

Carolin Köhler

February 2009

# jadice<sup>®</sup> document platform

## Version 4.2.x

### Developer's Guide

## Contents

1. GENERAL.....	5
1.1. ABOUT THIS DOCUMENTATION.....	5
1.1.1. GENERAL.....	5
1.1.2. FEEDBACK.....	5
1.2. ABOUT THIS PRODUCT.....	5
1.3. DISTRIBUTION.....	5
1.4. ONLINE-SERVICE.....	6
2. INTRODUCTION.....	7
2.1. FUNCTIONAL RANGE IN VERSION 4.2.....	7
2.2. PARTICULAR JADICE FEATURES.....	7
2.2.1. PROCESSING OF VERY LARGE DOCUMENTS.....	7
2.2.2. PROCESSING OF VIRTUAL DOCUMENTS.....	7
2.2.3. LARGE NUMBER OF DIRECTLY SUPPORTED DOCUMENT FORMATS.....	8
2.2.4. SIMPLE INTEGRATION POSSIBILITIES WITH THE JADICE INTEGRATOR API.....	8
2.2.5. THE MOST IMPORTANT TECHNICAL FUNCTIONALITIES AT A GLANCE.....	9
2.3. SYSTEM REQUIREMENTS.....	9
2.4. TERMS.....	10
2.4.1. THE DOCUMENT MODEL.....	10
2.4.1.1. DOCUMENTS.....	10
2.4.1.2. LAYERS.....	10
2.4.1.3. PAGES.....	11
2.4.1.4. PAGESEGMENTS.....	11
2.4.2. ANNOTATIONS.....	11
2.4.3. RESOURCES.....	12
2.5. FORMATS.....	12
3. THE JADICE INTEGRATOR API.....	14
3.1. AIMS.....	14
3.2. REALISATION.....	14
3.2.1. COMMANDS.....	14
3.2.2. ACTIONS.....	15
3.2.3. CONTEXT.....	16
3.3. CONFIGURATION FILES.....	17
3.3.1. COMMANDS.PROPERTIES.....	18
3.3.2. MENUCOMPONENTS.PROPERTIES.....	18
3.3.3. ACTIONS.PROPERTIES.....	18
4. CLASS SURVEY.....	19
4.1. VIEWER.....	19
4.2. DOCUMENT.....	20
4.3. PAGE.....	21
4.4. PAGESEGMENT.....	21
4.5. LOADER.....	22
4.6. FORMATINFO AND FORMATFILE.....	22

4.7. LOADLISTENER.....	23
4.8. RESOURCELOADER.....	24
4.8.1. RESOURCEFILELOADER.....	26
4.8.2. RESOURCEURLLOADER.....	26
4.8.3. RESOURCEGROUPLoader.....	26
4.8.4. RESOURCEMULTILOADER.....	27
4.9. SEEKABLEINPUTSTREAMS.....	27
4.9.1. RANDOMACCESSINPUTSTREAM.....	28
4.9.2. FILECACHEINPUTSTREAM.....	28
4.9.3. MEMORYINPUTSTREAM.....	29
4.10. ANNOTATION.....	29
4.10.1. CHANGES ON ANNOTATIONS.....	32
4.11. IMAGEPLUSANNOTATIONFORMATINFO.....	32
4.12. IMAGEPLUSANNOTATIONFILE.....	33
4.13. FILENET AND FILENETP8 ANNOTATIONS.....	33
4.14. RENDERCONTEXT.....	34
4.15. EDITPANES.....	34
4.16. BASICJADICEPANEL.....	36
4.17. ADDONS.....	37
4.17.1. CREATION.....	37
4.17.2. CALL BY COMMANDS.....	37
4.17.3. INTEGRATION IN DIFFERENT ENVIRONMENTS.....	38
4.18. JADICEBOOKMARK.....	38
4.19. DOCUMENTBOOKMARKHANDLER.....	39
4.20. PAGESORTER.....	40
4.20.1. SUPPORT OF POPUPMENUS IN THE PAGESORTER.....	40
4.21. NAVIGATORPANEL.....	41
4.22. LENS.....	41
4.22.1. HOVERLENS.....	42
4.23. GRADATIONCURVECONTROL.....	42
4.23.1. GRADATIONCURVE.....	43
4.23.2. GRADATIONCURVEFILEHANDLER.....	43
4.24. PRINTERJAVA2.....	44
4.25. PRINTMANAGER.....	44
4.26. FILEOPENER.....	45
4.27. DOCUMENTSAVER.....	45
4.28. DEMONSTRATION CLASSES.....	45
4.28.1. PARAMETER OF THE DEMONSTRATION CLASSES JADICEPANEL AND JADICEMDI.....	45
4.28.2. PARAMETER OF THE DEMO-APPLET JADICEAPPLET.....	46
5. TYPICAL APPLICATION EXAMPLES.....	47
5.1. EMBED VIEWER INTO A FRAME.....	47
5.2. LOADING PROCESS.....	48
5.2.1. SIMPLE LOADING PROCESS.....	48
5.2.2. ASSEMBLE DOCUMENTS.....	49

5.2.3. LAYER.....	51
5.2.4. SEEKABLEINPUTSTREAM.....	54
5.2.5. RESOURCELOADER.....	55
5.2.6. ANNOTATIONS.....	56
5.2.7. BOOKMARKS.....	57
5.2.8. GRADATION.....	58
5.3. SAVING.....	59
5.3.1. DOCUMENT.....	59
5.3.2. ANNOTATIONS.....	60
5.3.3. BOOKMARKS.....	61
5.3.4. GRADATION.....	62
5.4. ACTIONS-COMMANDS-CONTEXT.....	62
5.4.1. EMBEDDING OF MENUS, TOOLBARS, ACTIONS.....	63
5.4.1.1. CONTEXT.....	64
5.4.1.2. EMBEDDING.....	65
5.4.2. ADAPTATION OF ACTIONS.....	67
5.4.2.1. PROPERTIES.....	67
5.4.2.2. ADAPTING THE MENU OR TOOLBAR STRUCTURE.....	67
5.4.3. OWN COMMANDS.....	68
5.5. PRINTING.....	72
5.5.1. SIMPLE PRINTING.....	72
5.5.2. SETTINGS.....	72
5.5.3. ADAPTATION OF RENDERCONTEXT.....	74
6. LOGGING.....	75
6.1. JADICE® LOGGING FRAMEWORK FACADE.....	75
6.2. FIRST STEPS.....	75
6.2.1. LOG4J.....	75
6.2.2. SLF4J.....	76
6.3. POSSIBLE ERRORS.....	76
7. CONFIGURATION AND SETTINGS.....	77
7.1. THE MOST IMPORTANT SETTINGS IN DETAIL.....	78
8. JADICE INTEGRATOR API: SYNTAX DESCRIPTION OF THE CONFIGURATION FILES. .	83
8.1. THE FILE „COMMANDS.PROPERTIES“.....	83
8.2. THE FILE „MENUCOMPONENTS.PROPERTIES“.....	84
8.3. THE FILE "ACTIONS.PROPERTIES".....	85
8.4. THE FILE „JADICE-VIEWER.PROPERTIES“.....	87
9. JADICE PUBLIC API AND INTERNAL PACKAGES.....	89
9.1. JADICE PUBLIC API.....	89
9.2. JADICE PRIVATE API.....	89
10. DOCUMENT HISTORY.....	90

## 1. General

### 1.1. About this documentation

#### 1.1.1. General

This guide in hand is an introduction to the technical coherences of the jadice<sup>®</sup> document platform (in the following shortly called jadice).

This documentation is basically limited to the areas which are interesting to developers (subsequently called integrators) in order to integrate jadice<sup>®</sup> in their own applications.

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

An API reference in *javadoc* format is made available as a separate document.

#### 1.1.2. Feedback

If you come across any errors when using this documentation or if you like to suggest any improvements, please send a possibly detailed message to [solutions@levigo.de](mailto:solutions@levigo.de).

Your feedback helps us in further developing this documentation. Thanks a lot.

### 1.2. About this product

jadice has been developed from the platform and format independent document viewer jadice viewer, which due to its flexibility and power has been specially used in archiving parts, to a flexible components' solution for the use in professional document management.

As an easy-to-integrate Java toolbox with useful modules, elaborated interfaces and helpful additional components jadice offers the base for individual archive client solutions. Anyway, the document viewer has remained an essential part of jadice – with an advanced and expanded functional range, though.

### 1.3. Distribution

jadice is now available in two types of packaged variants each with the same functional range: as the usual all-in-one product and as a modular solution with functions combined in units.

Since both variants are contained in the distribution, the integrator may decide according to situation, use and solution, if he wants to use the hitherto existing all-in-one solution or just the required and desired modules.

These packaged variants have been made possible by jadice's new architectural plan which is based on a component structure. There component-libraries and their dependencies are combined to sensible units. This structure offers flexibility to the integrator to use only selected functional units, if the complete solution is to be as compressed and efficient as possible – e.g. in the domain of web applications where transmission rates are relevant. Alternatively the reliance of the all-in-one solution still remains by using the complete functional range.

When using the all-in-one variant all functionalities of the jadice document platform are available for the integrator, but updates and tests have also to be done for the whole solution.

The modular variant makes it possible for the integrator to use only the required modules. However, he always has to make sure that he is provided with all files and libraries depending from each other. Regarding updates and tests this variant is easy to handle, since only the respective module or the respective function has to be updated or tested.

A survey of all modules of the jadice document platform and their dependencies is available in the HTML distribution documentation.

#### Important note:

jadice document platform requires a Java Virtual Machine 1.5. or higher.

~ Use for JVM 1.5. or newer the libraries in folder **jdk15** or **lib-jdk15**.

### 1.4. Online-Service

For developers and integrators we have got an online-service helping you to state directly your wishes, problems, suggestions or similar concerning jadice Viewer and to forward it to the respective jadice developer.

Thereupon you will be automatically informed by e-mail about all statements, displaying of the problem's state up to its solution. Of course, you may add anytime further information or advice.

If you are interested, please contact [solutions@levigo.de](mailto:solutions@levigo.de).

A HTML documentation – updated to the latest version - with additional information is provided in English as part of the distribution. You can find it under

***jadice-documentplatform-<<Versionnumber>>-  
dist.dir\documentation.html***

## 2. Introduction

### 2.1. Functional range in version 4.2

With jadice in generation 4.2 you get a mighty toolbox for the work with different document data and formats. This paragraph is to give you a short introduction in the innovations of the jadice document platform.

Due to an improved support of the **PDF-** and **PDF/A-**format range the font types **Type1**, **CompactFont** and **Type0** can be processed. Furtheron the FileNet Image Format (**fni**) – a proprietary FileNet bi-level document format which has not been supported up to now – may be displayed and used in its full range.

The previous support of **FileNet P7** annotations has been expanded to the support of **FileNet P8** annotations with their full functional range.

Due to a new logging framework it is now possible to integrate easily and seamlessly other already existing logging systems by delegation into the jadice document platform. This is realized by a logging facade which offers an abstraction level of known logging systems like **Log4J**, **JDK 1.4 Logging** or **Logback** and thus for using them it often only needs an adaptation of the class path.

Further logging frameworks may be integrated via **SLF4J**.

### 2.2. Particular jadice features

#### 2.2.1. Processing of very large documents

- † e.g. technical sketches, architecture or site plans: TIFF-files with more than 20.000 pixel<sup>2</sup>
- † e.g. colour-pictures: considerably bigger compressed data volumes

#### 2.2.2. Processing of virtual documents

- † Documents can consist of different document data from different origin (e.g. TIFF and MO:DCA).
- † Compound documents are already displayed during the loading of the belonging document data, if this is allowed by the respective format.
- † Different document data can be added to the document consecutively (like pages) or on top of each other (as various page elements).
- † Processing of documents with layers (superposed / parallel page elements)
  - † e.g. letter-head and letter-text in separate files
  - † Pages consist of different layers which may come from different resources.

Due to this flexibility jadice's architecture offers the possibility to create virtual documents out of different physical documents and to use this composed structure with full functional range like a simple document.

Virtual documents are often used to represent processes and files belonging together.

### 2.2.3. Large number of directly supported document formats

Another specialty is the wide range of supported document formats. A listing of presently supported formats can be found in paragraph [2.5.Formats](#).

Picture files are rarely loaded with jadice from a local file system by an „open file“-dialogue. Jadice normally receives document and annotation data from a document management system.

Further information and a product survey are provided on the levigo website<sup>1</sup>.

### 2.2.4. Simple integration possibilities with the jadice Integrator API

jadice document platform offers very easy and flexible integration possibilities, i.e. an easy integration of viewer components with a flexible availability and possible adaptations of functions as well as the reusability and expansion of existing tools and actions.

- † Easy viewer integration:
  - † BasicJadicePanel: the simplest possibility to integrate a viewer
  - † with toolbar, annotation toolbar, status bar, optional menu bar
  - † changes in tools or menus only by textual adaptation of configuration files
- † Actions and commands: command processing with command and action patterns
  - † encapsulation (separation) in single actions
    - † actions update automatically availability (enable state)
    - † easy adaptation to corporate identity, corporate design
    - † flexible changing of features: icons, accessors, inputmaps etc.
  - † structuring of tool bar and menus by using the configuration file
  - † simple definition of different look & feels, as well as the accompanying menu and toolbar structures which can be used according to demand, e.g., for
    - † pure research
    - † processing
    - † administration etc.
- † Provided AddOns:
  - † AddOns are useful, directly integratable extensions for viewer components or for document processing.
  - † For integration AddOns offer functional components as JComponent with encapsulated functionality and/or additionally window elements as *Jframe/JInternalFrame* and/or alternatively as *Jrame/JInternalFrame*.

<sup>1</sup> See <http://www.levigo.de/en/document-management/>



### 2.2.5. The most important technical functionalities at a glance

- † Documents<sup>2</sup> are independent of format, i.e. only "containers" for pages.
- † Pages<sup>3</sup> are independent of format and composed of layers<sup>4</sup>.
- † „Occupied“ layers of single pages, so called page segments, carry the real document data and are thus format-specific.
- † A flexible displaying of contents by virtual documents which may have been created from different physical document sources. As such even complex structures like files or processes may be created and used.
- † The input/output processing (especially the loading of documents) is laid out on an optimum memory efficiency. If necessary this behaviour may be additionally adapted by the integrator to the environmental usage and the desired requirements.
- † Easy integration of supplied or proper functionalities and visual representations.

## 2.3. System requirements

Basically jadice works as a pure Java application platform-independently. With this all operating systems are supported for which a JVM Machine from 1.5.x or newer is provided.

- † JVM from version 1.5. or higher is recommended.
- † At least 256MB memory for jadice document platform are recommended. The required memory of the integrating application should be taken separately and it should be added to the heapsize of the jadice document platform in order to determine the recommended heapsize of the whole application. Then the required memory thus calculated may be provided to the application by using the VM parameter -xmx.

---

2 Instances of the class `com.levigo.jadice.docs.Document`

3 Instances of the class `com.levigo.jadice.docs.Page`

4 Instances of the class `com.levigo.jadice.docs.DocumentLayer`

## 2.4. Terms

### 2.4.1. The document model

[Chart 1](#) shows an outline of jadice's document model. This model consists of four elements which are further described in the following.

#### 2.4.1.1. Documents

In practise documents can be considered as a summary of pages which are processed as a unit in an application. On this occasion is to be noticed that no

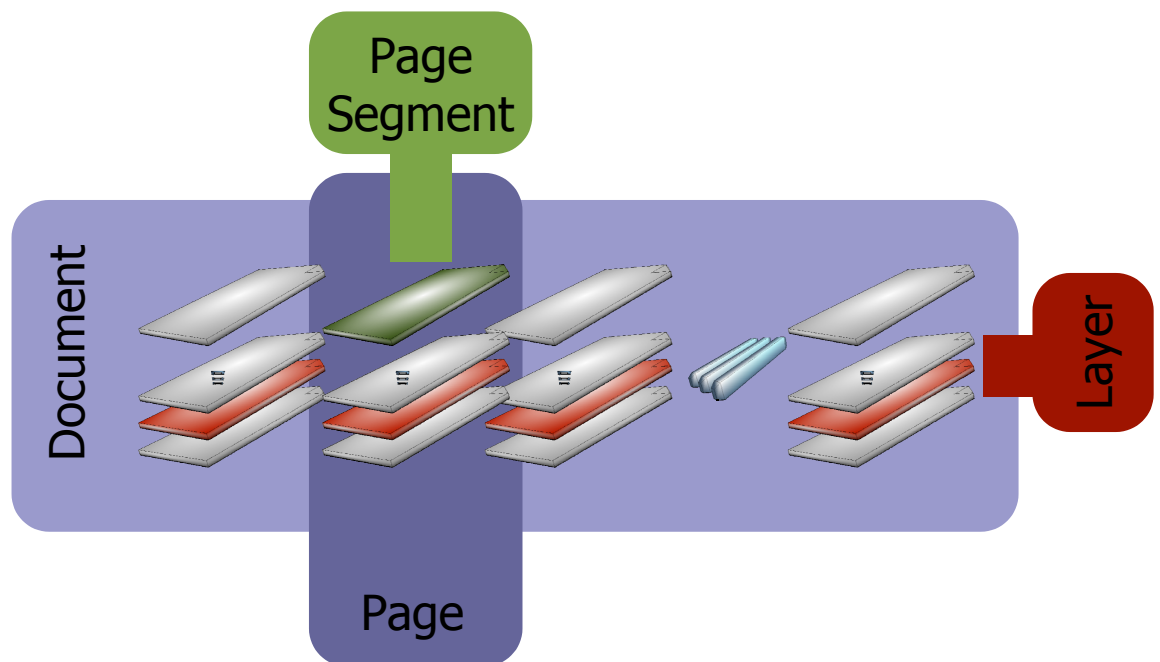


Chart1 - Document modell

firm and permanent connection need to exist between documents and data streams or data formats. A jadice document can obtain its pages from several data streams or data formats, but even the pages themselves can in regard of their presentation be composed of data of different data streams.

Thus jadice documents are of a two-dimensional structure:

- † they consist of a number (0-n) of logical pages and
- † a number (1-n) of presentation layers.
- † Page segments which lie in a corresponding displaying layer.

#### 2.4.1.2. Layers

The content of a page can be composed as a sum of the most different components like e.g. letter-head and letter-text. However, in its presentation the page appears as a unit. jadice uses layers for this purpose.

Layers present a „vertical“ division of the document in which page segments are positioned. Different layers are displayed, as a rule, directly superposed. This means that from the user's view all layers of a document form a closed unit. Thus layers are comparable to superposed transparent folios.

### 2.4.1.3. Pages

Pages usually represent the smallest unit displayed to the user. They consist, similar to the documents, of further objects so-called PageSegments. PageSegments cover within a page the positions determined by the document's layers. In other words: the document's layers provide a limited number of „places“ within the page as well as a logical order of these places. These places can be (but not necessarily) covered within the pages with PageSegments. Pages don't create images for displaying, they render themselves automatically into given graphic contexts like e.g. monitor, printer, etc.

### 2.4.1.4. PageSegments

PageSegments carry the actual document information. Thus they correspond directly with the according pages of a document data stream. Examples for PageSegments are:

- † The scanned picture of the front or back side of a physically present sheet of paper or a similar media with NCI-documents.<sup>5</sup>
- † An individually displayed page of a CI-document<sup>6</sup> like e.g. ASCII- or PTOCA.
- † Annotations set on a page.

PageSegments are integrated by means of layers into the page or the document and thus they get allocated a „vertical“ position within the page.

## 2.4.2. Annotations

Annotations are understood as

- † comments
- † notes
- † remarks
- † explanations
- † notices or
- † indications by arrows or coloured areas

which the user can set on a certain page in a document. Annotations are additional information to a document and do not change the original document. For this purpose annotation data are managed in an own annotation page segment.

These annotations can contain information in form of

- † text or
- † graphic objects for
  - † illustration / clarification
  - † highlighting or even for

<sup>5</sup> NCI: Non Coded Information like images, language, sound, video etc., which cannot be recognised and directly processed by the computer. A typical NCI application is the capturing of documents with scanners and their treating as facsimiles.

<sup>6</sup> CI: Coded Information like texts which exist as graphic characters and can directly be processed and displayed.

† fading out / hiding.

### 2.4.3. Resources

Special formats like AFP & MO:DCA support using elements like logos, forms, overlays and page segments.

These elements are called resources and can be found in the AFP- or MO:DCA-document itself (inline) or in an external resource.

## 2.5. Formats

The following formats are directly supported by jadice:

- † PDF/A
- † AFP & MO:DCA with following content
  - † PTOCA
  - † IOCA
  - † GOCA
  - † Page Segment
  - † Overlays & Forms
- † IOCA
- † TIFF
  - † uncompressed
  - † compressed
    - † RLE
    - † Packbits
    - † Fax G3 / G4 or CCITT T.4 and CCITT T.6
    - † JPEG (true-colour and grayscale)
    - † LZW
    - † DEFLATE
- † FileNet Banded Image
- † EBCDIC
- † ASCII
- † JPEG/JBIG/JFIF
- † GIF
- † BMP
- † PNG
  
- † Further optional formats via ImageIO-interface by using Java Advanced ImageIO Tools

- † JPEG2000
- † and further formats depending on Java Advanced ImageIO version like
  - † PNM
  - † (partly) FlashPix
  - † etc.
  
- † Further optional formats via ImageIO-interface as independent jadice products:
  - † DJVu
  - † DiCOM

## 3. The jadice Integrator API

### 3.1. Aims

One of the aims of the jadice document platform is to offer very simple integration possibilities with the lowest effort possible for integrators, but still high flexibility and adaptability. Therefore three points of interest have been traced:

† BasicJadicePanel – the simplest way to integrate a viewer

As JPanel it can be optionally integrated in existing layouts and it contains a toolbar with the most important viewer tools, an annotation toolbar for creating annotations, a status bar for the displaying of page numbers and of the zoom factor and optionally a menu bar if the embedding component is a frame able to set a menu bar. Adaptations, like appearances or hot keys as well as the structure of tools in toolbars or menus, can be simply reached by changing the configuration. Also compare paragraph [4.16.BasicJadicePanel](#).

