Carolin Köhler

Februar 2009

jadice® document platform

Version 4.2.x

Dokumentation für Entwickler

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

Inhaltsverzeichnis

1. Allgemeines

1.1. Über diese Dokumentation

1.1.1. Allgemeines

Der hier vorliegende Leitfaden führt in die technischen Zusammenhänge der jadice® document platform (im Folgenden kurz jadice genannt) ein.

Die Dokumentation beschränkt sich im Wesentlichen auf die Bereiche, die für Entwickler interessant sind (im Folgenden auch Integratoren genannt), um jadice® in eigene Applikationen zu integrieren.

Zur besseren Lesbarkeit werden Paketnamen nur in Fußnoten voll qualifiziert dargestellt.

Eine API-Referenz im *javadoc*-Format wird separiert in der jeweiligen Distribution versionsaktuell zur Verfügung gestellt.

1.1.2. Feedback

Sollten Ihnen bei der Verwendung dieser Dokumentation Fehler auffallen oder möchten Sie Verbesserungsvorschläge einbringen, senden Sie bitte eine möglichst detaillierte Nachricht an solutions@levigo.de.

Ihr Feedback hilft bei der Weiterentwicklung dieser Dokumentation. Vielen Dank.

1.2. Über das Produkt

jadice hat sich vom plattform- und formatunabhängigen Dokumentbetrachter jadice viewer, der aufgrund seiner Flexibilität und Leistungsstärke besonders im Archivbereich eingesetzt wurde, zur flexiblen Komponentenlösung für den Einsatz im professionellen Dokumentenmanagement entwickelt.

Als integrationsfreundliche Java Toolbox mit einsatzorientierten Modulen, durchdachten Schnittstellen und hilfreichen Zusatzkomponenten bietet jadice die Basis für individuelle Archivclient-Lösungen. Der Dokumentbetrachter ist dabei ein wesentlicher Bestandteil von jadice geblieben – allerdings mit weiterentwickeltem und erweitertem Funktionsumfang.

1.3. Auslieferung

jadice steht jetzt in zwei Paketierungsvarianten mit jeweils dem gleichen Funktionsumfang zur Verfügung: als bisherige Komplettlösung sowie in neuer Form als modulare Lösung mit zu Einheiten zusammengefassten Funktionen.

Da beide Varianten in der Auslieferung enthalten sind, kann der Integrator je nach Situation, Einsatz und Lösung selbst entscheiden, ob er die bisherige Auslieferungform als All-in-one-Lösung oder nur die benötigten und gewünschten Module einsetzen möchte.

Ermöglicht werden diese Paketierungsvarianten durch das neue Architekturkonzept von jadice, das auf einer Komponentenstruktur basiert. Dabei werden Komponenten-Librairies und ihre Abhängigkeiten zu sinnvollen Einheiten zusammengefasst. Diese Struktur liefert dem Integrator die Flexibilität nur ausgewählte funktionale Einheiten zu benutzen, wenn die Gesamtlösung so komprimiert und effizient wie möglich sein soll - beispielsweise im Umfeld von Webanwendungen, wenn Übertragungsraten

relevant sind. Alternativ besteht durch die All-in-one Lösung auch weiterhin die Sicherheit den kompletten Funktionsumfang einzusetzen.

Mit der All-in-one Variante hat der Integrator immer alle Funktionalitäten der jadice document platform zur Verfügung, doch müssen Updates und Test auch für die gesamte Lösung vorgenommen werden.

Die modulare Variante ermöglicht dem Integrator nur die benötigten Module zu benutzen. Dabei muss er jedoch immer sicher stellen, dass er alle voneinander abhängigen Dateien und Libraries zur Verfügung hat. Hinsichtlich von Updates und Tests ist diese Variante besonders pflegeleicht, da nur das betreffende Modul oder die betreffende Funktion aktualisiert und geprüft werden muss.

Eine Aufstellung aller Module der jadice document platform sowie deren Abhängigkeiten steht in der HTML-Auslieferungsdokumentation zur Verfügung.

Wichtiger Hinweis:

jadice document platform benötigt eine Java Virtual Machine ab 1.5. oder höher.

- Benutzen Sie für JVM 1.5. oder neuer die Libraries im Ordner **jdk15** bzw. **lib-jdk15**.

1.4. Online-Service

Für Entwickler und Integratoren haben wir einen Online-Dienst, mit dem Sie direkt Ihre Wünsche, Probleme, Verbesserungsvorschläge oder ähnliches zur jadice document platform angeben und an die entsprechenden jadice Entwickler weiterleiten können.
Im Folgenden werden Sie dann automatisch über alle Stellungnahmen, Anzeigen über den Status des Problems bis hin zur Lösung per eMail informiert. Sie selbst können natürlich auch jederzeit weitere Informationen oder Hinweise hinzufügen.
Bitte kontaktieren Sie bei Interesse solutions@levigo.de.

Eine HTML-Dokumentation mit zusätzlichen Informationen steht Ihnen in englischer Sprache als Teil der Auslieferung versionsaktuell zur Verfügung. Sie finden Sie unter
jadice-documentplatform-<<Versionsnummer>>-dist.dir|documentation.html

2. Einführung

2.1. Funktionsumfang der Version 4.2

Mit jadice in der Generation 4.2 erhalten Sie eine mächtige Toolbox für die unterschiedlichsten Aufgaben mit verschiedenen Dokumentdaten und Formaten. Dieser Abschnitt soll Ihnen eine kleine Einführung in die Neuerungen der jadice document platform geben.

Eine verbesserte Unterstützung des **PDF**- und **PDF/A-**Formatumfangs ermöglicht nun die Verarbeitung der Schrifttypen **Type1**, **CompactFont** und **Type0**. Des weiteren kann das bisher nicht unterstützte FileNet Image Format (**fni**), ein proprietäres FileNet bi-level Dokument-Format, jetzt auch angezeigt und in vollem Umfang genutzt werden.

Die bisherige Unterstützung von **FileNet P7**-Annotationen ist um die Unterstützung von **FileNet P8**-Annotationen mit vollem Funktionsumfang erweitert worden.

Dank eines neuen Logging-Frameworks ist es nun möglich, andere bereits bestehende Loggingsysteme durch Delegation einfach und nahtlos in die jadice document platform einzubinden. Verwirklicht wird dies durch eine Logging-Facade, die eine Abstraktionsebene von bekannten Logging-Systemen wie **Log4J**, **JDK 1.4 Logging** oder **Logback** bietet und somit für deren Verwendung meist nur eine Anpassung des Klassenpfades benötigt.

Weitere Logging-Frameworks können via **SLF4J** eingebunden werden.

2.2. Besondere jadice Merkmale

2.2.1. Verarbeitung sehr großer Dokumente

- z.B. technische Zeichnungen, Architektur- oder Lagepläne: TIFF-Dateien mit mehr als 20.000 Pixel[2]
- z.B. Farbbilder: erheblich größere komprimierte Datenmengen

2.2.2. Verarbeitung virtueller Dokumente

- jadice Dokumente können aus verschiedenen Dokumentdaten unterschiedlicher Herkunft (z.B. ein TIFF und ein MO:DCA) bestehen.
- Anzeige von zusammengesetzen jadice Dokumenten bereits während des Ladevorgangs der zugehörigen Dokumentdaten, sofern das entsprechende Format dies zulässt.
- Unterschiedliche Dokumentdaten können nacheinander (wie Seiten) oder übereinander (als verschiedene Seitenelemente) dem jadice Dokument hinzugefügt werden.
- Bearbeitung von Dokumenten mit Layern (übereinander-/ nebeneinanderliegende Seitenelemente)
  - z.B. Briefkopf und -text in getrennten Dateien
  - Seiten bestehen aus verschiedenen Schichten, die aus unterschiedlichen Ressourcen stammen können.

Durch diese Flexibilität bietet die jadice Architektur die Möglichkeit, virtuelle Dokumente aus verschiedenen physikalischen Dokumenten zu erstellen und diese zusammengesetzte Struktur mit vollem Funktionsumfang wie ein einfaches Dokument zu nutzen.

Virtuelle Dokumente werden oft auch genutzt, um zusammengehörige Vorgänge oder Akten zu repräsentieren.

2.2.3. Vielzahl direkt unterstützter Dokumentformate

Eine weitere Besonderheit stellt die breite Palette der unterstützten Dokument-Formate dar. Eine Auflistung der zur Zeit unterstützen Formate ist in Abschnitt 2.5.Formate zu finden.

Bilddateien werden bei jadice nur in seltenen Fällen über einen "Datei öffnen"-Dialog vom lokalen Dateisystem geladen. In der Regel erhält jadice Dokument- und Annotationsdaten von einem Dokumenten Management System.

Weiterführende Informationen und eine Produktübersicht findet sich auf der levigo Website[1].

2.2.4. Einfache Integrationsmöglichkeiten mit der jadice Integrator API

Die jadice document platform bietet sehr einfache und flexible Integrationsmöglichkeiten an.

Eine einfache Einbindung von Viewer-Komponeten mit flexibler Bereitstellung und Anpassungsmöglichkeiten von Funktionen sowie die Wiederverwendbarkeit und Erweiterbarkeit von bestehenden Tools und Actions.

- Einfache Viewer Integration:
  - BasicJadicePanel: die einfachste Möglichkeit einen Viewer einzubinden
  - mit Toolbar, Annotation Toolbar, Statusbar, optional Menubar
  - Veränderungen an Tools oder Menüs nur durch textuelle Anpassung der Konfigurationsdateien
- Actions und Commands: Kommandoverarbeitung mit Command- und Action Pattern
  - Kapselung in einzelne Actions
    - Actions aktualisieren eigenständig Verfügbarkeit (enable state)
    - einfache Anpassung an corporate identity, corporate design
    - flexible Veränderung von Eigenschaften: Icons, Accessors, InputMaps etc.
  - über Konfigurationsdatei Strukturierung von ToolBar und Menüs
  - einfache Definition von verschiedenen Look & Feels, sowie zugehörigen Menü und Toolbar Strukturen, die je nach Bedarf verwendet werden können z.B. für
    - reine Recherche

1 Siehe http://www.levigo.de/de/dokumentenmanagement/

- Sachbearbeitung
- Verwaltung etc.

- Bereitstellung von AddOns:
  - AddOns sind nützliche, direkt integrierbare Erweiterungen für Viewer-Komponenten oder zur Dokumentbearbeitung.
  - Zur Integration bieten die AddOns jeweils funktionale Komponten als JComponent mit gekapselter Funktionalität und/oder zusätzlich Fensterelemente als *Jframe|JInternalFrame*.und/oder alternativ als *Jrame|JInternalFrame.*

2.2.5. Die wichtigsten technischen Funktionalitäten auf einen Blick

- Dokumente[2] sind formatunabhängig, d.h. Dokumente sind nur „Behälter" für Seiten.
- Seiten[3] sind formatunabhängig und aus Ebenen[4] zusammengesetzt.
- „Besetzte" Ebenen einzelner Seiten, so genannte Seitensegmente, tragen die eigentlichen Dokumentdaten und sind damit formatspezifisch.
- Eine flexible Darstellung von Inhalten durch virtuelle Dokumente, die aus verschiedenen physikalischen Dokumentenquellen erstellt werden können. Als solches können auch komplexe Strukturen wie Akten oder Vorgänge einfach erstellt und genutzt werden.
- Die Eingabe/Ausgabe-Verarbeitung (insbesondere das Laden von Dokumenten) ist auf optimale Speichereffizienz ausgelegt. Bei Bedarf kann zusätzlich dieses Verhalten durch den Integrator feingranular auf die Einsatzumgebung und die gewünschten Anforderungen angepasst werden.
- Einfache Integration von mitgelieferten oder eigenen Funktionalitäten und visuellen Repräsentationen.

2.3. Systemvoraussetzung

Grundsätzlich arbeitet jadice als reine Java Applikation plattformunabhängig. Damit werden alle Betriebssysteme unterstützt, für die eine JVM Machine ab 1.5.x oder neuer zur Verfügung steht.

- Empfohlen ist die Verwendung einer JVM ab Version 1.5. oder höher.
- Empfohlen wird mindestens 256MB Heapsize für die Verwendung der jadice document platform. Der Speicherbedarf der integrierenden Anwendung sollte dabei gesondert berücksichtigt und der Heapsize der jadice document platform hinzugerechnet werden, um die empfohlene Heapsize der Gesamtanwendung zu bestimmen. Der so ermittelte Speicherbedarf kann dann mittels des VM-Parameters -xmx der Anwendung zur Verfügung gestellt werden.

2 Instanzen der Klasse com.levigo.jadice.docs.Document
3 Instanzen der Klasse com.levigo.jadice.docs.Page
4 Instanzen der Klasse com.levigo.jadice.docs.DocumentLayer

2.4. Begriffe

2.4.1. Das Dokumentenmodell

In Abbildung 1 ist das Dokumentenmodell von jadice skizziert. Das Modell besteht aus vier Elementen, die im Folgenden näher beschrieben werden.

2.4.1.1. Dokumente (Documents)

In der Praxis kann man Dokumente als eine Zusammenfassung von Seiten betrachten, die in einer Applikation als Einheit verarbeitet werden. Hierbei ist zu

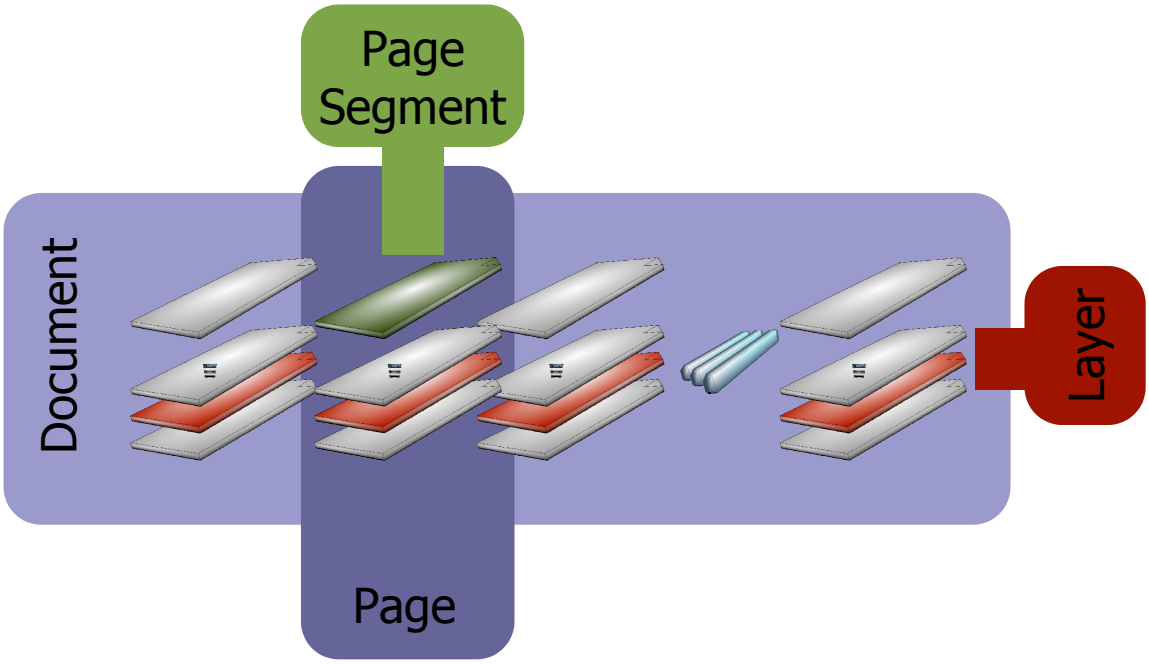

Abbildung 1 - Dokumentenmodell

beachten, dass kein feststehender, permanenter Zusammenhang zwischen jadice Dokumenten und physikalischen Dokumenten, Datenströmen bzw. Datenformaten bestehen muss. Ein jadice Dokument kann seine Seiten aus mehreren Datenströmen bzw. Datenformaten beziehen, aber auch die Seiten selbst können sich in ihrer Darstellung zusammensetzen aus Daten verschiedener Datenströme.

jadice Dokumente besitzen somit eine zweidimensionale Struktur:

- sie bestehen aus einer Anzahl (0-n) logischer Seiten (Pages) sowie
- einer Anzahl (1-n) von Darstellungsebenen (Layers) und
- Seitensegmenten, die in einer zugehörigen Darstellungsebene liegen.

2.4.1.2. Ebenen (Layers)

Der Inhalt einer Seite kann sich als Summe verschiedenster Bestandteile, z.B. Briefkopf und Brieftext zusammensetzen. In ihrer Darstellung erscheint die Seite jedoch als Einheit. jadice nutzt für diesen Zweck Ebenen.

Ebenen stellen eine „vertikale" Unterteilung eines jadice Dokuments dar, in denen Seitensegmente positioniert werden. Verschiedene Ebenen werden i.d.R. direkt überlagernd dargestellt. Dies bedeutet, dass alle Ebenen eines jadice

Dokuments aus Sicht des Benutzers eine geschlossene Einheit bilden. Ebenen sind somit vergleichbar mit übereinander liegenden, transparenten Folien.

2.4.1.3. Seiten (Pages)

Seiten stellen die üblicherweise kleinste, dem Benutzer präsentierte Einheit dar. Sie setzen sich, ähnlich der Dokumente, selbst aus weiteren Objekten zusammen, so genannten Seitensegmenten oder PageSegments. Seitensegmente belegen innerhalb der Seite die von den Ebenen des Dokuments vorgegebenen Positionen. Mit anderen Worten: Die Ebenen des Dokumentes geben eine endliche Anzahl von vertikalen Positionen oder „Plätzen" in der Seite sowie eine logische Reihenfolge dieser Plätze vor. Diese Plätze können dann (aber müssen nicht) in den Seiten von Seitensegmenten belegt sein. Seiten erstellen zur Darstellung keine Images, sie rendern sich selbstständig in gegebene Grafikkontexte, z.B. Monitor, Drucker, etc.

2.4.1.4. Seitensegmente (PageSegments)

Die Seitensegmente tragen die eigentliche Dokumentinformation. Sie korrespondieren direkt mit entsprechenden Seiten eines physikalischen Dokumentendatenstromes. Beispiele für Seitensegmente sind:

- Das gescannte Bild der Vorder- oder Rückseite eines physikalisch vorliegenden Blatt Papiers oder eines ähnlichen Mediums bei NCI-Dokumenten[5].
- Eine einzeln dargestellte Seite eines CI-Dokuments[6] wie beispielsweise ASCII- oder PTOCA.
- Die auf einer Seite angebrachten Annotationen.

Seitensegmente werden mittels Layers (Ebenen) in die Seite und somit auch in das Dokument eingebunden und erhalten dadurch eine „vertikale" Position innerhalb der Seite zugewiesen.

2.4.2. Annotationen

Unter Annotationen werden

- Kommentare
- Vermerke
- Anmerkungen
- Erläuterungen
- Notizen oder
- Hinweise durch Pfeile oder farblich hinterlegte Bereiche

verstanden, die der Benutzer in einem Dokument auf einer bestimmten Seite anbringen kann. Annotationen sind zusätzliche Informationen zu einem Dokument und verändern das eigentliche Dokument nicht. Zu diesem Zweck werden Annotationsdaten in einem eigenen Annotationsseitensegment verwaltet.

5 NCI: (Non Coded Information) nicht kodierte Informationen wie Bilder, Sprache, Ton, Video etc., die vom Rechner nicht erkannt und nicht direkt verarbeitet werden können. Eine typische NCI-Anwendung ist die Erfassung von Dokumenten mit Scannern und deren Behandlung als Faksimiles.

6 CI: (Coded Information) kodierte Informationen wie Texte, die auf Zeichenbasis vorliegen und direkt verarbeitet und angezeigt werden können.

Diese Annotationen können Informationen in Form von

- Text oder
- grafischen Objekten zur
  - Verdeutlichung
  - Hervorhebung oder gar zur
  - Ausblendung

enthalten.

2.4.3. Ressourcen

Spezielle Formate wie AFP und MO:DCA unterstützen die Verwendung von Elementen wie Logos, Forms, Overlays und Page Segmenten.

Diese Elemente werden Ressourcen genannt und können im AFP- oder MO:DCA-Dokument selbst (inline) oder in einer externen Ressource vorkommen.

