]. Otherwise the annotation data could be recognized by mistake as a MO:DCA document. In such a case annotations would be loaded as an independent document, would not be displayed in the usual appearance and could not be changed by the user.

Note:

The loading and saving of annotations must be done explicitly.

Note:

The loading of annotations must always be accompanied by their annotation format info class.

Exception:

FileOpener[9] is a utility class of the jadice® package. It performs automatically loading processes of documents and respective annotations.

5.1. Loading of VisualInfo / ImagePlus annotations

The following example describes how ImagePlus compatible annotations may be loaded.

```
File file2Load = new File("Myimage.tif");
```

6 com.levigo.jadice.docs.resource.Loader
7 com.levigo.jadice.docs.Document
8 com.levigo.jadice.formats.annoiplus.ImagePlusAnnotationFormatInfo
9 com.levigo.jadice.util.FileOpener

```
Loader loader = new Loader();
// load document...
int lastDot = file2Load.lastIndexOf(".");
if (lastDot > 0) {
  // try to find annotation file
  String annoFileName = file2Load.substring(0, lastDot);
  // Default Annotation Extension: „.T_L“
  File annoFile = new File(annoFileName + ".T_L");
  // load annotations, if file exists
  if (annoFile.exists())
    loader.loadDocument(
      new FileInputStream(annoFile),
      new ImagePlusAnnotationFormatInfo(),
      0);
}
```

Code example 3- Load VisualInfo / ImagePlus Annotations

5.2. Loading of FileNet annotations

The following example describes how FileNet compatible annotations may be loaded.

Note:

The method **FileNetAnnotationFile.convertToUTF8(InputStream)** serves as workaround, since the at the time of the implementation current FileNet Image Services Resource Adapter (Version 3.0a) interface provides a NOT compliant UTF-8 data stream. Already correct UTF-8 XML data should not be edited by this method, otherwise a correct data processing cannot be granted.

```
File file2Load = new File("Myimage.tif");
Loader loader = new Loader();
// load document...
int lastDot = file2Load.lastIndexOf(".");
if (lastDot > 0) {
  // try to find annotation file
  String annoFileName = file2Load.substring(0, lastDot);
  // Default Annotation Extension: „.xml“
  File annoFile = new File(annoFileName + ".xml");
  // load annotations, if file exists
  if (annoFile.exists())
    loader.loadDocument(
      FileNetAnnotationFile.convertToUTF8(

      new FileInputStream(annoFile))
      new FileNetAnnotationFormatInfo(),
      0);
}
```

Code example 4- Load FileNet annotations

5.3. Loading of FileNet P8 annotations

The following example shows, how FileNet P8 annotations may be loaded. Annotations are loaded separately from the archive. Each annotation has its own data stream. With the FileNetP8AnnotationInputXMLParser-Class several annotation data streams are united and written in a single data stream which then will be loaded by the FileNetP8AnnotationFile-Class.

```
// Create Input-handler
FileNetP8AnnotationInputXMLParser input =
  new FileNetP8AnnotationInputXMLParser();

// summarize all annotations
input.addAnnotationXMLStream(>Annotation 1 data stream<);
input.addAnnotationXMLStream(>Annotation 2 data stream<);
input.addAnnotationXMLStream(>Annotation 3 data stream<);
input.addAnnotationXMLStream(>Annotation x data stream<);

// Create data stream
ByteArrayOutputStream bos = new ByteArrayOutputStream();

// for a single- and multipaged document (one data stream)
// write all annotations
input.write(bos);

// for a composed document (multiple data streams) only
// the annotations for the designated page are saved here
input.write(bos, >Seitenindex des Dokuments<);

// retrieve data
byte[] annotationData = bos.toByteArray();
bos.close();

// create data stream for the loading process
ByteArrayInputStream bis =
  new ByteArrayInputStream(annotationData);

// load annotations
FileNetP8AnnotationFile file =
  new FileNetP8AnnotationFile(>jadice® Dokument<);
file.load(bis, document
  .getLayer(AnnotationPageSegment.DEFAULT_LAYER_NAME), 0,
  null);
// alternative: loading via loader-class
Loader loader = new Loader();
loader.loadDocument(bis,
  new FileNetP8AnnotationFormatInfo(), 0);
```

Code example 5- loading FileNet P8 annotations

6. Saving

Similar to the fact, that an approbate format information instance supports loading processes into certain formats, there are classes which support saving processes of certain formats (*FormatName*File). The naming of these classes follows a predetermined convention.

Example:

*FormatName*FormatInfo -> TIFFFormatInfo[10]

*FormatName*File -> TIFFFile[11]

For ImagePlus compatible annotations this class is accordingly called ImagePlusAnnotationFile[12], for FileNet compatible annotations this class is called FileNetAnnotationFile[13].

The saving is described here by using the example of the ImagePlusAnnotationFile class.

First an instance of ImagePlusAnnotationFile is created. In order to get access on the annotations to be saved, the document whose annotations are to be saved is passed to the ImagePlusAnnotationFile in the constructor.

In the same way as the loader provides different loading methods, each FormatNameFile instance allows a qualified saving (e.g. only certain pages or similar). For more detailed information see the jadice® integration documentation.

In code example 6 the simplest method has been chosen in order to save all annotations of a document. An OutputStream in which the annotation information is to be saved is simply passed to the ImagePlusAnnotationFile instance.

In code example 8 the saving of FileNet annotations is shown. Here attention should be payed to some points.

Note:

The loading and saving of annotations must be done explicitly.

In the following paragraphs a simple example shows how ImagePlus and FileNet compatible annotations may be saved in a file.

6.1. Saving of VisualInfo / ImagePlus annotations

VisualInfo and ImagePlus annotations allow saving additional information to the annotations.

Bear in mind that annotations with additionally saved information may be displayed in each VisualInfo client. However, as soon as the annotations are saved again by a VisualInfo client, the additional information is lost.