† AddOns - Useful viewer extensions directly integrable

Viewer extensions, also called AddOns, like the navigator or the page sorter, are available as JComponent with encapsulated functionality and uniform API, alternatively as a *JFrame* / *JInternalFrame*.

For further information about AddOns consult paragraph [4.17.AddOns](#).

† Command processing with Command and Action Pattern

All viewer specific tasks like zoom or rotation, but also the activating of AddOns were encapsulated in single commands which can be called up by actions. These actions are responsible for their GUI representation, e.g. by icon, accelerator or tooltip, and they are able to bundle commands. Behaviour and appearance of actions, but also which commands should be activated, are defined in a configuration file, so in case of adaptations integrators don't have to worry about more programming efforts. In addition to this it is possible with only little programming effort to expand already existing commands in their functionality according your own wishes or to integrate completely own commands.

The cooperation of actions and commands combined with an easy configuration ability offers a strong and flexible framework which makes adaptations in the easiest way with the least programming effort possible.

In addition to this different look&feels as well as menu, context-menu and toolbar structures can be freely defined in the configuration files. An example of such an application is described in chapter [5.4.Actions-Commands-Context](#).

### 3.2. Realisation

#### 3.2.1. Commands

The commands' processing and their binding to user face elements base on „command“ and „action“ design patterns. The action pattern is already used in SWING, the use of the command pattern serves for a further detaching of the command functionality.

Basic tasks like zooming, rotating or activating of AddOns are encapsulated within the jadice package into independent commands which can be called up

by actions and which take over the actual performance of actions. A command receives for the performance a number of objects, called context ([3.2.3.Context](#)), which reflect the user interface's state at the time of the action's performance. Each GUI-element can contribute own objects to this number of objects.

All commands are successors of the abstract basis class `AbstractCommand` which most important methods of are shortly described in the following:

† **doExecute(Collection)**

is called up by the enclosing `CommandAction` for realisation of the actual command. The arguments are the objects contained in the context.

† **checkQuickly(Collection)**

is called up by the enclosing `CommandAction` in order to perform an Enabled-Check. The arguments are the objects contained in the context. The checking methodology within this method should be performant, since the command state is very often checked. The command state is always verified, when the context has been changed (part of the Enabled-Check of the `CommandAction`). Context changes may be caused by changing the document structure, like the loading of further pages and page segments, but also changes in the displaying, like rotation or zoom, even the scrolling of a page within the viewer may cause under certain circumstances a checking of the commands' state.

† **checkDeeply(Collection)**

final test before calling up the "doExecute ()" method. The arguments are the objects contained in the context. In contrast to the „checkQuickly“ method which is very frequently caused „checkDeeply“ is only activated before calling the „doExecute“-method. It allows a complex state checking and context validation before the command is being performed and even allows, if the checking fails, to prevent the „doExecute“ from performing.

† **isAvailable()**

initial test before taking in a `CommandAction` in a menu or toolbar structure. Due to this method a command may verify during its creation, if it is basically available or not. If a command is not available because an edge condition is not fulfilled, e.g. particularly needed resources are not provided, this command automatically won't be created or taken up in the required structure.

In general the commands' practicability (verification of the enabled-state by „checkQuickly“ or „checkDeeply“) is tested along the following pattern:

- † Checking, if the expected argument types (classes) are contained in the expected number in the passed context objects.
- † Checking, if the command's performance is possible with the content of the passed arguments.

### 3.2.2. Actions

Actions<sup>7</sup> bind performable commands to GUI elements like buttons, toolbar buttons or menu items. They contribute on the one hand to the appearance of

---

<sup>7</sup> javax.swing.Action

the GUI element by providing e.g. icons or labels, on the other hand they also control the state of the GUI elements as e.g. the enabled/disabled state.

CommandActions<sup>8</sup> can additionally bundle one or more commands<sup>9</sup> (see also [3.2.1.Commands](#)) and perform in their "actionPerformed(...)" method.

Further on each CommandAction is bound to a context (see also [3.2.3.Context](#)) the changing of which causes an enabled-check. This check includes a check of all containing commands. Only if all of them are performable, the action is also set as performable.

Objects contained in the context are passed to the commands for performance. Thus the context does not only provide the preconditions to define the enabled state of a CommandAction, but its context objects influence also the performance of activated commands.

Extensions of CommandActions are in general not necessary, since the properties are defined by the configuration file *actions.properties*.

### 3.2.3. Context

Instances of the class Context<sup>10</sup> connect GUI elements, CommandActions and commands. GUI elements can be understood as semantic units. Example: a window contains a viewer instance, a menu bar and a toolbar. Each action which happens within this window may have an effect on the enabled state, but also on the way how the tools in the toolbar or the menu items in the menu bar are performed. But the tools or items may also work with the window's objects. Thus the window builds a logical unit in itself, the single elements of which depend on each other regarding their state and their performability.

A context object accompanies a GUI component (as a client property of a JComponent) – the window's RootPane in the example above – and may contain any number of objects which may be determinant for performance and state of activated elements within this component.

Furthermore instances of the class Context inform registered interested parties (ContextChanged-Listener<sup>11</sup>) about the context's changes. This regards changes on the contained objects, but also semantic changes of the context which may be provoked by calling the method „contextChanged()“.

When building up the GUI component CommandActions receive for creation a reference to a context object and register there as ContextChangeListener. With each context changing operation the CommandAction instance checks its state (enabled state) by querying all contained commands about their state (see [3.2.1.Commands](#) „checkQuickly(...)“ method). The state checking is based on the objects contained in the context.

CommandActions are performed in two steps. First a final test of all commands is done (see [#3.2.1.Commands](#) „checkDeeply(Collection)“ method), the result of which must be successful, before the commands are performed by the method „doExecute(Collection)“. At this the objects contained in the context are also passed.

Analogical to the hierarchical component structure of GUI elements their contexts can also be organised hierarchically. In order to be able to decide in more complex situations with several contexts which objects of different

8 com.levigo.util.swing.action.CommandAction

9 com.levigo.util.swing.action.Command

10 com.levigo.util.swing.action.Context

11 com.levigo.util.swing.action.ContextListener



contexts belong to the collection of objects of superior contexts, single contexts may be active or inactive. The contexts' activity state is oriented to the focussing state of the associated GUI elements. As a rule: a context is active, if a contained GUI element without an associated context is active.

Under certain circumstances CommandActions of the parent-component may in regard of state and performance depend on the state of the child-component reflected in its associated context. But since the CommandActions only know the elements of their context, objects of the child-context must be able to become temporarily part of the parent-context. For this contexts may be instantiated in three modes or states of aggregation:

- † **Context.NO\_CHILDREN** Individual context, all GUI child-components own no further context or the elements of a possible child-context are not relevant for CommandActions of the parent-context.
- † **Context.ACTIVE\_CHILD** GUI child-components may have contexts, but only elements of the active child-context are relevant for CommandActions of the parent-context and are temporarily added to this.
- † **Context.ALL\_CHILDREN** GUI child-components may have contexts, elements of all child-contexts are relevant for CommandActions of the parent-context and are temporarily added to this.

For de/registration of child-contexts at the parent-context the class Context provides the following methods:

† **addChildContext(Context)**

Adds the given context as child-context.

† **addToParentsContext()**

Adds the context to its parent-context. The context object is always filed as a client property of the associated GUI component. The parent-context is determined by browsing the component's hierarchy „from bottom to top“ for a parent-context. For this reason it is advised to perform this method correctly so that the component's hierarchy is already determined.

† **removeChildContext(Context)**

Removes the given child-context.

† **removeFromParentsContext()**

Removes the context of its parent-context. As already described in „addToParentsContext“-method, a determined component's hierarchy should be ascertained in order to perform the method correctly.

### 3.3. Configuration files

Configuration files describe appearance, keyboard settings and structure of menus and toolbars of supplied commands. For adaptation of these properties to the integrating environment, in order to add own commands or to create different look & feels for different possible applications, configuration files may be adapted.

Please note that the configuration files are supplied in internationalised versions and are used automatically according to the country code of the operating

system. Accordingly changes of these files should be always done in all versions.

In the following paragraphs functions, contents and the interaction of single configurations are explained in detail. The precise syntax of the files can be found in [8.jadice Integrator API: Syntax description of configuration files](#).

### 3.3.1. **commands.properties**

This configuration creates a mapping between a unique command name, which is used in other configurations as reference, and its realisation, the so called exact class reference.

Commands are created by reflection. In order to avoid instantiation mistakes, the path and class names must be stated correctly.

Further information are in paragraph [8.jadice Integrator API: Syntax description of configuration files](#).

### 3.3.2. **menucomponents.properties**

The menu component configuration defines the structure of menus, submenus, context menus and toolbars.

For this a clear name is given to a structure, e.g. a menu or a toolbar, which can be later used to create an instance of a menu for integration. Actions which should be contained in this structure are determined as a comma separated list of action-command-names. Action-Command-Names must be provided by referring *actions.properties* files.

Like this definitions of further substructures like submenus, context menus and similar may be determined by the menu components configuration.

Further information are in paragraph [8.jadice Integrator API: Syntax description of configuration files](#).

### 3.3.3. **actions.properties**

In this configuration properties of the CommandActions like Tool-Icon, Tooltip Text, Accessor and similar are defined. Beside the definition of the CommandAction's external appearance it is also determined in this configuration which commands should in which order get to performance by releasing the CommandActions.

Clear command terms from referring *commands.properties* configurations are used for the commands' identification.

Further information are in [8.jadice Integrator API: Syntax description of the configuration files](#).

## 4. Class survey

The following survey is limited on a description of classes most important for integrators.

A detailed description of the collaboration of single classes and precise application examples are found in paragraph [5. Typical application examples](#).

Interfaces, classes and methods are additionally documented in a separate API reference.

### 4.1. Viewer

The viewer<sup>12</sup> is one of the central classes of the jadice component architecture. It takes over the administration, handling and displaying of documents and pages. The viewer being a JComponent respectively JavaBean is easy to integrate in own architectures and layouts. It consists of a displaying area and appropriate scroll bars. At the same time the viewer offers from the integrator's point of view an interface of the most important functions for document/page viewing, like

- † scrolling within a document
- † document zoom
- † page zoom
- † document rotation
- † page rotation
- † zoom policy, etc.

A viewer instance is able to display or edit exactly one document. Registered listeners, e.g. the integrating application, are informed about the most important changes on the document, its displaying or on the viewer itself.

This happens by using `PropertyChangeEvents`<sup>13</sup> or `PropertyChangeListener`<sup>14</sup>. Interested users log in as `PropertyChangeListener` at the appropriate viewer instance: qualified (exactly for a defined property) or unqualified (for all properties). For a precise identification of the respective property, property names are declared in the viewer as public constants.

The viewer obtains for displaying a document<sup>15</sup>- reference. By using appropriate „getter“ and „setter“ methods a reference to this document can anytime be obtained or changed. There is no need to pay attention on the document's state or the moment when a document change is taking place. Properties of the class `Document` are described in chapter [4.2. Document](#).

The viewer supports - for an optimal displaying and higher-performance working - different zoom policies with document or page change. For example, newly set documents can always be displayed in the „ZoomToFit“ mode, i.e. the document is optimally adapted in height and width for displaying in the viewer. Other zoom policies are:

---

12 `com.levigo.jadice.Viewer`

13 See also `PropertyChangeListener` and `PropertyChangeSupport` in the Java 2 Platform API Specification

14 See also `PropertyChangeEvent` and `PropertyChangeSupport` in the Java 2 Platform API Specification

15 `com.levigo.jadice.docs.Document`

„**ZoomToWidth**“: adapt document's width optimally in viewer, suitable for documents with pages of same size.

„**ZoomToHeight**“: adapt document's height optimally in viewer, suitable for documents with pages of same size.

„**PageZoomToWidth**“: adapt page's width optimally in viewer, suitable for documents with different page sizes.

„**PageZoomToHeight**“: adapt page's height optimally in viewer, suitable for documents with different page sizes.

„**PageZoomToFit**“: adapt page's height and width optimally in viewer, suitable for documents with different page sizes.

„**ZoomDefault**“: the standard document zoomfactor is used for displaying.

The zoom policy is according to the set policy applied to page or document change and can respectively be queried and set for a viewer instance.

Zoom policies are less relevant than a page's or a document's zoom properties set by the user. As long as a document is referred to in a viewer instance, the viewer memorises page and document specific zoom and rotation settings done by the user. If e.g. the user sets page 3 to 200% page zoom, the viewer will always display page 3 with the zoom factor of 200%, even if it has meanwhile been scrolled to a different document page. With document changes the viewer abandons these render settings and the set zoom policy is used again.

Beyond the mere document viewing the viewer makes it possible to lay self-defined, active layers over the document's displaying. Such layers may render themselves, but may also receive input events like mouse or key events. Integrators may add on this way of page viewing (eventually active) decorations like e.g. a paperclip icon that performs an action on mouse click. In the jadice component structure such layers are called EditPanes.

For further information see [4.15.EditPanes](#).

## 4.2. Document

A document is a format independent container for pages and page layers. Instances of the class Document<sup>16</sup> contain one or more pages which may contain page segments in different layers (see also chart 1 in [2.4.1.The document model](#)).

A document instance offers the possibility to get qualified access to the contained pages and page layers, i.e. it is possible to add, delete and move pages and page layers to an instance of the class document by using public API methods.

All changes of the document's structure are publicised to registered interested parties (DocumentListener<sup>17</sup>). The interface DocumentListener defines an interface for information about changes on a document. Such changes may be e.g. adding/deleting or reasorting of pages, changing of pages, page segments or layers and modifying of the document's status. An according adapter class of the interface DocumentListener is also provided by the API.

<sup>16</sup> com.levigo.jadice.docs.Document

<sup>17</sup> com.levigo.jadice.docs.DocumentListener

In addition to this a name and a specific ResourceLoader<sup>18</sup> may be assigned to each document.

ResourceLoader enable the access on external resources for MO:DCA or AFP-documents. MO:DCA and AFP documents may contain or refer to internal (inline) or external resources. Such resources may be e.g. logos, letter heads or signatures. Access on such external resources of AFP or MO:DCA documents is effected by a ResourceLoader (see [4.8.ResourceLoader](#)).

Examples for creating and loading of documents are found in [5.Typical application examples](#).

### 4.3. Page

A document can contain one or more pages, a page can consist of different page segments (PageSegment<sup>19</sup>), but it is unlike a document visually displayed by the viewer as a page unit .

An instance of the class Page<sup>20</sup> allows access on the contained page segments and holds a reference to the document which contains the page. Beyond this the page offers information about the original, scaled, rotated and the displayed page size as a summarised unit of all page segments.

The according access methods are in detail:

† **getSize():**

original size in base units (converted on 7200dpi)

† **getRotatedSize():**

original size in base units (converted on 7200dpi), zoom factor and rotation included

† **getScaledSize():**

displayed page size (converted on device resolution), zoom factor included, rotation not included

† **getRenderedSize():**

the current page size (converted on device resolution), zoom factor and rotation included

### 4.4. PageSegment

Page segments are part of a page and carry the actual raw data information of the corresponding segment.

PageSegments<sup>21</sup> may thus offer the following information to integrators:

† data format of the raw data, e.g. Tiff or MO:DCA

† original resolution of the raw data

† provided and scaled size of the segment

† page containing the page segment

18 com.levigo.jadice.docs.resource.ResourceLoader

19 com.levigo.jadice.docs.PageSegment

20 com.levigo.jadice.docs.Page

21 com.levigo.jadice.docs.PageSegment

## 4.5. Loader

The Loader<sup>22</sup> is a central class for all document loading processes. Integrators may anytime create an instance by using the default constructor of the class and so fill a new or already existing document synchronously or asynchronously.

A created document can be passed already in its „unfilled“ state to the viewer for displaying. The viewer recognises automatically when the first page respectively the first page segment in a page has been loaded and displays it, though the loading process in the background has not been finished yet. This is an advantage in particular with slow network connections and large documents.

A loader can be used for one or more loading processes and provides for this purpose different loading methods. The different loading methods allow the integrators the simple loading process of a document, but also loading processes in a certain layer, from a particular page and/or in a particular format.

Loading processes normally proceed asynchronously in order to enable e.g. a faster reaction of the GUI without having to wait till the eventually longer lasting loading of the document has been finished. However, under certain circumstances it is of advantage to let proceed loading processes synchronously. If the integrator wants e.g. to attach in a document several image files in a specified sequence, the loader can be changed over to synchronous processing.

Notices about the progress of the loading process are transmitted to registered interested parties (LoadListener<sup>23</sup>) by activating the „loadStateChanged“ method. Normally these notices are activated in the current load-thread. In connection with GUI components it may be of advantage, though, to have notices activated on the Event Dispatch Thread. If this is requested, the loader may be simply changed by a corresponding method.

MO:DCA and AFP documents may contain internal (inline) or external resources. For the loading of external resources, which are often in different locations than the document raw data, ResourceLoader<sup>24</sup> may be used. They support the loading process by providing access on external resources. For this purpose ResourceLoaders should be set in the loader or in the document before the loading process starts. ResourceLoaders set in a document are particularly used for loading processes into this document. If the document has not indicated a ResourceLoader of its own, it uses the ResourceLoader registered in the loader. ResourceLoaders registered in the loader are kept static in order to enable a simple application-wide access and to may be used by other loader-instances.

For more details about ResourceLoaders see [4.8.ResourceLoader](#).

Detailed information about Loaders, LoadListeners, ResourceLoaders as well as their class and method signatures you will find in the jadice API documentation. An example for the loading of a document is described in [5.Typical application examples](#) beschrieben.

<sup>22</sup> com.levigo.jadice.docs.resource.Loader

<sup>23</sup> com.levigo.jadice.docs.resource.LoadListener

<sup>24</sup> com.levigo.jadice.docs.resource.ResourceLoader

## 4.6. FormatInfo and FormatFile

The API of the jadice document platform offers for each supported image format a format class which supports loading processes into the represented format. The naming of these classes follows a predetermined naming convention.

Format supporting classes for loading and saving:

*FormatNameFormatInfo* -> e.g. `TIFFFormatInfo`<sup>25</sup>

*FormatNameFile* -> e.g. `TIFFFile`<sup>26</sup>

*FormatNameFormatInfo* serves loading processes of a format

*FormatNameFile* serves saving processes.

Without any indication of a *FormatNameFormatInfo* instance the loader tries to define the format of the data to be loaded before the actual loading process starts. For this it is first checked which formats are provided as Format Services in the class path. Subsequently each *FormatInfo* instance found is requested, if it fits to the data stream to be loaded. If such a *FormatInfo* instance is found, the data stream's format is detected and the actual loading process starts.

But if the loader gets an instance of *FormatNameFormatInfo* for a loading process, the defining of the format may be omitted and the data's loading may begin immediately. Thus loading processes may be accelerated and format confusion avoided. Please note that if a *FormatNameFormatInfo* not fitting to the data stream is indicated, the loading process is cancelled and a search for alternative *FormatInfo* instances will not take place.

An example:

jadice supports annotations compatible to IBM ContentManager. These annotations are kept as MO:DCA structures in the archive. If annotations are loaded without specifying a *FormatNameFormatInfo*, during the loading process a MO:DCA-document and annotations as additional document information might be mixed-up. More about this in chapter [4.11.ImagePlusAnnotationFormatInfo](#) and [4.12.ImagePlusAnnotationFile](#).

Additionally FileNet-compatible annotations are supported. (Further information are found as part of the distribution in the manual *Annotations: Load – Save – Edit*)

## 4.7. LoadListener

The interface *LoadListener* is available for integrators, if they are interested in the flow of document loading processes. The loader informs registered *LoadListeners* by calling the „loadStateChanged“ method about the loading progress. The loading state is described by an instance of a *LoadEvents*<sup>27</sup>.

A *LoadEvent* contains the following information:

- † the relevant document
- † the relevant page
- † which kind of event it is (page loaded, loading process finished, loading error) and

<sup>25</sup> `com.levigo.jadice.formats.tiff.TIFFFormatInfo`

<sup>26</sup> `com.levigo.jadice.formats.tiff.TIFFFile`

<sup>27</sup> `com.levigo.jadice.docs.resource.LoadEvent`

† the event's origin. Mostly the corresponding *FormatNameFormatInfo* or loader instance in which the respective event has occurred.

In former jadice versions (2.x) it was necessary to wait for the end of a loading process, before a document could be set by means of a *LoadListener* into the viewer for displaying. This is not necessary any more.

So *LoadListeners* have no functional background in the present version. They serve merely informing purposes.

*LoadListeners* may be logged on in the loader in different implementations and instances and may be used for multiple loading processes without having to register again.

Notifications to *LoadListeners* are normally forwarded on the current load-thread. Under certain circumstances, e.g. in the interaction with GUI components, it may be interesting to maintain the call of the „loadStateChanged“ on the event dispatch thread. In such a case integrators can make the loader deviate all messages on the EDT by using the „**setSendNotificationsOnEDT**“ method.

#### 4.8. ResourceLoader

*ResourceLoader* classes are used for AFP or MO:DCA documents in which internal (inline) and external resources may be integrated within the document.

External resources are reloaded during the loading process of the document by means of *ResourceLoaders*. Resources are referenced in the document by a name and may be read by different implementations of *ResourceLoaders* as *InputStream*.

The class *ResourceLoader* represents the interface for implementation of the classes described in the following, but also for own implementations. The different *ResourceLoaders* of the jadice package are described in the following paragraphs.