2.5. Formate

Folgende Formate werden von jadice direkt unterstützt:

- PDF/A
- AFP & MO:DCA mit folgenden Inhalten
  - PTOCA
  - IOCA
  - GOCA
  - Page Segmente
  - Overlays & Forms
- IOCA
- TIFF
  - unkomprimiert
  - komprimiert
    - RLE
    - Packbits
    - Fax G3 / G4 bzw. CCITT T.4 und CCITT T.6
    - JPEG (true-color und grayscale)
    - LZW
    - DEFLATE
- FileNet Banded Image
- EBCDIC
- ASCII
- JPEG/JBIG/JFIF
- GIF
- BMP
- PNG

- Weitere optionale Formate über ImageIO Schnittstelle durch die Verwendung von Java Advanced ImageIO Tools
  - JPEG2000
  - und weitere Formate je nach Java Advanced ImageIO Version, wie
    - PNM
    - (teilweise) FlashPix
    - usw.

- Weitere optionale Formate über ImageIO Schnittstelle als eigenständige jadice Produkte:
  - DJVu
  - DiCOM

3. Die jadice Integrator API

3.1. Ziel

Eines der Ziele der jadice document platform ist es, sehr einfache Integrationsmöglichkeiten mit möglichst geringem Programmieraufwand gepaart mit großer Flexibilität und Anpassungsfähigkeit für Integratoren anzubieten. Dazu sind drei Ansatzpunkte verfolgt worden:

† BasicJadicePanel – die einfachste Möglichkeit einen Viewer einzubinden

Als JPanel ist es beliebig in bestehende Layouts einbindbar und beinhaltet eine Toolbar mit den wichtigsten Viewer Tools, eine Annotation Toolbar zur Erstellung von Annotationen, eine Statusbar zur Anzeige der Seitenzahl und des Zoom Faktors und optional eine Menubar, falls die einbettende Komponente ein Frame ist und eine Menubar setzen kann. Anpassungen, wie Erscheinungsbild oder Hot Keys, aber auch die Struktur der Tools in den Toolbars oder den Menüs kann einfach durch Veränderung der Konfiguration erreicht werden. Vergleichen Sie bitte auch Abschnitt 4.16.BasicJadicePanel.

† AddOns - Nützliche Viewer Erweiterungen direkt integrierbar

Viewer Erweiterungen, auch AddOns genannt, wie der Navigator oder der Seitensortierer, stehen als JComponent mit gekapselter Funktionalität und einheitlicher API zur Verfügung, alternativ als *JFrame / JInternalFrame.*

Weitere Informationen zu AddOns entnehmen Sie bitte Abschnitt 4.17.AddOns.

† Kommandoverarbeitung mit Command- und Action Pattern

Alle Viewer spezifischen Aufgaben, wie Zoomen oder Rotieren, aber auch das Aktivieren von AddOns wurden gekapselt in einzelnen Commands, die von Actions aufgerufen werden können. Diese Actions sind verantwortlich für ihre GUI Repräsentation, z.B. durch Icon, Accelerator oder Tooltip, und sind in der Lage Commands zu bündeln. Verhalten und Erscheinungsbild der Actions, aber auch welche Commands aktiviert werden sollen, werden über die Konfigurationsdateien definiert, so dass bei Anpassungen Integratoren kein weiterer Programmieraufwand entsteht. Zudem besteht die Möglichkeit mit sehr geringem Programmieraufwand bestehende Commands in ihrer Funktionalität nach eigenen Wünschen zu erweitern oder komplett eigene Commands einzubinden.

Das Zusammenwirken von Actions und Commands in Kombination mit einer einfachen Konfigurierbarkeit bietet ein starkes und flexibles Framework, das Anpassungen und Integration in bestehende Anwendungen auf einfachste Art mit möglichst geringem Programmieraufwand möglich macht.

Darüber hinaus sind über die Konfigurationsdateien verschiedene Look&Feels sowie Menü-, Kontextmenü- und Toolbar-Strukturen frei definierbar. Ein Anwendungsbeispiel dazu finden Sie im Kapitel 5.4.Actions-Commands-Context.

3.2. Umsetzung

3.2.1. Commands

Die Verarbeitung von Kommandos und das Binden von Kommandos an Elemente der Benutzeroberfläche basieren auf den „Command" und „Action" Design-Patterns. Das Action-Pattern wird in SWING bereits eingesetzt, die Verwendung des Command-Patterns dient der weiteren Entkopplung der Kommandofunktionalität.

Standardaufgaben wie Zoomen, Rotieren oder das Aktivieren von AddOns sind innerhalb der jadice Pakete in jeweils eigenständige Commands gekapselt, die von Actions aufgerufen werden können und die die eigentliche Ausführung der Aktionen übernehmen. Ein Command erhält zur Ausführung eine Anzahl von Objekten, Context (3.2.3.Context) genannt, die den Zustand der Benutzeroberfläche zum Zeitpunkt der Ausführung der Aktion widerspiegeln. Zu dieser Anzahl von Objekten kann jedes GUI-Element eigene Objekte beisteuern.

Alle Commands sind Nachfolger der abstrakten Basisklasse AbstractCommand, deren wichtigste Methoden im Folgenden kurz beschrieben sind:

- **doExecute(Collection)**

  wird aufgerufen von der umschließenden CommandAction zur Ausführung des eigentlichen Kommandos. Die Argumente sind die im Kontext beinhalteten Objekte.

- **checkQuickly(Collection)**

  wird aufgerufen von der umschließenden CommandAction, um ein Enabled-Check durchzuführen. Die Argumente sind die im Kontext beinhalteten Objekte. Die Überprüfungsmethodik innerhalb dieser Methode sollte möglichst performant sein, da Command Zustandsabfragen sehr häufig vollzogen werden. Der Zustand von Commands wird immer verifiziert, wenn der Kontext verändert wurde (Teil des Enabled-Checks der CommandAction). Kontext Veränderungen können ausgelöst werden durch Änderungen der Dokumentstruktur, wie das Nachladen von Seiten oder Seitensegmenten, aber auch Veränderungen der Darstellung, wie Rotationen oder Zoomen, selbst das Scrollen einer Seite innerhalb des Viewers kann unter bestimmten Umständen eine Zustandsprüfung von Commands auslösen.

- **checkDeeply(Collection)**

  abschließende Zustandsverifikation vor dem Aufruf der "doExecute ()" Methode. Die Argumente sind die im Kontext beinhalteten Objekte. Im Gegensatz zu der „checkQuickly" Methode, die sehr häufig ausgelöst wird, wird „checkDeeply" ausschließlich nur vor dem Aufruf der „doExecute"-Methode ausgelöst. Sie ermöglicht eine aufwendige Zustandsprüfung und Validierung des Kontextes, bevor es zur Ausführung des Commands kommt und erlaubt bei fehlgeschlagener Überprüfung sogar die Ausführung der „doExecute" Methode zu verwehren.

- **isAvailable()**

  initialer Test vor der Aufnahme einer CommandAction in eine Menu- oder Toolbar Struktur.

  Durch diese Methode kann ein Command bereits zur Erstellungszeit verifizieren, ob es grundsätzlich verfügbar ist oder nicht. Ist ein Command nicht verfügbar, weil gravierende Randbedingungen nicht erfüllt sind, z.B.

bestimmte, benötigte Ressourcen stehen nicht zur Verfügung, wird dieses Command automatisch nicht erzeugt bzw. nicht in der gewünschten Struktur aufgenommen.

Eine Prüfung der Ausführbarkeit von Commands (Verifizierung des Enabled-Status durch „checkQuickly" oder „checkDeeply") vollzieht sich im Allgemeinen nach folgendem Schema:

- Überprüfung, ob die erwarteten Argumenttypen (Klassen) in der erwarteten Anzahl in den übergebenen Kontext Objekten enthalten sind.
- Überprüfung, ob mit dem Inhalt der übergebenen Argumente die Ausführung des Commands möglich ist.

3.2.2. Actions

Actions[7] binden ausführbare Kommandos an GUI-Elemente wie Buttons, Toolbars-Buttons oder MenuItems. Sie tragen dabei einerseits zum Erscheinungsbild des GUI-Elements bei, indem sie z.B. Icons oder Labels liefern, steuern andererseits aber auch den Zustand der GUI-Elemente, wie z.B. den enabled/disabled-Zustand.

CommandActions[8] können zusätzlich ein oder mehrere Commands[9] (siehe auch 3.2.1.Commands) bündeln und in ihrer "actionPerformed(...)" Methode ausführen.

Darüber hinaus ist jede CommandAction an einen Kontext (siehe auch 3.2.3.Context) gebunden, dessen Änderung einen Enabled-Check zur Folge hat. Dieser Check schließt einen Check aller enthaltenen Commands ein. Nur wenn alle ausführbar sind, setzt sich auch die Action als ausführbar.

Die im Kontext enthaltenen Objekte werden den Commands zur Ausführung übergeben. Damit bestimmt der Kontext nicht nur über den Enabled-Status einer CommandAction, vielmehr beeinflussen seine Kontext Objekte auch die Ausführung der aktivierten Commands.

Erweiterungen von CommandActions sind im Allgemeinen nicht notwendig, da die Eigenschaften bestimmt werden durch die Konfigurationsdatei *actions.properties*.

3.2.3. Context

Instanzen der Klasse Context[10] bilden die Brücke zwischen GUI Elementen, CommandActions und Commands. GUI Elemente können als semantische Einheiten verstanden werden. Ein Beispiel: Ein Fenster beinhaltet eine Viewerinstanz, eine Menübar und eine Toolbar. Jede Aktion, die innerhalb dieses Fensters passiert, kann Auswirkungen auf den Zustand (enabled state), aber auch auf die Art der Ausführung der Tools in der Toolbar oder der Menu Items in der Menübar haben. Ebenso können aber auch die Tools oder Items mit Objekten des Fensters arbeiten. Damit bildet das Fenster in sich eine logische Einheit, dessen einzelne Elemente in ihrem Zustand und ihrer Ausführbarkeit voneinander abhängen.

7 javax.swing.Action
8 com.levigo.util.swing.action.CommandAction
9 com.levigo.util.swing.action.Command
10 com.levigo.util.swing.action.Context

Ein Kontextobjekt begleitet eine GUI-Komponente (als Client Property einer JComponent), im obigen Beispiel die RootPane des Fensters, und kann eine beliebige Anzahl von Objekten beinhalten, die bestimmend für Ausführung und Zustand von aktiven Elementen innerhalb dieser Komponente sein können.

Ferner informieren Instanzen der Klasse Context registrierte Interessenten (ContextChanged-Listener[11]) über Änderungen des Kontexts. Dies betrifft Änderungen an den enthaltenen Objekten, aber auch semantische Änderungen des Kontexts, die provoziert werden können durch einen Aufruf der Methode „contextChanged()".

Beim Aufbau der GUI-Komponente erhalten CommandActions zur Erstellung eine Referenz auf ein Kontextobjekt und registrieren sich als ContextChangedListener an diesem. Bei jedem Kontextänderungs-Ereignis überprüft die CommandAction Instanz ihren Zustand (enabled state), indem alle enthaltenen Commands nach ihrem Zustand befragt werden (siehe 3.2.1.Commands „checkQuickly(...)" Methode). Die Zustandsüberprüfung basiert auf den im Kontext enthaltenen Objekten.

Die Ausführung von CommandActions verläuft in zwei Schritten. Zunächst wird ein letzter Test aller Commands (siehe 3.2.1.Commands „checkDeeply(Collection)" Methode) vollzogen, dessen Ergebnis erfolgreich sein muss, bevor die Commands mittels der Methode „doExecute(Collection)" ausgeführt werden. Auch hierbei werden jeweils die im Kontext enthaltenen Objekte übergeben.

Analog der hierarchischen Komponentenstruktur von GUI-Elementen können auch deren Kontexte hierarchisch organisiert sein. Um in komplexeren Situationen mit mehreren Kontexten entscheiden zu können, welche Objekte der verschiedenen Kontexte, der Sammlung von Objekten übergeordneter Kontexte angehören, können einzelne Kontexte aktiv oder inaktiv sein. Der Aktivitätsstatus der Kontexte orientiert sich dabei grob am Fokussierungsstatus der assoziierten GUI-Elemente. Es gilt: ein Kontext ist aktiv, wenn ein beinhaltetes GUI Element, ohne assoziiertem Kontext, aktiv ist.

Unter bestimmten Umständen können CommandActions der Eltern-Komponente in Zustand und Ausführung abhängig sein von dem Zustand der Kind-Komponente, widergespiegelt in dessen assoziiertem Kontext. Da aber die CommandActions nur die Elemente ihres Kontextes kennen, müssen Objekte des Kind-Kontextes temporär Teil des Eltern-Kontextes werden können. Dazu können Kontexte in drei Modi bzw. Aggregationszuständen instantiiert werden:

- **Context.NO_CHILDREN** Einzelner Kontext, alle GUI Kind-Komponenten besitzen keinen weiteren Kontext oder die Elemente eines eventuellen Kind-Kontextes sind nicht relevant für CommandActions des Eltern-Kontextes.
- **Context.ACTIVE_CHILD** GUI Kind-Komponenten können Kontexte haben, aber nur die Elemente des aktiven Kind-Kontextes sind relevant für CommandActions des Eltern-Kontextes und werden diesem temporär hinzugefügt.
- **Context.ALL_CHILDREN** GUI Kind-Komponenten können Kontexte haben, Elemente aller Kind-Kontexte können relevant für CommandActions des Eltern-Kontextes sein und werden diesem Zweck temporär hinzugefügt.

Um Kind-Kontexte an deren Eltern-Kontext zu de-/registrieren, stellt die Klasse Context folgende Methoden zur Verfügung:

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

- **addChildContext(Context)**

  Fügt den gegebenen Kontext als Kind hinzu.

- **addToParentsContext()**

  Fügt den Kontext seinem Eltern-Kontext hinzu. Das Kontext Objekt wird stets als Client Property der assoziierten GUI Komponente abgelegt. Der Eltern-Kontext wird ermittelt, in dem die Komponentenhierachie „aufwärts" nach einem Eltern-Kontext durchsucht wird. Aus diesem Grund ist zur korrekten Ausführung dieser Methode angeraten, dass die Komponentenhierachie bereits festgelegt ist.

- **removeChildContext(Context)**

  Entfernt den gegebenen Kind-Kontext.

- **removeFromParentsContext()**

  Entfernt den Kontext von seinem Eltern-Kontext. Ebenso wie schon unter „addToParentsContext"-Methode beschrieben, sollte zur korrekten Ausführung der Methode eine festgelegte Komponenten-Hierachie sichergestellt sein.

3.3. Die Konfigurationsdateien

Die Konfigurationsdateien beschreiben das Erscheinungsbild, Tastenbelegungen und die Struktur von Menüs und Toolbars der mitgelieferten Commands. Zur Anpassung dieser Eigenschaften an die integrierende Umgebung, um eigene Commands hinzuzufügen oder um verschiedene Look & Feels für verschiedene Einsatzmöglichkeiten zu erstellen, können Konfigurationsdateien angepasst werden.

Bitte beachten Sie, dass die Konfigurationsdateien in internationalisierten Varianten vorliegen und automatisch entsprechend der Länderkennung des Betriebssystems verwendet werden. Änderungen an diesen Dateien sollten dementsprechend stets in allen Varianten getätigt werden.

In den folgenden Abschnitten werden die Aufgaben, Inhalte und das Zusammenspiel der einzelnen Konfigurationen detailliert erläutert. Die genaue Syntax der Dateien ist unter 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien zu finden.

3.3.1. commands.properties

Diese Konfiguration erzeugt eine Verknüpfung zwischen einem eindeutigen Command-Namen, der in den anderen Konfigurationen als Referenz verwendet wird, und dessen Verwirklichung, sprich der genauen Klassenreferenz.

Commands werden über Reflection erstellt. Um Instantiierungs-Fehler zu vermeiden, muss auf die korrekte Angabe von Pfad- und Klassennamen wie auch auf Groß- und Kleinschreibung geachtet werden.

Weitere Informationen sind in Abschnitt 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien zu finden.

3.3.2. menucomponents.properties

Die Menükomponenten-Konfiguration bestimmt die Zusammensetzung von Menüs, Sub-Menüs, Kontext-Menüs und Toolbars.

Dazu wird einer Struktur, z.B. einem Menü oder einer Toolbar, ein eindeutiger Name gegeben, über den später zur Integration eine Instanz eines Menüs erstellt werden kann. Die Actions, die in dieser Struktur enthalten sein sollen, werden als Komma separierte Liste von Action-Command-Namen festgelegt. Die Action-Command-Namen müssen dabei durch referenzierbare *actions.properties Dateien* zur Verfügung gestellt werden.

Auf gleichem Weg können Definitionen von anderen Substrukturen wie Unter-Menus, Kontext-Menüs und ähnlichem über die Menükomponenten-Konfiguration bestimmt werden.

Weitere Informationen sind in Abschnitt 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien zu finden.

3.3.3. actions.properties

In dieser Konfiguration werden die Eigenschaften der CommandActions, wie Tool-Icon, Tooltip-Text, Accessor und ähnliches, bestimmt. Neben der Definition der äußeren Erscheinung der CommandAction wird in dieser Konfiguration ebenso bestimmt, welche Commands in welcher Reihenfolge durch Auslösung der CommandAction zur Ausführung gelangen sollen.

Dabei werden zur Identifizierung der Commands die eindeutigen Command Bezeichner aus referenzierbaren *commands.properties* Konfigurationen verwendet.

Weitere Informationen sind in Abschnitt 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien zu finden.

4. Klassenüberblick

Der folgende Überblick beschränkt sich auf eine Beschreibung der für Integratoren wichtigsten Klassen.

Eine detaillierte Beschreibung des Zusammenwirkens der einzelnen Klassen und konkrete Anwendungsbeispiele finden sich in Abschnitt 5.Typische Anwendungsbeispiele.

Interfaces, Klassen und Methoden sind zusätzlich in einer separaten API-Referenz dokumentiert.

4.1. Viewer

Der Viewer[12] ist eine der zentralen Klassen der jadice Komponentenarchitektur. Er übernimmt die Verwaltung, Handhabung und Darstellung von Dokumenten und Seiten. Als JComponent bzw. JavaBean ist der Viewer einfach in eigene Architekturen und Layouts einbettbar. Er besteht aus einer Bildanzeigefläche und zugehörigen Scrollbars. Zugleich bietet der Viewer aus Sicht eines Integrators eine Schnittstelle der wichtigsten Funktionen zur Dokumenten-/Seiten- Ansicht, wie

- Blättern im Dokument
- Dokument Zoom
- Seiten Zoom
- Dokument Rotation
- Seiten Rotation
- Zoom Policy, etc.

Eine Viewerinstanz kann immer genau ein Dokument darstellen bzw. bearbeiten. Registrierte Listener, z.B. die integrierende Anwendung, werden über die wichtigsten Änderungen am Dokument, seiner Darstellung oder am Viewer selbst informiert.

Dies geschieht unter Verwendung von PropertyChangeEvents[13] bzw. PropertyChangeListener[14]. Interessenten melden sich als PropertyChangeListener qualifiziert (genau für eine definierte Eigenschaft) oder unqualifiziert (für alle Eigenschaften) bei der jeweiligen Viewerinstanz an. Zur konkreten Identifizierung der betreffenden Property sind Eigenschaftsnamen im Viewer als öffentliche Konstanten deklariert.

Der Viewer erhält zur Darstellung eine Document[15]-Referenz. Über entsprechende „getter"- und „setter"-Methoden kann jederzeit eine Referenz auf dieses Dokument erhalten bzw. verändert werden. Es muss dabei nicht auf den Zustand des Dokuments oder den Zeitpunkt, wann ein Dokumentenwechsel stattfindet, geachtet werden. Eigenschaften der Klasse Document sind in Kapitel 4.2.Document beschrieben.