10 com.levigo.jadice.formats.tiff.TIFFFormatInfo
11 com.levigo.jadice.formats.tiff.TIFFFile
12 com.levigo.jadice.formats.annoiplus.ImagePlusAnnotationFile
13 com.levigo.jadice.formats.annofilenet.FileNetAnnotationFile

In order to activate the information's saving, the following parameter must be set accordingly in the AnnotationInit.properties file:

vi.annotation.save-additional-info=<value>

value: true = information is saved

false = information is not saved

The default setting is *false*.

```
// similar to loading: ImagePlusAnnotationFormatInfo ->
// take ImagePlusAnnotationFile
ImagePlusAnnotationFile annoFile =
  new ImagePlusAnnotationFile(myViewer.getDocument());

try {
  annoFile.save(new FileOutputStream("Myimage.T_L"));
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

Code example 6- Save annotations

6.2. Saving of FileNet annotations

When saving FileNet annotations additional FileNet information must be indicated, so that the annotations may be saved correctly into the FileNet archive. This information is changed with each saving process by FileNet. In order to maintain a consistent state between the archive and the displayed annotations, the saving of FileNet annotations must be done in three steps:

- saving of annotations in the archive.
- deleting of annotations in the jadice® document to avoid a double appearance later on in the document.
- reloading of annotations by FileNet with updated FileNet information (e.g. time stamp, annotation identification number).

Thus after the saving all existing annotations have to be removed from the AnnotationPageSegment (see code example 7).

```
// get annotation layer
DocumentLayer annoLayer = document.getLayer
  (AnnotationPageSegment.DEFAULT_LAYER_NAME);
// Edit all pages
for (Iterator it = document.getPages().iterator();
  it.hasNext();) {
  Page aPage = (Page) it.next();
  // get AnnotationPageSegment for current pages
  AnnotationPageSegment aps = (AnnotationPageSegment)
    aPage.getPageSegment(annoLayer);
  if (aps != null) {
    // Deselect all annotations
    aps.deselectAllAnnotations(aps.getAnnotations());
    // Remove annotations
    aps.getAnnotations().clear();
    aps.getDeletedAnnotations().clear();
  }
}
```

Code example 7- Remove annotations from AnnotationPageSegment

Having successfully deleted the annotations in the AnnotationPageSegment, the annotations updated by FileNet must be reloaded now. The loading process of annotations from FileNet is described in chapter 5.2.

For a correct saving of annotations into the FileNet archive additional FileNet-specific information must be indicated.

The following information is required which is passed for the creation of a FileNetAnnotationFile object in its constructor:

FileNet archive definition:

The FileNet archive definition is composed of the following parameters:

library = FileNet Library Name (Default: „DefaultIMS")

domain = FileNet Domain Name (Default: „Imaging")

organisation = FileNet Organisation Name (Default: „FileNet")

FileNet document ID:

ID of the document on which the annotations are to be saved:

docId = FileNet document ID

FileNet client access permission:

Access permission of the client:

clientPermission = client access permission, here are possible „none", „change", „admin".

FileNet access permission:

Access permission of the client (user or group):

permissionTypeAppend,

permissionTypeRead,

permissionTypeWrite =

access permission, here are possible „user", „group".

FileNet user permission:

User permission of the client (user or group):

permissionNameAppend,

permissionNameRead,

permissionNameWrite =

user permission, here a valid FileNet user / FileNet user group must be indicated.

Default users are „(ANYONE)" and „(NONE)".

Permission control:

If the useDefaultPermission flag is set to „false", then already existing annotations are stored with the available (=loaded before) permission. Newly created annotations are saved with the permission defined in the constructor.

If the flag is set to „true", then all annotations are saved with the permission defined in the constructor.

```
// FileNet archive definition
String library = "DefaultIMS";
String domain = "Imaging";
String organisation = "FileNet";
// FileNet document ID
String docId = "100000";
// client access permission
String clientPermission = "change";
// access permission
String permissionTypeAppend = "user";
String permissionTypeRead = "user";
String permissionTypeWrite = "user";
// user permission
String permissionNameAppend = "(ANYONE)";
String permissionNameRead = "(ANYONE)";
String permissionNameWrite = "(ANYONE)";
// Flag for controling of access permission.
// false = with already existing annotations the available
// access permission is used, newly created annotations
// get assigned the access permission defined above.
// true = all annotations get assigned the access
// permission defined above.
boolean useDefaultPermission = false;
FileNetAnnotationFile file =
  new FileNetAnnotationFile(
  myViewer.getDocument(),
  library,
  domain,
  organisation,
  docId,
  clientPermission,
  permissionTypeAppend,
  permissionTypeRead,
  permissionTypeWrite,
  permissionNameAppend,
  permissionNameRead,
  permissionNameWrite,
  useDefaultPermission);
try {
  annoFile.save(new FileOutputStream("Myimage.xml"));
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}
```

Code example 8- Save annotations

6.3. Saving of FileNet P8 annotations

Similar to the saving mechanism of FileNet annotations, FileNet P8 have to be saved in a specified sequence of steps in order to maintain a consistent state between the archive and the displayed annotations:

- saving of annotations in the archive.
- deleting of annotations in the jadice® document to avoid a double appearance later on in the document.
- reloading of annotations from the archive.

Having successfully deleted the annotations in the AnnotationPageSegment, the annotations updated by FileNet must be reloaded now. The loading process of annotations from FileNet P8 is described in chapter 5.3.

There are 4 states which annotations could appear in that decide on the further saving process.

- **Unmodified**

  Unmodified annotations should not be saved in the archive.
- **Modified**

  Modified annotations must be saved in the archive, the time stamp will be updated. (concerns attribute F MODIFYDATE).
- **Added**

  Added annotations must be saved in the archive, the time stamp will be set (attributes F ENTRYDATE and F MODIFYDATE). The following attributes must be set additionally:

  - ID (attribute F_ID and F_ANNOTATEDID)

    New annotation-ID, must be retrieved from the archive.
  - Page number (attribute F_PAGENUMBER)

    With composed documents the corresponding page number has to be set additionally, the standard value is 1.
- **Deleted**

  Deleted annotations must be removed from the archive. The annotations are loaded using their ID and deleted afterwards.

One can get access to the different annotation states via an instance of the FileNetP8AnnotationOutputXMLParser[14] Class, or alternatively this is also possible via an instance of the FileNetP8AnnotationFile[15] Class.