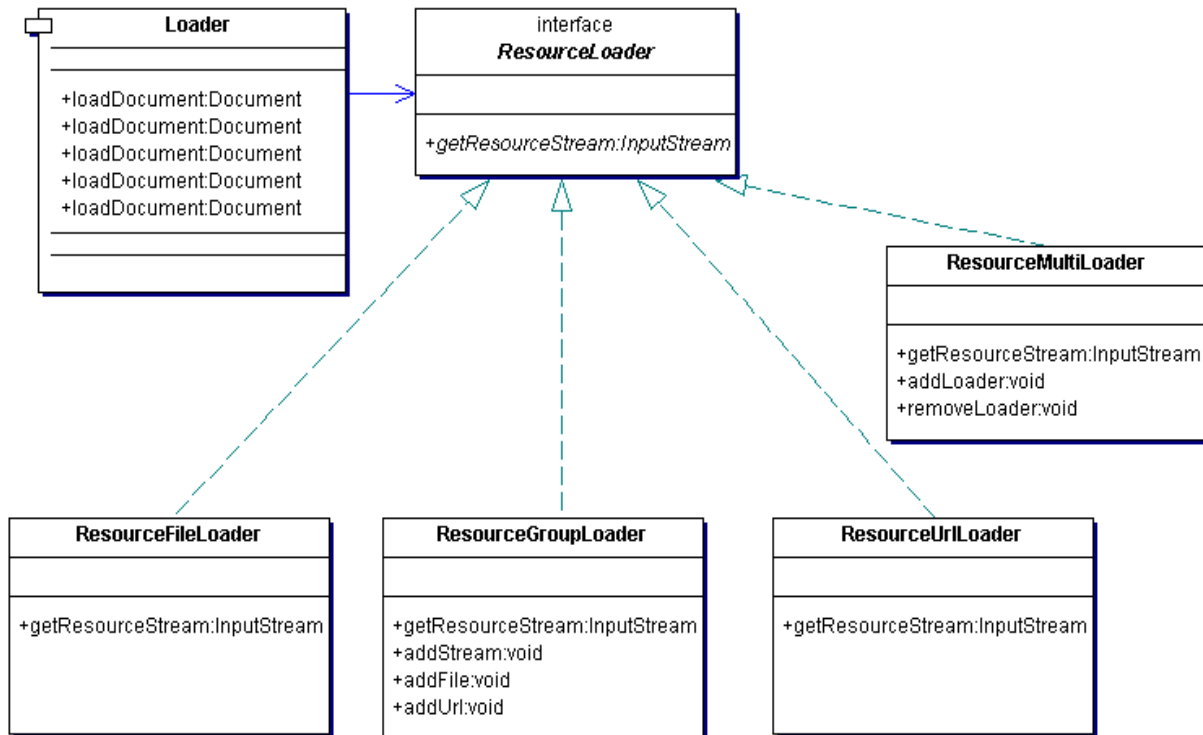


Chart 2 - Class diagram Loader / ResourceLoader

The ResourceLoader interface defines the method „**getResourceStream(String resourceName, String defaultExt)**“ which returns an InputStream of the resource data, if the searched resource is found and if an access is possible.

The ResourceLoader to be used during the loading process must be disclosed either to the document or to the loader by the method „**setResourceLoader(ResourceLoader resourceLoader)**“. On this way AFP documents get the possibility to access during the loading a corresponding ResourceLoader and consequently the correct resource.

In doing so keep this in mind:

- † ResourceLoader in the document:
  - † is prioritised to possibly registered ResourceLoader of the loader
  - † is a document's property, i.e. each document maintains a reference of its own on a possibly registered ResourceLoader.
- † ResourceLoader in the loader:
  - † is a loader's static property, thus easy application-wide access on the set ResourceLoader and validity for all loader instances.
  - † ResourceLoaders of a document are prioritised to registered ResourceLoader of the loader.

[Chart 2](#) gives a survey of the ResourceLoaders' class hierarchy. The ResourceLoader is an interface which integrators may use for their own implementations. The classes described in the following represent different

useful realisations of this interface which, if necessary, may be set joined in a MultiLoader, a loader instance or a document.

#### 4.8.1. ResourceFileLoader

The ResourceFileLoader<sup>28</sup> supplies InputStreams on the basis of file resources (files of the local file system or network resources visible by the file system).

A list of search paths (directories in the file system) for file resources is given to the constructor. The search paths are to be separated by the operating system specific constant **java.io.File.pathSeparator**.

By calling the method **getResourceStream(String resourceName, String defaultExt)** an InputStream is created using the transferred parameters, in which resourceName represents the file name without the preceding path and without extension and defaultExt represents the file name's extension.

#### 4.8.2. ResourceURLLoader

Resources accessible by URLs may be used by the class ResourceUrlLoader<sup>29</sup>.

The operating mode is to a large extent identical with that of the class ResourceFileLoader. Instead of the search path leading to the file a list of comma separated URLs is given to the constructor. Here it is also possible to transfer a single URL instead of the list.

The InputStream is as defined in the interface returned by **getResourceStream(String resourceName, String defaultExt)**.

#### 4.8.3. ResourceGroupLoader

External resources of AFP- or MO:DCA documents may be separately or in resource groups available. Resource groups are the summary of different resources to a data stream, in which contained resources may be maintained as InputStream by the ResourceLoader. The actual source of a resource group may either be saved as a file in the file system or in a document management / archive system. A document can get one or more resources from the very resource group or from different resource groups.

The ResourceGroupLoader<sup>30</sup> offers for instantiation two different constructors:

- † **ResourceGroupLoader()**: creates an „empty“ ResourceGroupLoader to which different resources may be added by using the different addXXX methods (see below).
- † **ResourceGroupLoader(InputStream is)**: creates a ResourceGroupLoader which is initialised with an InputStream to a resource.

Further resource groups may be added by using different addXXX methods.

- † **addFile(String fileName)**: adds a file resource

<sup>28</sup> com.levigo.jadice.docs.resource.ResourceFileLoader

<sup>29</sup> com.levigo.jadice.docs.resource.ResourceUrlLoader

<sup>30</sup> com.levigo.jadice.formats.modca.ResourceGroupLoader

† **addStream(InputStream is)**: adds a resource which is already available as InputStream

† **addUrl(String urlName)**: adds a URL resource

#### 4.8.4. ResourceMultiLoader

The class ResourceMultiLoader<sup>31</sup> offers the most flexible implementation of the ResourceLoader-Interface. It behaves like a simple ResourceLoader, but it unites in itself multiple registered ResourceLoaders. All implementations described before may be by using the methods

† **addLoader(ResourceLoader loader)** added or

† **removeLoader(ResourceLoader loader)** removed.

With that an application is able to integrate all available resources and corresponding ResourceLoaders into a ResourceLoader, to register in the loader or in the document and to access as usual per „getResourceStream(...)“ on the required resource InputStream, without having to care about where and how the actual resource is made available.

An example for the creation and registration of ResourceLoaders is provided in [5. Typical application examples](#).

#### 4.9. SeekableInputStreams

For an efficient and memory saving processing of large document data amounts the viewer tries, if allowed by the image format, to read, to process and to buffer dynamically only document data required for the current page displaying instead of keeping all image data completely in the memory.

This procedure requires data streams, whose file index may be positioned arbitrarily.

jadice viewer uses to this end SeekableInputStreams. The class SeekableInputStream<sup>32</sup> is an abstract basis class which extends the InputStream<sup>33</sup> by the following methods:

† seek(int) position the file index

† length() File size in bytes

† getFilePointer() position where file index is positioned

<sup>31</sup> com.levigo.jadice.docs.resource.ResourceMultiLoader

<sup>32</sup> com.levigo.jadice.io.SeekableInputStream

<sup>33</sup> java.io.InputStream

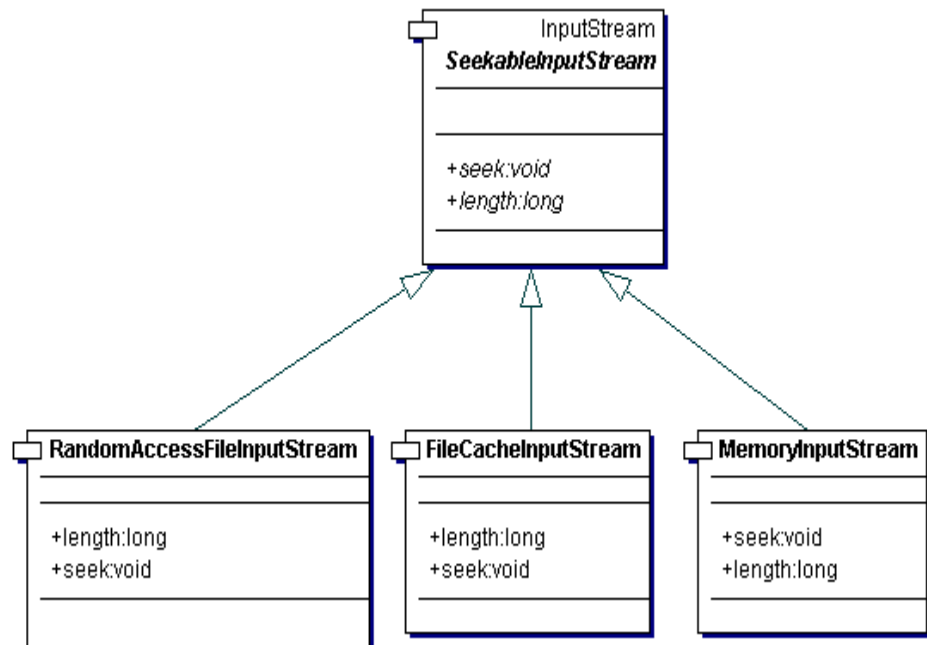


Chart 3 - class diagram of the most common SeekableInputStreams

[Chart 3](#) offers a survey of the most common implementations of the interface SeekableInputStream provided by the jadice API.

Using SeekableInputStreams is easy for integrators. jadice provides for certain types of data streams different sorts of SeekableInputStreams which wrap the actual data stream and may be delivered to the loader in the required „loadDocument“ method. Nothing more to do for the integrator.

The most common SeekableInputStreams are shortly described in the following paragraphs.

#### 4.9.1. RandomAccessInputStream

RandomAccessInputStream<sup>34</sup> represents a SeekableInputStream for locally available image data (i.e. as file in filesystem).

This class offers a constructor with a parameterised file (a file object of the image file) for instantiation. This file is opened with a particular FileInputStream which is kept for the following reading processes. During the processing in jadice the FileInputStream remains open to the document file and a file index is positioned within this file. It should be noticed that the source file must not be changed as long as the document is opened in the viewer. This leads, e.g. when using Windows, eventually to an access locking on the respective file.

#### 4.9.2. FileCacheInputStream

To process image data in the viewer a temporary file is created in which the read data are stored. After having used them, but at the latest on the next start

<sup>34</sup> com.levigo.jadice.io.RandomAccessInputStream

of the viewer, the temporary files, that are not required anymore, are deleted. The temporary files are stored in the temporary directory predetermined by the system, if the viewer is not configured differently.

For instantiation this class offers a constructor which takes in a given `InputStream` and creates a work file in the defined temporary directory. This temporary file is removed after use.

Another constructor offers the possibility to indicate apart from the `InputStream` a temporary directory and a flag showing if the temporary file is to be deleted after use.

### 4.9.3. `MemoryInputStream`

Image data are buffered in the central memory.

Similar to the `FileCacheInputStream` this class offers a constructor that takes in a given `InputStream` and keeps it for reading processes. A second constructor allows additionally an indication of the block size in which data are to be buffered in the central memory.

This sort of internal data administration is often used in context with applets. An advantage of this data storage is an extremely fast access on document raw data. However, the higher memory requirements due to the data storage in the central memory prove to be rather disadvantageous.

## 4.10. Annotation

Annotations are additional information to a document which may have the form of textual notes or graphic objects for highlighting. Annotations are displayed in an own layer above the document. Thus annotations occur to the user as an optical unit with the document. As additional document information annotations do not belong physically to the actual document. Changes on documents do not change the actual document.

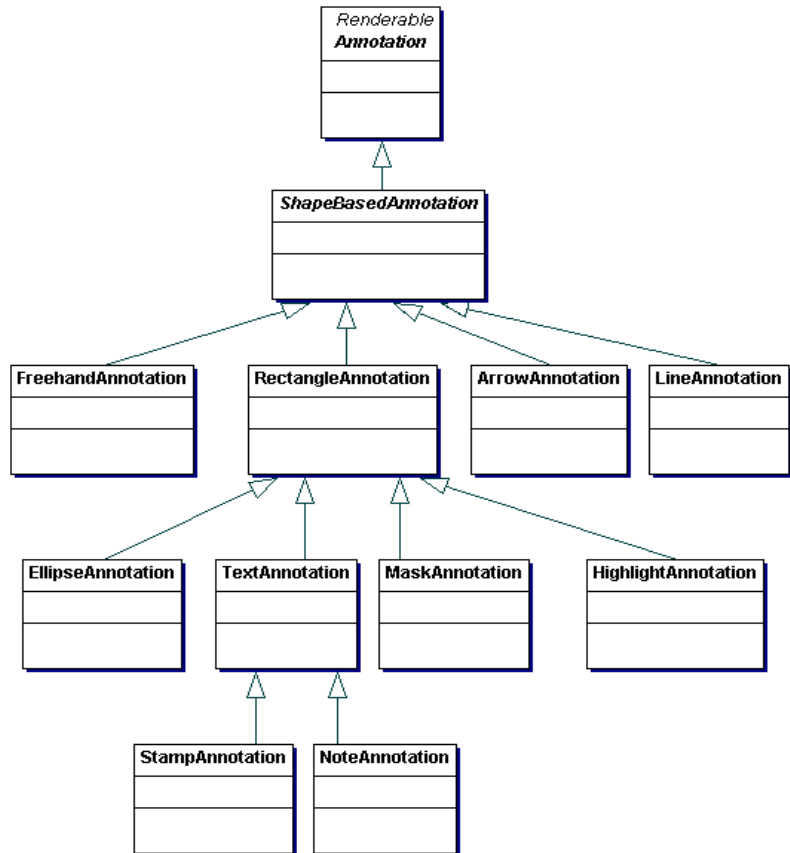


Chart 4 - Inheritance hierarchy annotations– rough survey

Provided types are:

† NOTE:

textual note, displayed as yellow post-it

† HIGHLIGHT:

highlighting, displayed as a filled and transparent rectangle

† MASK:

masking of parts of a displayed document by a filled, not transparent rectangle

† ARROW:

a pointer to a particular part of the document displayed as an arrow

† ELLIPSE:

a not filled ellipse, appropriate for framing a certain area, e.g. an amount, a date or a name

† RECTANGLE:

a not filled rectangle. Like the ellipse, appropriate for framing a certain area, e.g. an amount, a date or a name

† LINE:

a line e.g. to underline a certain word or number

† TEXT:

a textual note placed directly on the page with a transparent background, not to be saved

† FREEHAND:

a freehand draft. When using this type a polygon is created due to the mouse movement. Appropriate for highlighting of unequal areas which can't be framed by an ellipse or a rectangle

† STAMP:

a „stamp“ with transparent background, coloured frame, able to be rotated with a coloured textual content

[Chart 4](#) shows a rough survey of the annotations' class architecture. The class `Annotation`<sup>35</sup> is an abstract class which represents the basis of all annotations. `ShapeBasedAnnotation`<sup>36</sup> represents a further abstraction layer which forms the basis of all annotations based on geometrical forms (`Shape`<sup>37</sup>). As a concrete annotation this class is not relevant, for integrators, however, who want to define their own annotations it represents a useful basis.

With jadice viewer the main emphasis was put on a logical inheritance due to the displaying and processing properties of single annotation types. This grants on the one hand consistency and the avoiding of redundancies, on the other hand a flexible exchange of the annotation support for different archive systems or completely self-defined annotations is possible.

With user interaction annotations are administered by the viewer component. It allows users to create different types of annotations, to delete them and to change their properties. The type of an annotation to be created depends on the viewer's mode at the creation time. This mode is set indirectly by the user activating a corresponding tool of the annotation toolbar or by the integrator using the method `AnnotationCreatorPane`<sup>38</sup>.`setAnnotationMode(int)`.

If integrators like to direct the annotation administration themselves or to edit the programming of annotations, they find further information on this subject in the jadice viewer annotation documentation (jadice Viewer: Annotations – loading, saving, editing).

In order to load or to save annotations in the ImagePlus format, the classes `ImagePlusAnnotationFormatInfo`<sup>39</sup> and `ImagePlusAnnotationFile`<sup>40</sup> may be used. More about this is in the following chapter.

Respectively FileNet and FileNet P8 annotations may be loaded or saved with the aid of the classes `FileNetAnnotationFormatInfo`<sup>41</sup>, `FileNetAnnotationFile`, `FileNetP8AnnotationFormatInfo`<sup>42</sup> and `FileNetP8AnnotationFormatInfo`<sup>43</sup>. Further information about the use of FileNet annotations is found in the jadice viewer annotation documentation (jadice Viewer: Annotations – loading, saving, editing).

35 `com.levigo.jadice.annotation.Annotation`

36 `com.levigo.jadice.annotation.ShapeBasedAnnotation`

37 `java.awt.Shape`

38 `com.levigo.jadice.annotation.AnnotationCreatorPane`

39 `com.levigo.jadice.formats.annoiplus.ImagePlusAnnotationFormatInfo`

40 `com.levigo.jadice.formats.annoiplus.ImagePlusAnnotationFile`

41 `com.levigo.jadice.formats.annofilenet.FileNetAnnotationFormatInfo`

42 `com.levigo.jadice.formats.annofilenetp8.FileNetP8AnnotationFormatInfo`

43 `com.levigo.jadice.formats.annofilenetp8.FileNetP8AnnotationFile`



*Note:*

The loading and saving of annotations must be done explicitly and is e.g. **not** done automatically at the document's loading.

*Note:*

Not all archive systems support all annotation types provided by jadice. So e.g. FileNet does not know any ellipse or mask annotations; IBM Content Manager ImagePlus® for OS/390 or AS400 supports only highlight, note and mask annotations, whereas IBM ContentManager for Multiplatforms allows all types of annotations.

By a simple textual change on the configuration integrators may activate or deactivate the requested annotation types.

For further information about this subject see chapter [5.4.2.2. Adaptation of the menu or toolbar structure](#) or the jadice viewer annotation documentation (jadice Viewer: Annotations – loading, saving, editing).

#### 4.10.1. Changes on annotations

Changes on annotations are propagated with the aid of the interface AnnotationListener<sup>44</sup>. Applications that like to be informed about property changes of existing annotations like size, position, colour, etc. but also about the creating or removing of annotations may register as AnnotationListeners in the class AnnotationEventcaster<sup>45</sup>. A logon in the class AnnotationEventcaster is always carried out globally for all annotation changes within the current application independently of document or viewer instances. That means that a registered AnnotationListener is informed within the whole application about changes on annotations by the method

- **annotationChanged(AnnotationEvent e)**.

The supplied AnnotationEvent<sup>46</sup> provides information which annotation was changed in which way. For this the following methods of the class AnnotationEvent may be used:

- † **getAnnotation()** provides a reference to the changed annotation. With aid of this annotation access is possible on the enclosing annotation page segment, the corresponding page and the document.
- † **getEventType()** indicates kind of change, e.g. size, position, colour, selection state, etc.
- † **getNewValue()** indicates new property value of annotation, e.g. new colour.
- † **getOldValue()** indicates original property value, e.g. previous colour.

44 com.levigo.jadice.annotation.AnnotationListener

45 com.levigo.jadice.annotation.AnnotationEventcaster

46 com.levigo.jadice.annotation.AnnotationEvent



### 4.11. ImagePlusAnnotationFormatInfo

In order to avoid a mix-up of IBM VisualInfo/ImagePlus-compatible annotations with MO:DCA documents during the loading process, it is necessary to pass an instance of the class `ImagePlusAnnotationFormatInfo` to the loader in the `loadDocument()` method.

`ImagePlusAnnotationFormatInfo` is a format information for IBM VisualInfo/ImagePlus compatible annotations. This class ensures during the loading process that the loaded data are not interpreted as an independent MO:DCA document, but considered as additional document information, just as annotations, and that they are administered and displayed in a layer of their own „above“ the document.

An example of annotation loading is in chapter [5. Typical application examples](#).

*Note:*

Up to IBM ContentManager Vs. 7.x annotation properties were resolution independent. This has changed with version 8.x. With that up to version 7.x annotations could be loaded independently of the image document. From version 8.x on integrators should load annotations only after the document's loading process has finished, otherwise errors in the annotations' positioning and size are expected.

*Note:*

Like ImagePlus compatible annotations FileNet annotations may also be displayed and edited by the viewer. The respective `FormatInfo` class for FileNet annotations is `FileNetAnnotationFormatInfo`.

### 4.12. ImagePlusAnnotationFile

`ImagePlusAnnotationFile` is used for saving annotations in the ImagePlus format. In order to get access on the annotations to be saved, a document with annotations that are to be saved is passed in the constructor to the `ImagePlusAnnotationFile`.

Just like the loader provides different loading methods, each `FormatNameFile` instance offers also different loading methods for a general or qualified (e.g. only certain pages) saving. The easiest way to save all annotations of a document into an `OutputStream` is by calling the method „**save(OutputStream)**“.

See the jadice viewer API documentation for more detailed information.

### 4.13. FileNet and FileNetP8 annotations

Like ImagePlus compatible annotations FileNet and FileNetP8 annotations may also be displayed and edited by the viewer. The respective `FormatFile` class for FileNet annotations is `FileNetAnnotationFormatFile` respectively `FileNetP8AnnotationFormatFile`.

Further information about using FileNet and FileNetP8 annotations may be found in the jadice distribution in the annotation documentation (jadice viewer: Annotations – Loading, Saving, Editing).

#### 4.14. RenderContext

On some points of this document it is referred to the class `RenderContext`<sup>47</sup>.