12 com.levigo.jadice.Viewer
13 Siehe dazu auch PropertyChangeListener und PropertyChangeSupport in der Java 2 Platform API Specification
14 Siehe dazu auch PropertyChangeEvent und PropertyChangeSupport in der Java 2 Platform API Specification
15 com.levigo.jadice.docs.Document

Für eine optimale Bildanzeige und ein performanteres Arbeiten unterstützt der Viewer bei Dokument- oder Seitenwechsel verschiedene Zoom-Policies. Beispielsweise können neu gesetzte Dokumente immer im „ZoomToFit" Modus angezeigt werden, d.h. das Dokument wird optimal in Höhe und Breite zur Ansicht im Viewer eingepasst. Andere Zoom-Policies sind:

„ZoomToWidth": Dokument in der Breite optimal im Viewer einpassen, geeignet für Dokumente mit Seiten gleicher Größe.

„ZoomToHeight": Dokument in der Höhe optimal im Viewer einpassen, geeignet für Dokumente mit Seiten gleicher Größe.

„PageZoomToWidth": Seite in der Breite optimal im Viewer einpassen, geeignet für Dokumente mit verschiedenen Seitengrößen.

„PageZoomToHeight": Seite in der Höhe optimal im Viewer einpassen, geeignet für Dokumente mit verschiedenen Seitengrößen.

„PageZoomToFit": Seite in der Breite und Höhe optimal im Viewer einpassen, geeignet für Dokumente mit verschiedenen Seitengrößen.

„ZoomDefault": der Standard Dokument-Zoomfaktor wird zur Anzeige verwendet.

Die Zoom-Policy wird je nach gesetzter Policy bei Seiten- bzw. Dokumentwechsel angewendet und kann jeweils für eine Viewerinstanz erfragt und gesetzt werden.

Zoom-Policies stehen in der Relevanz niedriger als vom Anwender gesetzte Zoom-Eigenschaften einer Seite bzw. eines Dokuments. Solange ein Dokument in einer Viewer-Instanz referenziert wird, merkt sich der Viewer vom Anwender gesetzte Seiten und Dokument spezifische Zoom- und Rotationseinstellungen. Wird als Beispiel vom Anwender 200% als Seiten-Zoom der Seite 3 gesetzt, wird Seite 3 immer mit 200% Zoomfaktor vom Viewer dargestellt, auch wenn zwischenzeitlich auf eine andere Seite des Dokuments geblättert wurde. Bei Dokumentenwechsel werden diese Rendereinstellungen vom Viewer verworfen und die gesetzte Zoom-Policy kommt wieder zur Anwendung.

Zusätzlich über die reine Dokumenten-Ansicht hinaus ermöglicht der Viewer selbst definierte, aktive Schichten über die Dokumentdarstellung zu legen. Solche Schichten können sich selbst rendern, aber auch Input-Events, wie Maus- oder Tastatur-Ereignisse, empfangen. Integratoren können auf diesem Weg der Seiten-Ansicht (ggf. aktive) Dekorationen hinzufügen, z.B. ein „Büroklammer"-Icon, das auf Mausklick eine Aktion ausführt. Solche Schichten heißen in der jadice Komponentenarchitektur EditPanes.

Weitere Informationen entnehmen Sie bitte 4.15.EditPanes.

4.2. Document

Ein Dokument ist ein formatunabhängiger Behälter für Seiten und Seitenschichten. Instanzen der Klasse Document[16] enthalten eine oder mehrere Seiten, die in verschiedenen Schichten Seitensegmente beinhalten können. Zur Verdeutlichung betrachten Sie bitte Abbildung 1 in 2.4.1.Das Dokumentenmodell.

Es bietet die Möglichkeit auf die enthaltenen Seiten und Seitenebenen (Layer) qualifiziert zuzugreifen, d.h. einer Instanz der Klasse Document kann über

16 com.levigo.jadice.docs.Document

öffentliche API Methoden Seiten und Seitenebenen hinzugefügt, entfernt und verschoben werden.

Alle Änderungen der Struktur eines Dokuments werden an registrierte Interessenten (DocumentListener[17]) propagiert. Das Interface DocumentListener definiert eine Schnittstelle, um Informationen über Änderungen an einem Dokument zu erhalten. Solche Änderungen können z.B. das Hinzufügen, Löschen oder Umsortieren von Seiten, das Verändern von Seiten, Seitensegmenten oder Layern wie auch Änderungen des Dokumentstatus sein. Eine entsprechende Adapterklasse der Schnittstelle DocumentListener wird über die API ebenfalls angeboten.

Darüber hinaus kann jedem Dokument ein Name und ein spezifischer ResourceLoader[18] zugewiesen werden.

ResourceLoader ermöglichen den Zugriff auf externe Ressourcen für MO:DCA oder AFP-Dokumente. MO:DCA und AFP-Dokumente können interne (inline) Ressourcen beinhalten oder auf externe Ressourcen verweisen. Solche Ressourcen können z.B Logos, Briefköpfe oder Unterschriften sein. Der Zugriff auf externe Ressourcen von AFP oder MO:DCA Dokumenten erfolgt über einen ResourceLoader (siehe dazu auch 4.8.ResourceLoader).

Beispiele zum Erzeugen und Laden von Dokumenten finden sich unter 5.Typische Anwendungsbeispiele.

4.3. Page

Ein Dokument kann eine oder mehrere Seiten beinhalten, eine Seite wiederum kann aus verschiedenen Seitensegmenten (PageSegment[19]) bestehen, wird aber im Gegensatz zu einem Dokument visuell als eine (Seiten-)Einheit vom Viewer dargestellt.

Eine Instanz der Klasse Page[20] erlaubt Zugriff auf die enthaltenen Seitensegmente. Zusätzlich hält eine Seite eine Referenz auf das Dokument, welches die Seite enthält. Darüber hinaus bietet die Seite Angaben über die ursprüngliche, die skalierte, die rotierte und die dargestellte Größe der Seite als zusammengefasste Einheit aller Seitensegmente.

Die entsprechenden Zugriffsmethoden sind im Einzelnen:

- **getSize()**:

  ursprüngliche Größe in Base Units (umgerechnet auf 7200dpi)

- **getRotatedSize()**:

  ursprüngliche Größe in Base Units (umgerechnet auf 7200dpi), Zoom Faktor und Rotation berücksichtigt

- **getScaledSize()**:

  dargestellte Seitengröße (umgerechnet auf Device-Resolution), Zoom Faktor berücksichtigt, Rotation unberücksichtigt

- **getRenderedSize()**:

  die aktuelle Seitengröße (umgerechnet auf Device-Resolution), Zoom Faktor und Rotation berücksichtigt

17 com.levigo.jadice.docs.DocumentListener
18 com.levigo.jadice.docs.resource.ResourceLoader
19 com.levigo.jadice.docs.PageSegment
20 com.levigo.jadice.docs.Page

4.4. PageSegment

Seitensegmente sind Teile einer Seite und tragen die eigentliche Rohdateninformation des entsprechenden Segments.

PageSegments[21] können somit Integratoren folgende Informationen bieten:

- das Datenformat der Rohdaten, z.B Tiff oder MO:DCA
- die ursprüngliche Auflösung der Rohdaten
- die vorgegebene und die skalierte Größe des Segments
- die das Seitensegment beinhaltende Seite

4.5. Loader

Der Loader[22] ist die zentrale Klasse für alle Ladevorgänge von Dokumenten. Integratoren können jederzeit eine Instanz über den Default-Konstruktor der Klasse erstellen und damit ein neues oder bereits vorhandenes Dokument wahlweise synchron oder asynchron befüllen.

Ein erstelltes Dokument kann bereits in „unbefülltem" Zustand dem Viewer zur Ansicht übergeben werden. Der Viewer erkennt automatisch, wenn die erste Seite bzw. des erste Seitensegment in einer Seite geladen wurde und zeigt diese bereits an, obwohl im Hintergrund der Ladevorgang noch nicht abgeschlossen sein muss. Dies ist insbesondere bei langsamen Netzwerkverbindungen und großen Dokumenten von Vorteil.

Ein Loader kann für einen oder mehrere Ladevorgänge genutzt werden und stellt zu diesem Zweck verschiedene Lademethoden zur Verfügung. Die unterschiedlichen Lademethoden erlauben Integratoren den einfachen Ladevorgang eines Dokuments, aber auch Ladevorgänge in einen bestimmten Layer, ab einer bestimmten Seite und/oder ein vorbestimmtes Format zu laden.

Ladevorgänge verlaufen in der Regel asynchron, um z.B. eine schnellere Reaktion der GUI zu ermöglichen, ohne warten zu müssen, bis das evtl. länger andauernde Laden des Dokuments abgeschlossen ist. Unter bestimmten Umständen ist es jedoch von Vorteil, Ladevorgänge synchron ablaufen zu lassen. Möchte z.B. der Integrator mehrere Bilddateien in einer vorgegebenen Reihenfolge in einem Dokument „aneinander hängen", kann der Loader auf synchrone Verarbeitung umgestellt werden.

Benachrichtigungen über den Fortschritt des Ladevorgangs werden registrierten Interessenten (LoadListener[23]) durch Auslösen der „loadStateChanged" Methode übermittelt. Im Normalfall werden diese Benachrichtigungen im aktuellen Lade-Thread ausgelöst. Im Zusammenhang mit GUI Komponenten kann es jedoch von Vorteil sein, Benachrichtigungen auf dem Event Dispatch Thread auslösen zu lassen. Ist dies gewünscht, kann der Loader einfach über eine entsprechende Methode umgestellt werden.

MO:DCA und AFP Dokumente können interne (inline) oder externe Ressourcen beinhalten. Zum Laden von externen Ressourcen, die oft an anderen Stellen liegen als die Dokumentrohdaten, können ResourceLoader[24] verwendet werden. Sie unterstützen den Ladevorgang, indem sie den Zugriff auf externe Ressourcen zur Verfügung stellen. Zu diesem Zweck sollten ResourceLoader vor dem Ladevorgang im Loader oder im Dokument gesetzt werden.

21 com.levigo.jadice.docs.PageSegment
22 com.levigo.jadice.docs.resource.Loader
23 com.levigo.jadice.docs.resource.LoadListener
24 com.levigo.jadice.docs.resource.ResourceLoader

ResourceLoader, die in einem Dokument gesetzt wurden, werden für Ladevorgänge speziell in dieses Dokument genutzt. Hat das Dokument keinen eigenen ResourceLoader angegeben, nutzt es den im Loader angemeldeten ResourceLoader. Im Loader angemeldete ResourceLoader werden statisch gehalten, um einen einfachen applikationsweiten Zugriff zu ermöglichen und von anderen Loader-Instanzen genutzt werden zu können.

Näheres zu ResourceLoadern entnehmen Sie bitte 4.8.ResourceLoader.

Genauere Informationen zu Loadern, LoadListenern, ResourceLoadern, sowie deren Klassen- und Methodensignaturen entnehmen Sie bitte der jadice API Dokumentation. Ein Beispiel zum Laden eines Dokuments wird in 5.Typische Anwendungsbeispiele beschrieben.

4.6. FormatInfo und FormatFile

Die API der jadice document platform bietet zu jedem unterstützten Bild-Format jeweils eine Format-Klasse, die Ladevorgänge in das repräsentierte Format unterstützt. Die Benennung dieser Klassen folgt einer vorgegebenen Namenskonvention.

Format unterstützende Klassen zum Laden und Speichern:

*FormatName*FormatInfo -> Bsp. TIFFFormatInfo[25]

*FormatName*File -> Bsp. TIFFFile[26]

*FormatName*FormatInfo dient Ladevorgängen eines Format

*FormatName*File dient Speichervorgängen.

Ohne Angabe einer *FormatName*FormatInfo Instanz versucht der Loader vor dem eigentlichen Ladevorgang das Format der zu ladenden Daten zu bestimmen. Dazu wird zunächst überprüft, welche Formate im Klassenpfad als Format Services zur Verfügung stehen. Im Anschluss daran wird jede gefundene FormatInfo Instanz befragt, ob sie zu dem zu ladenden Datenstrom passt. Ist eine solche FormatInfo Instanz gefunden, ist das Format des Datenstrom ermittelt und der eigentliche Ladevorgang beginnt.

Erhält der Loader jedoch für einen Ladevorgang eine konkrete Instanz einer *FormatName*FormatInfo, kann die Formatbestimmung entfallen und sofort mit dem Laden der Daten begonnen werden. Somit können Ladevorgänge beschleunigt und Formatverwechselungen vermieden werden. Bitte beachten Sie jedoch, dass bei Angabe einer nicht zum Datenstrom passenden *FormatName*FormatInfo der Ladevorgang abgebrochen wird und eine Suche nach alternativen FormatInfo Instanzen nicht stattfindet.

Ein Beispiel:

jadice unterstützt IBM ContentManager kompatible Annotationen. Diese Annotationen werden vom Archiv als MO:DCA Strukturen gehalten. Lädt man Annotationen ohne Angabe eines *FormatName*FormatInfo, könnte es während des Ladevorgangs zur Verwechslung zwischen einem MO:DCA-Dokument und Annotationen als zusätzlichen Dokument Informationen kommen. Mehr dazu in den Kapiteln 4.11.ImagePlusAnnotationFormatInfo und 4.12.ImagePlusAnnotationFile.

Zusätzlich werden FileNet-kompatible Annotationen unterstützt. (Weitere Informationen hierzu finden Sie als Teil der Auslieferung im Handbuch Annotationen: Laden – Speichern – Bearbeiten)

25 com.levigo.jadice.formats.tiff.TIFFFormatInfo
26 com.levigo.jadice.formats.tiff.TIFFFile

4.7. LoadListener

Die Schnittstelle LoadListener steht Integratoren zur Verfügung, wenn ein Interesse am Ablauf von Ladeprozessen von Dokumenten besteht. Der Loader informiert registrierte LoadListener durch Aufruf der „loadStateChanged" Methode über den Ladefortschritt. Der Ladezustand wird über eine Instanz eines LoadEvents[27] beschrieben.

Ein LoadEvent beinhaltet folgende Angaben:

- das betreffende Dokument
- die entsprechende Seite
- welcher Art das Ereignis ist (Seite geladen, Ladevorgang beendet, Ladefehler) und
- die Quelle bzw. den Ursprung des Ereignisses. Meistens die entsprechende *FormatName*FormatInfo oder Loader Instanz, in der das jeweilige Ereignis aufgetreten ist.

In früheren jadice Versionen (2.x) war es notwendig mit Hilfe eines LoadListeners das Ende eines Ladevorgangs abzuwarten, bevor ein Dokument einem Viewer zur Anzeige übergeben werden konnte. Dies ist nicht mehr notwendig.

Damit haben LoadListener in der vorliegenden Version keinen funktionalen Hintergrund mehr. Sie dienen rein informellen Zwecken.

LoadListener können in verschiedenen Ausprägungen und Instanzen am Loader angemeldet sein und für mehrere Ladevorgänge genutzt werden, ohne wieder neu registriert zu werden.

Benachrichtigungen an LoadListener werden im Normalfall auf dem aktuellen Lade-Thread weitergeleitet. Unter gewissen Umständen kann es z.B. im Zusammenspiel mit GUI Komponenten, interessant sein, den Aufruf der „loadStateChanged" auf dem Event Dispatch Thread zu erhalten. In einem solchen Fall können Integratoren über die „**setSendNotificationsOnEDT**" Methode den Loader veranlassen alle Nachrichten auf den EDT umzuleiten.

4.8. ResourceLoader

ResourceLoader-Klassen finden für AFP- oder MO:DCA- Dokumente Verwendung, bei denen im Dokument interne (inline) und externe Ressourcen eingebunden sein können.

Externe Ressourcen werden während des Ladevorganges des Dokuments mit Hilfe von ResourceLoadern nachgeladen. Ressourcen werden im Dokument über einen Namen referenziert und können über verschiedene Implementierungen von ResourceLoadern als InputStream gelesen werden.

Die Klasse ResourceLoader stellt das Interface zur Implementierung der im folgenden beschriebenen Klassen, aber auch für eigene Implementierungen dar. Die verschiedenen ResourceLoader der jadice API werden in den folgenden Abschnitten beschrieben.

27 com.levigo.jadice.docs.resource.LoadEvent

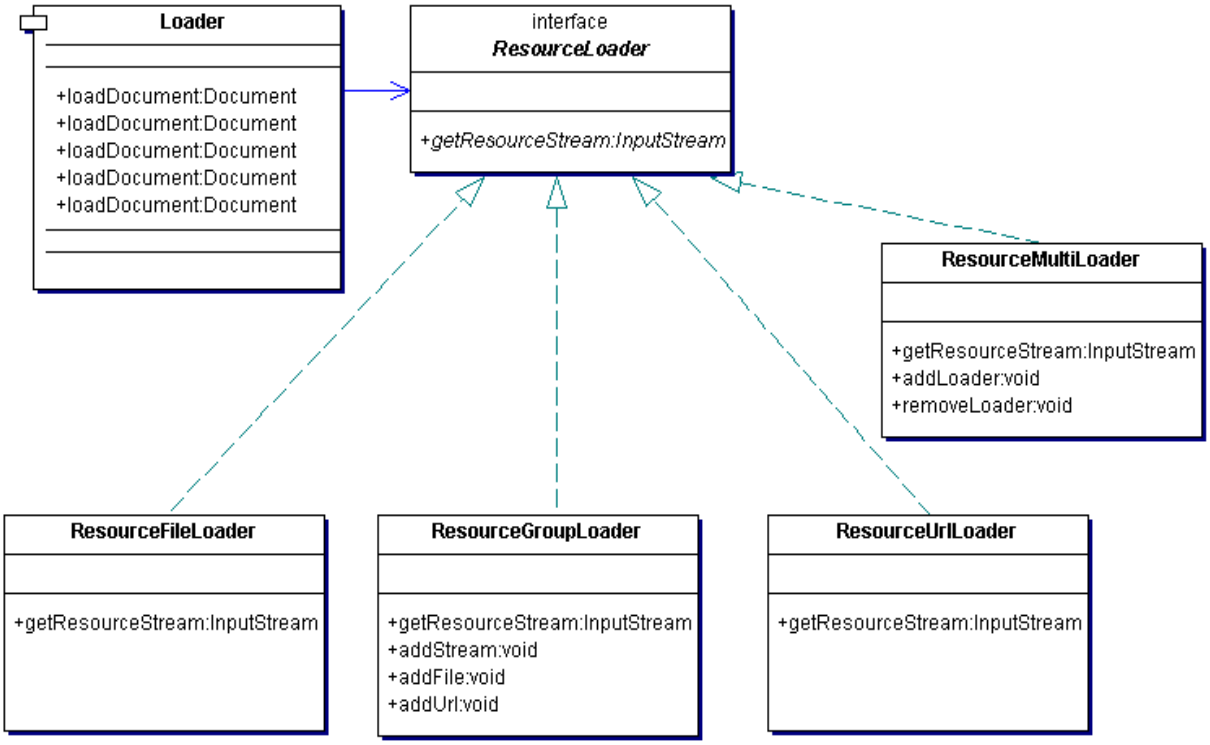

Abbildung 2 - Klassendiagramm Loader / ResourceLoader

Das ResourceLoader-Interface definiert die Methode „**getResourceStream(String resourceName, String defaultExt)**", die einen InputStream der Ressourcedaten zurück gibt, sofern die gesuchte Ressource gefunden und ein Zugriff möglich ist.

Der während des Ladevorgangs zu verwendende ResourceLoader muss entweder dem Dokument oder dem Loader über die Methode „**setResourceLoader(ResourceLoader resourceLoader)**" bekannt gegeben werden. Auf diesem Weg erhalten AFP-Dokumente die Möglichkeit während des Ladens auf einen entsprechenden ResourceLoader und damit auf die korrekte Ressource zuzugreifen.

Dabei ist zu beachten

- ResourceLoader im Dokument:
  - wird priorisiert vor evtl. registriertem ResourceLoader des Loaders.
  - ist eine Eigenschaft des Dokuments, d.h. jedes Dokument unterhält eine eigene Referenz auf einen evtl. registrierten ResourceLoader.
- ResourceLoader im Loader:
  - ist eine statische Eigenschaft des Loaders, somit einfacher, applikationsweiter Zugriff auf den gesetzten ResourceLoader und Gültigkeit für alle Loaderinstanzen.
  - ResourceLoader eines Dokuments werden priorisiert vor registriertem ResourceLoader des Loaders.