```
// create FileNetP8AnnotationFile
FileNetP8AnnotationFile file =
  new FileNetP8AnnotationFile(>jadice document<);

// saving annotations
```

14 com.levigo.jadice.formats.annofilenetp8.FileNetP8AnnotationOutputXMLParser
15 com.levigo.jadice.formats.annofilenetp8.FileNetP8AnnotationFile

```
ByteArrayOutputStream os = new ByteArrayOutputStream();
file.save(os);
byte[] annotationData = os.toByteArray();
os.close();

// create Output-handler
ByteArrayInputStream is = new
ByteArrayInputStream(annotationData);
FileNetP8AnnotationOutputXMLParser output =
    new FileNetP8AnnotationOutputXMLParser(is);

org.w3c.dom.Document[] annotations = null;

//##########################################################
// Unmodified annotations
//##########################################################

annotations = output.getUnmodifiedAnnotations();
// alternative:
annotations = file.getUnmodifiedAnnotations(null);

// handle all annotations
for (int i = 0; i < annotations.length; i++) {
  org.w3c.dom.Document anno = annotations[i];
  // no saving necessary, only debug-output here
  // retrieve annotations-ID
  String id =
    FileNetP8AnnotationXMLUtils.getAnnotationID(anno);
  System.out.println("Anno " + id + " unmodified !");
}

//##########################################################
// Modified annotations
//##########################################################

annotations = output.getModifiedAnnotations();
// alternative:
annotations = file.getModifiedAnnotations(null);

// handle all annotations
for (int i = 0; i < annotations.length; i++) {
  org.w3c.dom.Document anno = annotations[i];
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  FileNetP8AnnotationXMLUtils.write(bos, anno);
  byte[] data = bos.toByteArray();
  bos.close();
  // retrieve annotations-ID
  String id =
    FileNetP8AnnotationXMLUtils.getAnnotationID(anno);
  // create new datastream for saving to the archive
    ByteArrayInputStream bis = new
    ByteArrayInputStream(data);

  // here the annotation has to be loaded from the archive
  // with the ID and the data stream has to written over
  // the annotation´s object.
  // >>>>>> access to the archive

  System.out.println("Anno " + id + " updated !");
}
```

```
//##########################################################
// Added annotations
//##########################################################

annotations = output.getAddedAnnotations();
// alternative:
annotations = file.getAddedAnnotations(null);

// handle all annotations
for (int i = 0; i < annotations.length; i++) {
  org.w3c.dom.Document anno = annotations[i];
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  FileNetP8AnnotationXMLUtils.write(bos, anno);
  byte[] data = bos.toByteArray();
  bos.close();

  // here a new annotation´s object has to be created in
  // the archive and the ID has to be retrieved.
  // >>>>>> access to the archive

  String id = "New ID";
  // set new ID
  FileNetP8AnnotationXMLUtils.setAnnotationID(anno, id);
  // possibly set page number for multi-documents
  FileNetP8AnnotationXMLUtils.setPageNumber(anno,
    >page number<);
  // create new data stream for saving to the archive
  ByteArrayInputStream bis = new
    ByteArrayInputStream(data);

  // here the data stream has to be written over the
  // annotation´s object.
  // >>>>>> access to the archive

  System.out.println("Anno " + id + " added !");
}

//##########################################################
// Deleted annoations
//##########################################################

annotations = output.getDeletedAnnotations();
// alternative:
annotations = file.getDeletedAnnotations(null);

// handle all annotations
for (int i = 0; i < annotations.length; i++) {
  org.w3c.dom.Document anno = annotations[i];
  // retrieve annotation´s ID
  String id =
    FileNetP8AnnotationXMLUtils.getAnnotationID(anno);

  // here the annotation has to be loaded form the archive
  // with the ID and than deleted.
  // >>>>>> access to the archive

  System.out.println("Anno " + id + " deleted !");
}

// remove all annotations from the document
file.removeAllAnnotationsFromDocument(null);
```

Code example 9– Save annotations

7. Access

This paragraph explains how integrators get direct access on annotations belonging to pages.

In order to explain the access, first some preliminaries about the jadice® document and page model are necessary.

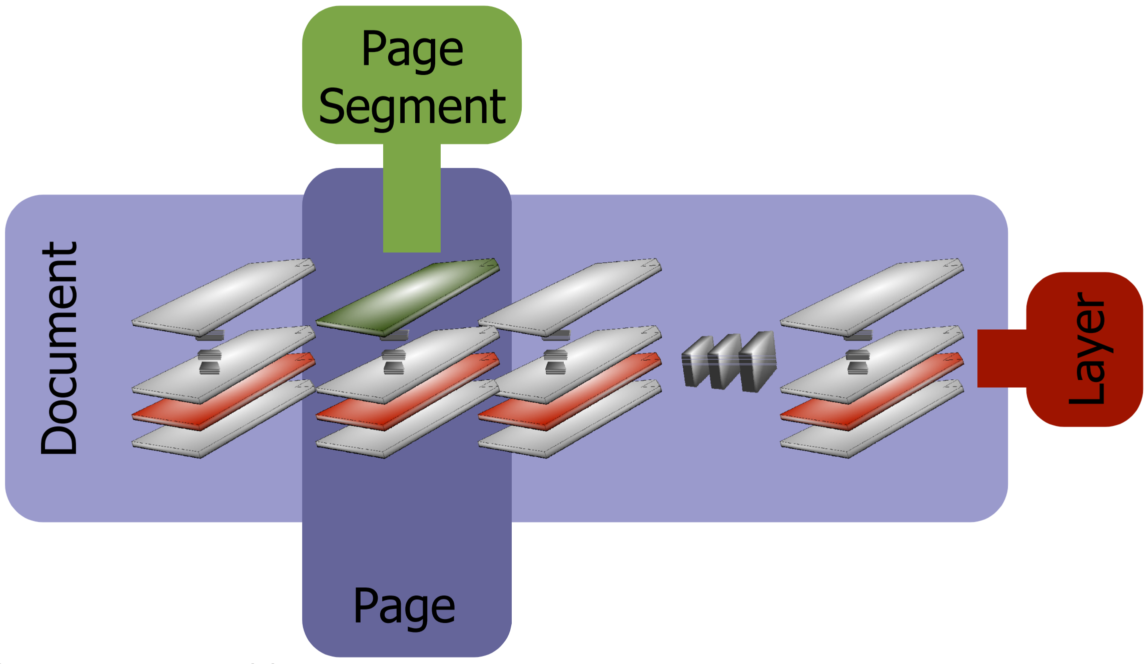

Chart 2 Document model

A document consists of one or more pages which are considered to be a unit in the application. A page may consist of multiple layers which are, however, visualised as a unit. These layers are called displaying levels or simply layers. In these layers page segments are provided which represent data from different data sources. Example: one layer is a letter-head, another one is a textual letter content.

For a detailed description of the document model see the jadice® integration documentation or the jadice® API reference.

Since annotations are not part of a document, but an additional information source, annotations are administrated and displayed in a page segment of their own, the so-called AnnotationPageSegment[16].

7.1. ...on the AnnotationPageSegment

As described in the paragraph before, AnnotationPageSegment is the central class to get direct references on annotations currently belonging to a page.

16 com.levigo.jadice.annotation.AnnotationPageSegment

In the following a code example is provided in which access on the AnnotationPageSegment of a page is described step by step.