Normally integrators hardly work directly with instances of this class, but the properties and tasks of the class `RenderContext` are still presented at this point for a better understanding.

For the displaying of a page no images are created in the jadice viewer, but the pages render themselves including all their segments into the provided graphic contexts e.g. display, printer, etc.

Such a rendering process is always accompanied by an instance of the class `RenderContext`. `RenderContext` encapsulates different displaying features which determine stringently the rendering process and thus also the displaying properties.

The displaying attributes are divided into direct attributes like zoom, rotation or similar, and `ProcessingSettings`<sup>48</sup>. Direct-attributes may be requested directly and also changed by an instance of the class `RenderContext`.

`ProcessingSettings` are attributes of a special nature which summarize and describe displaying properties of a very particular type, e.g. the `AnnotationRenderSettings`<sup>49</sup>. 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.

Further information may be taken from the API documentation and the jadice Viewer Annotation Documentation.

In addition to this the `RenderContext` provides affine transformations for an easy conversion between the document's and the device's coordinate system.

As already mentioned in chapter [2.4.1. The document model](#) the pages contained in documents may get their content from multiple data streams or data formats, but even the pages themselves may consist of data of different data streams. Since these image data may be of different formats and different resolution, the viewer maintains internally a document coordinate system and transforms only for displaying into the particular device coordinates. According to this page segments administrate their data in document coordinates. A transformation in consideration of the zoom factor and the rotation only takes place during the rendering process into a provided graphic context.

#### 4.15. EditPanels

Jadice viewer offers the possibility to extend the viewer's functionality in a flexible way. For this document-independent displaying layers, so called `EditPanels` which, similar to page segments, overlay the document transparently, may be added to the viewer. These layers may influence the documents' displaying, but also receive input events and activate event-directed user

<sup>47</sup> `com.levigo.jadice.docs.RenderContext`

<sup>48</sup> `com.levigo.jadice.docs.ProcessingSettings`

<sup>49</sup> `com.levigo.jadice.annotation.AnnotationRenderSettings`

interactions. In contrast to page segments, which are part of a page or a document, EditPane instances do not belong to a page, but are document independent displaying layers of a viewer.

The basis class of all EditPanes is the AbstractEditPane<sup>50</sup> which offers amongst others the following methods for extension:

† **render(...)**

Own implementations may overwrite this method in order to place additional hints on the page. These may be e.g. page decorations, text, signs, icons and much more. The method gets as parameter a Graphics Context (Graphics2D), a RenderControls<sup>51</sup> (RenderContext with the viewer's current displaying properties) and a RenderObserver. The Graphics Object may be used to display the required object or text. RenderContext supplies eventually needed information about zoom, rotation or similar. If a RenderObserver has been passed, it may be used – if necessary - to activate asynchronously a repaint in so called dirty-regions.

† **getEditEventListener()**

EditPanes can receive mouse or keyboard InputEvents. For their processing by EditPanes this method should return an implementation of an EditEventListener<sup>52</sup>. An EditEventListener is an interface which provides different methods for the processing of mouse or keyboard events. A corresponding adapter class is provided by EditEventAdapter<sup>53</sup>.

EditPanes suit e.g. to display integration-specific annotations on a page in any form and eventually to react actively on InputEvents.

Activating an EditPane is done by registering on the viewer. For this purpose the viewer offers the methods:

† **addEditPane(anEditPane)**

† **removeEditPane(anEditPane)**

A list of all EditPanes registered in a viewer instance may be received by means of the viewer method:

† **Collection getEditPanes()**

*An application example:*

*The integrating application has got additional information to a page. This is to be visualised for the user by a „paper clip“ icon on the page. If the user clicks on the icon, a window with the additional information is to pop up.*

*This can be easily realised by an EditPane. The EditPane renders, if additional information are available, a „paper clip“ icon in its „render“ method. The related EditEventListener reacts on mouse events. After checking if the mouse click has taken place on the icon, the window displaying the additional information is opened.*

50 com.levigo.jadice.AbstractEditPane

51 com.levigo.jadice.docs.RenderControls

52 com.levigo.jadice.EditEventListener

53 com.levigo.jadice.EditEventAdapter

Another typical application example for an EditPane is the class HoverLens. Further information can be taken from [4.2.2.1.HoverLens](#).

## 4.16. BasicJadicePanel

The easiest way how to integrate a viewer is offered by the class BasicJadicePanel<sup>54</sup>. BasicJadicePanel offers a complete, fully functional viewer sight and is as JPanel<sup>55</sup> easy to embed into the user interface of integrating applications.

This class contains:

- † a viewer instance
- † a toolbar with the most important displaying and editing possibilities
- † an annotation toolbar for creating annotations (to be folded in and out)
- † a status bar
- † a context menu
- † (optional) a menu bar

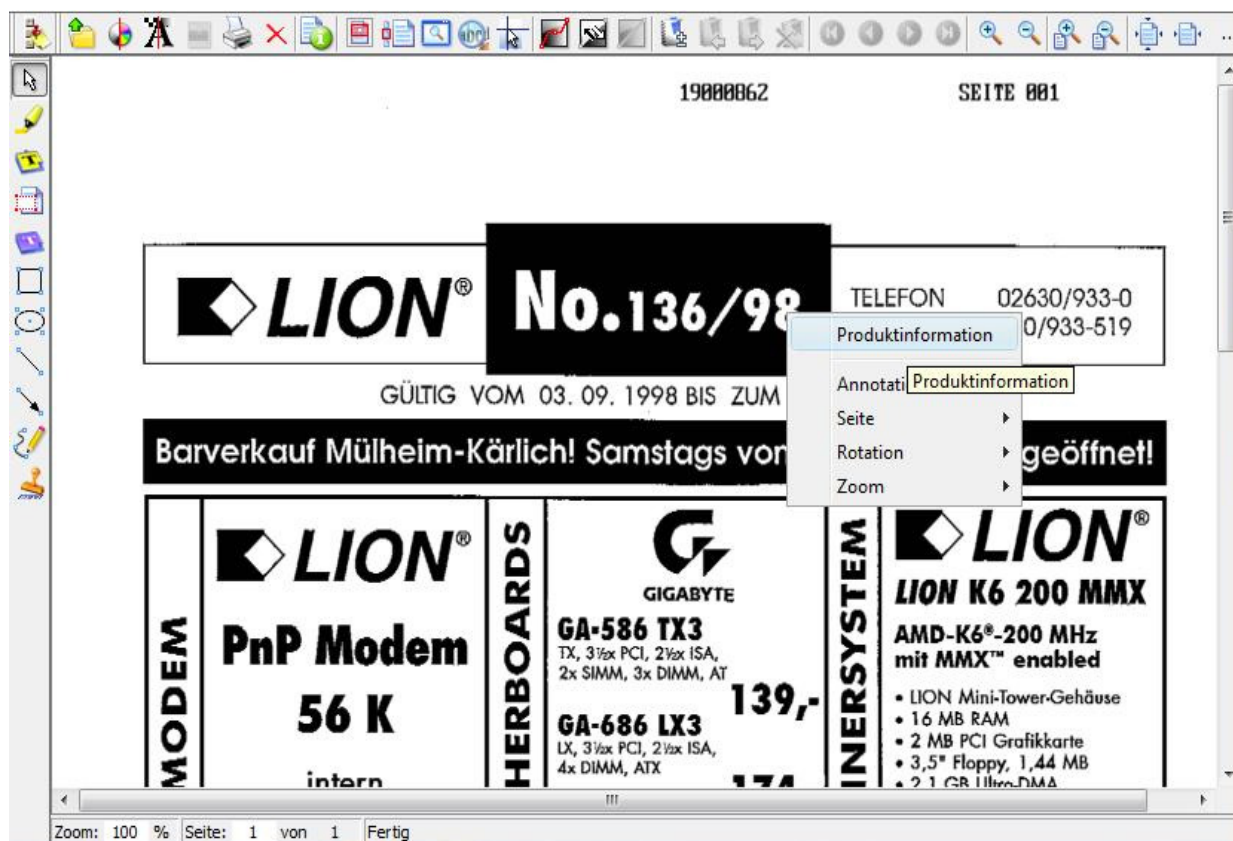


Chart5 - BasicJadicePanel

<sup>54</sup> com.levigo.jadice.gui.BasicJadicePanel

<sup>55</sup> javax.swing.JPanel

Menus and toolbars are based on the new action and command concept of the viewer. Accordingly no further programming effort is necessary for the changing or adapting of single tools respectively of the structure of menus or toolbars. Only the required changes in the configuration files *menucomponents.properties* and *actions.properties* must be done. Own tools may be integrated as well.

More detailed information are in [3.3.Configuration files](#), [5.4.Actions-Commands-Context](#) and [8.jadice Integrator API: Syntax description of the Configuration files](#).

BasicJadicePanel is an extension of the class AbstractJadicePanel<sup>56</sup>. AbstractJadicePanel only unites in itself a viewer and an interacting status bar. This class leaves the implementation of toolbars and menus to its descendants. If integrators don't want to use the viewer's action and command concept, it may be inherited directly of this class – but then toolbars and menus have to be created by the integrators themselves.

As a further aid how toolbars and menubars may be created, but also as a basis for own developments, the class BasicJadicePanel is also provided in the distribution as sourcecode in the example-src directory.

## 4.17. AddOns

Beside the displaying of documents the jadice document platform offers some helpful and useful extensions like document or page survey, bookmarks and similar which may be used in interaction with a viewer instance.

In the following all these classes are called AddOns and are described in the chapters [4.18.JadiceBookmark](#) to [4.23.GradationCurveControl](#).

Properties belonging to all AddOns are summarised in this very chapter.

### 4.17.1. Creation

An AddOn interacts always with a viewer instance respectively a document or the pages contained within. For this reason a viewer instance must always be associated to an AddOn. This may already happen at the creation of an AddOn or later by using the provided method „**setViewer(Viewer)**“. With that each AddOn can always be bound to a new viewer instance or to put it in other words: an AddOn can serve different viewers, but only one at the same time.

Available constructors:

- † Default- constructor
- † constructor with a viewer instance

### 4.17.2. Call by commands

The jadice viewer package provides in general two implementations of commands for the displaying of the respective AddOn:

- † **ToggleAddOnName**

---

<sup>56</sup> com.levigo.jadice.gui.AbstractJadicePanel

Example: ToggleSorter. Shows/hides the respective AddOn embedded in a JFrame

† **ToogleInternalAddOnName**

for use in MDI environments: Shows/hides the respective AddOn embedded in a JInternalFrame.

For own implementations these commands can be overwritten or own commands can be registered.

Further information concerning commands are in chapter [3.The jadice Integrator API](#), [5.4.Actions-Commands-Context](#) and [8.jadice Integrator API: Syntax description of configuration files](#).

### 4.17.3. Integration in different environments

If a viewer instance is associated to an AddOn, relevant changes for this AddOn are recognised and accordingly updated. Bookmarks for example update independently their page number, if pages of a document are resorted within the viewer.

If AddOns are applied in an environment with multiple viewer instances, they should be informed by the method „**setViewer(Viewer)**“ about the viewer active at that time in order to be able to update themselves correctly.

For an easy integration all AddOns are descendants of javax.swing.JComponent. In addition to this jadice package offers corresponding JFrames or JinternalFrames.

† **AddOnNameFrame**

as JFrame, with access method on the embedded AddOn and a „setViewer(Viewer)“ method which allows a switching between viewer instances.

† **AddOnNameInternalFrame**<sup>57</sup>

as JInternalFrame, with access method on the embedded AddOn and a „setViewer(Viewer)“ method which allows a switching between viewer instances.

### 4.18. JadiceBookmark

Bookmarks are useful devices, if preferably worked with documents containing multiple pages and if certain pages are to be displayed without long searching. In such a case these pages may be bookmarked.

For a comfortable control of markers set within the document, instances of the class JadiceBookmark<sup>58</sup> are used in the jadice viewer. This class represents a bookmark within the jadice package and can be set as a marker to a particular page. In addition to this displaying properties - current at the time of the bookmark's creation – of the page like rotation, zoom or similar are also administrated. When activating the bookmark at a later date the corresponding page is displayed again according to its displaying properties saved in the mark.

---

<sup>57</sup>

<sup>58</sup> com.levigo.jadice.addon.bookmarks.JadiceBookmark

A bookmark carries the following properties:

- † page number
- † page
- † document
- † rotation
- † zoom factor
- † Page position in viewer (Pan-Point), useful for displaying of pages which are larger than those of the viewer. The page view is automatically scrolled by the Pan Point e.g. in the right corner at the bottom.
- † A textual description of the marker to be identified by the user

The programme internal use and creation of bookmarks is specified in chapter [4.19.DocumentBookmarkHandler](#).

The jadice package offers to the end user commands for a direct control and administration of page markers:

† **BookmarkToggleCommand**<sup>59</sup>

This command creates/removes a bookmark on/from the current page. If a bookmark exists on the current page, it is selected. The activating of the command deletes the according bookmark. Otherwise the command is unselected and the activating creates a bookmark with the current settings like page number, rotation, zoom factor and pan-point.

† **BookmarkBrowsingCommand**<sup>60</sup>

With this command it may be scrolled between existing bookmarks. The direction, if it should be scrolled to the preceding or the next bookmark of the current document, may be defined as a command parameter. Predefined in the jadice delivery version are:

† **NextBookmark and PrevBookmark**<sup>61</sup>

† **BookmarkRemovalCommand**<sup>62</sup>

According to configuration this command deletes all bookmarks of a particular document or all existing bookmarks. Already predefined in the jadice delivery version are:

† **RemoveBookmarks and RemoveAllBookmarks**<sup>63</sup>

## 4.19. DocumentBookmarkHandler

The central class for programme internal editing and administrating of bookmarks is the DocumentBookmarkHandler<sup>64</sup>. An instance of this class may be received by the static method „**getInstance()**“.

59 com.levigo.jadice.addon.bookmarks.BookmarkToggleCommand

60 com.levigo.jadice.addon.bookmarks.BookmarkBrowsingCommand

61 In com.levigo.jadice.resources.properties.commands.properties

62 com.levigo.jadice.addon.bookmarks.BookmarkRemovalCommand

63 In com.levigo.jadice.resources.properties.commands.properties

64 com.levigo.jadice.addon.bookmarks.DocumentBookmarkHandler

Integrators can use this class in order to load from a property object already existing bookmarks before their further editing or to save it in a property object after the editing.

Beyond this the DocumentBookmarkHandler offers different methods in order to carry out the bookmark administration in a totally programme internal way:

- † add bookmark
- † delete single bookmark
- † delete all bookmarks of a page
- † delete all bookmarks of a document
- † edit bookmark
- † activate bookmark (-> a given viewer displays the corresponding page according to bookmark settings)
- † access on
  - † all bookmarks
  - † all bookmarks of a document
  - † all bookmarks of a page
  - † number of bookmarks
  - † query, if bookmarks have been changed
- † loading
- † saving

Under certain circumstances it may be of advantage to get informed about changes on bookmarks. For this purpose integrators may register implementations of the interface `BookmarkListener`<sup>65</sup> at the `DocumentBookmarkHandler`.

Using the `DocumentBookmarkHandler` makes it much easier for integrators to edit bookmarks, since administration and synchronisation are already part of the handler and need not to be realised by the integrator.

## 4.20. PageSorter

Jadice is able to display a document's pages minimised in form of thumbnails. Using the mouse one or more pages may be selected and moved by „drag and drop“. The page order is automatically adapted in the viewer. The page displayed in the viewer as well as all selected thumbnails are colour-highlighted. Double-clicking on a `ThumbnailPanel` activates this page in the viewer.

If the `PageSorter` is to be used as a mere page survey of a document, the sorting functionality may be switched off by the method **„setSortingEnabled(boolean)“**.

The `PageSorter`'s functionality is provided by the class `PageSorter`<sup>66</sup> which being a `JComponent` can very easily be integrated in the user interface of the integrating application.

Being a viewer `AddOn` the comments in [4.17.AddOns](#) apply for the `PageSorter`.

<sup>65</sup> `com.levigo.jadice.addon.bookmarks.BookmarkListener`

<sup>66</sup> `com.levigo.jadice.addon.pagesorter.PageSorter`



### 4.20.1. Support of PopupMenus in the PageSorter

The PageSorter supports the use of self-defined JPopupMenu in order to extend its functionality and usability.

Integrators may add self-defined PopupMenus for different purposes to the PageSorter:

- † a JPopupMenu for displaying on minimised pages:  
For functions referring to pages (e.g. „remove page“).
- † a JPopupMenu for displaying on the background of the used panel:  
For functions applying globally (e.g. „activate/deactivate sorting“).

Depending on the position (on a page or between the pages in the PageSorter) in which the context menu is requested by the user and if such a corresponding popup menu is set, either the global or the page referring popup menu opens. Of course, it is also possible to set the very same context menu as global and page referring menu.

The functions belonging to the offered menu items are to be put into realisation by the integrator.

Menus may be set in the PageSorter by using the following methods of the PageSorter:

- † **setPanelPopupMenu(JPopupMenu popupMenu)**  
Sets JPopupMenu to be displayed on the panel
- † **setThumbnailPopupMenu(JPopupMenu popupMenu)**  
Sets JPopupMenu to be displayed on the thumbnails

### 4.21. NavigatorPanel

For the editing of large pages or if a very high zoom factor is set in the viewer, jadice viewer offers a page survey of a single page.

The current page is presented in a minimised way as a thumbnail, whereas the section displayed by the viewer is visualised as a small transparent rectangle over the page. This rectangle can be moved by the mouse in order to navigate within a page, i.e. to scroll the displayed page sector in the viewer.

The page survey is provided by the class NavigatorPanel<sup>67</sup> and being a JComponent it may be embedded in any way into the integrating application. The navigator works in two different modes:

- † page is displayed with rotation 0°
- † page is displayed with rotation used in the viewer

This mode may be requested by the method „**getFollowViewerRotation()**“ or set by „**setFollowViewer-Rotation(boolean)**“. In order to call additionally the user's attention to the chosen mode, a description text integrated in the thumbnail view of the page may be set for each mode.

Being a viewer AddOn the comments in [4.17.AddOns](#) apply for the NavigatorPanel.

<sup>67</sup> com.levigo.jadice.addon.navigator.NavigatorPanel

## 4.22. Lens

jadice viewer supports a lens view for a particularly close displaying of page details. The lens displays an extremely magnified page sector according to the mouse-cursor position in the viewer.

To fix the lens a simple mouseclick on the corresponding passage in the viewer is enough, thus the fixed page detail may be for example compared with a different passage of the page. The fixed state is visualised to the user by a „frozen“ marking, the text of this marking may be requested/adapted by „getter-/setter“ methods.

The zoom scaling may be requested or changed by mouseclicking into the lens or by the aid of the methods „**getScale()/setScale()**“.

The lens is provided by the class `Lens`<sup>68</sup> and being a viewer AddOn the comments in [4.17.AddOns](#) apply.

### 4.22.1. HoverLens

In contrast to the lens in a separate window, there is alternatively the `HoverLens`<sup>69</sup>. A `HoverLens` corresponds to a lens bound to mouse movements and hovering above the page view in the viewer.

A `HoverLens` may present itself in two shapes (rectangle or round) and in different sizes, in which case the desired look may be determined by the method `setHoverShape(...)` and the dimension `setHoverSize(Dimension)`.

A fixing of the `HoverLens` is also possible and may be set or released by the ctrl.-key combined with a mouseclick.

Just as in the case of the lens the zoom scaling may be requested or changed by mouseclicking or by the methods „**getScale()/setScale()**“.

As an extension of the class `EditPane` (see also [4.15.EditPanes](#)) `HoverLens` instances do not represent a `JComponent` which is to be integrated. Instead of that for the de-/activating of a `HoverLens` the corresponding instance is de-/registered in the viewer. This happens by:

```
† viewerInstance.addEditPane(hoverLensInstance)
† viewerInstance.removeEditPane(hoverLensInstance)
```

## 4.23. GradationCurveControl

In order to change the displaying of image data jadice offers a changeable transmission curve of scaled pixel intensity to displayed pixel intensity on the display. The transmission curve is provided as a `GradationCurve` instance in the `RenderContext`.

The class `GradationCurveControl`<sup>70</sup> visually presents a gradation curve and allows to define, to move or to delete the curve's nodes. A first order spline interpolation takes place over the curve's nodes in order to get a continued curve progression.

<sup>68</sup> `com.levigo.jadice.addon.lens.Lens`

<sup>69</sup> `com.levigo.jadice.addon.lens.HoverLens`

<sup>70</sup> `com.levigo.jadice.addon.gradation.GradationCurveControl`

An instance of a GradationCurveControl may edit any gradation curves or even interact with a viewer instance.

Thus this class works in two different modes:

- **a gradation curve is set** the indicated curve is changed, an interaction with the viewer does not take place. A gradation curve is represented by the class GradationCurve, compare [4.23.1.GradationCurve](#).

- **a viewer instance is set** the gradation curve used by the viewer is applied, an interaction with a viewer takes place.

Being a viewer AddOn the comments in [4.17 AddOns](#) apply for the GradationCurveControl.

### 4.23.1. GradationCurve

This class contains the specifying points (data background) of a gradation curve, more precisely of an illustration between the pixel and displayed intensity.

As an extension of the class NaturalCubic-Spline1D<sup>71</sup> the gradation curve defines itself by an amount of key points which due to the nature of gradation curves must conform to certain properties.

Gradation curves may be held and edited totally independent of viewer instances. They may be used in various ways, e.g. gradation curves can be defined for document printing or can be used for own luminosity commands as fixed settings.

Properties of gradation curves may be loaded or saved as properties object.

Please note that gradation curves only effect a change when displaying image data! Text and other render elements like lines, shapes or similar remain untouched in their displaying.