Abbildung 2 gibt einen Überblick über die Klassenhierarchie der ResourceLoader. Der ResourceLoader ist ein Interface, das Integratoren für

eigene Implementierungen nutzen können. Die im Folgenden beschriebenen Klassen stellen verschiedene nützliche Verwirklichungen dieser Schnittstelle dar, die, wenn nötig, zusammengefügt in einem MultiLoader, einer Loaderinstanz oder einem Dokument gesetzt werden können.

4.8.1. ResourceFileLoader

ResourceFileLoader[28] stellt InputStreams auf Basis von File-Ressourcen (Dateien des lokalen Dateisystems oder über das Dateisystem sichtbare Netzwerkressourcen) bereit.

Dem Konstruktor wird eine Liste der Suchpfade (Verzeichnisse im Dateisystem) für File-Ressourcen mitgegeben. Die Suchpfade sind durch die betriebssystem-spezifische Konstante **java.io.File.pathSeparator** zu trennen.

Mit Aufruf der Methode **getResourceStream(String resourceName, String defaultExt)** wird ein InputStream unter Verwendung der übergebenen Parameter erzeugt, wobei resourceName den Dateinamen ohne vorangehenden Pfad und ohne Erweiterung darstellt und defaultExt die Erweiterung des Dateinamens.

4.8.2. ResourceURLLoader

Über URLs erreichbare Ressourcen können über die Klasse ResourceUrlLoader[29] genutzt werden.

Die Funktionsweise ist weitestgehend mit der der Klasse ResourceFileLoader identisch. Anstelle des Suchpfades zur Datei wird dem Konstruktor eine Liste kommagetrennter URLs mitgegeben. Auch hier kann statt der Liste wieder eine einzelne URL übergeben werden.

Der InputStream wird wie im Interface definiert über **getResourceStream(String resourceName, String defaultExt)** zurückgegeben.

4.8.3. ResourceGroupLoader

Externe Ressourcen von AFP- oder MO:DCA-Dokumenten können einzeln oder in Resource Groups vorliegen. Resource Groups sind die Zusammenfassung verschiedener Ressourcen zu einem Datenstrom, wobei beinhaltete Ressourcen durch den ResourceLoader als InputStream erhalten werden können. Die tatsächliche Quelle einer Resource Group kann dabei entweder als Datei auf dem Dateisystem oder in einem Dokumenten Management-/ Archiv-System gespeichert sein. Ein Dokument kann eine oder mehrere Ressourcen aus ein und derselben Resource Group oder aus verschiedenen Resource Groups beziehen.

Der ResourceGroupLoader[30] bietet zur Instantiierung zwei verschiedene Konstruktoren:

- **ResourceGroupLoader()**: erzeugt einen 'leeren' ResourceGroupLoader, dem über die verschiedenen addXXX-Methoden (s.u.) verschiedene Ressourcen eingefügt werden können.
- **ResourceGroupLoader(InputStream is)**: erzeugt einen ResourceGroupLoader, der mit einem InputStream zu einer Ressource initialisiert wird.

28 com.levigo.jadice.docs.resource.ResourceFileLoader
29 com.levigo.jadice.docs.resource.ResourceUrlLoader
30 com.levigo.jadice.formats.modca.ResourceGroupLoader

Weitere Resource Groups können über verschiedene addXXX-Methoden hinzugefügt werden.

- **addFile(String fileName)**: fügt eine File-Ressource hinzu
- **addStream(InputStream is)**: fügt eine Ressource hinzu, die bereits als InputStream vorliegt
- **addUrl(String urlName)**: fügt eine URL-Ressource hinzu

4.8.4. ResourceMultiLoader

Die Klasse ResourceMultiLoader[31] bietet die flexibelste Implementierung des ResourceLoader-Interfaces. Sie verhält sich wie ein einfacher ResourceLoader, vereinigt in sich aber mehrere angemeldete ResourceLoader. Sämtliche zuvor beschriebenen Implementierungen können über die Methoden

- **addLoader(ResourceLoader loader)** hinzugefügt oder über
- **removeLoader(ResourceLoader loader)** wieder entfernt werden.

Eine Applikation kann damit alle zur Verfügung stehenden Ressourcen und korrespondierende ResourceLoader in einen ResourceLoader integrieren, am Loader oder im Dokument anmelden und wie gewohnt per „getResourceStream(...)" auf den gewünschten Ressourcen-InputStream zugreifen, ohne darauf Rücksicht nehmen zu müssen, wo und in welcher Art die eigentliche Ressource zur Verfügung gestellt wird.

Ein Beispiel für die Erstellung und Registrierung von ResourceLoadern findet sich in 5.Typische Anwendungsbeispiele.

4.9. SeekableInputStreams

Zur effizienten und speicherschonenden Verarbeitung großer Dokument-Datenmengen versucht der Viewer, sofern es das Bildformat erlaubt, nur die für die aktuelle Seitendarstellung benötigten Dokument-Daten zu lesen, zu verarbeiten und dynamisch zu puffern, statt alle Bild-Daten komplett im Speicher zu halten.

Für dieses Vorgehen werden Datenströme benötigt, deren Dateizeiger beliebig positioniert werden kann.

Der jadice viewer verwendet dafür SeekableInputStreams. Die Klasse SeekableInputStream[32] ist eine abstrakte Basisklasse, die die Klasse InputStream[33] um folgende Methoden erweitert:

- seek(int) — Positioniere den Dateizeiger
- length() — Größe der Datei in Bytes
- getFilePointer() — die Position, an der der Dateizeiger positioniert ist

31 com.levigo.jadice.docs.resource.ResourceMultiLoader
32 com.levigo.jadice.io.SeekableInputStream
33 java.io.InputStream

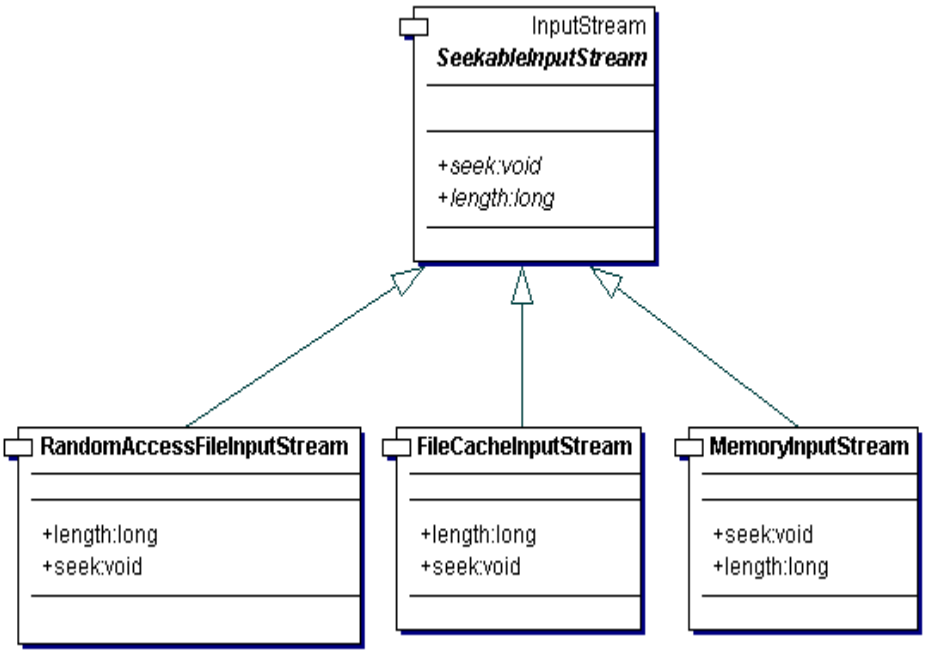

Abbildung 3 - Klassendiagramm der gebräuchlichsten SeekableInputStreams

Abbildung 3 bietet einen Überblick der gebräuchlichsten von der jadice API zur Verfügung gestellten Implementierungen der Schnittstelle SeekableInputStream.

Die Verwendung von SeekableInputStreams stellt sich für Integratoren einfach dar. Jadice stellt für bestimmte Typen von Datenströmen verschiedene Arten von SeekableInputStreams zur Verfügung, die den eigentlichen Datenstrom wrappen und dem Loader in der gewünschten „loadDocument" Methode übergeben werden können. Weiteres muss vom Integrator nicht getan werden.

Die gebräuchlichsten SeekableInputStreams werden in den folgenden Abschnitten kurz erläutert.

4.9.1. RandomAccessInputStream

RandomAccessInputStream[34] stellt ein SeekableInputStream für lokal verfügbare Bilddaten (d.h. als Datei im Dateisystem) dar.

Diese Klasse bietet einen Konstruktor mit einem parametrisierten File (ein File Objekt der Bilddatei) zur Instantiierung an. Diese Datei wird mit einem bestimmten FileInputStream geöffnet, der für die folgenden Lesevorgänge gehalten wird. Während der Verarbeitung in jadice bleibt der FileInputStream zu der Dokument-Datei offen und ein Dateizeiger wird innerhalb dieser Datei positioniert. Zu beachten ist hierbei, dass die Quelldatei, während das Dokument im Viewer geöffnet ist, nicht verändert werden darf. Dies führt, z.B. unter Windows, ggf. auch zu einem Sperren des Zugriffes auf die betreffende Datei.

34 com.levigo.jadice.io.RandomAccessInputStream

4.9.2. FileCacheInputStream

Zur Verarbeitung der Bilddaten im Viewer wird eine temporäre Datei erzeugt, in der bereits gelesene Daten abgelegt werden. Nach Gebrauch, spätestens jedoch beim nächsten Start des Viewers, werden die nicht mehr benötigten temporären Dateien gelöscht. Die temporären Dateien werden, falls der Viewer nicht anders konfiguriert ist, in dem durch das System vorgegebenen Temporär-Verzeichnis abgelegt.

Zur Instantiierung bietet diese Klasse einen Konstruktor, der einen gegebenen InputStream aufnimmt und eine Bearbeitungsdatei im gesetzten Temporär-Verzeichnis anlegt. Die dabei erzeugte Temporär-Datei wird nach Gebrauch wieder entfernt.

Ein weiterer Konstruktor bietet die Möglichkeit neben dem InputStream ein Temporär-Verzeichnis und ein Flag, ob die temp. Datei nach Gebrauch zu löschen ist, anzugeben.

4.9.3. MemoryInputStream

Bilddaten werden im Hauptspeicher zwischengespeichert.

Ähnlich dem FileCacheInputStream bietet diese Klasse einen Konstruktor, der einen gegebenen InputStream aufnimmt und für Lesevorgänge hält. Ein zweiter Konstruktor erlaubt zusätzlich eine Angabe der Blockgröße, in der Daten im Hauptspeicher gepuffert werden sollen.

Diese Art der internen Datenverwaltung wird oft im Kontext von Applets bevorzugt. Ein Vorteil dieser Datenhaltung ist ein extrem schneller Zugriff auf die Dokumentrohdaten. Nachteilig ist ein erhöhter Speicherbedarf durch die Datenhaltung im Hauptspeicher.

4.10. Annotation

Annotationen sind zusätzliche Informationen zu einem Dokument, die in Form von textuellen Anmerkungen oder grafischen Objekten zur Hervorhebung vorkommen können. Annotationen werden in einer eigenen Ebene über dem Dokument dargestellt. In dieser Form stellen sich Annotationen dem Anwender als optische Einheit mit dem Dokument dar. Als zusätzliche Dokumentinformationen gehören Annotationen aber physikalisch nicht zum eigentlichen Dokument. Änderungen an Annotationen verändern damit das ursprüngliche Dokument nicht.

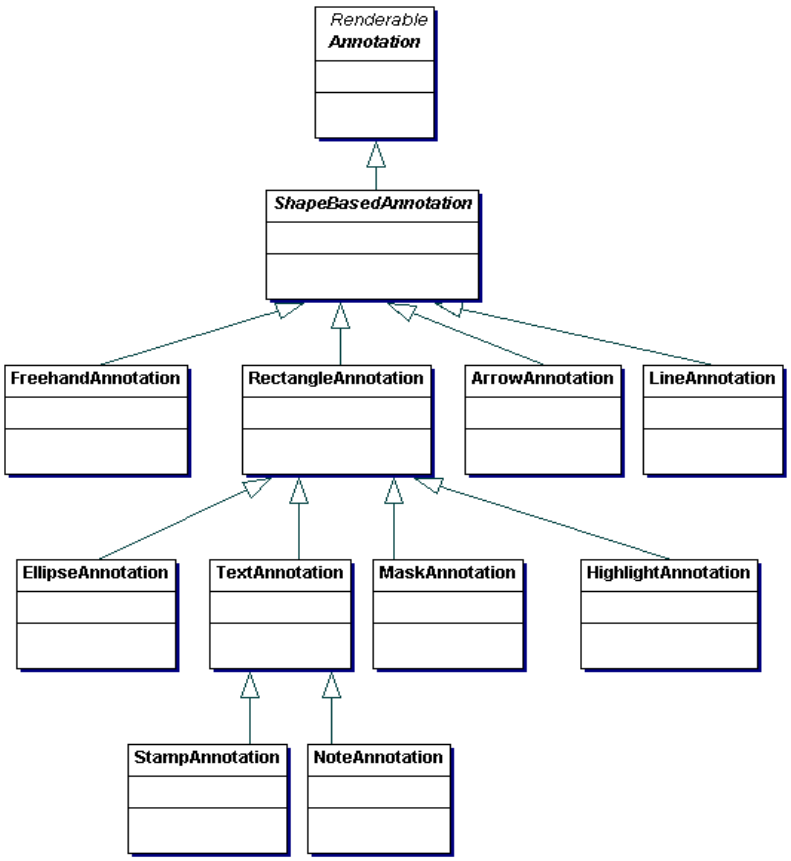

Abbildung 4 - Vererbungshierarchie Annotationen – Grober Überblick

Zur Verfügung gestellte Typen sind:

- NOTE:

  textuelle Anmerkung, dargestellt als gelbe Haftnotiz

- HIGHLIGHT:

  Hervorhebung, dargestellt durch ein gefülltes und transparentes Rechteck

- MASK:

  Maskierung von Teilen eines dargestellten Dokuments durch ein gefülltes, nicht transparentes Rechteck

- ARROW:

  ein Hinweis auf eine bestimmte Stelle des Dokuments durch einen Pfeil

- ELLIPSE:

  eine nicht gefüllte Ellipse, geeignet um einen bestimmten Bereich einzurahmen, z.B. einen Betrag, ein Datum oder einen Namen

- RECTANGLE:

  ein nicht gefülltes Rechteck. Ebenso wie die Ellipse, geeignet um einen bestimmten Bereich einzurahmen, z.B. einen Betrag, ein Datum oder einen Namen

† LINE:

eine Linie, z.B. zur Unterstreichung eines bestimmten Wortes oder einer bestimmten Zahl

† TEXT:

eine mit transparentem Hintergrund direkt auf der Seite angebrachte Textbemerkung

† FREEHAND:

eine Freihandzeichnung. Bei Erstellung wird ein Polygon aufgrund der Mausbewegung erstellt. Geeignet um ungleichmäßige Bereiche, die nicht mit einer Ellipse oder Rechteck zu umranden sind, hervorzuheben.

† STAMP:

ein „Stempel" mit transparentem Hintergrund, einem farbigen Rahmen, rotierbar mit farbigem textuellem Inhalt

Abbildung 4 stellt einen groben Überblick der Klassenarchitektur von Annotationen dar. Die Klasse Annotation[35] ist eine abstrakte Klasse, die die Basis aller Annotationen darstellt. ShapeBasedAnnotation[36] stellt eine weitere Abstraktionsebene dar, die allen Annotationen, die auf geometrischen Formen (Shape[37]) basieren, zugrunde liegt. Als konkrete Annotation ist diese Klasse nicht von Relevanz, für Integratoren jedoch, die eigene Annotationen definieren möchten, stellt sie eine nützliche Basis dar.

In der jadice API steht eine logische Vererbung, aufgrund der Darstellungs- und Verarbeitungseigenschaften einzelner Annotationstypen, im Vordergrund. Damit können zum einen Konsistenz und Vermeidung von Redundanzen gewährleistet werden, zum anderen ist ein flexibler Austausch des Annotations-Support für verschiedene Archivsysteme oder komplett selbst definierte Annotationen möglich.

Die Verwaltung von Annotationen übernimmt bei Userinteraktion die Viewer Komponente. Er erlaubt Benutzern, Annotationen verschiedener Typen anzulegen, zu löschen und ihre Eigenschaften zu ändern. Der Typ einer anzulegenden Annotation hängt davon ab, in welchem Modus sich der Viewer zum Zeitpunkt der Erstellung befindet. Dieser Modus wird vom Anwender indirekt durch die Aktivierung eines entsprechenden Tools der Annotations-Toolbar oder vom Integrator über die Methode AnnotationCreatorPane[38].setAnnotationMode(int) gesetzt.

Sollten Integratoren die Annotations-Verwaltung selbst steuern oder Annotationen programmtechnisch bearbeiten wollen, finden sie weiterführende Informationen zu diesem Thema in der jadice viewer Annotation Dokumentation (jadice viewer: Annotationen – Laden, Speichern, Bearbeiten).

Um Annotationen im ImagePlus-Format zu laden oder zu speichern, können die Klassen ImagePlusAnnotationFormatInfo[39] und ImagePlusAnnotationFile[40] verwendet werden. Näheres dazu finden Sie in folgendem Kapitel.

Analog können FileNet und FileNet P8 Annotationen mithilfe der Klassen FileNetAnnotationFormatInfo[41], FileNetAnnotationFile,

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

FileNetP8AnnotationFormatInfo[42] und FileNetP8AnnotationFormatInfo[43] geladen bzw. gespeichert werden. Weitere Information über die Verwendung von FileNet Annotionen finden sich in der jadice viewer Annotation Dokumentation (jadice viewer: Annotationen – Laden, Speichern, Bearbeiten).

Hinweis:

Das Laden und Speichern von Annotationen muss explizit durchgeführt werden und wird z.B. **nicht** etwa automatisch beim Laden des Dokuments erledigt.

Hinweis:

Nicht alle Archivsysteme unterstützen sämtliche von jadice zur Verfügung gestellten Annotationstypen. Beispielsweise kennt FileNet keine Ellipse- oder Mask-Annotationen; IBM Content Manager ImagePlus® für OS/390 oder AS400 unterstützt nur Highlight-/ Note und Mask-Annotationen, während der IBM ContentManager for Multiplatforms alle Annotationstypen erlaubt.

Durch eine einfache textuelle Änderung an der Konfiguration können Integratoren die gewünschten Annotationstypen de-/ oder aktivieren.

Für weitere Informationen zu diesem Thema sei an dieser Stelle auf Kapitel 5.4.2.2.Anpassen der Menü- oder Toolbar-Struktur und auf die jadice viewer Annotation Dokumentation (jadice viewer: Annotationen – Laden, Speichern, Bearbeiten) hingewiesen.

4.10.1. Veränderungen an Annotationen

Veränderungen an Annotationen werden mithilfe der Schnittstelle AnnotationListener[44] propagiert. Anwendungen, die über Eigenschaftsänderungen von bestehenden Annotationen, wie Größe, Position, Farbe usw., aber auch über das Anlegen bzw. Entfernen von Annotationen informiert werden möchten, können sich als AnnotationListener an der Klasse AnnotationEventcaster[45] registrieren. Eine Anmeldung an der Klasse AnnotationenEventcaster erfolgt stets global für alle Annotationsänderungen innerhalb der laufenden Anwendung, unabhängig von Document oder Viewerinstanzen. Das heißt: Ein registrierter AnnotationListener wird applikationsweit über alle Änderungen an Annotationen über die Methode

- **annotationChanged(AnnotationEvent e)**

informiert.

Der mitgelieferte AnnotationEvent[46] gibt Auskunft welche Annotation in welcher Art verändert wurde. Dazu können folgende Methoden der Klasse AnnotationEvent verwendet werden:

† **getAnnotation()** liefert eine Referenz auf die veränderte Annotation. Mithilfe der Annotation ist ein Zugriff auf das umschließende Annotationsseitensegment, die zugehörige Seite und das Dokument möglich.

† **getEventType()** gibt die Art der Veränderung an, z.B. Größe, Position, Farbe, Selektionsstatus usw.

† **getNewValue()** gibt den neuen Eigenschaftswert der Annotation an, z.B. die neue Farbe.