```
public AnnotationPageSegment getAnnotationPageSegment(Page
  page) {
  AnnotationPageSegment aps = null;
  // search displaying level in which annotations are
  // provided
  Document doc = page.getParentDocument();
  DocumentLayer layer =
    doc.getLayer(AnnotationPageSegment.DEFAULT_LAYER_NAME)
    ;

  // if the layer is not provided, an
  // AnnotationPageSegment does not exist
  if (layer != null) {
    // get the layer's page segment
    aps = (AnnotationPageSegment)
      page.getPageSegment(layer);
  }
  return aps;
}
```

Code example 10- Reference on AnnotationPageSegment

In order to get a reference on a certain page segment, first it must be detected in which layer the searched for page segment is provided. In general there is only one layer for annotations.

A reference on the annotation default layer is got by the layer access methods of the document.

If such a layer is not provided, the whole document has got no annotations at all and thus no AnnotationPageSegment exists in this document.

Otherwise the page may now be queried for the segment in this layer by using the annotation layer. This segment is the searched for instance of AnnotationsPageSegment.

7.2. ...on annotations of a page

An instance of the class AnnotationPageSegment offers by using simple „getter"-methods access on all annotations of a page respectively on all selected annotations of a page.

```
// Access method from code example 8
AnnotationPageSegment aps =
  getAnnotationPageSegment(aPage);

// all annotations of a page
Collection allAnnotations = aps.getAnnotations();
// all selected annotations of a page
Collection allSelectedAnnotations =
  aps.getSelectedAnnotations();
```

Code example 11- Reference on all or all selected annotations of a page

8. Changing of annotations

For a better understanding of the following chapters the connection between the document and the device coordinate system is shortly explained here.

As shown in chart 2 jadice® document model (compare also the jadice® integration documentation), the pages contained in documents may get their content from multiple data streams or data formats, but also the pages themselves may consist in their displaying of data from different data streams. Since these image data may be of different format and resolution, the viewer maintains internally a document coordinate system and transforms only for displaying into the respective device coordinates. Accordingly page segments, like the AnnotationPageSegment as well, keep their data in document coordinates.

Consequently position and size information as well as changes on annotations are to be understood in document coordinates. For an easy conversion of device to document and document to device coordinates the class RenderContext[17] offers the according affine transformations. More details about the class RenderContext may be read in chapter 10 or in the integration documentation of the jadice® document platform. Application examples of the according affine transformations are provided in the following chapters.

8.1. Add and delete

The link between annotations and the document's page is the AnnotationPageSegment. Hence adding and deleting happens by using the AnnotationPageSegment.

```
// Access method from code example 8
AnnotationPageSegment aps =
  getAnnotationPageSegment(aPage);

if (aps != null){
  // delete a particuler annotation
  aps.deleteAnnotation(anAnnotation);

  // delete all selected annotations
  aps.deleteSelectedAnnotations();
}
```

Code example 12- Deleting of annotations

If no AnnotationPageSegment exists to a given page, this page has not got any annotations. So a deleting procedure is obsolete. Otherwise the AnnotationPageSegment offers two methods for deleting. By the „deleteAnnotation“ method a particular annotation may be removed in a qualified way; by the „deleteSelectedAnnotations“ all selected annotations are removed.

If an annotation is to be added to a page, it must be provided that an AnnotationPageSegment exists for this page.

17 com.levigo.jadice.docs.RenderContext

In code example 10 it has been described how to get access on the AnnotationPageSegment. But if the document does not contain any annotations yet, it may happen that the document does not contain any annotation layer or the page does not contain an AnnotationPageSegment yet. In the following example the access method on the AnnotationPageSegment from code example 10 has been thus extended that now an according layer or AnnotationPage-Segment is created.

```
public AnnotationPageSegment getAnnotationPageSegment(Page
  page) {
  AnnotationPageSegment aps = null;

  // search displaying level in which annotations are
  // provided
  Document doc = page.getParentDocument();
  DocumentLayer layer =
    doc.getLayer(AnnotationPageSegment.DEFAULT_LAYER_NAME)
    ;

  // if the layer does not exist, create it
  if (layer == null)
    layer = doc.addLayer(
      AnnotationPageSegment.DEFAULT_LAYER_NAME,
      DocumentLayer.TOP);

  // get page segment of the layer
  aps = (AnnotationPageSegment)
    page.getPageSegment(layer);

  // if the page segment does not exist, create it
  if (null == aps) {
    aps = new AnnotationPageSegment(page);
    page.addPageSegment(aps, layer);
  }