### 4.23.2. GradationCurveFileHandler

GradationCurveFileHandler<sup>72</sup> is a utility class for loading and saving gradation curves in/from a file system.

An object of the class GradationCurveFileHandler may either be instantiated for a GradationCurve or a GradationCurveControl and it provides different methods for the loading/saving of gradations.

† openGradationCurveFromFile()

A file selection is displayed, the file selected by the user is loaded.

† openGradationCurveFromFile(File)

The indicated file is loaded.

† openGradationCurveFromFile(String)

The file indicated by a file name is loaded.

† saveGradationCurveToFile()

<sup>71</sup> com.levigo.util.math.NaturalCubicSpline1D

<sup>72</sup> com.levigo.jadice.util.GradationCurveFileHandler

A file selection is displayed, a file with the indicated name is saved in the path selected by the user.

† `saveGradationCurveToFile(File)`

The information is saved to the indicated file.

† `saveGradationCurveToFile(String)`

The information is saved to the file indicated by the file name.

## 4.24. PrinterJava2

`PrinterJava2`<sup>73</sup> is the central class for document printing out of jadice. This class is derived from the abstract basic class `AbstractPrinter`<sup>74</sup> which offers basic printing functionality and may be used by integrators to realise own printers.

An instance of the class `PrinterJava2` can be obtained by the default-constructor, alternatively a constructor with a parameterised `PrinterJob` can be used.

Before starting to print a document it is necessary to set a document instance providing the pages to be printed. Alternatively an array of the pages to be printed may be set instead of a document.

Furtheron a `RenderContext` may be passed in order to determine displaying properties of the rendering process within the provided printer device. As a standard the viewer's `RenderContext` can be used here, but an instance adapted by the integrator is also possible, so e.g. a render context with a special gradation curve for printing processes or adapted `AnnotationsRenderingHints` for the displaying or hiding of annotations. Some specifications of the `RenderContext` are ignored when printing, since they are unchangeably provided by the printing device. This refers to indications like zoom, rotation and similar.

Further optional printing information may be given for a `PrinterJava2` instance:

† indication of the page format by getter-/setter methods or as user input (dialogue)

† indication of the pages to be printed by getter-/setter methods or as user input (dialogue)

† adapt page optimally into printing range, e.g. by adapting the page size or even by automatic rotating

† if the printing process should be done synchronously / asynchronously

Beyond this an implementation of the interface `PageDecorator`<sup>75</sup> may be indicated. `PageDecorator` is an interface for the modification of printing results, e.g. for the indication of the page number, the document name, user indications etc. Two methods are provided for this:

† **`decoratePreRender(...)`**

Rendering of additional information before the rendering of the page.

† **`decoratePostRender(...)`**

Rendering of additional information after the rendering of the page

The final call of the method „**`print()`**“ results in the document's printing.

<sup>73</sup> `com.levigo.jadice.docs.printer.PrinterJava2`

<sup>74</sup> `com.levigo.jadice.docs.printer.AbstractPrinter`

<sup>75</sup> `com.levigo.jadice.docs.printer.PageDecorator`

Further information may be read in the API documentation.

#### 4.25. PrintManager

The class `PrintManager`<sup>76</sup> is used to simplify printing processes. Integrators who only want to initiate an asynchronous standard printing or who want to have a standard page format defined by the user may do this without any complex configurations of `PrinterJava2` instances by using one of the many static methods of the class `PrintManager`.

Further information about this class are in the API documentation.

#### 4.26. FileOpener

An interesting utility class of the `jadice` package is the class `FileOpener`<sup>77</sup>. Instances of this class make it very easily possible to load image data and eventually corresponding annotations from a file system.

An instance of the `FileOpener` may be used in order to load an image file with eventually corresponding annotation information. If no document to be loaded has been indicated, the user is asked by displaying of a dialogue box for file selection to select a document. After a confirmed selection a corresponding loading process is initiated. All further steps of the loading process as well as the search for a corresponding annotation file are automatically taken over by this class.

#### 4.27. DocumentSaver

Corresponding to the class `FileOpener` the class `DocumentSaver`<sup>78</sup> is available for developers. With the aid of this class `jadice` documents (see also paragraph [4.2Document](#)) may be saved in the local file system or in a free selectable `OutputStream`.

The `DocumentSaver` supports the saving of all formats which may be displayed by the viewer. However, format conversions or document changes are not carried out and supported by this class.

#### 4.28. Demonstration classes

In your `jadice` distribution there are different demonstration classes which allow to get to know `jadice` document platform a bit. For integrators these classes beside further demo classes are additionally available in the source code.

For further information see the distributed documentation. The following information is considered as a supplement to this documentation.

---

<sup>76</sup> `com.levigo.jadice.docs.printer.PrintManager`

<sup>77</sup> `com.levigo.jadice.util.FileOpener`

<sup>78</sup> `com.levigo.jadice.util.DocumentSaver`

#### 4.28.1. Parameter of the demonstration classes **JadicePanel** and **JadiceMDI**

The exemplified applications **JadicePanel** and **JadiceMDI** contained in the default package process an „-open“ parameter which allows directly at the programme's starting to indicate an image file to be opened.

† -Open

Opens the document indicated fully qualified in the path.

Ex.: `-open=c:\dokuments\test1.tif`

The command line call is as follows:

```
java -cp lib-all-in-one/jdk15/jadice-documentplatform-<version>-all.jar;
```

```
JadicePanel -open=c:\dokuments\test1.tif
```

or

```
java -Xmx256m -cp lib-all-in-one/jdk15/jadice-documentplatform-<version>-all.jar;
```

```
JadicePanel -open=c:\dokuments\test1.tif
```

with a heightened heap-size.

#### 4.28.2. Parameter of the demo-applet **JadiceApplet**

The exemplified applet **JadiceApplet** also contained in the default package processes the following parameters:

† **LOADITEM**

document to be opened at the start

† **IOHANDLER**

class of the IOhandler to be used, fully qualified class name of implementation, without any specification the standard IOhandler of the jadice package is used.

† **LOADERBASE**

provides the place where the documents may be found and from where they may be loaded. Without any specification the applet's Codebase is used.

† **RESBASE**

provides the place where external resources are to be found, without any specification the applet's Codebase is used.

† **jadice.viewer.tmps-path** (only for applets with write / read permission) defines the temporary directory to be used. If the applet has got write permission in a local directory, this may be used to cache already read document data, compare [4.9.2FileCacheInputStream](#). Without any indication temporary files will not be used.

## 5. Typical application examples

The following paragraphs show clearly the connection between the classes presented in paragraph [4.Class survey](#). On the basis of some exemplified applications it is shown how jadice may be integrated in own applications.

### 5.1. Embed viewer into a frame

In order to get a closer look on the jadice viewer and as a basis for further examples in this chapter, a window containing a viewer is created by the class TestViewerFrame.

```
public class TestViewerFrame extends JFrame{
    private BasicJadicePanel viewerPanel;
    public TestViewerFrame(){
        super("Test the - jadice \u00ae viewer");
        setContentPane(viewerPanel = new BasicJadicePanel());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
    }
    /**
     * Returns a reference to embedded viewer
     * @return Viewer
     */
    public Viewer getViewer() {
        return viewerPanel.getViewer();
    }
    /**
     * Opens a Frame containing a viewer
     * @param args
     */
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new TestViewerFrame().setVisible(true);
            }
        });
    }
}
```

Code example 1 - Creating a viewer window

The easiest method to embed a jadice viewer into a component's hierarchy is to use the class BasicJadicePanel. It combines beside a viewer an interacting status bar, toolbars and a context menu and thus it offers a complete, functional viewer representation.

For interested integrators the class BasicJadicePanel may be found as a source code example in the jadice document platform distribution under

<distribution-directory\example-src\viewer

It is a very helpful possibility to access on the contained viewer instance, as described in the following examples. For this reason the method „**getViewer()**“ has been added.

Performing the class the following application window appears:

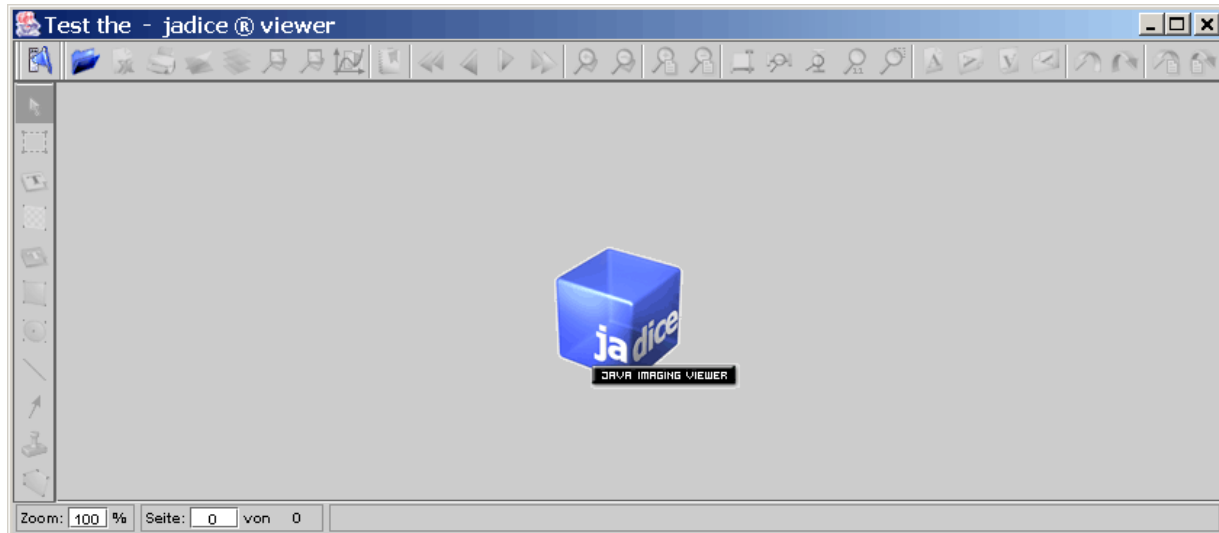



Chart 6 - Viewer Frame

In order to be able to work with the viewer, a document must be loaded now. This may happen in two different ways:

- † by the user: by pressing the button „open“ 
  - A file selection appears. The user may now select an image file and display it in the viewer.
- † by the programme: by initiating a loading process. This will be explained in detail in the following paragraph.

Instead of an instance of the BasicJadicePanel a viewer instance may also be - analogous to the example above – directly embedded into a component's hierarchy. However, in such a case it is up to the integrator to create toolbars, status bars, menu and context structures himself and to embed them accordingly. Ideas how toolbars or menus may be created programmatically are offered in the source code of the class BasicJadicePanel which is provided in the distribution in the directory example-src.

## 5.2. Loading process

### 5.2.1. Simple loading process

In order to explain a simple, programme-directed loading process of an image document available in the file system the class Test ViewerFrame is upgraded with the method „**loadAnImage(...)**“.

```
public class TestViewerFrame extends JFrame{
```



```
...
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            TestViewerFrame viewerFrame = new TestViewerFrame();
            viewerFrame.setVisible(true);
            viewerFrame.loadAnImage("resources\\Fax.tif");
        }
    });
}
/**
 * Loads an image into embedded viewer
 * @param imageFileName file name of image to load
 */
public void loadAnImage(String imageFileName) {
    Loader loader = new Loader();
    getViewer().setDocument(loader.getDocument());

    try {
        loader.loadDocument(new FileInputStream(
            imageFileName), 0);
    } catch (FileNotFoundException e) {
        System.err.println("Image not found: "+imageFileName);
        e.printStackTrace();
    } catch (IOException e) {
        System.err.println("Could not read image data: "
            +imageFileName);
        e.printStackTrace();
    }
}
}
```

Code example 2 - Simple loading process

First a loader instance is created. The document which is later filled by the loader is directly set in the viewer. The viewer updates the displaying automatically as soon as the first page is loaded.

The simplest way to load a document completely is offered by the method „**loadDocument(InputStream, firstPage)**“ of the loader. The parameters consist of a data `InputStream` and the index of the first document target page to be filled. By indicating „0“ as index of the target page the file is loaded at the beginning of the document. The indexing of pages within the loader is 0-based.

At this point it should be mentioned that image files with jadice are only in rare cases loaded by the local file system. As a rule jadice accesses on a document management system and gets documents from there. Since the „**loadDocument()**“ method of the loader expects only one `InputStream`, it is free to choose where this stream gets its data from and it is left to the integrating application to provide such a data stream.

### 5.2.2. Assemble documents

Jadice documents need not to exist of data of one source. The document model (compare [2.4.1.The document model](#)) shows clearly that documents may consist of different data sources. In this paragraph it is exemplified how pages of different sources may be assembled to a virtual jadice document. Beyond

this pages may also obtain their page segments from different data sources, this is specified in [5.2.3Layer](#).

To demonstrate it simply in this example the very same Single Page Tiff has been linked together ten times in a row. Once the loading process has been finished, the document thus contains 10 pages. Like that different image documents, with one and more pages, in different formats, may be also linked together.

```
public class MultipleTiffLoadSample {
    private static TestViewerFrame viewerFrame = null;
    public static void main(String[] args) throws Exception
    {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                TestViewerFrame viewerFrame = new TestViewerFrame();
                viewerFrame.setVisible(true);
                // Ladevorgang asynchron starten
                Thread t = new Thread("Load 10 Tiffs in a row...") {
                    public void run() {
                        load10TiffsInRow();
                    }
                };
                t.setPriority(Thread.MIN_PRIORITY);
                t.start();
            }
        });
    }
    /**
     * Loads 10 Tiffs in a row
     */
    private static void load10TiffsInRow() {
        Loader loader = new Loader();
        // set empty document in viewer
        viewerFrame.getViewer().setDocument(
            loader.getDocument());
        // synchronous loading process
        loader.setSynchronousLoading(true);
        // LoadListener, only for displaying,
        // when a raw document
        // has been loaded and added to the jadice document
        loader.addLoadListener(new LoadListener() {
            public void loadStateChanged(LoadEvent e) {
                if (e.getType() == LoadEvent.LOAD_COMPLETE)
                    System.err.println(
                        "Document Fax.tif is loaded");
            }
        });
        for (int i = 0; i < 10; i++) {
            try {
                loader.loadDocument(
                    new FileInputStream("resources\\Fax.tif"),
                    loader.getDocument().getPageCount());
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
}  
}  
}
```

*Code example 3 Assemble documents*

In this example as well the class `TestViewerFrame` is used for the viewer's displaying, but not its „`loadAnImage(...)`“ method. The „`loadAnImage(...)`“ method passes at each call and for each raw data stream to be loaded a new document to the viewer instance and thus it can't be used in this example.

In [Code example 3](#) first a `TestViewerFrame` is created, made visible and a multiple loading process is started. To avoid that the loading process blocks the current thread, the method „`load10TiffsInRow()`“ is called asynchronously. The asynchronous starting of the loading process is not necessarily needed, but in certain contexts this procedure may be very useful in order to grant a smooth programme flow and it is only taken in here for demonstrating purposes.

In the method „`load10TiffsInRow()`“ first a loader instance is created. Afterwards, like in the preceding example, the loader's document is set in the embedded viewer for displaying. At this the „`getViewer()`“ method of the class `TestViewerFrame` is used for the first time. By this method the test frame can be flexibly used from the outside, a property which is often applied in the following examples.

At this point it should be again and in particular pointed out, that a loader instance may be used for any number of loading processes. These loading processes fill the document that has been set to the loader. If no document has been set, the loader creates automatically a new `DocumentInstance`. If different documents should be loaded by a loader instance, the document to be loaded may be changed by using the method „`setDocument(Document)`“. Since the loader, if not set differently, works asynchronously, this should not be done during a current loading process, though. Integrators are responsible for changes of the document instance to be filled.

To keep the correct order loading processes of different image sources should be synchronised, so that really the next image document is only loaded, when the preceding has already been loaded and added to the document instance which is to be loaded. For this purpose the loader is set on synchronous processing by the method „`setSynchronousLoading(true)`“.

After that a `LoadListener` is registered on the loader which has no functional meaning in this example and has only been added for exemplification. Each time when one of the ten documents to be loaded has finished its loading process the `LoadListener` issues a corresponding message on the console. Please note that `LoadListener` instances have to be registered only once per loading process. In the same way multiple `LoadListeners` may be registered in parallel.

In the next step the actual loading process is performed. Since the loader works synchronously, loading processes may be sequentially started by the aid of a „for“-loop. Like already in the last example a data `InputStream` and a page index are passed as parameters.

To make sure that a newly loaded raw document will be added at the end of the jadice document which is to be loaded, the current page number of the document is indicated as page index of the „`loadDocument(...)`“. Since the pages are added synchronously to the document by the loader and the loader's page indexing is 0-based, the requested page order is granted.

### 5.2.3. Layer

In the last paragraph pages of different sources were loaded and assembled to a document.

Documents may not only consist of different pages from different sources, but even pages as well may be composed of different layers. Compare also [2.4.1.The document model](#).

This paragraph describes an example how a page background (in this case a fax preprint) and a textual content may be loaded in different page segments.

```
/**
 * Loads data into two different layers
 */
private static void doLayeredLoad() {
    loader.setSynchronousLoading(true);
    // create layers to load into
    DocumentLayer backgroundLayer =
        loader.getDocument().addLayer(
            "background", DocumentLayer.BOTTOM);
    DocumentLayer overlayLayer =
        loader.getDocument().addLayer(
            "overlay", DocumentLayer.ABOVE_BOTTOM);

    // start loading
    try {
        loader.loadDocument(
            new FileInputStream(
                "resources\\levigoFaxPapier.afp"),
            backgroundLayer,
            0);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    try {
        loader.loadDocument(
            new FileInputStream(
                "resources\\FaxContent.txt"),
            overlayLayer,
            0);
    } catch (FileNotFoundException e1) {
        e1.printStackTrace();
    } catch (IOException e1) {
        e1.printStackTrace();
    }
}
```

Code example 4 Loading of a layer

First the loader is set again in a synchronous processing mode. This is in technical regards not really necessary for the loading of page segments, but it makes sure that first the page background is loaded and displayed before the textual context is inserted. Normally the loading process should proceed so

quickly that the user does not even realise it, but with slow network connections e.g. this displaying order is more suggestive.

In order to be able to load in different page segments, first these layers must be created in the document. A layer thus exists for all pages of the document, even if they possibly don't cover all layers with PageSegments. A layer is defined by a unique name and is inserted at a particular vertical position of the document levels. So e.g. an AnnotationPage segment should always lie on the highest level on top of all page segments; in this example the text displayed over the fax background. For this reason the layer „background“ is created on the position DocumentLayer.BOTTOM and the layer „overlay“ on the position DocumentLayer.ABOVE\_BOTTOM.

Afterwards the loading processes are initiated. For this – like in the preceding examples - a data InputStream and a page index are passed to the „**loadDocument(...)**“ method. Additionally a layer is defined in which the loaded data are positioned vertically as page segment in the page.

After the realisation the loaded page looks like the following:

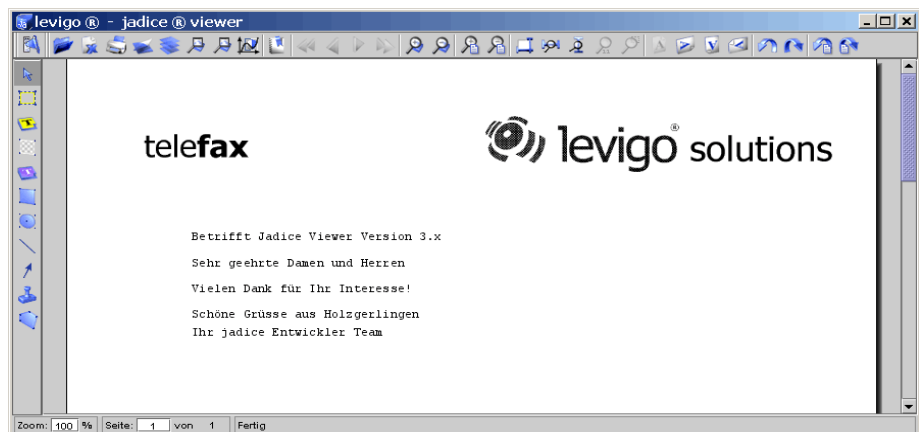


Chart 7 - Page segments loaded in layers

If the „**loadDocument()**“ call for the „overlay“-layer of the textual fax content is commented out, as expected only the page background appears.

See also the following chart.

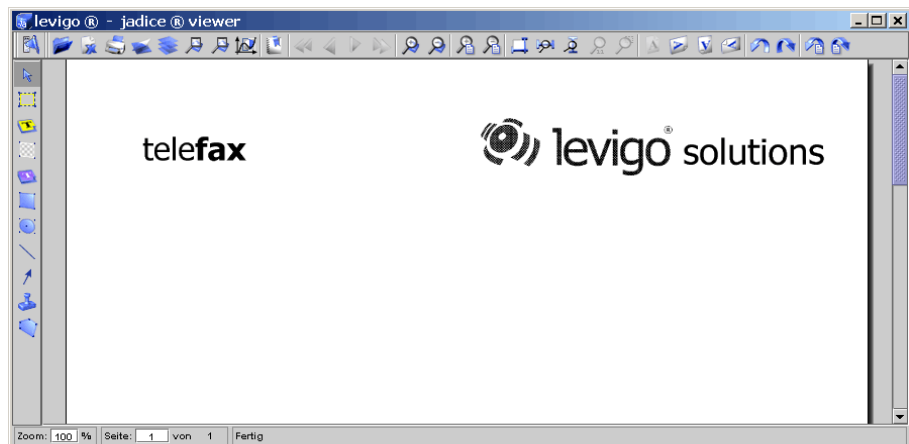


Chart 8 - Only background level loaded

## 5.2.4. SeekableInputStream

For an efficient and memory sparing processing of large documents the viewer tries, if allowed by the image format, to read, to process and to cache dynamically only document data necessary for the current page segment instead of holding all image data completely in the memory.