42 com.levigo.jadice.formats.annofilenetp8.FileNetP8AnnotationFormatInfo
43 com.levigo.jadice.formats.annofilenetp8.FileNetP8AnnotationFile
44 com.levigo.jadice.annotation.AnnotationListener
45 com.levigo.jadice.annotation.AnnotationEventcaster
46 com.levigo.jadice.annotation.AnnotationEvent

† **getOldValue()** gibt den ursprünglichen Eigenschaftswert an, z.B. die vorherige Farbe.

4.11. ImagePlusAnnotationFormatInfo

Um eine Verwechselung von IBM VisualInfo/ImagePlus-kompatiblen Annotationen mit MO:DCA-Dokumenten während des Ladevorgangs zu vermeiden, ist es notwendig, eine Instanz der Klasse ImagePlusAnnotationFormatInfo dem Loader in der loadDocument()-Methode zu übergeben.

ImagePlusAnnotationFormatInfo ist eine Formatinformation für IBM VisualInfo/ImagePlus-kompatible Annotationen. Diese Klasse sorgt während des Ladevorgangs dafür, dass die geladenen Daten nicht als eigenständiges MO:DCA-Dokument interpretiert, sondern als zusätzliche Dokument-Informationen, eben als Annotationen, angesehen und in einer eigenen Schicht „über" dem Dokument verwaltet und dargestellt werden.

Ein Beispiel zum Laden von Annotationen findet sich unter 5.Typische Anwendungsbeispiele.

Hinweis:

Bis IBM ContentManager Vs. 7.x waren Annotationseigenschaften auflösungsunabhängig. Dies hat sich mit Version 8.x geändert. Damit konnten bis zur Version 7.x Annotationen unabhängig vom Bild-Dokument geladen werden. Ab Version 8.x sollten Integratoren Annotationen erst laden, nachdem der Ladevorgang des Dokuments beendet ist, anderenfalls sind Fehler in der Positionierung und Größe von Annotationen zu erwarten.

Hinweis:

Ebenso wie ImagePlus-kompatible Annotationen können auch FileNet Annotationen mit dem Viewer angezeigt und bearbeitet werden. Die entsprechende FormatInfo Klasse für FileNet Annotationen ist FileNetAnnotationFormatInfo.

4.12. ImagePlusAnnotationFile

ImagePlusAnnotationFile dient zum Speichern von Annotationen im ImagePlus Format. Um einen Zugriff auf die zu speichernden Annotationen zu erhalten, wird dem ImagePlusAnnotationFile im Konstruktor ein Dokument übergeben, dessen Annotationen zu speichern sind.

Ebenso wie der Loader verschiedene Lade-Methoden zur Verfügung stellt, bietet auch jede FormatNameFile-Instanz verschiedene Speicher-Methoden, um generell oder qualifiziert (z.B. nur bestimmte Seiten o.ä.) zu speichern. Die einfachste Art alle Annotationen eines Dokuments in einen bestimmten OutputStream zu speichern ist ein Aufruf der Methode „**save(OutputStream)**".

Genauere Informationen entnehmen Sie bitte der jadice viewer API Dokumentation.

4.13. FileNet und FileNet P8 Annotationen

Ebenso wie ImagePlus-kompatible Annotationen können auch FileNet und FileNetP8 Annotationen mit dem Viewer angezeigt und bearbeitet werden. Die entsprechende FormatFile Klasse für FileNet Annotationen ist FileNetAnnotationFormatFile bzw. FileNetP8AnnotationFormatFile.

Weitere Information über die Verwendung von FileNet und FileNet P8 Annotionen finden sich in jadice Auslieferung unter Annotation-Dokumentation (jadice viewer: Annotationen – Laden, Speichern, Bearbeiten).

4.14. RenderContext

An einigen Stellen dieses Dokuments wird auf die Klasse RenderContext[47] Bezug genommen.

Im Normalfall arbeiten Integratoren eher weniger mit Instanzen dieser Klasse, dennoch werden an dieser Stelle zum besseren Verständnis die Eigenschaften und Aufgaben der Klasse RenderContext vorgestellt.

Zur Darstellung einer Seite werden im jadice viewer keine Images erzeugt, vielmehr rendern sich die Seiten wie auch alle ihre Segmente selbstständig in gegebene Grafikkontexte, z.B. Bildschirm, Drucker, etc.

Ein solcher Renderprozess wird stets von einer Instanz der Klasse RenderContext begleitet. Der RenderContext kapselt verschiedene Darstellungsattribute, die bindend den Render Prozess und damit auch die Eigenschaften der Darstellung bestimmen.

Die Darstellungs-Attribute unterteilen sich in Direkt-Attribute, wie Zoom, Rotation oder ähnliches, und ProcessingSettings[48]. Direkt-Attribute können von einer Instanz der Klasse RenderContext direkt erfagt und auch verändert werden.

ProcessingSettings sind Attribute spezieller Natur, die Darstellungseigenschaften ganz bestimmter Art zusammenfassen und beschreiben, z.B. der AnnotationRenderSettings[49]. Mit AnnotationRenderSettings können Sichtbarkeitseigenschaften von Annotationen gesetzt werden. Dabei ist es möglich *alle* Annotationen un-/sichtbar zu machen oder *nur* Annotationen bestimmten Typs.

Weiterführende Informationen können der API-Dokumentation und der jadice viewer Annotation Dokumentation entnommen werden.

Desweiteren stellt der RenderContext affine Transformationen zur einfachen Umrechnung zwischen Dokument- und Device Koordinatensystem zur Verfügung.

Wie in Kapitel 2.4.1.Das Dokumentenmodell bereits erwähnt, können die in Dokumenten enthaltenen Seiten ihren Inhalt aus mehreren Datenströmen bzw. Datenformaten beziehen, aber auch die Seiten selbst können sich aus Daten verschiedener Datenströme zusammensetzen. Da diese Bilddaten verschiedenen Formats und unterschiedlicher Auflösung sein können, unterhält der Viewer intern ein Basis-Dokument-Koordinatensystem und transformiert nur zur Darstellung in die jeweiligen Devicekoordinaten. Dementsprechend verwalten Seitensegmente ihre Daten in Dokumentkoordinaten. Eine

47 com.levigo.jadice.docs.RenderContext
48 com.levigo.jadice.docs.ProcessingSettings
49 com.levigo.jadice.annotation.AnnotationRenderSettings

Transformation unter Berücksichtigung des Zoom-Faktors und der Rotation findet nur während des Renderprozesses in einen gegebenen Grafikkontext statt.

4.15. EditPanes

Der jadice viewer bietet die Möglichkeit, die Funktionalität des Viewers flexibel zu erweitern. Dazu können dem Viewer dokumentunabhängige Darstellungsschichten, so genannte EditPanes, hinzugefügt werden, die, ähnlich wie Seitensegmente, das Dokument transparent überlagern. Diese Schichten können die Darstellung von Dokumenten beeinflussen, aber auch Inputevents empfangen und eventgesteuerte Benutzerinteraktionen auslösen. Im Gegensatz zu Seitensegmenten, die Teil einer Seite bzw. eines Dokuments sind, gehören EditPane Instanzen nicht zu einer Seite, sondern sind dokumentunabhängige Darstellungsschichten eines Viewers.

Die Basisklasse aller EditPanes ist die AbstractEditPane[50], die unter anderem folgende Methoden zu Erweiterung bietet:

- **render(...)**

  Eigene Implementierungen können diese Methode überschreiben, um zusätzliche Hinweise auf der Seite zu platzieren. Dies können z.B. Seitendekorationen, Text, Zeichen, Icons und vieles mehr sein. Als Parameter erhält die Methode einen Graphics Context (Graphics2D), ein RenderControls[51] Objekt (RenderContext mit den aktuellen Darstellungseigenschaften des Viewers) und einen RenderObserver. Das Graphics Objekt kann genutzt werden, um das gewünschte Objekt oder Text darzustellen. Der RenderContext liefert dazu eventuell benötigte Angaben über Zoom, Rotation oder ähnliches. Falls ein RenderObserver übergeben wurde, kann dieser genutzt werden, um - sofern nötig - asynchron Dirty-Regions-Repaints auszulösen.

- **getEditEventListener()**

  EditPanes können Maus- oder Tastatur-InputEvents empfangen. Diese können von EditPanes verarbeitet werden, indem diese Methode eine Implementierung eines EditEventListener[52]s zurückgibt. Ein EditEventListener ist eine Schnittstelle, die verschiedene Methoden zur Verarbeitung von Maus- oder Tastatur-Events bereitstellt. Eine entsprechende Adapter-Klasse wird durch EditEventAdapter[53] zur Verfügung gestellt.

EditPanes eignen sich beispielsweise dazu, integrationsspezifische Anmerkungen auf einer Seite in beliebiger Form darzustellen und ggf. aktiv auf InputEvents zu reagieren.

Die Aktivierung einer EditPane findet durch eine Registrierung an einer bestimmten Viewer Instanz statt. Dazu bietet der Viewer die Methoden:

- **addEditPane(anEditPane)**
- **removeEditPane(anEditPane)**

50 com.levigo.jadice.AbstractEditPane
51 com.levigo.jadice.docs.RenderControls
52 com.levigo.jadice.EditEventListener
53 com.levigo.jadice.EditEventAdapter

Eine Liste aller an einer Viewer Instanz registrierten EditPanes kann mithilfe der Viewer Methode:

† **Collection getEditPanes()**

erhalten werden.

Ein Anwendungsbeispiel:

Die integrierende Applikation hat zu einer Seite zusätzliche Informationen. Dies soll dem Benutzer durch ein „Büroklammer"-Icon auf der Seite visualisiert werden. Wenn der Benutzer das Icon anklickt, soll sich ein Fenster mit der Zusatzinformation öffnen.

Dies kann einfach durch eine EditPane realisiert werden. Die EditPane rendert, falls zusätzliche Informationen vorhanden sind, ein „Büroklammer"-Icon in seiner „render" Methode. Der zugehörige EditEventListener reagiert auf Maus-Events. Nach Überprüfung, ob der Mausklick auf dem Icon stattgefunden hat, wird das Fenster zur Ansicht der zusätzlichen Information geöffnet.

Ein weiteres typisches Anwendungs-Beispiel für ein EditPane ist die Klasse HoverLens. Weitere Informationen dazu in 4.22.1.HoverLens.

4.16. BasicJadicePanel

Die einfachste Art einen Viewer einzubinden bietet die Klasse BasicJadicePanel[54]. BasicJadicePanel bietet eine komplette, voll funktionale Viewer Ansicht und ist als JPanel[55] einfach in die Benutzungsoberfläche integrierender Applikationen einzubetten.

Diese Klasse enthält

† eine Viewerinstanz

† eine Toolbar mit den wichtigsten Anzeige- und Bearbeitungsmöglichkeiten

† eine Annotation-Toolbar zur Erstellung von Annotationen (ein- und ausklappbar)

† eine Statusbar

† ein Kontext-Menü

† (optional) eine Menübar

54 com.levigo.jadice.gui.BasicJadicePanel
55 javax.swing.JPanel

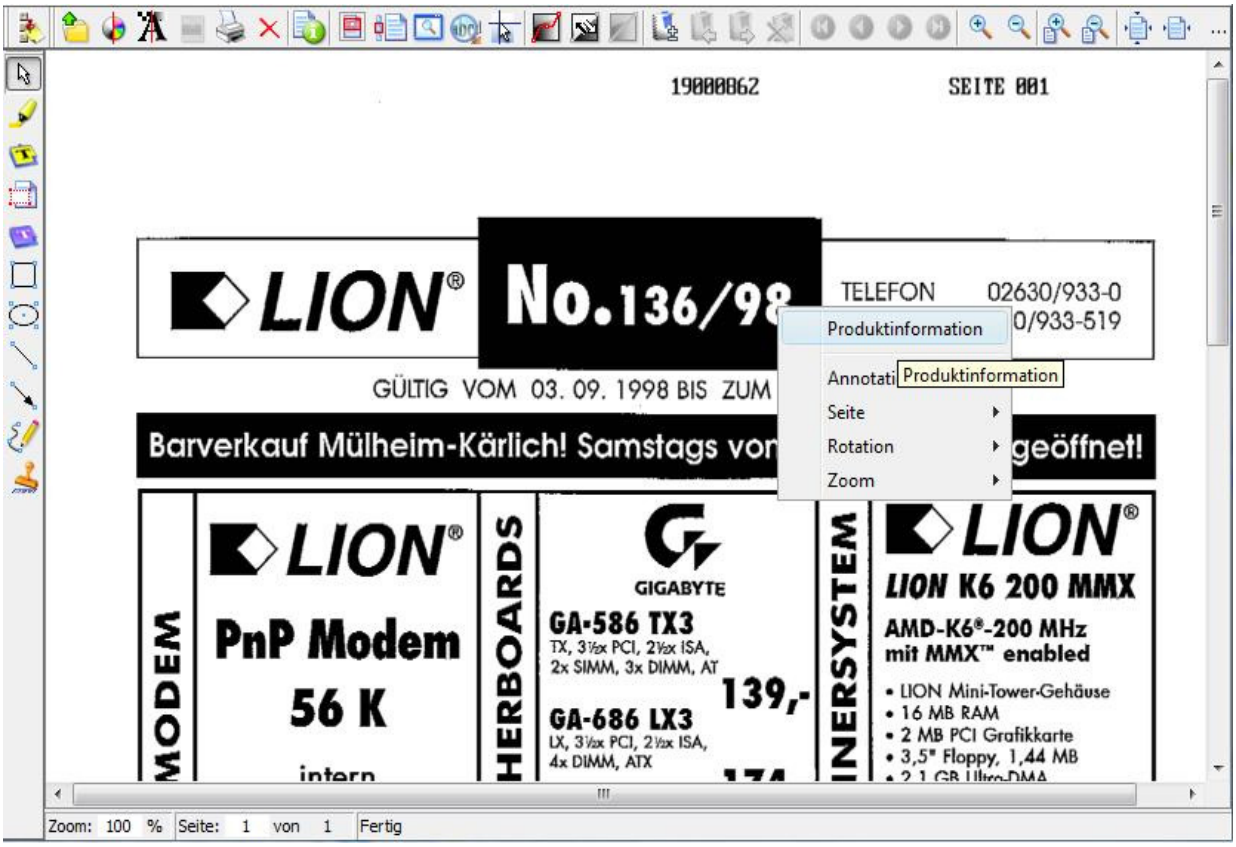

Abbildung 5 - BasicJadicePanel

Menüs und Toolbars basieren auf dem neuen Action- und Command Konzept des Viewers. Dementsprechend ist zur Veränderung oder Anpassung der einzelnen Tools bzw. der Struktur von Menüs oder Toolbars kein weiterer Programmieraufwand nötig. Lediglich müssen die gewünschten Änderungen an den Konfigurations-Dateien *menucomponents.properties* und *actions.properties* vorgenommen werden. Ebenso können eigene Tools integriert werden.

Genauere Informationen finden sich unter 3.3.Die Konfigurationsdateien, 5.4.Actions-Commands-Context und 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien.

BasicJadicePanel ist eine Erweiterung der Klasse AbstractJadicePanel[56]. AbstractJadicePanel vereinigt lediglich einen Viewer und eine interagierende Statusbar in sich. Die Implementierung der Toolbars wie auch der Menüs überlässt diese Klasse ihren Nachfahren. Sollten Integratoren nicht das Action- und Command Konzept des Viewers nutzen wollen, kann direkt von dieser Klasse geerbt werden - Toolbars und Menüs sind dann jedoch von den Integratoren selbst zu erstellen.

Als Hilfestellung wie Toolbars und Menubars erstellt werden können, aber auch als Basis für eigene Entwicklungen, wird die Klasse BasicJadicePanel in der Auslieferung ebenfalls als Sourcecode im example-src Verzeichnis zur Verfügung gestellt.

56 com.levigo.jadice.gui.AbstractJadicePanel

4.17. AddOns

Neben dem Anzeigen von Dokumenten bietet die jadice document platform einige hilfreiche und nützliche Erweiterungen an, wie eine Dokument- bzw. Seiten-Übersicht, Lesezeichen und ähnliches, die im Zusammenspiel mit Viewer Instanzen genutzt werden können.

Im Folgenden werden alle diese Klassen AddOns genannt und in den Kapiteln 4.18.JadiceBookmark bis 4.23.GradationCurveControl beschrieben.

Eigenschaften, die allen AddOns eigen sind, sind in diesem Kapitel zusammengefasst.

4.17.1. Erzeugung

Ein AddOn interagiert immer mit einer Viewerinstanz bzw. einem Dokument oder den darin enthaltenen Seiten. Aus diesem Grund muss einem AddOn stets eine Viewerinstanz assoziiert werden. Dies kann bereits bei der Erstellung eines AddOns passieren oder später über eine jeweils vorhandene Methode „**setViewer(Viewer)**". Damit ist jedes AddOn stets in der Lage an eine neue Viewerinstanz gebunden zu werden oder anders formuliert: Ein AddOn kann verschiedene Viewer bedienen, aber jeweils nur einen zum gleichen Zeitpunkt.

Vorhandene Konstruktoren:

- Default- Konstruktor
- Konstruktor mit einer Viewerinstanz

4.17.2. Aufruf über Commands

Das jadice viewer Paket liefert im Allgemeinen zwei Implementierungen von Commands zur Anzeige des jeweiligen AddOns:

- **Toggle*AddOnName***

  Beispiel: ToggleSorter. Zeigt/Versteckt das entsprechende AddOn eingebettet in einem JFrame.

- **ToogleInternal*AddOnName***

  zur Nutzung in MDI-Umgebungen: Zeigt/Versteckt das entsprechende AddOn eingebettet in einem JInternalFrame.

Für eigene Implementierungen können diese Commands überschrieben oder eigene Commands registriert werden.

Weiterführende Informationen zum Thema Commands finden sich unter Kapitel 3.Die jadice Integrator API, 5.4.Actions-Commands-Context und 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien.

4.17.3. Integration in verschiedene Umgebungen

Ist eine Viewerinstanz zu einem AddOn assoziiert, werden für das AddOn relevante Veränderungen erkannt und entsprechend aktualisiert. Beispielsweise aktualisieren Lesezeichen eigenständig ihre Seitennummer, wenn Seiten eines Dokuments im Viewer umsortiert wurden.

Werden AddOns in einer Umgebung mit mehreren Viewerinstanzen eingesetzt, sollten sie über den jeweils aktiven Viewer mittels der Methode „**setViewer(Viewer)**" informiert werden, um sich korrekt aktualisieren zu können.

Zur einfachen Integration sind alle AddOns Nachfahren von javax.swing.JComponent. Zusätzlich bietet das jadice Paket korrespondierende JFrames bzw. JInternalFrames.

- ***AddOnName*Frame**

  als JFrame, mit Zugriffsmethode auf das eingebettete AddOn und einer „setViewer(Viewer)" Methode, die ein Umschalten zwischen Viewerinstanzen erlaubt.

- ***AddOnName*InternalFrame**[57]

  als JInternalFrame, mit Zugriffsmethode auf das eingebettete AddOn und einer „setViewer(Viewer)" Methode, die ein Umschalten zwischen Viewerinstanzen erlaubt.

4.18. JadiceBookmark

Lesezeichen sind ein nützliches Hilfsmittel, wenn bevorzugt mit mehrseitigen Dokumenten gearbeitet wird und bestimmte Seiten ohne langes Suchen angezeigt werden sollen. In einem solchem Fall kann man diese Seiten mit einem Lesezeichen belegen.

Zur bequemen Ansteuerung von Dokument zugehörigen Marken werden im jadice viewer Instanzen der Klasse JadiceBookmark[58] verwendet. Diese Klasse repräsentiert im jadice Paket ein Lesezeichen und kann zu einer bestimmten Seite als Marke gesetzt werden. Zusätzlich werden auch die zum Zeitpunkt der Erstellung des Lesezeichens aktuellen Darstellungseigenschaften der Seite wie Rotation, Zoom und ähnliches verwaltet. Bei späterer Aktivierung des Lesezeichens wird dann die entsprechende Seite mit den in der Markierung abgelegten Darstellungseigenschaften wieder angezeigt.