  return aps;
}
```

Code example 13- Reference on AnnotationPageSegment, extended

By using the access method from code example 11 you may now add annotations and be sure that a corresponding layer exists in the document and that the page owns an AnnotationPageSegment.

The actual adding is now simply done by using the method „addAnnotation" of the AnnotationPageSegment.

Note:

Adding and deleting of annotations propagates a „PageModified" event to all registered DocumentListeners[18] of the document.

18 com.levigo.jadice.docs.DocumentListener

8.2. Size and position

All annotations descend from the basis class Annotation[19]. This class provides for size and position changes each a corresponding method. So you may use these methods independently of the annotation type. The only exception at the time being is the StampAnnotation[20]: This annotation defines its size independently due to its properties.

The methods in detail:

- **Annotation.setLocation(Point)** – for position changing
- **Annotation.setSize(Dimension)**- for size changing.

Note:

The original position of the arrow annotation always refers to the arrowhead.

Note:

All size and position information are to be understood in document coordinates.

Note:

After changes on annotations integrators should have the viewer re-rendered. Example: myViewer.repaint();

Note:

Please keep in mind: Direct changes on annotations DO NOT propagate a „PageModified" event to registered DocumentListeners[21] of the document!

8.3. Other changes

According to their type the single annotations have got different specifications, e.g. foreground colour, background colour, text, etc. Detailed information about each annotation type are in the jadice® API.

Integrators are free to change attributes of annotations. However, possible compatibility limitations have to be paid attention to.

Note:

After changes on annotations integrators should have the viewer re-rendered. Example: myViewer.repaint();

This „repaint" demand will happen automatically in a future version of the jadice® and will not have to be started manually.

Note:

Please keep in mind: Changes directly on annotations DO NOT propagate a „PageModified" event to all registered DocumentListeners[22] of the document!

19 com.lavigo.jadice.annotation.Annotation
20 com.levigo.jadice.annotation.StampAnnotation
21 com.levigo.jadice.docs.DocumentListener
22 com.levigo.jadice.docs.DocumentListener

8.4. Events of annotation changes

In order to be able to react on annotation changes, it is possible to set a class which implements the AnnotationListener[23] interface by the static addAnnotationListener method of the class AnnotationEventCaster[24].

At each change of annotation properties or states the annotationChanged method of the listener class is called and an AnnotationEvent class is passed which contains information about the respective change.

Information from the AnnotationEvent[25] class:

- type of event
- concerned annotation class
- old and new value of property which has been changed

The propagation of change events may be suppressed for annotations. The behaviour is controled by the **setDoFireAnnotationEvents** method of the class Annotation.

23 com.levigo.jadice.annotation.AnnotationListener
24 com.levigo.jadice.annotation.AnnotationEventcaster
25 com.levigo.jadice.annotation.AnnotationEvent

9. Create annotations

This chapter concentrates on creating annotations by programming. For this purpose the single steps are explained by using an example in which a rectangle annotation is created on a random position with a random size. Further on it is referred to chapter 8.1 Add and delete.

In code example 14 first a random rectangle is created. In the following this rectangle determines position and size of the RectangleAnnotation[26] to be newly created. Since size and position information are to be set in document coordinates, the rectangle must be transformed in document coordinates.

```
// create random rectangle
Rectangle annoBoundsDev = new Rectangle(
   (int)Math.round(Math.random()*300),
   (int)Math.round(Math.random()*300),
   (int)Math.round(Math.random()*500),
   (int)Math.round(Math.random()*500));

// transform rectangle in document coordinates
Rectangle annoBoundsDoc =
  getDoc2DevTransformationForAnnos()
  .createTransformedShape(annoBoundsDev).getBounds();

// create annotation
RectangleAnnotation newRectangleAnnotation =
  new RectangleAnnotation(
    annoBoundsDoc.x,annoBoundsDoc.y,
    annoBoundsDoc.width,annoBoundsDoc.height);
// set colour to red
newRectangleAnnotation.setForegroundColor(Color.red);

// add annotation to AnnotationPageSegment
// getAnnotationPageSegment() from code example 11
getAnnotationPageSegment().addAnnotation(
  newRectangleAnnotation);
```

Code example 14- Create rectangle annotation

The transformation into document coordinates happens in two steps. First a corresponding transformation is created in the **„createDoc2DevTransformationForAnnos“** method, then it is transformed by using the „createTransformedShape“ method of the class AffineTransform[27]. The functionality of the method **„createDoc2DevTransformationForAnnos“** is described lateron in this chapter. More details about affine transformations are provided in the Java 2 Platform API Specification.

Size and position of the new annotation being now defined, a new instance of a RectangleAnnotation is created. For this the only public constructor is used which is provided by RectangleAnnotation. For a better differentiation between RectangleAnnotations created by programming and created by jadice®, a red foreground colour has been set here to the annotation. RectangleAnnotations created by jadice® are yellow by default.

26 com.levigo.jadice.annotation.RectangleAnnotation
27 java.awt.geom.AffineTransform

In the last step the newly created annotation is added to the page. For this a reference on the AnnotationPageSegment has been created by using code example 13. As described in chapter 8.1 the annotation is added to the page.

The question about the affine transformation remains open. As already mentioned, instances of the class RenderContext[28] provide affine transformations which allow a simple conversion between device and document coordinates. An instance of the RenderContext with the current settings of zoom, rotation or similar is always provided by the viewer.

Page segments do not create images during the displaying process, but they render their data information in consideration of the set zoom and rotation factor into a given device context. Due to this position and size information of annotations are always to be understood with rotation 0° and zoom 100.

In code example 15 first the viewer's current RenderContext is cloned. Thus the required instance of a RenderContext may be set on zoom 100 and rotation 0° without changing the document displaying of the viewer.

```java
public AffineTransform getDoc2DevTransformationForAnnos()
{
  RenderContext rc = myViewer.getRenderContext().clone();
  // annotations are always created with zoom 100 and
  // rotation 0.
  // Zoom and rotation are first used during the
  // rendering.
  rc.setRotation(0);
  rc.setZoomFactor(100);

  // getInverseTransform()  -> Device to document
  // coordinates transformation
  return rc.getInverseTransform();
}
```

Code example 15- Affine transformation

The central methods of the RenderContext for the transformation between document and device coordinates are the following:

- **RenderContext.getAffineTransform()** - provides a transformation from document into device coordinates
- **RenderContext.getInverseTransform()** - provides a transformation from device into document coordinates

Newly created annotations require position and size information in document coordinates. To convert this information the inverse transformation of the cloned instance of the RenderContext is returned.

28 com.levigo.jadice.docs.RenderContext

9.1. Create/change by using the AnnotationCreator Interface

In order to enable the user to create interactively own annotations or annotations with different, predefined properties, the interface AnnotationCreator[29] may be used. A class created by the integrator and complying with this interface may be registered by using the static **setAnnotationCreator** method of the class Annotation.

If a user creates interactively an annotation by using the mouse, the createAnnotation method of the registered instance of the interface AnnotationCreator is called. The selected annotation type and the position of the annotation to be created are passed as parameters. The default annotation types are available as constants of the class Annotation. More details about this are in the API description of the class Annotation.

In the createAnnotation method an annotation can now be created and adapted with the desired properties (see code example 14). If no annotation is returned, the default annotation corresponding to the annotation type is created.

In the following example a text annotation with the size of 150x50 pixel and green background colour is created.