Jadice uses for this procedure SeekableStreams. [Example 5](#) makes it clear, how it is tried to find a suitable SeekableInputStream.

```

public Document load(File aFile){
    Loader loader = new Loader();
    SeekableInputStream seekInputStream = null;
    try {
        seekInputStream =
            new RandomAccessFileInputStream(aFile);
    } catch (Exception e) {
        try {
            seekInputStream = new FileCacheInputStream(
                new FileInputStream(aFile));
        } catch (Exception g) {
            seekInputStream =
                new MemoryInputStream(
                    new FileInputStream(aFile));
        }
    }
    return loader.loadDocument(seekInputStream, 0);
}

```

Code example 5 Selection of a suitable data stream

The method „**load(File)**“ has to load a given file and to return it as a document.

First it is tried to create a `RandomAccessFileInputStream` which makes it possible to position within the file. This data stream type works on a document

source - physically existing in a file system – in which a pointer is directly positioned in the data source.

If this fails, it is tried to use a `FileCacheInputStream`. A `FileCacheInputStream` buffers read data in a temporary file. So slow reading processes (e.g. due to a bad network connection) must only be performed once and if required. However, data which have been already read and buffered as a temporary file are quickly available. In order to create a `FileCacheInputStream` the application must at least have write permission in the set temporary directory.

As last alternative remains a `MemoryInputStream` which buffers read data in the central memory. This is the fastest variant, since data are only accessed in the central memory, but it may with very large data amounts enlarge significantly the central memory requirement of the application.

It is left to the integration to choose depending on the environment of use and the specific conditions of application the most pragmatic type of `SeekableInputStreams` and to use them for loading processes. If choosing the most appropriate `SeekableInputStream` is to be left to jadice, jadice offers two possibilities as static methods of the class `Loader`:

③ `Loader.prepareSeekableInputStream(InputStream)`

Creates from the given `Inputsteam` a suitable `Seekablestream` and returns this one as method return value.

③ `Loader.prepareSeekableInputStream(InputStream, boolean)`

Creates from the given `Inputsteam` a suitable `Seekablestream` and returns this one as method return value.

The Boolean parameter defines, if a temporary file buffering is to be taken into consideration when evaluating the most appropriate `SeekableInputStream`.

### 5.2.5. ResourceLoader

`ResourceLoader` are used for AFP or MO:DCA documents which may have embedded in the document not only internal (inline) but also external resources. External resources are dynamically supplied during the document's loading process with the aid of `ResourceLoaders`.

In this example we assume that different resources are found in the directory „C:\afp\res“ and beyond this a resource directory „myResources“ with the required resources is provided on „[www.myServer.com](http://www.myServer.com)“. All these resources should always be available for all of the following loading processes.

```
public ResourceLoader getResourceLoader() {
    ResourceFileLoader resLoader = new
ResourceFileLoader("C:\\afp\\res");

    ResourceMultiLoader multiLoader =
        new ResourceMultiLoader();
    multiLoader.addLoader(
        new ResourceUrlLoader(
            "http:\\www.myServer.com\\myResources"));
    multiLoader.addLoader(resLoader);
}
```

```
return multiLoader;
}
```

Code example 6 A ResourceLoader out of different ResourceLoaders

In order to be available for all loading processes, the creating ResourceLoader must be registered directly on the loader. If the ResourceLoader was registered on the document, it would be only available for the loading process of this document, but not for loading processes into other documents.

Then the registration on the loader can be performed with the method „**setResourceLoader(...)**“.

Example:

```
loader.setResourceLoader(getResourceLoader());
```

On the loader only one ResourceLoader may be registered at a time. However, in this example two ResourceLoaders are required: One ResourceFileLoader which provides all resources of the directory „C:\afp\res“ and a second one which makes all resources on „www.myServer.com“ available in the resource directory „myResources“. The solution is an instance of the class ResourceMultiLoader which implements the interface ResourceLoader and thus may work as a ResourceLoader, but it may also take in different ResourceLoaders by the method „**addLoader()**“. If a resource is requested, all registered ResourceLoaders in the ResourceMultiLoader are queried about this resource and the first successful result is returned.

In [Code example 6](#) first a ResourceFileLoader is created. Then a ResourceUrlLoader is created. It may be directly instantiated by a list of one or more URLs containing resources.

Afterwards the ResourceFileLoader as well as the ResourceUrlLoader are added to an instance of the class ResourceMultiLoader.

### 5.2.6. Annotations

In the following paragraph it is exemplified how ImagePlus compatible annotations may be loaded. FileNet and FileNet P8 annotations may be loaded analogously. But instead of an instance of the class ImagePlusAnnotationFormatInfo an instance of the class FileNetAnnotationFormatInfo respectively FileNetP8AnnotationFormatInfo must be passed to the loader's method „**loadDocument(...)**“. In order to keep it concise at this point it is not dwelled on the loading process of the actual image document, but only on the annotations.

```
File file2Load = new File("myimage.tif");
Loader loader = new Loader();
// load document...

int lastDot = file2Load.lastIndexOf(".");
if (lastDot > 0) {
    // try to look for annotation file
    String annoFileName = file2Load.substring(
        0, lastDot);
}
```



```
// Default vi annotation extension: „.T_L“
File annoFile = new File(annoFileName + ".T_L");
// if file exists, do load the annotations
if (annoFile.exists())
    loader.loadDocument(
        new FileInputStream(annoFile),
        new ImagePlusAnnotationFormatInfo(),
        0);
}
```

Code example 7 - Load annotations

First it is tried to find a corresponding annotation file to the provided image file „file2Load“. ImagePlus compatible annotation files usually have got the same name like the image document and „.T\_L“ as suffix. If such a file exists, it is used for the loading of annotations. Here it should be pointed out that annotation data are usually not provided as a file, but as a stream from an archive or similar.

Format information describe the format in which a document is provided. If no format information is given to the loader in the „loadDocument“ method, the loader tries to define the format itself. Since ImagePlus annotations are MO:DCA structures, it is necessary to load ImagePlus compatible annotations always with ImagePlusAnnotationFormatInfo<sup>79</sup>. Otherwise these are mixed up with MO:DCA data and are loaded accordingly as a document in place of annotations.

*Note:*

Loading and saving of annotations must be performed explicitly by the integrator and it is **not** done automatically by jadice when loading the main document.

### 5.2.7. Bookmarks

Bookmarks are saved, loaded and administrated by the class DocumentBookmarkHandler.

DocumentBookmarkHandler allows loading and saving of bookmarks as Properties<sup>80</sup>.

[Example 8](#) describes a loading process from a property file.

```
public void loadBookmarks(Document doc, String
bmkFileName) {
    try {
        File fBookmarks = new File(bmkFileName);

        if (fBookmarks.exists() && fBookmarks.canRead()) {
            // create and load bookmark properties
            Properties bookmarksProps = new Properties();
            bookmarksProps.load(new FileInputStream(fBookmarks));

            // load bookmarks into the BookmarkHandler
            DocumentBookmarkHandler.getInstance().load(
```

<sup>79</sup> com.levigo.jadice.formats.annoipus.ImagePlusAnnotationFormatInfo

<sup>80</sup> java.util.Properties

```
        bookmarksProps,  
        doc,  
        doc.getName());  
    } else {  
        // reset BookmarksHandler state  
        DocumentBookmarkHandler.getInstance()  
            .removeBookmarksForDocument(doc);  
    }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Code example 8 - Load bookmarks

First it is checked, if the indicated Bookmark Properties file is available and if it may be accessed for reading. If so, an empty property object is created in which the bookmark data are loaded in the next step. The filled properties object is then made available to the DocumentBookmarkHandler. The DocumentBookmarkHandler initialises JadiceBookmarks out of the given properties which are then provided by the DokumentBookmarkHandler for further use.

Properties objects may contain bookmark entries of any number of different documents. To make sure that only the bookmarks belonging to the document are loaded, an explicit bookmark identification of the DocumentBookmarkHandler's loading method must be indicated additionally. In this example simply the document's name has been used. But principally any other character string may be used as identifier, it is only important that it is the same which was used when saving the bookmarks.

With the aid of this identification bookmarks from different saving processes and of different documents may be administrated in a properties object.

### 5.2.8. Gradation

Gradation data are processed by two different object types of the jadice package: on the one hand by instances of the class GradationCurve and on the other as AddOn in form of a GradationCurveControl instance.

For both object types processing gradation data the utility class GradationCurveFileHandler supports the loading process from a file system. This class is instantiated with a GradationCurve or a GradationCurveControl which is to be filled with gradation points.

The class GradationCurveFileHandler offers different loading methods for this which may be used according to requirements. A detailed listing is found in [4.23.2.GradationCurveFileHandler](#). In the following example the method „**openGradationCurveFromFile**“ was used. This method first opens a file selection dialogue which after having selected a gradation data file effects a loading process into the passed GradationCurve or GradationCurveControl object.

Alternatively gradation data may also be loaded from a provided data stream. Compare the method „**loadGradationCurveFromStream**“ in [Example 9](#).

```
public void loadGradationCurve(GradationCurve aCurve){
    new GradationCurveFileHandler(aCurve)
        .openGradationCurveFromFile();
}
public void loadGradationCurveIntoGradPanel(
    GradationCurveControl aCurveControl){
    new GradationCurveFileHandler(aCurveControl)
        .openGradationCurveFromFile();
}
public void loadGradationCurveFromStream(
    GradationCurve aCurve, InputStream is){
    new GradationCurveFileHandler(aCurve)
        .openGradationCurveFromStream(is);
}
```

Code example 9 - Load gradation from local file system or InputStream

If the gradation data are not available in the local file system, the class GradationCurve offers additionally the possibility to insert data by a properties object.

```
public GradationCurve loadGradationData(
    String gradationIdentifier,
    InputStream gradationDataStream ) {
    GradationCurve newCurve = new GradationCurve();
    try {
        // create and load gradation properties
        Properties gradationProps = new Properties();
        gradationProps.load(gradationDataStream);

        // load gradation into the GradationCurve
        newCurve.load(gradationProps, gradationIdentifier);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return newCurve;
}
```

Code example 10- Load gradation from a data stream

[Example 10](#) shows the loading process in an instance of the class GradationCurve. For this purpose the method „**loadGradationData(...)**“ gets a corresponding DataInputStream and a gradation identifier. This identifier identifies uniquely the gradation data to be loaded and it should correspond to the identifier used in the preceding loading processes of the gradation data. As a result the method returns a new created GradationCurve initialised with the data of the InputStream.

First a property object is created which loads the data of the provided data stream. Then the GradationCurve is initialised with these properties und the passed gradation identifier, before it is returned as a method result.

### 5.3. Saving

This paragraph is mainly about examples which describe in detail the saving into a particular format or go into concrete parts of a document like annotations. In the standard case (which is the saving of a simple, not assembled document with possible annotations) the functionality of the class [4.27.DocumentSaver](#) should be enough for integrators and developers. The DocumentSaver supports the saving of all formats which may be displayed by the viewer. Format conversion and document changes are not supported.

#### 5.3.1. Document

Similar to format information which support loading processes of particular formats, there are classes which support saving processes into particular formats (*FormatNameFile*). See also [4.6.FormatInfo and FormatFile](#).

```
private void saveDocument () {
    TIFFFile tiffFile =
    new TIFFFile (getViewer ().getDocument ());
    try {
        tiffFile.save (new FileOutputStream ("Myimage.tif"));
    } catch (FileNotFoundException e) {
        e.printStackTrace ();
    } catch (IOException e) {
        e.printStackTrace ();
    }
    saveAnnotations ();
}
```

Code example 11 Save Tiff document

In the example a document that has been created of TIFF data is to be saved into a file „myimage.tif“.

For this purpose first an instance of the class TIFFFile is created and the document to be saved is passed to the constructor.

Then the TIFF-data of the document are saved by the method „**save(OutputStream)**“. If the target format differs from the format of the original document, an error occurs. Format conversions are not carried out.

The saving as well as the loading underlies the responsibility of the integrating application. Compound documents from different formats must be saved according to their configuration. For this all *FormatNameFile* classes provide different „**save(...)**“ methods which allow a saving of particular pages and/or layers.

#### 5.3.2. Annotations

Like in the preceding paragraph for the saving of annotations a particular format-specific saving class is used especially for annotations.

```
private void saveAnnotations() {
    // similar to anno load:
    // ImagePlusAnnotationFormatInfo ->
    // use here ImagePlusAnnotationFile
    ImagePlusAnnotationFile annoFile =
        new ImagePlusAnnotationFile(
            getViewer().getDocument());

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

Code example 12- Save annotations

The example describes how annotations of a document may be stored in a file.

First an instance of this class is created. In order to get access on the annotations to be loaded, the document with its annotations to be saved is passed to `ImagePlusAnnotationFile` in the constructor. Finally a call of the „save“-method saves the annotations in the passed `OutputStream`.

Note:

Loading and saving of annotations must be done explicitly and is **not** done automatically when loading the document.

For a format-specific saving of annotation data the following classes may be used:

- † `ImagePlusAnnotationFile` (for IBM ImagePlus and IBM ContentManager compatible annotations)
- † `FileNetAnnotationFile` (for FileNet annotations)
- † `FileNetP8AnnotationFile` (for FileNet P8 annotations)

### 5.3.3. Bookmarks

As already presented in paragraph [5.2.7.Bookmarks](#), the persistence of bookmarks is provided by `Properties` objects and a unique identifier.

```
public void saveBookmarkData(Properties bmProperties,
    Document doc, String documentBookmarkIdentifier) {

    // get a reference to bookmark handler
    DocumentBookmarkHandler bmHandler =
        DocumentBookmarkHandler.getInstance();
    // save bookmark data into the properties object
    bmHandler
```

```
.saveBookmarksForDocument (bmProperties, doc,  
    documentBookmarkIdentifier);  
}
```

Code example 13- Save bookmarks

In the code example first a reference on the DocumentBookmarkHandler is detected by the static method „**getInstance()**“.

In the next step the saving of bookmarks is effected by a call of the method **saveBookmarksForDocument**. For being called this method expects the following parameter:

- ③ A reference of the document the bookmarks of which are to be saved.
- ③ A Properties Object in which the bookmarks are to be saved.
- ③ A unique bookmark identification which allows at a later date to read bookmarks out of a Properties Object to the corresponding document.

#### 5.3.4. Gradation

In the following example gradation data set in the viewer are to be saved in a given OutputStream with a provided identifier. This identifier serves for identification within a new loading process.

```
public void saveGradationData (  
    String gradationIdentifier,  
    OutputStream gradationDataStream ) {  
    GradationCurve curveToSave =  
        getViewer().getRenderContext()  
            .getImageRenderSettings().getGradationCurve();  
    try {  
        // create a properties object to store gradation  
        // data into it  
        Properties gradationProps = new Properties();  
  
        // save gradation data into properties object  
        curveToSave.save (gradationProps, gradationIdentifier);  
  
        // save properties object into output stream  
        gradationProps.store (  
            gradationDataStream,  
            "Test Viewer Frame Gradation Data");  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Code example 14- Save gradation

The gradation curve used by the viewer is saved in the viewer's RenderContext so that by the aid of the methods

† **Viewer#getRenderContext()** and

† **RenderContext#getImageSettings()** and

† **ImageRenderSettings#getGradationCurve()**

a reference to the gradation curve to be saved may be obtained.

In the next step a property object is created which is passed to the gradation curve with a unique gradation identifier for the intake of gradation data.

After the gradation data have been stored in the properties object, the properties instance saves itself in the passed OutputStream. The second parameter of the method „**store(...)**“ serves as data-header which precedes the saved gradation properties as a comment. Indicating a header is optional and if no header is requested it may be indicated with „null“.

## 5.4. Actions-Commands-Context

A central aspect of the jadice document platform is to offer very easy integration possibilities with a possibly small programming and adaptation effort. This chapter demonstrates how easily jadice components may be integrated and adapted in own applications by the aid of the jadice Integrator API.

For a better understanding of the following paragraphs see also the chapters [3.The jadice Integrator API](#) and [8.jadice Integrator API: Syntax description of configuration files](#).

### 5.4.1. Embedding of menus, toolbars, actions

As introducing example in this paragraph first a window is created which contains a viewer instance, a corresponding toolbar and a menu bar. Menu bar and toolbar are created by the aid of the jadice Integrator API.

```
public class CommandsTestFrame extends JFrame {  
  
    // parent context  
    private Context context = null;  
    // containing viewer  
    private Viewer viewer = null;  
  
    CommandsTestFrame () {  
        super ("Commands Test Frame");  
        setDefaultCloseOperation (EXIT_ON_CLOSE);  
  
        viewer = new Viewer ();  
        initContext ();  
        initGui ();  
        pack ();  
        setVisible (true);  
    }  
    private void initContext () {  
        // ... see 5.4.1.1  
    }  
    private void initGui () {  
        getRootPane (). setJMenuBar (getCommandsMenuBar ());  
  
        JPanel contentPane = new JPanel (new BorderLayout ());
```

```

contentPane.add(viewer, BorderLayout.CENTER);
contentPane.add(
    getCommandsToolBar(), BorderLayout.NORTH);
setContentPane(contentPane);
}

private JToolBar getCommandsToolBar() {
    // ... see 5.4.1.2
}

private JMenuBar getCommandsMenuBar() {
    // ... see 5.4.1.2
}

public static void main(String[] args) {
    new CommandsTestFrame();
}
}

```

Code example 15- Exemplified window

The methods „**initContext()**“, „**getCommandsToolBar()**“ and „**getCommandsMenuBar()**“ are described in the following paragraphs.

#### 5.4.1.1. Context

For the creation of **CommandActions**, no matter if these are the performable part of a menu item or of a button, a context object is required. A context reflects by means of the contained objects the current state of the associated GUI-component and thus presents the basis for state and performability of **CommandActions**. See also chapter [3. Die jadice Integrator API](#).

```

private void initContext() {
    // Create a context instance with the RootPane as owner
    // and the NO_CHILDREN as aggregation mode.
    // NO_CHILDREN means that there are no other context
    // objects in the context hierachy, whose context objects
    // need to be aggregated on a context changed event
    // which triggers a command action update.
    context =
        new Context(getRootPane(), Context.NO_CHILDREN);
    // Add context objects, here just the embedded viewer
    // instance
    context.add(viewer);

    // React to property changes of the viewer by correct
    // enabling/disabling the commands/actions.
    // Therefore trigger a context changed event whenever a
    // property change of the viewer happened.
    viewer.addPropertyChangeListener(
        new PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent evt) {
                context.contextChanged();
            }
        });
}
}

```



*Code example 16- Create context*

Accordingly in the method „**initContext()**“ first a context object is created to the constructor of which two parameters are passed, a Context Owner and an aggregation mode.

In the following the parameters are described in detail:

† The window's RootPane as Context Owner

Each context instance has got an associated GUI element as Context Owner. A Context Owner serves two aspects.

On the one hand it controls the state of the context object. If the Context Owner is active as a GUI element, the context is also active. In context hierarchies the context elements are according to the set aggregation mode composed of the proper context elements and the elements of no, all or the active child-contexts. So the contexts' activity determines depending on the set aggregation mode the precise composition of context elements and thus influences with context changes the state (performable, not performable, performed, not performed) of associated CommandActions.

On the other hand the Context Owner represents as GUI-element a contained component's hierarchy. Context hierarchies are to be composed in correlation to the component hierarchies of their Content Owners.

† Context.NO\_CHILDREN as context aggregation mode

The aggregation mode determines the composition of context elements in context hierarchies. For this there are three modes which are offered as static constants of the class Context:

† Context.ALL\_CHILDREN – The context contains all elements of its own as well as all elements of all child-contexts.

† Context.ACTIVE\_CHILD – In this mode the context elements are composed of the elements of the context object and the elements of the active child-contexts.

† Context.NO\_CHILDREN – This mode is appropriate, if no context hierarchy is available or if all context objects and the corresponding CommandActions act absolutely independently of other context objects and their elements.

See also [3.2.3.Context](#) and the jadice API documentation.

In the next step objects required by the CommandActions are added to the context. All commands only need the viewer, thus only one viewer instance is included in the context. Information which context objects are expected by which jadice commands for performance are provided in the jadice API documentation. With the following methods further context objects might be created at this point and placed into a hierarchy.

† Context#addToParentsContext()

† Context#addChildContext(Context)

† Context#removeFromParentsContext()

† Context#removeChildContext(Context)

For this simple example, however, no context hierarchy is required.

In order to translate changes within the viewer, e.g. that a page of a document has been loaded and is displayed, into updating events of the `CommandActions`, a `PropertyChangeListener` which transforms `PropertyChangeEvents` in `ContextChangedEvent`<sup>81</sup> is added to the viewer. This mechanism is a current method to make sure that viewer specific commands in their state always blend well with the viewer's state.

#### 5.4.1.2. Embedding

After a context object has been created in the last paragraph, in the method „`initGUI()`“ the graphic user interface of the `CommandsTestFrame` is assembled.