Ein Lesezeichen trägt die folgenden Eigenschaften:

- Seitennummer
- Seite
- Dokument
- Rotation
- Zoomfaktor
- Position der Seite im Viewer (Pan-Point), nützlich für Seitendarstellungen, die größer als die des Viewer sind. Über den Pan Point wird die Seitenansicht automatisch beispielsweise in die untere rechte Ecke gescrollt.
- eine Textbeschreibung der Marke zur Identifizierung durch den Benutzer

Die programminterne Verwendung und Erstellung von Lesezeichen wird im Kapitel 4.19.DocumentBookmarkHandler näher beschrieben.

Das jadice Paket bietet dem Enduser Commands zur direkten Steuerung und Verwaltung von Seiten-Markierungen:

- **BookmarkToggleCommand**[59]

57
58 com.levigo.jadice.addon.bookmarks.JadiceBookmark
59 com.levigo.jadice.addon.bookmarks.BookmarkToggleCommand

Dieses Command erzeugt bzw. entfernt ein Lesezeichen zur aktuellen Seite. Existiert ein Lesezeichen zur aktuellen Seite, ist es selektiert. Die Aktivierung des Commands löscht das jeweilige Lesezeichen. Andernfalls ist das Command unselektiert und eine Aktivierung erzeugt ein Lesezeichen mit den aktuellen Einstellungen wie Seitennummer, Rotation, Zoomfaktor und Pan-Point.

- **BookmarkBrowsingCommand**[60]

  Mithilfe dieses Commands kann zwischen vorhandenen Lesezeichen geblättert werden. Die Richtung, ob zum vorherigen Lesezeichens oder zum nächsten Lesezeichen des aktuellen Dokuments geblättert werden soll, kann als Command Parameter definiert werden. Vordefiniert in der jadice Auslieferungsversion sind bereits:

  - **NextBookmark und PrevBookmark**[61]

- **BookmarkRemovalCommand**[62]

  Dieses Command löscht je nach Konfiguration alle Lesezeichen eines bestimmten Dokuments oder alle zur Zeit vorhandenen Lesezeichen. Vordefiniert in der jadice Auslieferungsversion sind bereits:

  - RemoveBookmarks und RemoveAllBookmarks[63]

4.19. DocumentBookmarkHandler

Die zentrale Klasse zur programminternen Bearbeitung und Verwaltung von Lesezeichen ist der DocumentBookmarkHandler[64]. Eine Instanz dieser Klasse ist über die statische Methode „**getInstance()**" zu erhalten.

Integratoren können diese Klasse nutzen, um bereits vorhandene Lesezeichen vor einer Verarbeitung aus einem Property Objekt zu laden oder nach der Bearbeitung in einem Property Objekt zu speichern.

Darüber hinaus bietet der DocumentBookmarkHandler verschiedene Methoden, um die Bookmark-Verwaltung komplett programmintern abzuwickeln:

- Lesezeichen hinzufügen
- einzelne Lesezeichen löschen
- alle Lesezeichen einer Seite löschen
- alle Lesezeichen eines Dokuments löschen
- Lesezeichen editieren
- Lesezeichen aktivieren (ein gegebener Viewer zeigt die jeweilige Seite entsprechend Bookmark Einstellungen an)
- Zugriff auf
  - alle Lesezeichen
  - alle Lesezeichen eines Dokument
  - alle Lesezeichen einer Seite

60 com.levigo.jadice.addon.bookmarks.BookmarkBrowsingCommand
61 Zu finden in com.levigo.jadice.resources.properties.commands.properties
62 com.levigo.jadice.addon.bookmarks.BookmarkRemovalCommand
63 Zu finden in com.levigo.jadice.resources.properties.commands.properties
64 com.levigo.jadice.addon.bookmarks.DocumentBookmarkHandler

- Anzahl der Lesezeichen
- Abfrage, ob Lesezeichen verändert wurden

- Laden
- Speichern

Unter bestimmten Umständen kann es von Vorteil sein, über Veränderungen an Lesezeichen informiert zu werden. Für diesen Zweck können Integratoren Implementierungen des Interface BookmarkListener[65] beim DocumentBookmarkHandler registrieren.

Durch die Benutzung des DocumentBookmarkHandlers vereinfacht sich für Integratoren die Bearbeitung an Lesezeichen erheblich, da die Verwaltung und Synchronisation bereits Teil des Handlers sind und nicht durch den Integrator verwirklicht werden müssen.

4.20. PageSorter

jadice verfügt über die Möglichkeit, die Seiten eines Dokuments verkleinert in Form von Thumbnails darzustellen. Mittels der Maus können eine oder mehrere Seiten selektiert und per Drag and Drop verschoben werden. Die Reihenfolge der Seiten wird automatisch im Viewer angepasst. Die im Viewer angezeigte Seite, ebenso wie alle selektierten Thumbnails sind farblich hevor gehoben. Ein Doppelklick auf ein ThumbnailPanel aktiviert diese Seite im Viewer.

Soll der PageSorter als reine Seitenübersicht eines Dokuments genutzt werden, kann die Sortierungsfunktionalität mittels der Methode „**setSortingEnabled(boolean)**" ausgeschaltet werden.

Die Funktionalität des Seitensortierers wird durch die Klasse PageSorter[66] bereitgestellt, die als JComponent sehr einfach in die Benutzungsoberfläche der integrierenden Applikation einbettbar ist.

Als Viewer AddOn gelten für den PageSorter die Ausführungen unter 4.17.AddOns.

4.20.1. Unterstützung von PopupMenus im PageSorter

Der PageSorter unterstützt die Verwendung von selbst definierten JPopupMenus, um seine eigene Funktionalität und Bedienbarkeit zu erweitern.

Integratoren können selbstdefinierte Popup-Menüs für unterschiedliche Zwecke dem PageSorter hinzufügen:

- ein JPopupMenu zur Anzeige auf den verkleinerten Seiten:
  Für Funktionen, die seitenbezogen sind (z.B. „Seite entfernen")
- ein JPopupMenu zur Anzeige auf dem Hintergrund des verwendeten Panel:
  Für Funktionen, die global gelten (z.B. „Sortieren Aktivieren/Deaktivieren").

Je nachdem, an welcher Position (auf einer Seite oder zwischen den Seiten im Seitensortierer) das Kontext-Menü vom Benutzer angefordert wird und ob ein entsprechendes Popup-Menü gesetzt ist, öffnet sich das globale oder das seitenbezogene Popup-Menü. Selbstverständlich ist es ebenfalls möglich das gleiche Kontext-Menü als globales und seitenbezogenes Menü zu setzen.

65 com.levigo.jadice.addon.bookmarks.BookmarkListener
66 com.levigo.jadice.addon.pagesorter.PageSorter

Die Umsetzung der zu den angebotenen Menüpunkten gehörenden Funktionen ist vom Integrator zu erstellen.

Menüs können im Seitensortierer über folgende Methoden des PageSorters gesetzt werden:

† **setPanelPopupMenu(JPopupMenu popupMenu)**
Setzt das auf dem Panel anzuzeigende JPopupMenu

† **setThumbnailPopupMenu(JPopupMenu popupMenu)**
Setzt das auf den Thumbnails anzuzeigende JPopupMenu

4.21. NavigatorPanel

Zur Bearbeitung von großen Seiten oder wenn im Viewer ein sehr hoher Zoomfaktor gesetzt ist, bietet der jadice viewer eine Seiten-Übersicht einer einzelnen Seite.

Die aktuelle im Viewer dargestellte Seite wird verkleinert als Thumbnail präsentiert, wobei der vom Viewer dargestellte Ausschnitt als kleines transparentes Rechteck über der Seite visualisiert wird. Dieses Rechteck kann mittels der Maus bewegt werden, um innerhalb einer Seite zu navigieren, d.h. den dargestellten Seiten-Ausschnitt im Viewer zu verschieben.

Die Seitenübersicht wird durch die Klasse NavigatorPanel[67] bereitgestellt und kann als JComponent beliebig in die integrierende Applikation eingebunden werden. Der Navigator arbeitet in zwei verschiedenen Modi:

† die Seite wird mit Rotation 0° angezeigt

† die Seite wird in der im Viewer verwendeten Rotation angezeigt

Dieser Modus kann durch die Methode „**getFollowViewerRotation()**" erfragt, bzw. durch „**setFollowViewer-Rotation(boolean)**" gesetzt werden. Um zusätzlich den Benutzer auf den gewählten Modus aufmerksam zu machen, kann für jeden Modus ein Beschreibungstext gesetzt werden, der in der Thumbnail-Ansicht der Seite integriert wird.

Als Viewer AddOn gelten für das NavigatorPanel die Ausführungen unter 4.17.AddOns.

4.22. Lens

Zur besonders genauen Darstellung von Seiten-Details unterstützt der jadice viewer eine Lupenansicht. Die Lupe stellt einen stark vergrößerten Seitenausschnitt entsprechend der Position des Maus-Cursors im Viewer dar.

Zur Fixierung der Lupe genügt ein einfacher Mausklick an die entsprechende Stelle im Viewer, somit kann bspw. das fixierte Seitendetail mit einer anderen Stelle der Seite verglichen werden. Der fixierte Zustand wird dem Benutzer durch eine „Frozen"-Markierung visualisiert, der Text dieser Markierung kann über entsprechende „getter-/setter" Methoden erfragt/angepasst werden.

Die Zoom-Stufen können über Mausklick in die Lupe oder unter Zuhilfenahme der Methoden „**getScale()/setScale()**" erfragt bzw. verändert werden.

Die Lupe wird durch die Klasse Lens[68] bereitgestellt und als Viewer AddOn gelten die Ausführungen unter 4.17.AddOns.

67 com.levigo.jadice.addon.navigator.NavigatorPanel
68 com.levigo.jadice.addon.lens.Lens

4.22.1. HoverLens

Als Alternative zur Lupe in einem separaten Fenster, gibt es die HoverLens[69]. Eine HoverLens entspricht einer über der Seitenansicht im Viewer schwebenden Lupe, die an die Bewegung der Maus gebunden ist.

Eine HoverLens kann sich in zwei Formen (rechteckig oder rund) und in verschiedenen Größen darstellen, wobei das gewünschte Aussehen über die Methode "**setHoverShape(...)**" und die Dimension „**setHoverSize(Dimension)**" bestimmt werden kann.

Eine Fixierung der HoverLens ist ebenfalls möglich und kann über Strg-(oder auch Ctrl-) Taste kombiniert mit einem Mausklick gesetzt bzw. gelöst werden.

Ebenso wie bei der Lupe können die Zoom-Stufen über Mausklick oder über die Methoden „**getScale()/setScale()**" erfragt bzw. verändert werden.

Als Erweiterung der Klasse EditPane (siehe auch 4.15.EditPanes) stellen Instanzen von HoverLens keine JComponent dar, die zu integrieren ist. Stattdessen wird zur De-/Aktivierung einer HoverLens die entsprechende Instanz am Viewer de-/registriert. Dies geschieht mittels:

- **viewerInstance.addEditPane(hoverLensInstance)**
- **viewerInstance.removeEditPane(hoverLensInstance)**

4.23. GradationCurveControl

Zur Änderung der Darstellung von Bilddaten bietet jadice eine veränderbare Übertragungskurve von skalierter Pixelintensität zu dargestellter Pixelintensität auf dem Bildschirm. Die Übertragungskurve ist als Instanz der GradationCurve im RenderContext hinterlegt.

Die Klasse GradationCurveControl[70] stellt eine Gradationskurve visuell dar und erlaubt es, Stützstellen der Kurve zu definieren, zu verschieben oder zu löschen. Über die Stützstellen der Kurve findet eine Spline-Interpolation erster Ordnung statt, um einen kontinuierlichen Kurvenverlauf zu erhalten.

Eine Instanz einer GradationCurveControl kann beliebige Gradationskurven bearbeiten oder auch mit einer Viewer Instanz interagieren.

Dazu arbeitet diese Klasse in zwei unterschiedlichen Modi:

- eine Gradationskurve ist gesetzt die angegebene Kurve wird verändert, eine Interaktion mit einem Viewer findet nicht statt. Eine Gradationskurve wird durch die Klasse GradationCurve repräsentiert, vgl. Sie dazu bitte 4.23.1.GradationCurve.

- eine Viewer Instanz ist gesetzt es wird die vom Viewer benutzte Gradationskurve verwendet, eine Interaktion mit einem Viewer findet statt.

Als Viewer AddOn gelten für die GradationCurveControl die Ausführungen unter 4.17 AddOns.

4.23.1. GradationCurve

Diese Klasse enthält die beschreibenden Punkte (Datenhintergrund) einer Gradationskurve, genauer einer Abbildung zwischen der Pixel- und dargestellter Intensität.

69 com.levigo.jadice.addon.lens.HoverLens
70 com.levigo.jadice.addon.gradation.GradationCurveControl

Als Erweiterung der Klasse NaturalCubic-Spline1D[71] definiert sich die Gradationskurve über eine Menge von Eckpunkten, die aufgrund der Natur von Gradationskurven bestimmten Eigenschaften genügen müssen.

Gradationskurven können vollkommen losgelöst von Viewer Instanzen gehalten und verarbeitet werden. Ihre Verwendung ist vielfältig, beispielsweise können Gradationskurven zum Dokumentendruck definiert oder als feste Einstellungen hinter eigene Intensitäts-Commands gelegt werden.

Eigenschaften von Gradationskurven können als Properties Objekt geladen bzw. gespeichert werden.

Bitte beachten Sie, dass Gradationskurven eine Änderung der Darstellung nur bei Bilddaten bewirken! Text und andere Renderelemente wie Linien, Shapes oder ähnliches bleiben in ihrer Anzeige unberührt.

4.23.2. GradationCurveFileHandler

GradationCurveFileHandler[72] ist eine Utility Klasse zum Laden und Speichern von Gradationskurven in/von einem Dateisystem.

Ein Objekt der Klasse GradationCurveFileHandler kann entweder für eine GradationCurve oder eine GradationCurveControl instantiiert werden und stellt verschiedene Methoden zum Laden bzw. Speichern von Gradationen zur Verfügung.

- **openGradationCurveFromFile()**

  Eine Dateiauswahl wird angezeigt, die vom Benutzer gewählte Datei wird geladen.

- **openGradationCurveFromFile(File)**

  Die angegebene Datei wird geladen.

- **openGradationCurveFromFile(String)**

  Die über einen Dateinamen angegebene Datei wird geladen.

- **saveGradationCurveToFile()**

  Eine Dateiauswahl wird angezeigt, an dem vom Benutzer gewählten Pfad wird eine Datei des angegebenen Namens gespeichert.

- **saveGradationCurveToFile(File)**

  Die Angaben werden in die angegebene Datei gespeichert.

- **saveGradationCurveToFile(String)**

  Die Angaben werden in die über den Dateinamen angegebene Datei gespeichert.

4.24. PrinterJava2

PrinterJava2[73] ist die zentrale Klasse zum Druck von Dokumenten aus dem jadice. Diese Klasse ist abgeleitet von der abstrakten Basisklasse AbstractPrinter[74], die grundsätzliche Druckfunktionalität bietet und von Integratoren genutzt werden kann, um eigene Printer zu verwirklichen.

71 com.levigo.util.math.NaturalCubicSpline1D
72 com.levigo.jadice.util.GradationCurveFileHandler
73 com levigo.jadice.docs.printer.PrinterJava2
74 com levigo.jadice.docs.printer.AbstractPrinter

Eine Instanz der Klasse PrinterJava2 ist über den Default-Konstruktor zu erhalten, alternativ kann ein Konstruktor mit einem parametrisierten PrinterJob verwendet werden.

Vor dem Start eines Dokumentendrucks ist es notwendig eine Dokument Instanz anzugeben, deren Seiten gedruckt werden sollen. Alternativ kann statt eines Dokuments auch ein Array von zu druckenden Seiten gesetzt werden.

Weiterhin kann ein RenderContext übergeben werden, um Darstellungs-Eigenschaften des Render-Prozesses in dem gegebenen Drucker-Device zu bestimmen. Standardmäßig kann hier der RenderContext des Viewers verwendet werden, aber auch eine durch den Integrator angepasste Instanz ist möglich, so z.B. ein RenderContext mit einer speziellen Gradationskurve für Druckprozesse oder angepassten AnnotationsRenderSettings, zum Anzeigen oder Verbergen von Annotationen.

Einige Angaben des RenderContextes werden beim Druck ignoriert, da sie durch die Ausgabe in ein Druck-Device unabänderlich vorgegeben sind. Dies betrifft Angaben wie Zoom, Rotation und ähnliches.

Weitere optionale Angaben zum Druck können einer PrinterJava2 Instanz vorgegeben werden:

- Angabe des Seitenformats über getter-/setter Methoden oder als Benutzer-Eingabe (Dialog)
- Angabe der zu druckenden Seiten über getter-/setter Methoden oder als Benutzer-Eingabe (Dialog)
- ob Seiten optimal in den Druckbereich eingepasst werden sollen, z.B. durch Anpassen der Seitengröße oder ggf. durch automatisches Rotieren
- ob der Druckprozess synchron oder asynchron vollzogen werden soll

Darüber hinaus kann eine Implementierung der Schnittstelle PageDecorator[75] angegeben werden. Ein PageDecorator ist eine Schnittstelle zum Modifizieren von Druckergebnissen, z.B. zur Angabe der Seitennummer, der Dokumentenbezeichnung, Benutzerangaben etc. Dazu werden zwei Methoden bereitgestellt:

- **decoratePreRender(...)**

  Rendern der Zusatzinformation vor dem Rendern der Seite.

- **decoratePostRender(...)**

  Rendern der Zusatzinformation nach dem Rendern der Seite.

Der abschließende Aufruf der Methode „**print()**" führt schließlich zum Ausdruck des Dokuments.

Weiterführende Informationen können der API-Dokumentation entnommen werden.

4.25. PrintManager

Die Klasse PrintManager[76] dient der Vereinfachung von Druckprozessen. Integratoren, die einfach nur einen asynchronen Standarddruck initiieren oder ein Standard-Seitenformat vom Anwender bestimmen lassen möchten, können dies ohne aufwendige Konfigurationen von PrinterJava2 Instanzen durch eine der vielen statischen Methoden der Klasse PrintManger erledigen.

75 com.levigo.jadice.docs.printer.PageDecorator
76 com.levigo.jadice.docs.printer.PrintManager

Weiterführende Informationen zu dieser Klasse können der API-Dokumentation entnommen werden.

4.26. FileOpener

Eine interessante Utility Klasse des jadice Pakets ist die Klasse FileOpener[77]. Instanzen dieser Klasse ermöglichen auf sehr einfache Art Bilddaten und eventuell zugehörige Annotationen von einem Datei-System zu laden.

Eine Instanz des FileOpeners kann genutzt werden, um eine Bilddatei mit eventuell zugehörigen Annotationsinformationen zu laden. Falls kein zu ladendes Dokument angegeben wurde, wird der Benutzer durch Anzeigen eines Dateiauswahldialogs aufgefordert, ein Dokument auszuwählen. Bei bestätigter Auswahl wird ein entsprechender Ladevorgang initiiert. Alle weiteren Schritte des Ladevorgangs wie auch die Suche nach einer korrespondierenden Annotationsdatei übernimmt diese Klasse selbstständig.

4.27. DocumentSaver

Korrespondierend zur Klasse FileOpener steht Entwicklern die Klasse DocumentSaver[78] zur Verfügung. Mithilfe dieser Klasse können jadice Dokumente (siehe auch Abschnitt 4.2Document) im lokalen Dateisystem oder in einem frei wählbaren OutputStream gespeichert werden.

Der DocumentSaver unterstützt das Speichern von allen Formaten, die der Viewer anzeigen kann. Formatkonvertierungen oder Änderungen an Dokumenten werden jedoch nicht durch diese Klasse vorgenommen und unterstützt.

4.28. Demonstrationsklassen

In Ihrer jadice Auslieferung befinden sich verschiedene Demonstrationsklassen, die Ihnen erlauben, die jadice document platform ein wenig kennen zu lernen. Für Integratoren sind diese Klassen neben weiteren Demoklassen zusätzlich im Quelltext vorhanden.

Weiteres entnehmen Sie bitte der Auslieferungs-Dokumentation. Die folgenden Angaben sind als Ergänzung zu dieser Dokumentation zu verstehen.