```java
public Annotation createAnnotation(int annotationMode,
  Point atPoint) {
  Annotation anno = null;
  if (annotationMode == Annotation.TEXT) {
    RenderContext rcDef = new RenderContext();
    Dimension size = new Dimension(
      rcDef.deviceToBase(150), rcDef.deviceToBase(50));
    TextAnnotation annoText = new
      TextAnnotation(atPoint.x, atPoint.y,
          size.width, size.height, "New Anno");
    annoText.setBackgroundColor(Color.green);
    anno = annoText;
  }
  return anno;
}
```

Code example 16- AnnotationCreator

29 com.levigo.jadice.annotation.AnnotationCreator

10. RenderContext

At some points of this document, particularly in the following chapter, it is referred to the class RenderContext[30].

For a better understanding the properties and duties of the class RenderContext are shortly presented here.

In contrast to the viewer Vs.2.x, images are not created for the displaying of pages, but the pages and consequently all of their segments render themselves automatically into given device contexts, e.g. monitor, printer, etc.

A rendering process is always accompanied by an instance of the class RenderContext. The RenderContext encapsulates different displaying attributes which determine the rendering process stringently.

The displaying attributes are divided into direct attributes like zoom, rotation or similar and ProcessingSettings[31]. ProcessingSettings are attributes of a special nature which describe displaying properties of a very particular type, e.g. AnnotationRenderSettings[32]. Visibility properties of annotations may be set by AnnotationRenderSettings. Thus it is possible to make all annotations in/visible or just annotations of a certain type. For more details refer to chapter 11.

Further on the RenderContext offers transformations for a simple conversion between document and device coordinate system. More information: document model in chapter 6, chapter 7 and the jadice® integration documentation.

30 com.levigo.jadice.docs.RenderContext
31 com.levigo.jadice.docs.ProcessingSettings
32 com.levigo.jadice.annotation.AnnotationRenderSettings

11. Visibility

Under certain circumstances it may be sensible to hide all or particular annotations.

Example: Although the annotations are to be visible in the viewer, the printing of the document should be performed without annotations.

jadice® supports two types for the changing of the annotations' visibilty. On the one hand all annotations may be faded in/out, on the other hand all annotations of a particular type may be switched on/off.

For this purpose use the class RenderContext[33], more precisely the com.levigo.jadice.annotation.AnnotationRenderSettings[34] which are contained in the class RenderContext.

```
// Instance of the current RenderContext from the viewer
RenderContext rc = viewer.getRenderContext();
AnnotationRenderSettings settings =
  rc.getAnnotationRenderSettings();
System.out.println("All annotations visible ? "
  + settings.isAnnotationRenderingEnabled());
settings.setAnnotationRenderingEnabled(!settings
  .isAnnotationRenderingEnabled());
System.out.println("All annotations visible ? "
  + settings.isAnnotationRenderingEnabled());
```

Code example 17- Annotation visibility

11.1. Visibility of all annotations

In code example 17 first the global visibility of all annotations is queried by the method **„isAnnotationRenderingEnabled()“** and logged out on the console.

The visibility of all annotations may be changed with the respective method **„setAnnotationRenderingEnabled(boolean)“**.

!

Note:

Integrators should have the viewer re-rendered after changes on ProcessingSettings.

Example: myViewer.repaint();

11.2. Visibility of particular annotations

Similar to the querying or setting of the visibility of all annotations, the visibility of annotations of a particular type may also be changed. The only difference is to pass additionally a definition of the annotation type to the respective method as a parameter.

As shown in code example 18, the class object of the appropriate annotation type is additionally passed as a parameter to the respective methods. This

33 com.levigo.jadice.docs.RenderContext
34 com.levigo.jadice.annotation.AnnotationRenderSettings

determines which annotation type is affected by the visibility change or the state inquiry.

```
// Instance of the current RenderContext from the viewer
RenderContext rc = viewer.getRenderContext();
AnnotationRenderSettings settings =
  rc.getAnnotationRenderSettings();
System.out.println("All text annotations visible ? "
  + settings.isAnnotationRenderingEnabled(
  TextAnnotation.class));
settings.setAnnotationRenderingEnabled(
  TextAnnotation.class, !settings
  .isAnnotationRenderingEnabled(TextAnnotation.class));
System.out.println("All text annotations visible ? "
  + settings.isAnnotationRenderingEnabled(
  TextAnnotation.class));
```

Code example 18– Annotation visibility

Note:

Integrators should have the viewer re-rendered after changes on ProcessingSettings.

Example: myViewer.repaint();

12. Property editors

By using property editors attributes of one or more annotations of the same type may be changed.

At the time being jadice® viewer supports IBM ContentManager, FileNet and FileNet P8 as compatible annotations. Accordingly the property editors of the jadice® package allow annotation changes only to value ranges which do not break the VI- or FileNet-compatibility.

In jadice® document platform demo-classes the property editors are to be activated by the context menu.

The central class AnnotationPropertyEditorFactory[35] is for integrators who would like to embed property editors in their own application environment. This class offers two possibilities for the editors' integration:

† `getAnnotationPropertyEditorAsPanel(...)` - This method provides a JPanel containing the attribute editors of the annotations to be changed.

† `getAnnotationPropertyEditorAsDialog(...)` - This method provides a modal dialogue containing the attribute editors of the annotations to be changed.

12.1. Write own editors

Integrators are free to create and embed own property editors.

Example: In order to keep the corporate design of the integrating application or to change the value ranges of the editors.

There are property editors belonging to each annotation attribute. With the annotations to be changed the composition of single attribute editors to an annotation editor is detected which is then returned by the **AnnotationPropertyEditorFactory** as JPanel or as dialogue. If the user confirms changes performed in the editor, these are automatically updated in the regarding annotations.

The basis class of all attribute editors is AbstractPropertyEditor[36]. AbstractPropertyEditor is a JPanel which may be simply integrated in the mechanism mentioned above.

Creating own AbstractPropertyEditors the integrator decides on the layout and the components contained therein. However, two methods are to be implemented:

† **getValue()** - Provides the current value of the regarding attribute. Transmitting the new attribute value to the annotation, when the changes are confirmed by the user.

† **setValue(...)** - Sets the annotation's current value in the editor.

Having created an own editor, it must be communicated to the PropertyEditorFactory for a particular attribute.

For this no further programming effort is necessary.

35 com.levigo.jadice.annotation.edit.AnnotationPropertyEditorFactory
36 com.levigo.jadice.annotation.edit.AbstractPropertyEditor

In the package com.levigo.jadice.resources.properties all configuration files describing the jadice® document platform are stored. The file „AnnotationEdit.properties" contains all information about annotation property editors.

Within this file different key-value pairs are provided in the paragraph „Annotation Properties". Attribute editors are logged in by the key „attributName.editor". The value to be set is the class name of the newly created editor. Attribute editors are instantiated by reflection. So it is absolutely necessary to indicate the class name correctly. More details concerning the file AnnotationEdit.properties are in chapter 12.

Example:

```
rotation.editor=my.package.MyRotationEditor
```

Code example 19- Own editor

Note:

It is advised to copy these configuration files and to place them in the class path in front of the jadice® modules. So performed changes are recognised, but the original configuration remains untouched.

Note:

Please keep in mind that the configuration files are provided in internationalised variants. Changes on the configuration should be always done in all variants.

13. The configuration file: AnnotationEdit.properties

Note:

The settings' suffix for VisualInfo / ImagePlus annotations starts with „**vi**.“, for FileNet annotations with „**fn.**“.

If no suffix is defined, the VisualInfo / ImagePlus setting is used.

Using the configuration file AnnotationEdit.properties settings for the following topics are possible:

- Which properties are changeable for which annotation type?
- Which editor is to be used for which attributes?
- Text resources for GUI elements of the attribute editors/dialogue

For a better orientation the settings are shortly specified in the following. All settings are defined by key-value pairs.

13.1. Which properties are changeable for which annotation type?

This information can be found under the title „Annotation Editor Mapping“.

Per annotation type it may be defined which attributes of this very annotation type are to be changeable. Here the key corresponds to the qualifiedly named annotation class, the value is a comma-separated list of changeable attributes.

Example:

```
#
# Annotation Editor Mapping
#
...
com.levigo.jadice.annotation.FreehandAnnotation=foreground
Color,lineWidth
```

Code example 20- Annotation Editor Mapping

This line tells that the changeable attributes of the freehand-annotation are the foreground colour and the line width.

13.2. Which editor is to be used for which attributes?

This information may be found under the title „Editors“.

At this point you may define which editor is to be used for the changing of an attribute. The key is composed of the attribute's name and the suffix „**editor**“. The value corresponds to the qualifiedly named editor class. Since editors are created by reflection, take care at this point to write the class name correctly.

Example:

```
#
# Editors
#
```

```
...
foregroundColor.editor=com.levigo.jadice.annotation.edit.B
aseColorEditor
```

Code example 21- Editors

By this specification the class BaseColorEditor is defined as editor for the foreground colour.

13.3. Text resources for GUI elements

This information can be found under the title „Titles and Strings".

In this paragraph property editors define text resources for the GUI-elements used by themselves. These are e.g. the field naming of attribute-input-fields, a dialogue title, but also error report texts.

Example:

```
#
# Titles and Strings
#
...
foregroundColor.title=foreground colour
```

Code example 22- Titles and Strings

!

Note:

It is advised to copy these configuration files and to place them in the class path in front of the jadice® modules. So performed changes are recognised, but the original configuration remains untouched.

Note:

Please keep in mind that the configuration files are provided in internationalised variants. Changes on the configuration should be always done in all variants.

Note:

Changes on this configuration file could, depending on the connected archiving system and the respectively supported annotation type, lead to compatibility problems.

14. The configuration file: AnnotationInit.properties

!

Note:

The settings' suffix for VisualInfo / ImagePlus annotations starts with „**vi.**", for FileNet annotations with „**fn.**".

If no suffix is defined, the VisualInfo / ImagePlus setting is used.

Using the configuration file AnnotationInit.properties settings to define the looks of a newly created annotation are possible:

† Definition of colour for fore-/background and margin (all annotation types)

† Property setting:

- transparency
- fill background
- displaying of margin
- line width
- rotation
- displaying of annotation as icon
- displaying of annotation numbers

† Minimum size of a newly created annotation (all annotation types)

† Font size and style (note-, stamp- and text annotation)

† Minimum size of the text input window (note-, stamp- and text annotation)

14.1. General properties (VisualInfo / ImagePlus and FileNet annotations)

General properties are independent of the annotation type and apply for VisualInfo / ImagePlus and FileNet annotations. A suffix is not used with the parameter names.

<u>Annotation numbers:</u>

```
annotation.show-number-on-start=<value>
```

Parameter, if the annotation numbers are to be displayed immediatelly at the start.

Value: true = numbers are displayed at the start.
false = numbers are not displayed.

```
annotation.number-display-mode=<value>
```

Parameter for the definition of the number displaying (behaviour when zooming).

Value: 0 = number size depends on zoom factor.
1 = number size is not enlarged, if zoom factor is greater than 100.
2 = number size is not reduced, if zoom factor is less than 100.
3 = number size is independent of the zoom factor.

`annotation.number-display-size=<value>`

Parameter for setting of number size.

Value: font size.

14.2. Annotation properties

The definition is composed as follows:

<**suffix**>.<**annotation type**>.<**parameter**>

suffix = fn or vi

annotation type = annotation type (package name + class name)

Description of possible parameters:

- **defaultSize** (type = dimension, default=1,1)
  Size when creating an annotation.
  -> all annotations except stamp

- **editFrameMinSize** (type = dimension, default=1,1)
  Size of text input window.
  -> note, text, stamp

- **foregroundColor** (type = colour, default=colour.yellow)
  Foreground colour.
  -> arrow, ellipse, freehand, highlight, line, rectangle, stamp, text

- **backgroundColor** (type = colour, default=Colour.white)
  Background colour.
  -> arrow, ellipse, freehand, highlight, line, rectangle, stamp, text

- **transparent** (type = boolean, default=false)
  Background transparency.
  -> rectangle, stamp, text

- **iconify** (type = boolean, default=false)
  Icon displaying.
  -> note

- **linePainted** (type = boolean, default=true)

Draw margin line.

-> arrow, ellipse, freehand, highlight, line, rectangle, stamp, text

- **borderColor** (type = colour, default=colour.yellow)

  Margin colour.

  -> ellipse, rectangle, stamp, text

- **lineWidth** (type = int, default =3)

  Line width.

  -> arrow, ellipse, freehand, line, rectangle, stamp, text

- **filled** (type = boolean, default=false)

  Fill background.

  -> ellipse, rectangle, stamp, text

- **fontFace** (type = String, default=Sansserif)

  Font type.

  -> note, stamp, text

- **fontSize** (type = int, default=36)

  Font size.

  -> note, stamp, text

- **fontBold** (type = boolean, default=false)

  Bold font.

  -> note, stamp, text

- **fontItalic** (type = boolean, default=false)

  Italic font.

  -> note, stamp, text

- **headAngle** (type =int, default=20)

  Angle of arrowhead.

  -> arrow

- **headLength** (type =int, default=25)

  Length of arrowhead.

-> arrow

- **allowResize** (type =boolean, default=false)
  Change size.
  -> freehand

- **gap** (type = int, default=10)
  Space between font and margin.
  -> stamp

- **rotation** (type = int, default=20)
  Rotation.
  -> stamp

- **userName** (type = String, default=unknown)
  User name.
  -> history

- **pattern** (type = String, default=<---[Created by <user> at <date> <time>]--->)
  Pattern for info line.
  <user> = Is replaced by user name.
  <date> = Is replaced by current date.
  <time> = Is replaced by current time.
  -> history

- **date** (type = String, default=DDMMYYYY)
  Displaying of date in info line.
  Possible patterns: DDMMYYYY, YYYYMMDD
  -> history

- **time** (type = String, default=24H_DEFAULT)
  Displaying of time in info line.
  Possible patterns: 12H_AM_PM, 24H_DEFAULT
  -> history

- **patternPosition** (type = String, default = PATTERN_HISTORY _AT_BOTTOM)

Position of info line.

Possible patterns: PATTERN_HISTORY_AT_BOTTOM , PATTERN_HISTORY_AT_TOP

-> history

The following example shows property changes of a VisualInfo annotation.