```
private JToolBar getCommandsToolBar() {
    JToolBar aToolBar =
        DefaultMenuComponentFactory
            .getInstance(
                "/com/levigo/jadice/resources/properties/"+
                "menucomponents.properties")
            .getToolBar("jadiceToolbar", context);

    aToolBar.setFloatable(false);

    return aToolBar;
}
private JMenuBar getCommandsMenuBar() {
    JMenuBar aMenuBar = new JMenuBar();

    JMenu menu =
        DefaultMenuComponentFactory
            .getInstance(
                "/com/levigo/jadice/resources/properties/"+
                "menucomponents.properties")
            .getMenu("file", context);
    aMenuBar.add(menu);

    return aMenuBar;
}
```

Code example 17- References to menus and toolbars

Toolbar and menu structures are defined by the configuration `menucomponents.properties`. For more detailed information see [3.3.The Configuration files](#) and [8.jadice Integrator API: Syntax description of the configuration files](#).

A jadice toolbar and a file menu were defined in this file and are referenced in this example for creation.

A reference on a defined structure is created by the class `DefaultMenuComponentFactory`<sup>82</sup>. The instance corresponding to a configuration is obtained by the method „`getInstance(ConfigurationName)`“ which creates the required structure

81 `com.levigo.swing.action.ContextChangedEvent`

82 `com.levigo.util.swing.action.DefaultMenuComponentFactory`

according to method call and definition. For this the following methods are offered:

† **getMenu(MenuName, Context)**

creates a menu defined by the name „MenuName“, the second parameter is the context required for the creation of CommandActions.

† **getContextMenu(MenuName, Context)**

creates a context menu defined by the name „MenuName“, the second parameter is the context required for the creation of CommandActions.

† **getToolBar(ToolBarName, Context)**

creates a menu defined by the name „ToolBarName“, the second parameter is the context required for the creation of CommandActions.

In order to use a different configuration, the required name is passed to the method „**getInstance(...)**“.

Take note that structures may be defined as qualified or unqualified. With an unqualified definition any structure may be created, e.g. menu or toolbar. Otherwise only the structure which has been indicated in the configuration can be created. Compare also [8.jadice Integrator API: Syntax description of configuration files](#).

If no structure, but only a reference to a certain CommandAction is required, this may be made possible by the following call.

Example:

```
DefaultActionFactory
  .getInstance (
    "/com/levigo/jadice/resources/properties/"+
    "actions.properties")
  .getAction(context, "OpenDocument ")
```

As descendant of AbstractAction<sup>83</sup> each CommandAction may be bound as a performable element to appropriate components e.g. to a button or a menu item.

## 5.4.2. Adaptation of Actions

### 5.4.2.1. Properties

Under certain circumstances it may be required to change the properties of a CommandAction, for example a change of the tooltip or of the menu text.

Part of the file menu is a command called „OpenDocument“ which opens local image documents and loads them in the viewer. If you want, for example, that the menu entry is made out to „Open file“ instead of „Open“ only, change the *actions.properties* configuration respectively the localised variant *actions\_de.properties* configuration as follows.

Example:

```
#actions.properties
...
OpenDocument.ShortDescription = open
...
```

<sup>83</sup> javax.swing.AbstractAction

```
# ... change to
OpenDocument.ShortDescription = open file
...
```

A precise description which properties are changeable and in which way they may be adapted is provided in chapter [8.jadice Integrator API: Syntax description of the configuration files](#).

*Note:*

The configuration details are read and evaluated once when starting the viewer. Thus changes on configuration files take effect only after the viewer's restart.

*Note:*

Bear in mind that the configuration files are provided in internationalised variants. In order to avoid inconsistencies configuration changes should always be done in all variants.

#### 5.4.2.2. Adapting the menu or toolbar structure

In order to define own menu or toolbar structures or to change existing structures the configuration *menucomponents.properties* respectively the localised variant *menucomponents\_de.properties* must be adapted.

An example: The viewer's context menu may be thus restricted, that it now contains the product information only.

For this the definition of the context menu is thus shortened that it now contains only the CommandAction for the displaying of the product information.

Example:

```
#menucomponents.properties
...
mainContextMenu.actions.contextmenu=ProductInfo
...
```

A precise description which properties are changeable and in which way they may be adapted may be found in paragraph [8.jadice Integrator API: Syntax description of the configuration files](#).

*Note:*

The configuration details are read and evaluated once when starting the viewer. Thus changes on configuration files take effect only after the viewer's restart.

*Note:*

Bear in mind that the configuration files are provided in internationalised variants. In order to avoid inconsistencies configuration changes should always be done in all variants.

### 5.4.3. Own commands

If the functionality of the provided commands does not cover the needs of the integrating application, own commands may be embedded simply and with little effort.

As an example a command which opens a local image document and loads it into the viewer is created in this chapter. In a further step this command is embedded in the file menu of the CommandsTestFrame.

```
package my.tests.commands;

/**
 * This command needs a viewer instance to load documents
 * into the context objects.
 */
public class ADocumentOpener extends AbstractCommand {
    protected void doExecute(Collection args) {
        // get access to viewer instance
        Viewer viewer = (Viewer) getClassFromArguments(
            args, Viewer.class);
        if (viewer != null){
            // use FileOpener to load an image
            new FileOpener(viewer).openDocumentFromFile();
        }
    }

    /**
     * Is executed whenever an action is executed. Its return
     * value is used to decide, if an action will be executed
     * or aborted. Could show a message, if it fails.
     * @see AbstractCommand#checkDeeply(Collection)
     */
    public boolean checkDeeply(Collection args) {
        // checks, if a viewer instance is available
        return isArgumentValid(
            AbstractCommand.ONE, Viewer.class, args);
    }

    /**
     * Is executed whenever the context changes. Its return
     * value is used to decide, if an action gets enabled or
     * disabled.
     * @see AbstractCommand#checkQuickly(Collection)
     */
    public boolean checkQuickly(Collection args) {
        // checks, if a viewer instance is available
        return isArgumentValid(
            AbstractCommand.ONE, Viewer.class, args);
    }
}
```

Code example 18- Create a command

The abstract basis class of all commands is the class `AbstractCommand`<sup>84</sup>. It requires for extension the realisation of three methods:

<sup>84</sup> com.levigo.util.swing.action.AbstractCommand

**† doExecute(Collection)**

This method is called in order to perform the command. The parameter contains the available context elements.

**† checkQuickly(Collection)**

This method is called, when the context has changed. The return value determines the command's state (active, not active). Since this method is called very often in order to always have a correct status (enabled, disabled), complex tests should not be done within this method.

**† checkDeeply(Collection)**

This method is called only before the command is performed. Here more complex tests may also be done. The return value determines, if the command is finally performed or not.

In order to perform the ADocumentOpener command a viewer instance in which the document is to be loaded is necessary as context element. The methods „**checkQuickly(...)**“ and „**checkDeeply(...)**“ use for the checking of this precondition the method „**isArgumentValid(...)**“.

This and other useful utility methods of the class AbstractCommand are described in the following:

**† isArgumentValid(CountCondition,Class,Collection)**

Checks, if objects of the indicated class are provided according the CountCondition in the indicated context objects. The different CountConditions are described in the jadice API documentation.

**† getClassFromArguments(Collection,Class)**

Detects an object of the indicated class from the given context objects.

**† getClassesFromArguments(Collection,Class)**

Detects all objects of the indicated class from the given context objects

The method „**doExecute(...)**“ first detects the viewer instance in which a document is to be loaded. In the next step it uses the class FileOpener<sup>85</sup> to select an image file and to have it loaded in the viewer. More information about the class FileOpener are in chapter [4.26.FileOpener](#) or in the jadice API documentation.

Upon completion of the ADocumentOpener command it must be registered for embedding in the configuration files. For this a mapping is defined in the *commands.properties* between a unique command name, e.g. ADocOpener, and the realisation (my.tests.commands.ADocumentOpener).

Example:

```
#commands.properties
...
ADocOpener=my.tests.commands.ADocumentOpener
...
```

Take care to indicate the class name correctly, since commands within the jadice Integrator API are instantiated by reflection.

<sup>85</sup> com.levigo.jadice.util.FileOpener

In the next step a `CommandAction` is defined which is to be embedded later in the file menu of the `CommandsTestFrame`. For this purpose the configuration *actions.properties* is extended by the following lines.

Example:

```
#actions.properties
...
MyOpener.commands = ADocOpener
MyOpener.ShortDescription = Mein Opener
MyOpener.LongDescription = Mein toller Opener
MyOpener.SmallIcon = defaulticons.TB_OPEN
...
```

In the first line it is indicated which commands should be effected when performing the `CommandAction` `MyOpener`. In doing so the name defined in the file *commands.properties* is used. The next line indicates the text which e.g. would appear in the menu or on a button. `LongDescription` corresponds to Tooltip text and `SmallIcon` defines the icon to be used. More details to possible specifications and syntax are in [8.jadice Integrator API: Syntax description of the configuration files](#).

In order to define finally `MyOpener` as part of the file menu, the configuration *menucomponents.properties* is adapted as follows:

Example:

```
#menucomponents.properties
...
file.actions= MyOpener, CloseDocument
file.name=Datei
...
```

The first line says that the file menu consists of `MyOpener` and the `CommandAction` `CloseDocument`. The second line has not been changed and indicates the menu's name.

All in all it may be said that registering a new command in the configuration files is done in three steps.

† Step 1 – `Commands.properties`

Defining the new command's unique name for the use in further configuration files, connected with the indication of the realising class.

† Step 2 – `Action.properties`

Defining of properties of the corresponding `CommandAction`, like for example tooltip text or icon.

† Step 3 – `menucomponents.properties`

Creating structures like menu, submenu or toolbar structures.

[Chart 9](#) shows the newly created command embedded in the file menu of the `CommandsTestFrame`.

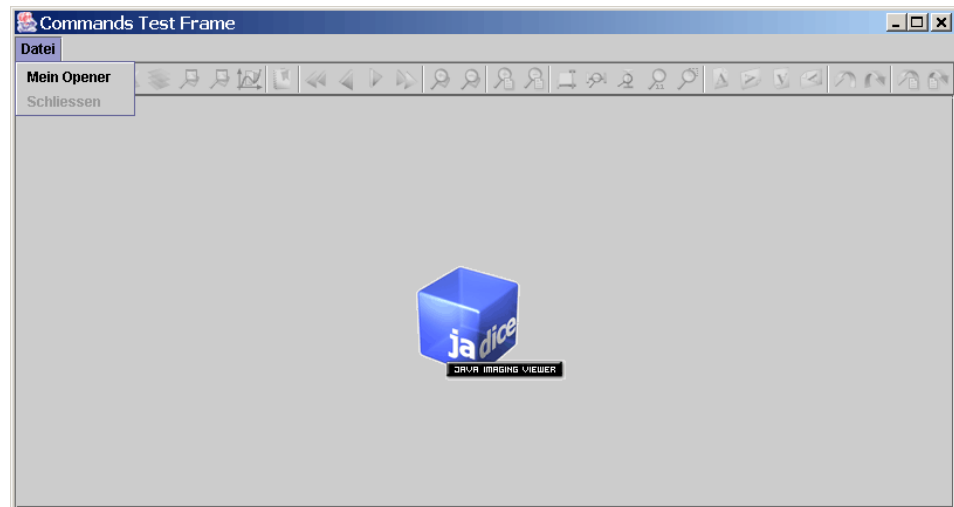


Chart 9 - Embed new command

### Note:

The configuration details are read and evaluated once when starting the viewer. Thus changes on configuration files take effect only after the viewer's restart.

### Note

Bear in mind that the configuration files are provided in internationalised variants. In order to avoid inconsistencies configuration changes should always be done in all variants.

## 5.5. Printing

### 5.5.1. Simple printing

PrinterJava2 is the central class for the printing of documents out of jadice. A simple example is the method „**simplePrint()**“ which extends the class TestViewerFrame.

```
public void simplePrint() {
    PrinterJava2 printer = new PrinterJava2();

    // what to print
    printer.setDocument(getViewer().getDocument());
    // how to print
    printer.setRenderContext(getViewer().getRenderContext());
    // or
    //printer.setRenderContext(new RenderContext());

    // ...and go on
    printer.print();
}
```

Code example 19 - Simple printing



First an instance of the class `PrinterJava2` is created by the default constructor. Alternatively a constructor with a parameterised `PrinterJob` may also be used, in case the integrating application wants to use an adapted `PrinterJob`.

Necessary printing information are a document, the pages to be printed and a `RenderContext` with details for the displaying on a printing device.

For this purpose the method „**simplePrint()**“ passes the document set in the viewer as well as the viewer's `RenderContext` to the `PrinterJava2` instance. With that the viewer's render settings are taken over for printing. Alternatively a `RenderContext` adapted especially for printing may be set. An example to this is in [5.5.3Adaptation of the RenderContext](#).

In the next step the printing process is started. If no other specifications are set by the user in an eventually opened printing dialogue, all pages of the document are printed. Whether a printing or page format dialogue is opened, depends on the set configuration or which specifications have been given by the calling of corresponding methods of the class `PrinterJava2`. See also the following paragraph [7.Configuration and settings](#).

### 5.5.2. Settings

The class `PrinterJava2` allows some optional settings which are specified in the following. All default-settings for printing are described in chapter [7.Configuration and settings](#).

```
PrinterJava2 printer = new PrinterJava2 ();

// what to print
printer.setDocument(myViewer.getDocument());
// how to print
printer.setRenderContext(myViewer.getRenderContext());

// optional...
// show page format Dialog
printer.setShowPageDialog(true);
// or setPageFormat(PageFormat)
// show printer Dialog
printer.setShowPrintDialog(true);
// or setPageSelection(int[])
// enlarge pages to paper size (if they are smaller than
// the paper)
printer.setEnlargePageToPaper(true);
// shrink pages to paper size (if they are larger than the
// paper)
printer.setShrinkPageToPaper(true);
// rotate pages to fit into page format
printer.setOptimizeRotation(true);
// print asynchronously
printer.setAsynchronousPrinting(true);

// and go on...
printer.print();
```

Code example 20- Settings for printing

A page format may be set directly with the method „**setPageFormat(PageFormat)**“ or may be defined by the user in a page format dialogue. The display of a page format dialogue is controled by the method „**setShowPageDialog(boolean)**“. If neither of the two possibilities is used by the integrating application, the page format set in the configuration is used.

It is similar with the required page selection. If only particular pages are to be printed, this may be directly specified by the method „**setPageSelection(int[])**“. It must be noted that the pagination is, like also in the document or in the loader, zero-based. Alternatively the user may specify the required pages in a printing dialogue which may be displayed by the method „**setShowPrinterDialog(boolean)**“. Further the target printer may be defined in this dialogue. As a standard PrinterJava2 prints on the system's default-printer. Take also note of the settings for displaying the printer dialogue in the configuration.

For better printing results the pages to be printed may be optimally adapted in the printable range. For this purpose the following methods are available:

† **setEnlargePageToPaper(boolean)**

Enlarge small pages optimally into printing range.

† **setShrinkPageToPaper(boolean)**

Minimize large pages optimally into printing range.

† **setOptimizeRotation(boolean)**

Rotate pages optimally in printing range.

Whether the printing process is to be performed asynchronously or synchronously may be defined by the method „**setAsynchronousPrinting(boolean)**“. The default setting is asynchronous.

### 5.5.3. Adaptation of RenderContext

In general the viewer's RenderContext may be used for printing, but also an instance adapted by the integrator is possible. For example by indicating a gradation curve for bi-level formats or AnnotationsRenderingHints for the displaying or hiding of annotations.

Changes on the gradation curve of the print-RenderContext are effected by a corresponding setter-method of the RenderContext.

In the following example the RenderContext is thus to be adapted that, independent of the annotations displayed in the viewer, the printout shows only the document without annotations.

For this an adapted RenderContext which is created by the method „**getNoAnnotationsVisibleRenderContext()**“ may be passed to an instance of the class PrinterJava2.

```
public RenderContext
    getNoAnnotationsVisibleRenderContext () {
    RenderContext rc =
```

```
(RenderContext) getViewer().getRenderContext().clone();
AnnotationRenderSettings annoRenderSettings = rc
    .getAnnotationRenderSettings();

annoRenderSettings.setAnnotationRenderingEnabled(false);

return rc;
}
```

*Code example 21- Adaptation of RenderContext*

First the viewer's RenderContext is cloned, so that the changes do not influence the viewer's displaying.

The viewer supports two methods in order to change the visibility of annotations. On the one hand all annotations can be hid/shown, on the other hand all annotations of a particular type can be switched off/on.

For this the class AnnotationRenderSettings<sup>86</sup> is used which is contained as a ProcessingSetting<sup>87</sup> in the class RenderContext. A RenderContext maintains different ProcessingSettings which cover each particular sorts of rendering properties. AnnotationRenderSettings define the visibility of annotations or just of particular annotation types.

The visibility of annotations is set in the example above by the method „**setAnnotationRenderingEnabled (boolean)**“.

Please note that some details of the RenderContext are ignored for printing, since they are unchangeably predetermined by the output device. This refers to zoom, rotation and similar.

<sup>86</sup> com.levigo.jadice.annotation.AnnotationRenderSettings

<sup>87</sup> com.levigo.jadice.docs.ProcessingSettings

## 6. Logging

### 6.1. jadice® Logging Framework Facade

At the initial release of the jadice® document platform version 4.1 a new logging framework facade has been introduced.

Up to now, i.e. in all jadice versions before 4.1.x, in order to get jadice specific messages the interface `LogAdapter`<sup>88</sup> had to be implemented in existing logging systems of the target application and it had to be introduced to the jadice Logging mechanism.

With the new jadice Logging Facade this requirement has been simplified essentially. The new framework allows a straight forward integration of the most popular and wide-spread logging systems and frameworks, like Log4J, SLF4J and via SLF4J JDK 1.4 Logging, Logback, JCL, x4Juli and many more. If you already use one of the frameworks mentioned, the output of jadice messages in one of these logging systems is often only a simple classpath modification.

If no particular logging delegation, i.e. a particular logging framework, is defined and provided by the classpath, a simple default logging will be used. This simple logging will log any (non-debug) messages onto `System.out`.

This simple default logger, which is provided as fallback, simplifies the developing phase. The desired target logging system is not needed any longer at compile time, only at run time the corresponding framework must be available in the classpath. Even if it is not strictly necessary, the target logging system may still be integrated at compile time. Also during the developing phase a different logging system may be used as in the final application.

### 6.2. First steps

Using jadice Logging Framework Facade is very easy. First it has to be sorted out which target logging system is to be used.

The distribution of the jadice document platform provides two implementations of Logging Delegates for this. If Log4J is to be used, add the corresponding Log4J implementation to the application's classpath. For using a different Logging Framework instead take its corresponding SLF4J implementation into the classpath.

A detailed description where the desired implementation of the Logging Delegate may be found in the distribution, if and which further steps are necessary now, is described in the following paragraphs.

Both Logging Delegates contained in the distribution do not need any further specific configuration. An adaptation of the target logging system's behaviour, e.g. of the log level or similar, may be directed by the configuration possibilities of the respective target logging system.

#### 6.2.1. Log4J

The implementation of the Logging Delegate for Log4J corresponds to the following naming convention:

---

<sup>88</sup> `com.levigo.util.log.LogAdapter`

③ `logging-log4j-<version>.jar`

In the distribution you may find the corresponding Log4J delegate, each fitting to the jadice modules being used, under the following directory structure:

③ `lib-jdk15/logging`

③ `lib-all-in-one/jdk15/logging`

Please insert the corresponding Log4J Delegate into the application's classpath. If a log4j configuration is already available on the classpath, no further configuration is needed.

For details about the configuration please see the [log4j homepage](#) and the [log4j manual](#). The jadice® document platform itself does not rely on a specific configuration and does not make any settings on the target logging system.

### 6.2.2. SLF4J

The naming convention of SLF4J's logging delegate corresponds to the following pattern:

③ `logging-slf4j-<version>.jar`

Like for Log4J you will find the corresponding SLF4J delegate, each fitting to the jadice modules being used, under the following directory structure:

③ `lib-jdk15/logging`

③ `lib-all-in-one/jdk15/logging`

Please include the corresponding SLF4J delegate into the application's classpath. Additionally the `slf4j-api-<version>.jar` and a logging implementation is needed. For details about slf4j and supported types of logging delegates and implementations see the [SLF4J homepage](#) or [SLF4J Manual](#), please.

### 6.3. Possible errors

If errors occur (loading of framework facade impossible, etc.), you will find detailed help regarding the error reports in the current distribution of the HTML documentation (`documentation.html`) which is provided in the distribution of the jadice document platform.

## 7. Configuration and settings

In jadice document platform licence and configuration data are separated. The configuration file of the jadice package is called „Jadice.properties“ and it is located in the default package of the jadice document or the jadice All-in-one Jar. It contains specific settings – as for example:

- † settings of the viewer's performance
- † settings of certain AddOns' performance
- † operating system specific settings
- † print settings
- † settings for external AFP/MO:DCA resources
- † cache size
- † lifetime of temp. files

etc.

It will be dwelt on the single parameters in the course of this chapter.

The configuration details are read and evaluated once when starting jadice. Thus changes on the configuration file take effect only after jadice's restart. jadice looks first in its working directory for the configuration. If it does not find it there, it is searched for in the class path. In the case of applets it is - for security reasons - only searched in the class path.

The configuration file should always remain in the default package, own adaptations may be done directly in this file or in a copy in the working directory or class path. However, it is advisable to make own adaptations only in a copy in order to be able to fall back to the default settings.

Integrators get access on the configuration's details by the class `JadicePreferenceHolder`<sup>89</sup>.

Example:

```
JadicePreferenceHolder.getInstance()  
    .getPreferenceStoreByName(JadicePreferenceHolder.JADICE_C  
    ONFIGURATION)
```

This call returns an object of the class `PreferenceStore`<sup>90</sup> containing the loaded configuration data. A `PreferenceStore` resembles a properties object, but it allows qualified and type-safe access on the configuration settings.

The jadice family administrates properties and settings in `PreferenceStores`. The class `PreferenceStore` is an interface which, similar to the class `Properties`<sup>91</sup>, administrates access on data by key-value pairs. The advantage of `PreferenceStores` is the encapsulated access on the contained data. Being an interface own implementations may be registered which allow to use data from any sources, e.g. from a database, a file-system, the intra-/extranet or similar.

It is optional to integrators to have own implementations administrated by the `JadicePreferenceHolder`. Take further details from the jadice API documentation, particularly in regard of the classes `PreferenceStore`, `JadicePreferenceHolder`, `PreferenceStoreHolder` and `PropertiesPreferenceStore`.

<sup>89</sup> `com.levigo.jadice.util.JadicePreferenceHolder`

<sup>90</sup> `com.levigo.util.preferences.PreferenceStore`

<sup>91</sup> `java.util.Properties`

## 7.1. The most important settings in detail

Option	Purpose
Printing – Initialisation of the printer class	
<code>jadice.viewer.show-print-dialog=true</code>	Is a printing dialogue to be displayed?
<code>jadice.viewer.show-pageformat-dialog=true</code>	Is a page format dialogue to be displayed?
<code>jadice.viewer.printer-page-format-enabled=true</code>	Is a default page format to be set?
<code>jadice.viewer.printer-page-format-size-x=210</code> <code>jadice.viewer.printer-page-format-size-y=297</code> <code>jadice.viewer.printer-page-format-border-x-left=10</code> <code>jadice.viewer.printer-page-format-border-x-right=10</code> <code>jadice.viewer.printer-page-format-border-y-top=10</code> <code>jadice.viewer.printer-page-format-border-y-bottom=10</code> <code>jadice.viewer.printer-page-format-orientation=1</code>	Definition of a default page format, it is <u>only</u> used, if the use of a default page format is activated, i.e. the property <i>jadice.viewer.printer-page-format-enabled</i> = <i>true</i> . Size information in mm. Page format: 0 = landscape; 1 = portrait
Default Print Commands	
<code>jadice.viewer.printer.commands.DefaultPrintMode=PrintAll</code>	Determines the default printing mode of the jadice printing commands. Three modes are available: <ul style="list-style-type: none"> <li>~ PrintAll - document and annotations are printed.</li> <li>~ PrintOnlyDocument – Only the document is printed.</li> <li>~ PrintOnlyAnnotations – Only the annotations are printed.</li> </ul> Default value: PrintAll
<code>jadice.viewer.printer.commands.DefaultPrintAdjusting=FitPrint</code>	Defines when using the jadice printing commands how documents are adapted (scaled) in the printable range. The following values are possible: <ul style="list-style-type: none"> <li>~ OrigSizePrint – Prints the document in original size.</li> <li>~ ShrinkPrint – Adapts the document into printable range by shrinking.</li> <li>~ EnlargePrint – Adapts the document into printable range by enlarging.</li> <li>~ FitPrint – Adapts the document always into the printable range, i.e. according to page size the displaying of the document is zoomed in or out.</li> </ul> Default value: FitPrint

Option	Purpose
Target system specific adaptations	
jadice.viewer.printjobname.maxlength=40	Under windows it may happen that print jobs with a too long print job name are not accepted and cancelled without further error warning. This setting defines the maximum length of print job names which are initiated out of the jadice package. Possible values: positive whole numbers greater than 0. If an invalid value is given or this setting is commented out, no limitation of the print job name is done.
jadice.viewer.printer-transparent-fix=auto	Depending on the graphics card, the monitor setting or the print driver printing problems may occur with transparent image areas or image elements. In the enabled state a workaround is activated which avoids this problem, but results in a bigger printing output. Possible values: true, false, auto Default value: auto
Hover Lens	
jadice.hover-lens.use-click-scaling=true	Setting, if the lens's enlargement factor is to be changeable by mouseclicks. Possible values: true, false Default value: true
jadice.hover-lens.default-scale=150	Initial zoom, lens's value as percentage. Possible values: Whole numbers greater than 0. Default value: 150, corresponds 150%
jadice.hover-lens.click-scale-step=25	Zoom step as percentage, in which the lens's enlargement factor changes on mouseclick (only important, if <i>jadice.hover-lens.use-click-scaling = true</i> ). Possible values: Whole numbers greater than 0. Default value: 25, corresponds 25%
jadice.hover-lens.shape=1	Shape of hovering lens. Possible values: 1 - rectangle 2 – round Default value: 2, corresponds round. Note: integrators may define any shape for the lens by API, however, using the configuration file only these



Option	Purpose
	two shapes are offered.
jadice.hover-lens.size.width=150 jadice.hover-lens.size.height=150	Size of hovering lens in pixel. Possible value: Whole numbers greater than 0. Default value: Width: 150 Height: 150
jadice.hover-lens.autoscroll.mode=true	Setting, whether the lens initiates an autoscroll behaviour as soon as the mouse leaves the document area. Possible values: true, false Default value: true
<b>Page Sorter – minimised displaying</b>	
jadice.sorter.show-page-numbers=false	Defines, if the page sorter shows page numbers. Possible values: true, false Default value: false
jadice.sorter.single-click-navigates=false	Setting, if by single clicking in the page sorter a page number change is to be effected in the viewer or not.  Possible values: ~ true page navigation is effected by single clicking on a page in the page sorter. ~ false A single click on a page in the sorter selects this very page, a double click effects a viewer's turn over on this page. Default value: false
<b>Zoom Policy</b>	
jadice.viewer.apply.zoom-policy.resize=true	Setting, if the zoom policy is to be applied even with size changes of the viewer. Note: ~ Zoom policy behaviour has always got minor priority than user settings. I.e., if the user himself changes the zoom setting, the zoom policy is ignored. ~ Is only supported in connection with zoom policy „fit“, „fit width“, „fit height“. Possible values: true, false. Default value: false.
jadice.viewer.zoom-policy=2	Zoom setting for reloaded documents. Possible values: 1 – Keeps the zoom value.

Option	Purpose
	2 - „fit“-mode, adapts document into viewer. 4 - „fit width“-mode, adapts document horizontally into viewer. 8 - „fit height“-mode, adapts document vertically into viewer. 16 - "100%" mode, displays the document in its original size. 32 - "page-fit" mode, adapts each page into viewer, unless the user has not defined any other pages or any other document zoom value. 64 – "page-fit width" mode, adapts each page horizontally into viewer, unless the user has not defined any other pages or any other document zoom. 128 – "page-fit height" mode, adapts each page vertically into viewer, unless the user has not defined any other pages or any other document zoom. Default value: 1
<b>Afp-specific</b>	
jadice.viewer.afp-resource-extension=ovl;300	Registrating of special file extensions for external Afp resources.
jadice.viewer.afp-resource-path=d:\afp_res\	Registrating of a particular Afp resource directory.
<b>Temporary files (-&gt; FileCacheInputStream)</b>	
jadice.viewer.delete-overaged-tmps = TRUE	Automatical removing of remaining temporary files.
jadice.viewer.overaged-tmps-lifetime=0	Number of days during which the temp. files must not be deleted. Default value: system temp. directory
jadice.viewer.tmps-path=c:/temp	Path for temp. files
<b>Annotations</b>	
jadice.viewer.annotation.type=vi	Indicates which kinds of annotations are processed. The annotation type defines the structure of the annotation toolbar and the properties of the annotation editors. vi = Visual Info compatible annotations; fn = FileNet compatible annotations fnp8 = FileNet P8 compatible annotations Default value: vi
jadice.viewer.annotation.creation.mode=0	Indicates how the tools for the creation of annotations behave.

Option	Purpose
	<p>Possible values:</p> <ul style="list-style-type: none"><li>(0) nonpermanent mode = The tool is automatically deselected after an annotation has been created;</li><li>(1) permanent mode = If a tool is selected, it remains selected as long as the user deselects it;</li><li>(2) both = If the tool has been selected by a single mouseclick, the behaviour corresponds to the „nonpermanent“ mode, with doubleclicking to the „permanent“ mode.</li></ul> <p>Default value: 0</p>

## 8. jadice Integrator API: Syntax description of the configuration files

The configuration files of the jadice document platform contained in the distribution package may be found under **com.levigo.jadice.properties.\*** or **com.levigo.jadice.graphics.\***. They are described in detail in the following paragraphs.

*Note:*

For the creation of an own configuration take into account that the configuration files contain references on corresponding configurations. These references must be adapted accordingly.

If own configuration are not to be created, it is advised to copy the jadice configuration files and to place them in the class path ahead of the jadice document platform modules. So changes which have been done are recognised, but the original configuration remains untouched.

*Note:*

Bear in mind that the configuration files are provided in internationalised variants. In order to avoid inconsistencies configuration changes should always be done in all variants, at least, however, in the variant corresponding to the current locale.

### 8.1. The file „commands.properties“

This configuration creates a mapping between a unique command name which is used as reference in other configurations and its realisation.

In general the syntax may be provided as follows:

Option	Purpose
CommandName=CommandPfad.CommandKlassenName	The command identifier is a freely selectable, but unique name which is used in other configurations as reference. The specification of the command realisation consists of its path and class name. Pay attention to case sensitivity here, since commands are created by reflection.

Example:

```
MyTestCommand1=test.myTests.AllTestClasses$ATestCommand
or
MyTestCommand2=test.myTests.ATestCommandClass
```

In the first line a command is described which is called **MyTestCommand1** and which has been realised as inner class named **ATestCommand** of the enclosing class **AllTestClasses**. In the second line another command is provided which is called **MyTestCommand2** and is presented by the class **ATestCommandClass**.

## 8.2. The file „menucomponents.properties“

The menucomponents-configuration defines the structure of menus, sub-menus, context-menus and toolbars, called structures in the following. In the following paragraph [8.3.The file „actions.properties“](#) a corresponding *actions.properties* is defined which describes the properties of the CommandActions.

The syntax of a structure may now be described as follows, considering that the black-written „name“ is a freely defined, but unique structure identifier. Red statements are fixed expressions:

Option	Purpose
Definition of a structure	
name.name=TestMenuName	Name of a structure. With menus it is additionally used as menu name for displaying in a menubar.
name.actions=CommandName1, CommandName2	Definition of the commands contained in the structure which are used to create menu inputs or toolbuttons. Commands are provided as a comma-separated list. This way of definition may be used for all structures.
name.actions.toolbar=CommandName1,CommandName2	Analogical definition of the contained commands, especially only for toolbar structures.
name.actions.contextmenu=CommandName1, CommandName2	Analogical definition of the contained commands, especially only for context menus.
name.actions.menu=CommandName1, CommandName2	Analogical definition of the contained commands, especially only for menu structures.
name.menuState=	Only for checkbox / radiobutton menu items. Describes the initial selection state. Value: selected all other statements – not selected
name.menuType=	Only for menu items: Without specification a normal menu item is created.  Possible values: † visibilityEnabled – item which is only visible, if enabled † checkbox – checkbox item † visibilityEnabledCheckbox – checkbox item which is only visible, if enabled † radiobutton

Option	Purpose
	† visibilityEnabledRadiobutton – radiobutton item which is only visible, if enabled † iconmenu – simple menu item according to set look & feel with icon displaying, if set † without specification – simple menu item according to set look & feel, without icon
Structure defining settings	
{name}	Substitution part-menu
{>name}	Substitution sub-menu
	Separator

Hereto an example:

```
PartMenu.actions=Command1,Command2
SubMenu.actions=Command4,Command3
SubMenu.name=a SubMenu
SuperMenu.actions=Command5,|,{PartMenu},|,{>SubMenu}
SuperMenu.name=a super menu
```

With the definition above a menu titled „a super menu“ is created that consists of Command5, a separator, Command1, Command2, another separator and a submenu named „a SubMenu“ which is composed of Command4 and Command3.

The indication of the corresponding configuration file *actions.properties* is defined as follows:

```
resource.defaultactions=/com/levigo/jadice/resources/properties/actions.properties
resource.actions.default=defaultactions
```

Further action configurations may be specified as follows:

```
resource.moreActions=/my/action/definitions/actions.properties
```

In order to differentiate, if a command is defined by the default-action configuration or any other configuration, the configuration identifier is prefixed.

Example:

```
PartMenu.actions=Command1,moreActions .Command2
```

Here the part-menu consists of *Command1*, defined by the default-action configuration, and *Command2*, defined by the configuration named *moreActions*.

## 8.3. The file "actions.properties"

The file *actions.properties* describes the properties of CommandActions. This involves e.g. which commands are to be activated in which order for the action's performance, which icon is to be displayed, if a tooltip is to be offered, etc.

Additionally a corresponding command configuration (*commands.properties*) and an icon description are each indicated by

```
# icon description
icons.defaulticons=/com/levigo/jadice/resources/graphics/j
adice-viewer
resource.icons.default=defaulticons
# Commands configuration
resource.commands=/com/levigo/jadice/resources/properties/
commands.properties
resource.commands.default=commands
```

As also in the menucomponent.properties further action configurations may be defined, it is possible to define different command configurations in the action.properties.

Example:

```
resource.mycommands=/my/command/definitions/commands.prope
rties
```

Please compare also [8.1.The file „commands.properties“](#) and [8.2.The file „menucomponents.properties“](#).

The possible specifications per action are listed in the following table, considering that the black written „name“ is a freely defined, but unique action identifier. Red statements are fixed expressions:

Option	Purpose
Definition of a Command Action	
name.SmallIcon=defaulticons.TB_OPEN	Icon definition referred to defaulticons-reference. Example: TB_OPEN – icon name in icon reference
name.commands=	Comma-separated list of commands which are to be activated in the actionPerformed method.
name.ShortDescription=short description	Name, e.g. for menu entry
name.LongDescription= description text	Tooltip text, specifications may be provided in HTML.
name.AcceleratorKey= VK_N + CTRL_MASK	Only for menu items: Hot Key for menu items Modifier plus key Alt-, Shift- or Ctrl-Mask + KeyEvent.Key Analogical identifier as provided by the class KeyEvent <sup>92</sup> or the class InputEvent <sup>93</sup> are to

Option	Purpose
	be used.
name.MnemonicKey= VK_N	<p>Only for menu items: Mnemonic key for navigation through menus.</p> <p>Note: Mnemonic keys are always connected with the modifier ALT-MASK.</p>
name.InputMap= VK_PLUS	<p>Entry in InputMap of context owner in mode: „WhenInFocused-Window“.</p> <p>If no component of the used component's hierarchy consumes the indicated key-event, the event is used to activate the command. Key bindings preallocated by the used look&amp;feel principally prioritize command bindings.</p> <p>Key combinations may be indicated by different key identifiers connected with a plus sign. Example: SHIFT_MASK + VK_A creates an entry in the InputMap which effects a command at the key-event Shift-A.</p> <p>Note: KeyBindings predefined by the system or by the used look&amp;feel principally prioritize the specifications made here. If a KeyBinding is to be configured differently as provided by the system or the look&amp;feel, the integrator has to make sure that the predefined KeyBindings have been removed. Principally jadice does not change by itself any predefined settings.</p>

## 8.4. The file „jadice-viewer.properties“

All icons of the jadice package have been combined in a PNG-file (jadice-viewer.png) because of lucidity and resource improvement. Being a web format PNG can display icons very well (alpha channel with variable transparency, gamma correction and similar) with an optimised compression which is between 5% to 25% better than GIF.

Adapted icons need not to be provided in PNG format, in the same way GIF and JPEG formats are also supported.

The *jadice-viewer.properties* configuration describes the provided icons in a body. In this table, too, the red statements are fixed expressions.

92 java.awt.event.KeyEvent

93 java.awt.event.InputEvent



Option	Purpose
<code>extension=png</code>	<p>Format of icon file; the icon file has got the same name like the configuration, the suffix results from the format.</p> <p>Possible values: png, gif, jpeg</p>
<code>icon.name.rectangle=0,0,24,24</code>	<p>Where in the icon file is the corresponding icon with the name „name“?</p> <p>All icons are described in position and dimension by this specification.</p> <p><i>name</i> is a freely selectable, but unique identifier for this icon and is used in the configuration <i>actions.properties</i> as reference.</p> <p>Example: 0,0 corresponds to origin of icon position. 24,24 corresponds to icon dimension.</p>

## 9. jadice Public API and internal Packages

### 9.1. jadice Public API

jadice® document platform offers to developers and integrators a powerful Public API for integration in specific applications and solutions of customers. Current Javadoc information of the Public API are provided in the distribution in the directory

**<distributiondirectory>/javadoc .**

This directory contains several Jar-files with Javadoc information about the Public API of the different modules of the jadice® document platform.

All classes and methods contained and described in these Javadoc documentations are part of the Public jadice® API and may be used freely to integrate jadice® components and functionalities.

The classes, methods and functionalities – described by the Javadoc information - of the Public jadice® API are fully supported and maintained by levigo solutions.

In case of questions or problems with the Public jadice® API or if additional information about further modules are required, integrators or developers may address levigo solutions at any time. We would like to support you.

### 9.2. Jadice Private API

Apart from the Public API the distribution package contains internal classes, methods, functionalities and part-functionalities with the following name **\*.internal.\***.

These structures are part of the internal **Private** jadice® API and must not be used directly by developers and integrators.

levigo solutions reserves the right to delete, remove, rename or to change in its functionality all contents of the **Private** jadice® API, classes, methods and functionalities. There is no guarantee that the classes, methods and functionalities of the **Private** jadice® API are completely provided regarding single versions or releases.

Technical support in case of problems due to the direct use of the **Private** jadice® API is excluded from the software maintenance and will not be supported in any case.

Any direct use of the internal **Private** jadice® API frees levigo solutions from all warranty and liability claims. levigo solutions is not liable in any way for all direct, indirect, accidental, special, exemplary or other damage which occurs or might occur by the direct use of the internal **Private** jadice® API.

## 10. Document history

Version	Date	Author	Modifications
0.1	14.07.03	Oliver Suck	Developer's documentation of Vs 2.x as template
	16.07.03	Jörg Henne	- New in Version 3.x - Document / Layer concepts
	06.08.03	Carolin Köhler	- Online Service - 2.2 Extension of concepts and descriptions - 2.3 Format updating
	28.08.03	Carolin Köhler	- Updates - Viewer part1
	29.08.03	Carolin Köhler	- Viewer part2 - Document - Demonstration classes
	01.09.03	Carolin Köhler	- Page - PageSegment - Loader
	02.09.03	Carolin Köhler	- Loader description: resources - ResourceLoader in general - Annotations' hierarchy
	08.09.03	Carolin Köhler	- RenderContext
	09.09.03	Carolin Köhler	- ResourceLoader, hierarchy, diagramme - ResourceFileLoader - ResourceURLLoader - ResourceGroupLoader - ResourceMultiLoader - LoadListener
	10.09.03	Carolin Köhler	- SeekableInputStreams, diagramme
	15.09.03	Carolin Köhler	- FormatInfo, FormatFile - ImagePlusAnnotationFormatInfo - ImagePlusAnnotationFile
	16.09.03	Carolin Köhler	- JadiceBookmark
	17.09.03	Carolin Köhler	- BookmarkPanel
	18.09.03	Carolin Köhler	- SeekableInputStreams - RandomAccess, FileInput, Memory - BookmarkPanel subsections - JadiceBookmarkHandler - Involved classes - PageSorter
	19.09.03	Carolin Köhler	- EditPanels - PrinterJava2 - NavigatorPanel
	22.09.03	Carolin Köhler	- AddOns, creation, call, integration - Navigator part 2 - Lens

Version	Date	Author	Modifications
			<ul style="list-style-type: none"> <li>- HoverLens</li> <li>- GradationCurveControl</li> <li>- GradationCurve</li> <li>- GradationCurveFileHandler</li> <li>- Demo class revise</li> </ul>
	23.09.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- correction, revision, rounding off chapter 1, 2, 4</li> <li>- preparation chapter 5, 6</li> </ul>
	24.09.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- chapter 6 finished</li> <li>- BasicJadicePanel, AbstractJadicePanel</li> <li>- preparation chapter 3</li> </ul>
	25.09.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- chapter 5.1</li> <li>- chapter 5.2.1</li> <li>- chapter 5.2.2</li> <li>- chapter 5.2.3</li> <li>- chapter 5.2.4</li> <li>- chapter 5.2.5</li> <li>- chapter 5.2.6</li> <li>- chapter 5.2.7</li> <li>- chapter 5.2.8</li> </ul>
	26.09.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- chapter 5.5</li> <li>- chapter 5.5.1</li> <li>- chapter 5.5.2</li> <li>- chapter 5.5.3</li> </ul>
	29.09.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- chapter 5.3.1</li> <li>- chapter 5.3.2</li> <li>- chapter 5.3.3</li> <li>- chapter 5.3.4</li> <li>- chapter 7</li> <li>- chapter 7.0.1</li> <li>- chapter 7.0.2</li> <li>- chapter 7.0.3</li> <li>- chapter 7.0.4</li> </ul>
	30.09.03	Carolin Köhler	Document corrections, layout <ul style="list-style-type: none"> <li>- chapter 3</li> <li>- chapter 3.1</li> <li>- chapter 3.2</li> <li>- chapter 3.3</li> </ul>
	01.09.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- Revision chapter 4</li> </ul>
	02.09.03	Carolin Köhler	Final corrections, layout <ul style="list-style-type: none"> <li>- chapter 5.4</li> <li>- chapter 5.4.1</li> <li>- chapter 5.4.2</li> <li>- chapter 5.4.3</li> </ul>
	30.10.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- chapters 6,7 extended by new settings</li> </ul>
	04.11.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- chapter 7 extended by new settings</li> </ul>
	22.12.03	Carolin Köhler	<ul style="list-style-type: none"> <li>- new template</li> <li>- Updates and extensions of the jadice package added to documentation</li> </ul>

Version	Date	Author	Modifications
	19.02.04	Carolin Köhler	- different minor corrections - DocumentSaver
	02.03.04	Carolin Köhler	- Configuration parameter updated
	19.03.04	Carolin Köhler	- Hyperlinks updated
	26.08.04	Carolin Köhler	- Configuration parameter updated
	15.09.04	Carolin Köhler	- Configuration parameter updated
2.0	06.12.05	Jelkica Ćirilović Carolin Köhler	Complete revision
	28.02.07	F. Fernandes C. Köhler J. Ćirilović	Updated to jadice version 3.1
4.1	31.03.08	C. Köhler J. Ćirilović	Updated to jadice version 4.1
	03.07.08	C. Köhler J. Ćirilović	Updated and translated
4.2	18.02.09	C. Köhler J. Ćirilović	Updated to jadice version 4.2