4.28.1. Parameter der Demonstrationsklassen JadicePanel und JadiceMDI

Die im Default Package enthaltenen Beispiel-Applikationen **JadicePanel** und **JadiceMDI** verarbeiten ein „-open" Parameter, der es erlaubt, direkt beim Programmstart eine zu öffnende Bilddatei anzugeben.

- **-Open**

  Öffnet das im voll qualifiziert angegebenen Pfad stehende Dokument.

  Bsp.: *-open=c:|dokumente|test1.tif*

  Der Kommandozeilenaufruf lautet wie folgt:

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

  JadicePanel -open=c:|dokumente|test1.tif

  oder

77 com.levigo.jadice.util.FileOpener
78 com.levigo.jadice.util.DocumentSaver

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

JadicePanel -open=c:|dokumente|test1.tif

mit erhöhter Heap-Size

4.28.2. Parameter des Demo-Applets JadiceApplet

Das ebenfalls im Default Package enthaltene Beispiel-Applet JadiceApplet verarbeitet folgende Parameter:

- **LOADITEM**

  das beim Start zu öffnende Dokument

- **IOHANDLER**

  die Klasse des zu verwendenden IOHandlers, voll qualifizierter Klassenname der Implementierung, ohne Angabe wird der Standard IOHandler des jadice Pakets verwendet.

- **LOADERBASE**

  gibt den Ort, an dem die Dokumente zu finden sind und von dem aus sie geladen werden können. Ohne Angabe wird die Codebase des Applets verwendet.

- **RESBASE**

  gibt den Ort, an dem externe Ressourcen zu finden sind. Ohne Angabe wird die Applets Codebase verwendet.

- **jadice.viewer.tmps-path** (nur für Applets mit Schreib-/Lese-Recht)
  bestimmt das zu verwendende Temporär-Verzeichnis. Falls das Applet Schreibrechte in einem lokalen Verzeichnis hat, kann dieses genutzt werden, um gelesene Dokument-Daten zu cachen, vgl. 4.9.2FileCacheInputStream.
  Ohne Angabe werden keine temporären Dateien verwendet.

5. Typische Anwendungsbeispiele

Die folgenden Abschnitte verdeutlichen die Zusammenhänge zwischen den in Abschnitt 4.Klassenüberblick vorgestellten Klassen. Anhand von Anwendungsbeispielen wird dargestellt, wie jadice in eigene Applikationen integriert werden kann.

5.1. Viewer in einem Frame einbetten

Um den jadice viewer näher kennen zu lernen und als Basis für weitere Beispiele in diesem Kapitel, wird mit der Klasse TestViewerFrame ein Fenster erstellt, das einen Viewer enthält.

```
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 Beispiel 1 – Erstellen eines Viewer Fensters

Die einfachste Methode, einen jadice viewer in eine Komponentenhierachie einzubetten, ist die Nutzung der Klasse BasicJadicePanel. Sie kombiniert neben einem Viewer eine interagierende Statusbar, Toolbars und ein Kontext-Menü und bietet so eine komplette funktionale Viewer-Repräsentation.

Für interessierte Integratoren ist die Klasse BasiceJadicePanel als Source Code Beispiel in der Auslieferung unter

<Auslieferungsverzeichnis\example-src\viewer

zufinden.

Hilfreich, wie sich in folgenden Beispielen zeigen wird, ist die Möglichkeit auf die enthaltene Viewer Instanz zugreifen zu können. Aus diesem Grund wurde die Methode „**getViewer()**" hinzugefügt.

Führt man die Klasse aus, so erscheint folgendes Anwendungsfenster:

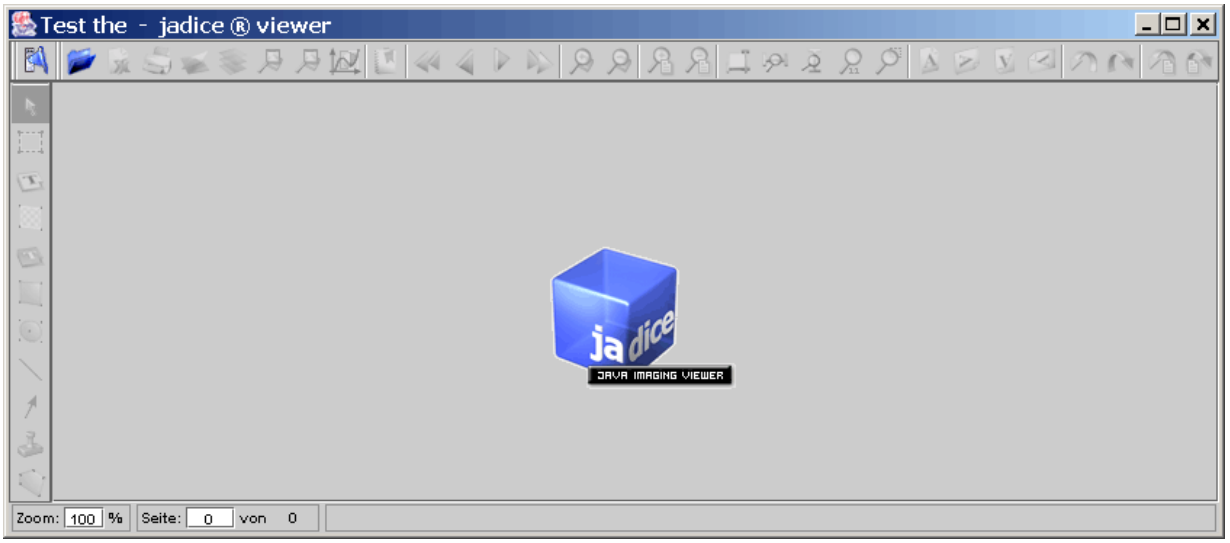

Abbildung 6 - Viewer Frame

Um mit dem Viewer arbeiten zu können, muss nun noch ein Dokument geladen werden. Dies kann nun auf zwei Wegen geschehen:

- Benutzer: durch Betätigung der Schaltfläche „öffnen" 
  Es erscheint eine Datei-Auswahl. Der Benutzer kann nun eine entsprechende Bild-Datei auswählen und sich im Viewer anzeigen lassen.
- programmatisch: durch Initiierung eines Ladevorganges. Dies wird im nächsten Abschnitt genauer erläutert.

Selbstverständlich kann auch statt einer Instanz des BasicJadicePanels analog zu obigem Beispiel eine Viewer Instanz direkt in eine Komponentenhierachie eingebunden werden. In einem solchen Fall muss der Integrator jedoch Toolbars, Statusbar, Menu- und Kontextstrukturen selbst erstellen und entsprechend einbinden. Anregungen, wie Toolbars oder Menüs programmatisch erstellt werden, kann der Sourcecode der Klasse BasicJadicePanel bieten, der in der Auslieferung im Verzeichnis example-src zu finden ist.

5.2. Ladevorgang

5.2.1. Einfacher Ladevorgang

Zur Erläuterung eines einfachen, programmtechnisch gesteuerten Ladevoganges von einem im Datei-System vorliegenden Bild-Dokument wird die Klasse Test ViewerFrame um die Methode „**loadAnImage(...)**" erweitert.

```
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 Beispiel 2 – Einfacher Ladevorgang

Zunächst wird eine Loader Instanz erstellt. Das Dokument, das später durch den Loader befüllt wird, wird direkt in den Viewer gesetzt. Der Viewer aktualisiert seine Ansicht automatisch, sobald die erste Seite geladen wurde.

Die einfachste Art ein Dokument vollständig zu laden bietet die Methode „**loadDocument(InputStream, firstPage)**" des Loaders. Die Parameter setzen sich aus einem Daten-InputStream und dem Index der ersten zu befüllenden Zielseite des Dokuments zusammen. Durch die Angabe von „0" als Index der Zielseite wird die Datei an den Anfang des Dokuments geladen. Die Indexierung von Seiten innerhalb des Loaders ist 0-basierend.

An dieser Stelle sei erwähnt, dass Bilddateien bei jadice eher in seltenen Fällen vom lokalen Dateisystem geladen werden. In der Regel erhält jadice Dokument Daten aus einem Dokumenten Management System. Da die „**loadDocument()**" Methode des Loaders jedoch nur einen InputStream erwartet, ist es freigestellt, woher dieser Stream seine Daten bezieht und der integrierenden Anwendung überlassen, einen solchen Datenstrom bereit zu stellen.

5.2.2. Dokumente zusammensetzen

jadice Dokumente müssen nicht nur aus Daten einer Quelle bestehen. Anhand des Dokumentenmodells (vgl. 2.4.1.Das Dokumentenmodell) wird deutlich, dass Dokumente sich aus unterschiedlichen Daten-Quellen zusammensetzen können. In diesem Abschnitt wird ein Beispiel aufgezeigt, wie man Seiten

verschiedenen Ursprungs zu einem virtuellen jadice Dokument zusammensetzt. Darüber hinaus können Seiten ihre Seitensegmente ebenfalls aus verschieden Datenquellen erhalten, dies wird im Abschnitt 5.2.3Layer näher beschrieben.

Zur Vereinfachung wurde in diesem Beispiel ein und dasselbe Single Page Tiff zehnmal aneinander gereiht. Nach abgeschlossenem Ladevorgang beinhaltet das Dokument somit 10 Seiten. Mit gleicher Art können aber auch unterschiedliche Bilddokumente, ein- und mehrseitige, verschiedenen Formats aneinander gefügt werden.

```
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();
     // Viewer ein leeres Dokument setzen
    viewerFrame.getViewer().setDocument(
       loader.getDocument());
    // Ladevorgang synchron
    loader.setSynchronousLoading(true);
    // LoadListener, hier lediglich zum Anzeigen wann
    // jeweils ein Rohdokument
    // geladen und dem jadice Dokument hinzugefügt wurde
    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 Beispiel 3 Dokumente zusammensetzen

Auch in diesem Beispiel wird die Klasse TestViewerFrame zur Anzeige des Viewers verwendet, nicht aber ihre „**loadAnImage(...)**" Methode. Die „**loadAnImage(...)**" Methode übergibt bei jedem Aufruf pro zu ladendem Rohdatenstrom der Viewer Instanz ein neues Dokument und ist damit für dieses Beispiel nicht zu gebrauchen.

Im Code Beispiel 3 wird zunächst ein TestViewerFrame erzeugt, sichtbar gemacht und ein multipler Ladevorgang gestartet. Damit der Ladevorgang den aktuellen Thread nicht blockiert, wird die Methode „**load10TiffsInRow()**" asynchron aufgerufen. Der asynchrone Start des Ladeprozesses ist nicht unbedingt notwendig, aber in bestimmten Zusammenhängen kann dieses Verfahren sehr nützlich sein, um einen flüssigen Programmablauf zu gewähren und ist hier zu Anschauungszwecken aufgenommen worden.

In der Methode „**load10TiffsInRow()**" wird zunächst eine Loader Instanz erstellt. Dann wird, wie auch im vorangegangenen Beispiel, dem eingebetteten Viewer das Dokument des Loaders zur Anzeige gesetzt. Hierbei findet die „**getViewer()**" Methode der Klasse TestViewerFrame ihren ersten Einsatz. Durch diese Methode wird das Test Frame flexibel von „außen" nutzbar, eine Eigenschaft, die auch in den folgenden Beispielen oft zur Anwendung kommt.

An dieser Stelle sei nochmals besonders darauf aufmerksam gemacht, dass eine Loader Instanz für beliebig viele Ladevorgänge genutzt werden kann. Diese Ladevorgänge befüllen das Dokument, das dem Loader gesetzt wurde. Ist kein Dokument gesetzt, erstellt der Loader eine neue Document Instanz eigenständig. Sollen verschiedene Dokumente mit einer Loader Instanz beladen werden, kann über die Methode „**setDocument(Document)**" das zu beladende Dokument verändert werden. Da der Loader, wenn nicht anders gesetzt, asynchron arbeitet, sollte dies jedoch nicht während eines laufenden Ladevorganges vollzogen werden. Änderungen an der zu befüllenden Dokument Instanz liegen in der Verantwortlichkeit des Integrators.

Zur Wahrung der korrekten Reihenfolge sollten die Ladevorgänge von verschiedenen Bildquellen synchronisiert sein, damit auch wirklich das nächste Bild-Dokument erst geladen wird, wenn das vorhergehende geladen und der zu beladenden Dokument Instanz hinzugefügt wurde. Zu diesem Zweck wird der Loader mit der Methode „**setSynchronousLoading(true)**" auf synchrone Verarbeitung gesetzt.

Nachfolgend wird ein LoadListener am Loader registriert, der in diesem Beispiel keine funktionale Bedeutung hat und vielmehr zu Anschauungszwecken hinzugefügt wurde. Jedes Mal, wenn eines der zehn zu ladenden Dokumente seinen Ladevorgang abgeschlossen hat, gibt der LoadListener eine entsprechende Meldung auf der Konsole aus. Beachten Sie bitte, dass LoadListener Instanzen nur einmal pro Ladevorgang registriert werden müssen. Ebenso können mehrere LoadListener parallel registriert sein.

Im nächsten Schritt wird der eigentliche Ladevorgang vollzogen. Da der Loader synchron arbeitet, können mittels einer „for"-Schleife die Ladevorgänge sequentiell gestartet werden. Als Parameter werden, wie auch schon im letzten Beispiel, ein Daten InputStream und ein Seitenindex übergeben.

Um sicherzustellen, dass ein neu geladenes Rohdokument am Ende des zu beladenen jadice Dokuments angefügt wird, wird als Seitenindex der „**loadDocument(...)**" jeweils die aktuelle Seitenzahl des Dokuments angegeben. Da die Seiten synchron durch den Loader dem Dokument hinzugefügt werden und die Seitenindexierung des Loaders 0-basierend ist, wird die gewünschte Reihenfolge der Seiten gewährleistet.

5.2.3. Layer

Im letzten Abschnitt wurden Seiten verschiedenen Ursprungs geladen und zu einem Dokument zusammengesetzt.

Dokumente können aber nicht nur aus verschiedenen Seiten aus unterschiedlichen Quellen bestehen, sondern auch Seiten selbst können sich aus verschiedenen Ebenen zusammensetzen. Vergleichen Sie dazu bitte auch 2.4.1.Das Dokumentenmodell.

Dieser Abschnitt beschreibt ein Beispiel wie ein Seitenhintergrund (hier ein Fax-Papier Vordruck) und ein textueller Inhalt in verschiedene Seitensegmente geladen werden können.

```
/**
 * 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 Beispiel 4 Laden eines Layers

Zunächst wird der Loader wieder in synchrone Verarbeitungsweise gesetzt. Dies ist technisch zum Beladen von Seitensegmenten nicht zwingend notwendig, stellt aber sicher, dass zunächst der Seitenhintergrund geladen und angezeigt wird, bevor der textuelle Inhalt eingefügt wird. Im Normalfall sollte der Ladevorgang so schnell ablaufen, dass der Benutzer dies nicht merkt, bei langsamen Netzwerkverbindungen beispielsweise ist diese Reihenfolge der Anzeige jedoch suggestiver.

Um in verschiedene Seitensegmente laden zu können, müssen zunächst diese Schichten im Dokument angelegt werden. Ein Layer existiert dann für alle Seiten des Dokuments, auch wenn diese möglicherweise nicht alle Layer mit Seitensegmenten belegen. Ein Layer bestimmt sich über einen eindeutigen Namen und wird an einer bestimmten vertikalen Position der Dokument-Ebenen eingefügt. Beispielsweise sollte ein AnnotationPageSegment stets auf oberster Ebene über allen Seitensegmenten liegen. In diesem Beispiel soll der Text über dem Fax Hintergrund dargestellt werden. Aus diesem Grund wird der Layer „background" an der Position DocumentLayer.BOTTOM und der Layer „overlay" an der Position DocumentLayer.ABOVE_BOTTOM angelegt.

Anschließend werden die Ladevorgänge initiiert. Dazu wird der „**loadDocument(...)**" Methode wie auch in den vorangehenden Beispielen ein Daten InputStream und ein Seitenindex übergeben. Zusätzlich wird aber auch der Layer bestimmt, in dem die geladenen Daten als Seitensegment in der Seite vertikal positioniert werden.

Bei Ausführung sieht die geladene Seite so aus:

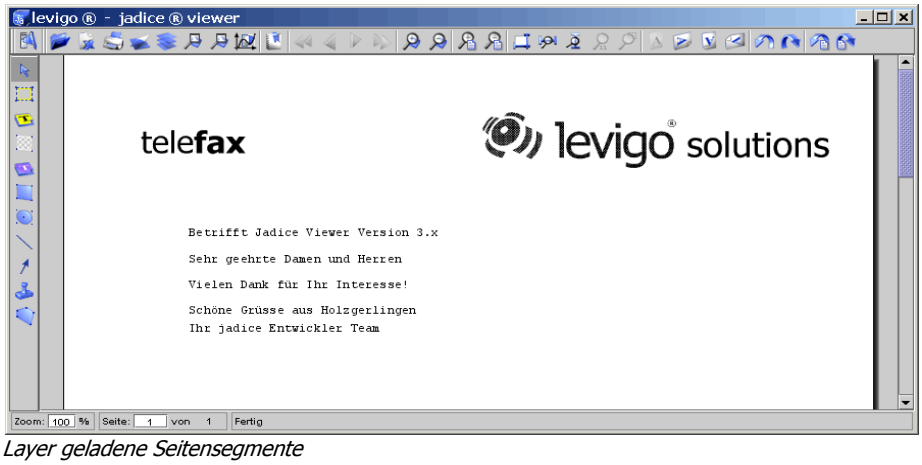

Abbildung 7 - In Layer geladene Seitensegmente

Kommentiert man den „**loadDocument()**" Aufruf für den „Overlay"-Layer des textuellen Fax-Inhalts aus, erscheint, wie zu erwarten war, nur der Seitenhintergrund.

Sehen Sie dazu die folgende Abbildung.

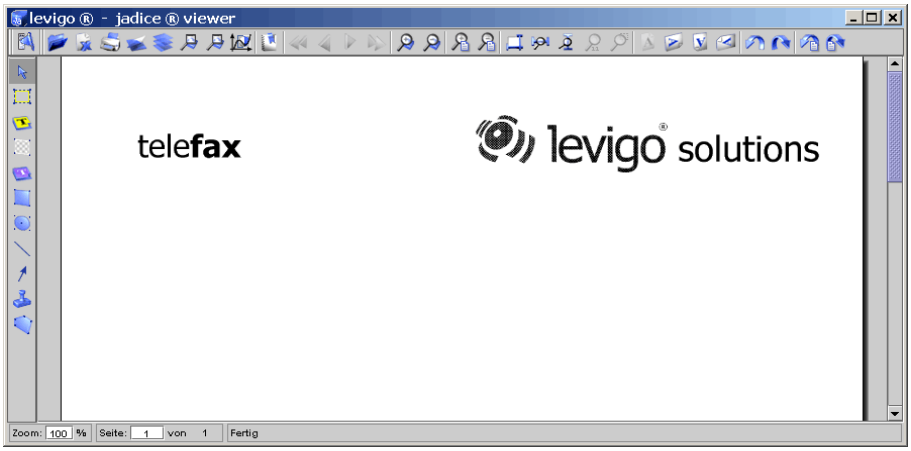

Abbildung 8 - Nur Hintergrund Ebene geladen

5.2.4. SeekableInputStream

Zur effizienten und speicherschonenden Verarbeitung von großen Dokumenten versucht jadice, sofern es das Bildformat erlaubt, nur die für den aktuellen Seitenausschnitt benötigten Dokumentdaten zu lesen, zu verarbeiten und dynamisch zu cachen, statt alle Bild-Daten vollständig im Speicher zu halten.