```
vi.com.levigo.jadice.annotation.TextAnnotation.lineWidth=1
vi.com.levigo.jadice.annotation.TextAnnotation.foregroundC
olor=255
vi.com.levigo.jadice.annotation.TextAnnotation.defaultSize
=50,50
vi.com.levigo.jadice.annotation.TextAnnotation.text=Text-
Annotation
vi.com.levigo.jadice.annotation.TextAnnotation.editFrameMi
nSize=100,75
```

Code example 23 / annotation properties

14.3. Annotation´s text settings

The text displaying of the text based annotations can be adjusted, the following settings for text adjustment are for note, text and history annotations.

Only whole words are recognised at the text adjustment.

Note: <CR> stands for an entered line-break at the input window, the # represent the window margin.

˜ adjustTextToWindow=false

  Text will NOT be adjusted to the displaying window size

The behaviour can be controlled with the keepEditText parameter:

˜ keepEditText=false (Default)

  Text will not be adjusted. Text from the input window will be aligned with the line-breaks.

```
Text input window:
######################
#This is a line      #
#with text.          #
#                    #
######################

Displaying window:
######################
#This is a line      #
#                    #
#                    #
######################
```

```
Annotation displaying 1 / without line-break
```

```
Text input window:
######################
#This is line 1      #
#With text.<CR> #
#Here starts line 2. #
######################

Displaying window:
######################
#This is line 1. #
#Here is line 2.      #
#                     #
######################
```

```
Annotation displaying 2 / with line-break
```

˜ keepEditText=true

The text is taken from the text input window, at a break in the input window a line-break will be generated.

```
Text input window:
######################
#This is a line       #
#with text.           #
#                     #
######################

Displaying window:
######################
#This is a line       #
#with text.           #
#                     #
######################
```

```
Annotations displaying 3 / without line-break
```

```
Text input window:
######################
#This is the 1.       #
#line with text.<CR> #
#Here comes line 2.   #
######################

Displaying window:
######################
#This is the first    #
#line with text.      #
#Here comes line 2. #
```

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

Annotation displaying 4 / with line-break

˜ adjustTextToWindow=true (Default)

The keepEditText Parameter should not be used here (keepEditText=false), as additional line-breaks could occur eventually that may cause a confusing behaviour.

Text will be adjusted to the size when increasing/decreasing the annotation. The text in the input window will not be changed, i.e. text will ONLY be changed when displayed.

Using the **AdjustTextToWindowNewLineMode** parameter the behaviour of entered line-breaks can be controlled additionally (only works with adjustTextToWindow=true).

˜ adjustTextToWindowNewLineMode=0 (Default)

Entered line-breaks will NOT be recognised at displaying.

```
Text input window:
######################
#This is a line      #
#with text.          #
#                    #
######################

Display window enlarged:
#################################
#This is a line with text.      #
#                               #
#                               #
#################################

Display window downsized:
##################
#This is a       #
#line with text  #
#                #
##################
```

Annotation displaying 5 / without line-break

```
Text input window:
#####################
#This is the 1.     #
#line with text.<CR> #
#Here is line 2.    #
#####################

Display window enlarged:
######################################
#This is the 1. line with text. Here  #
#is line 2.                           #
#                                     #
######################################

Display window downsized:
#################
#This is the 1.  #
#line with text. #
#Here is line 2  #
#################
```

Annotation displaying 6 / with line-break

- adjustTextToWindowNewLineMode=1

Entered line-breaks will be recognised at displaying.

```
Text input window:
#####################
#This is line       #
#with text.         #
#                   #
#####################

Display window enlarged:
################################
#This is a line with text      #
#                              #
#                              #
################################

Display window downsized:
#################
#This is a       #
#line with text  #
#                #
#################
```

Annotation displaying 7 / without line-break

```
Text input window:
```

```
######################
#This is the 1.      #
#line with text.<CR> #
#Here is line 2.     #
######################

Display window enlarged:
#######################################
#This is the 1. line with text.       #
#Here is line 2.                      #
#                                     #
#######################################

Display window downsized:
###########
#This is  #
#the 1.   #
#line with#
#text.    #
#Here     #
#is       #
#line 2.  #
#         #
###########
```

Annotation displaying 8 / with line-break

15. Document history

Version	Date	Author	Changes
1.0	02.09.03	Carolin Köhler	chapter 1 - 5
	03.09.03	Carolin Köhler	chapter 6
	04.09.03	Carolin Köhler	chapter 7, partly chapter 8
	05.09.03	Carolin Köhler	chapter 8 - 11
	08.09.03	Carolin Köhler	chapter 12
1.1	08.12.03	M. Grossmann	Upgrading of FileNet Annotations chapter 3, 4, 5, 6, 12, 13
1.2	20.01.04	M. Grossmann	Changes regarding FileNet Annotations chapter 4
1.3	22.04.04	M.Grossmann	Changes in chapter 3, 5, 6 Upgrading in chapter 4, 14
1.4	23.01.06	M.Grossmann	Changes in chapter 5 Upgrading in chapter 9
2.0	06.02.06	Carolin Köhler M.Grossmann	Text revised
4.1.0	April 2008	Jan Henne	Translation of new and updated chapter and paragraphs.
4.1.1	April 2008	Carolin Köhler M.Grossmann	Text revised
4.2.0	February 2008		Upgrade