Für dieses Vorgehen verwendet jadice SeekableStreams. Anhand von Beispiel 5 wird verdeutlicht, wie versucht wird, einen passenden SeekableInputStream zu finden.

```
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 Beispiel 5 Auswahl eines geeigneten Datenstromes

Die Methode „**load(File)**" soll eine gegebene Datei laden und als Dokument zurückgeben.

Zunächst wird versucht einen RandomAccessFileInputStream zu erstellen, der ein Positionieren innerhalb der Datei ermöglicht. Dieser Datenstromtyp arbeitet auf einer in einem Dateisystem physikalisch vorhandenen Dokumentquelle, in der ein Lesezeiger direkt in der Datenquelle positioniert wird.

Falls dies scheitert, wird versucht einen FileCacheInputStream zu benutzen. Ein FileCacheInputStream puffert gelesene Daten in einer temporären Datei. Damit müssen langsame Lesevorgänge (z.B. bedingt durch eine schlechte Netzwerkverbindung) nur einmalig und bei Bedarf ausgeführt werden. Bereits gelesene und als Temporär-Datei gepufferte Daten sind jedoch schnell wieder zur Verfügung. Zur Erstellung eines FileCacheInputStreams muss die Anwendung zumindest in dem gesetzten Temporär-Verzeichnis Schreibrechte haben.

Als letzte Alternative verbleibt ein MemoryInputStream, der gelesene Daten im Hauptspeicher puffert. Diese Variante ist die schnellste, da Datenzugriffe ausschließlich im Hauptspeicher erfolgen, sie kann aber bei sehr großen Datenmengen den Hauptspeicherbedarf der Applikation erheblich vergrößern.

Der Integration ist es überlassen nach Einsatzumgebung und anwendungsspezifischen Randbedingungen den pragmatischsten Typ von SeekableInputStreams auszuwählen und für Ladevorgänge zu verwenden. Falls jadice die Auswahl des günstigsten SeekableInputStream überlassen werden soll, bietet jadice dazu zwei Möglichkeiten als statische Methoden der Klasse Loader an:

- Loader.prepareSeekableInputStream(InputStream)

  Erzeugt aus dem gegebenen Inputsteam einen passenden Seekablestream und gibt diesen als Methodenrückgabewert zurück.

- Loader.prepareSeekableInputStream(InputStream, boolean)

  Erzeugt aus dem gegebenen Inputsteam einen passenden Seekablestream und gibt diesen als Methodenrückgabewert zurück. Der boolsche Parameter legt fest, ob bei der Ermittlung des günstigsten SeekableInputStreams Temporär-Datei Pufferung berücksichtigt werden soll oder nicht.

5.2.5. ResourceLoader

ResourceLoader finden für AFP oder MO:DCA Dokumente Verwendung, bei denen im Dokument nicht nur interne (inline), sondern auch externe Ressourcen eingebunden sein können. Externe Ressourcen werden während des Ladevorgangs des Dokuments mithilfe von ResourceLoadern dynamisch nachgeladen.

In diesem Beispiel gehen wir davon aus, dass im Verzeichnis „C:\afp\res" verschiedene Resources zu finden sind und darüber hinaus auf „www.myServer.com" ein Ressourcen-Verzeichnis „myResources" mit benötigten Ressourcen hinterlegt ist. Alle diese Ressourcen sollen stets für alle folgenden Ladevorgänge zur Verfügung stehen.

```
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 Beispiel 6 Ein ResourceLoader aus verschiedenen ResourceLoadern

Um für alle Ladevorgänge zur Verfügung zu stehen, muss der zu erstellende ResourceLoader am Loader direkt registriert werden. Würde der ResourceLoader am Dokument angemeldet, würde dieser nur für den Ladevorgang des Dokuments bereitgestellt, nicht aber für Ladevorgänge in andere Dokumente.

Die Anmeldung am Loader kann dann mit der „**setResourceLoader(...)**" Methode vollzogen werden.

Beispiel:

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

Am Loader kann jeweils immer nur ein ResourceLoader angemeldet sein. In diesem Beispiel werden jedoch zwei ResourceLoader benötigt: Einen ResourceFileLoader, der alle Ressourcen des Verzeichnisses „C:\afp\res" zur Verfügung stellt und einen Zweiten, der alle Ressourcen auf „www.myServer.com" im Ressourcen-Verzeichnis „myResources" bereitstellt. Die Lösung ist eine Instanz der Klasse ResourceMultiLoader, die das Interface ResourceLoader implementiert und damit als ResourceLoader arbeiten kann, aber auch verschiedene ResourceLoader über die Methode „**addLoader()**" in sich aufnehmen kann. Kommt nun eine Anfrage nach einer Ressource, so werden alle registrierten ResourceLoader im ResourceMultiLoader nach dieser Ressource befragt und das erste erfolgreiche Ergebnis zurückgegeben.

In Code Beispiel 6 wird zunächst ein ResourceFileLoader erstellt. Dann wird ein ResourceUrlLoader erzeugt. Dieser kann mittels einer Liste von einer oder mehreren Ressourcen beinhaltenden URLs direkt instantiiert werden.

Nachfolgend werden der ResourceFileLoader wie auch der ResourceUrlLoader einer Instanz der Klasse ResourceMultiLoader hinzugefügt.

5.2.6. Annotations

Im folgenden Abschnitt wird anhand eines Beispiels aufgezeigt, wie ImagePlus kompatible Annotationen geladen werden können. FileNet und FileNet P8 Annotationen können analog geladen werden. Es muss lediglich statt einer Instanz der Klasse ImagePlusAnnotationFormatInfo der Methode „**loadDocument(...)**" des Loaders eine Instanz der Klasse FileNetAnnotationFormatInfo bzw. FileNetP8AnnotationFormatInfo übergeben werden. Zur Übersicht wird an dieser Stelle nicht näher auf den Ladevorgang des eigentlichen Bilddokuments eingegangen, sondern ausschließlich auf die Annotationen.

```
File file2Load = new File("MeinBild.tif");
 Loader loader = new Loader();
// Dokument laden...
```

```
      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 Beispiel 7 – Annotationen laden

Zunächst wird versucht, zu der gegebenen Bilddatei „file2Load" eine korrespondierende Annotationsdatei zu finden. ImagePlus kompatible Annotationsdateien führen üblicherweise den gleichen Namen wie das Bilddokument und haben als Suffix „.T_L". Existiert eine solche Datei, wird diese zum Laden der Annotationen verwendet. An dieser Stelle sei darauf hingewiesen, dass Annotationsdaten in der Regel nicht als Datei, sondern als Stream aus einem Archiv oder ähnlichem vorliegen.

Formatinformationen beschreiben das Format, in dem ein Dokument vorliegt. Gibt man dem Loader in der „**loadDocument**" Methode keine Formatinformation an, versucht der Loader das Format selbst zu bestimmen. Da ImagePlus Annotationen MO:DCA Strukturen sind, ist es notwendig, ImagePlus kompatible Annotationen immer mit ImagePlusAnnotationFormatInfo[79] zu laden. Anderenfalls könnten diese mit MO:DCA Daten verwechselt und dementsprechend als Dokument statt als Annotationen geladen werden.

Hinweis:

Das Laden und Speichern von Annotationen muss immer explizit vom Integrator durchgeführt werden und wird **nicht** automatisch von jadice beim Laden des Hauptdokuments erledigt.

5.2.7. Bookmarks

Bookmarks werden über die Klasse DocumentBookmarkHandler gespeichert, geladen und verwaltet.

Der DocumentBookmarkHandler erlaubt das Laden und Speichern von Bookmarks als Properties[80].

Beispiel 8 beschreibt einen Ladevorgang aus einer Properties Datei.

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

79 com.levigo.jadice.formats.annoiplus.ImagePlusAnnotationFormatInfo
80 java.util.Properties

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

Code Beispiel 8 – Bookmarks laden

Zunächst wird überprüft, ob die angegebene Bookmark Properties Datei vorhanden ist und lesend darauf zugegriffen werden kann. Wenn ja, wird ein leeres Properties Objekt erzeugt, in das die Bookmark-Daten im nächsten Schritt geladen werden. Das gefüllte Properties Objekt wird dann dem DocumentBookmarkHandler zur Verfügung gestellt. Der DocumentBookmarkHandler erstellt aus den gegebenen Properties JadiceBookmarks, die dann zur weiteren Nutzung über den DokumentBookmarkHandler zur Verfügung stehen.

Properties Objekte können Bookmark Einträge von beliebig vielen verschiedenen Dokumenten beinhalten. Zur Sicherstellung, dass nur die zum Dokument gehörigen Bookmarks geladen werden, muss zusätzlich eine eindeutige Bookmark Kennung der Lademethode des DocumentBookmarkHandler angegeben werden. Hier im Beispiel ist der Einfachheit halber der Name des Dokuments verwendet worden. Grundsätzlich kann jedoch jede beliebige andere Zeichenkette als Kennung verwendet werden, wichtig ist nur, das es die gleiche ist wie jene, die beim Abspeichern der Bookmarks verwendet wurde.

Mit Hilfe dieser Identifizierung können Bookmarks aus verschiedenen Speichervorgängen und von verschiedenen Dokumenten in einem Properties-Objekt verwaltet werden.

5.2.8. Gradation

Gradationsdaten werden von zwei unterschiedlichen Objekttypen des jadice Pakets verarbeitet: Zum einen von Instanzen der Klasse GradationCurve und zum anderen als AddOn in Form einer GradationCurveControl Instanz.

Für beide Arten von Gradationsdaten verarbeitenden Objekten unterstützt die Utility Klasse GradationCurveFileHandler den Ladevorgang aus einem Dateisystem. Instantiiert wird diese Klasse mit einer GradationCurve oder einer GradationCurveControl, die mit Gradationspunkten befüllt werden soll.

Die Klasse GradationCurveFileHandler bietet dazu verschiedene Lademethoden an, die je nach Bedarf genutzt werden können. Eine detaillierte Auflistung ist unter 4.23.2.GradationCurveFileHandler zu finden. Im folgenden Beispiel wurde die Methode „**openGradationCurveFromFile**" verwendet. Diese Methode öffnet zunächst einen Dateiauswahldialog, der nach Auswahl der Gradationsdaten-Datei einen Ladevorgang in das übergebene GradationCurve oder GradationCurveControl Objekt auslöst.

Alternativ ist auch ein Laden der Gradationsdaten aus einem gegebenen Datenstrom möglich. Vergleichen Sie dazu die Methode „**loadGradationCurveFromStream**" aus Beispiel 9.

```
public void loadGradationCurve(GradationCurve aCurve){
 new GradationCurveFileHandler(aCurve)
   .openGradationCurveFromFile();
}
public void loadGradationCurveIntoGradPanel(
        GradationCurveControl aCurveControl){
 new GradationCurveFileHandler(aCurveControl)
     .openGradationCurveFromFile();
}

public void loadGradationCurveFromStream(
  GradationCurveaCurve, InputStream is){
  new GradationCurveFileHandler(aCurve)
   .openGradationCurveFromStream(is);
}
```

Code Beispiel 9 – Gradation aus lokalem Dateisystem oder InputStream laden

Liegen die Gradationsdaten nicht im lokalen Dateisystem bereit, bietet die Klasse GradationCurve zusätzlich die Möglichkeit Daten über ein Properties Objekt einzufügen.

```
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 Beispiel 10– Gradation aus einem Datenstrom laden

Beispiel 10 zeigt den Ladevorgang in eine Instanz der Klasse GradationCurve. Dazu erhält die Methode „**loadGradationData(...)**" einen entsprechenden Daten-InputStream und einen Gradationsbezeichner. Dieser Bezeichner identifiziert eindeutig die zu ladenden Gradationsdaten und sollte dem verwendeten Bezeichner des vorangegangenen Speichervorgangs der Gradationsdaten entsprechen. Als Ergebnis gibt die Methode eine neu erzeugte und mit den Daten des InputStreams initialisierte GradationCurve zurück.

Zunächst wird ein Properties-Objekt erzeugt, das die Daten des gegebenen Datenstroms lädt. Dann wird die GradationCurve mit diesen Properties und der übergebenen Gradationskennung initialisiert, bevor es als Methoden Ergebnis zurückgegeben wird.

5.3. Speichern

Dieser Abschnitt befasst sich im Schwerpunkt mit Beispielen, die detailliert auf das Speichern in ein bestimmtes Format bzw. konkrete Teile eines Dokuments, wie Annotationen, eingehen. Im Standardfall, sprich beim Speichern eines einfachen nicht zusammengesetzten Dokuments mit eventuellen Annotationen, sollte die Funktionalität der Klasse 4.27.DocumentSaver für Integratoren und Entwickler ausreichend sein. Der DocumentSaver unterstützt das Speichern von allen Formaten, die der Viewer anzeigen kann. Eine Formatkonvertierung und Änderungen an Dokumenten werden nicht unterstützt.

5.3.1. Dokument

Ähnlich wie Formatinformationen Ladevorgänge von bestimmten Formaten unterstützen, gibt es Klassen, die Speichervorgänge in bestimmte Formate (*FormatName*File) unterstützen. Mehr dazu auch in 4.6.FormatInfo und FormatFile.

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

Code Beispiel 11 Tiff Dokument speichern

Im Beispiel soll ein Dokument, das aus TIFF-Daten erstellt wurde, in eine Datei „MeinBild.tif" gespeichert werden.

Dazu wird zunächst eine Instanz der Klasse TIFFFile erstellt und dem Konstruktor das zu speichernde Dokument übergeben.

Dann werden mittels der Methode „**save(OutputStream)**" die TIFF-Daten des Dokuments gespeichert. Sollte das Zielformat und das Format des Ursprungsdokuments unterschiedlich sein, tritt ein Fehler auf. Formatkonvertierungen werden nicht durchgeführt.

Ebenso wie das Laden obliegt auch das Speichern der Verantwortlichkeit der integrierenden Anwendung. Zusammengesetzte Dokumente aus verschiedenen Formaten müssen entsprechend ihrer Zusammensetzung abgespeichert werden. Dazu stellen alle *FormatName*File Klassen verschiedene „**save(...)**" Methoden zur Verfügung, die ein Abspeichern von bestimmten Seiten und/oder Layer erlauben.

5.3.2. Annotations

Wie auch im vorigen Abschnitt wird zum Speichern von Annotationen eine bestimmte formatspezifische Speicher-Klasse speziell für Annotationen verwendet.

```
private void saveAnnotations(){
   // similiar 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 Beispiel 12- Annotationen speichern

Das Beispiel beschreibt, wie die Annotationen eines Dokuments in eine Datei abgelegt werden können.

Zunächst wird eine Instanz dieser Klasse erzeugt. Um einen Zugriff auf die zu speichernden Annotationen zu erhalten, wird dem ImagePlusAnnotationFile im Konstruktor das Dokument übergeben, dessen Annotationen zu speichern sind. Abschließend speichert ein Aufruf der save Methode die Annotationen in den übergebenen OutputStream.

Hinweis:

Das Laden und Speichern von Annotationen muss explizit durchgeführt werden und wird **nicht** etwa automatisch beim Laden des Dokuments erledigt.

Zur formatspezifischen Speicherung von Annotationsdaten können folgende Klassen verwendet werden:

- ImagePlusAnnotationFile (für IBM ImagePlus- und IBM ContentManager kompatible Annotationen)
- FileNetAnnotationFile (für FileNet Annotationen)
- FileNetP8AnnotationFile (für FileNet P8 Annotationen)

5.3.3. Bookmarks

Wie bereits im Abschnitt 5.2.7.Bookmarks vorgestellt, wird die Persistenz von Bookmarks über Properties Objekte und einer eindeutigen Kennung zur Verfügung gestellt.

```
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 Beispiel 13- Bookmarks speichern

Im Code Beispiel wird zunächst eine Referenz auf den DocumentBookmarkHandler über die statische Methode „**getInstance()**" ermittelt.

Im nächsten Schritt wird das Speichern der Bookmarks durch einen Aufruf der Methode **saveBookmarksForDocument** ausgelöst. Zum Aufruf erwartet diese Methode folgende Parameter:

- ③ Eine Referenz des Dokuments, dessen Bookmarks gespeichert werden sollen.
- ③ Ein Properties Objekt, in dem die Bookmarks abgelegt werden sollen.
- ③ Eine eindeutige Bookmark-Kennung, mithilfe deren zu einem späteren Zeitpunkt Bookmarks wieder aus einem Properties Objekt einem Dokument zugehörig ausgelesen werden können.

5.3.4. Gradation

Im folgenden Beispiel sollen die im Viewer gesetzten Gradations-Daten mit einer vorgegebenen Kennung in einem gegebenen OutputStream gespeichert werden. Die Kennung dient zur Identifizierung bei einem erneuten Ladevorgang.

```
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 Beispiel 14- Gradation speichern

Die vom Viewer verwendete Gradationskurve ist im RenderContext des Viewers abgelegt, so dass mithilfe der Methoden

- Viewer#getRenderContext() und
- RenderContext#getImageSettings() und
- ImageRenderSettings#getGradationCurve()

eine Referenz auf die zu speichernde Gradationskurve erhalten werden kann.

Im nächsten Schritt wird ein Properties-Objekt erzeugt, das der Gradationskurve mit einer eindeutigen Gradationskennung zur Aufnahme der Gradationsdaten übergeben wird.

Nachdem nun die Gradationsdaten in dem Properties-Objekt abgelegt wurden, speichert sich die Properties-Instanz in den übergebenen OutputStream. Der zweite Parameter der Methode „**store(...)**" dient als Daten-Header, der den gespeicherten Gradationsproperties als Kommentar vorangestellt ist. Die Angabe eines Headers ist optional und kann, falls kein Header erwünscht, mit „null" angeben werden.

5.4. Actions-Commands-Context

Ein zentraler Aspekt der jadice document platform ist es, sehr einfache Integrationsmöglichkeiten mit möglichst geringem Programmier- und Anpassungsaufwand anzubieten. Dieses Kapitel zeigt, wie einfach mittels der jadice Integrator API jadice Komponenten in eigene Anwendungen integriert und angepasst werden können.

Zum besseren Verständnis der folgenden Abschnitte beachten Sie bitte auch die Kapitel 3.Die jadice Integrator API und 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien.

5.4.1. Einbinden von Menüs, Toolbars, Actions

Als einführendes Beispiel wird in diesem Abschnitt zunächst ein Fenster erstellt, das eine Viewer Instanz, eine korrespondierende Toolbar und eine Menübar beinhaltet. Menübar und Toolbar werden unter Zuhilfenahme der jadice Integrator API erzeugt.

```
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 Beispiel 15- Beispiel Fenster

Die Methoden „**initContext()**", „**getCommandsToolBar()**" und „**getCommandsMenuBar()**" werden in den folgenden Abschnitten beschrieben.

5.4.1.1. Context

Zur Erstellung von CommandActions, unerheblich ob diese der ausführbare Teil eines Menüpunkts oder eines Buttons sind, wird ein Context Objekt benötigt. Ein Kontext spiegelt über die enthaltenen Objekte den aktuellen Zustand der assoziierten GUI-Komponente wider und stellt damit die Basis für Zustand und Ausführbarkeit von CommandActions dar. Bitte beachten Sie dazu auch Kapitel 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 Beispiel 16- Context erstellen

Dementsprechend wird zunächst in der Methode „**initContext()**" ein Context Objekt erstellt, dessen Konstruktor zwei Parameter übergeben werden, ein Context Owner und ein Aggregationsmodus.

Im folgenden werden die Parameter näher erläutert:

† die RootPane des Fensters als Context Owner

Jede Context Instanz hat ein assoziiertes GUI Element als Context Owner. Ein Context Owner dient zwei Aspekten.

Zum Einen steuert er den Zustand des Context Objekts. Ist der Context Owner als GUI Element aktiv, ist auch der Context aktiv. In Context Hierarchien setzen sich die Context Elemente entsprechend dem gesetzten Aggregationsmodus aus den Kontext eigenen Elementen und den Elementen keines, aller oder nur der aktiven Kind-Kontexte zusammen. Damit bestimmt die Aktivität von Kontexten in Abhängigkeit von dem gesetzten Aggregationsmodus die konkrete Zusammensetzung von Context Elementen und beeinflusst somit bei Kontext Veränderungen den Zustand (ausführbar, nicht ausführbar, ausgeführt, nicht ausgeführt) von assoziierten CommandActions.

Zum Anderen repräsentiert der Context Owner als GUI Element eine beinhaltete Komponentenhierachie. Context Hierachien sollten korreliert zu den Komponentenhierachien ihrer Context Owner zusammengesetzt werden.

- Context.NO_CHILDREN als Context Aggregationmodus

  Der Aggragationsmodus bestimmt die Zusammensetzung von Context Elementen in Context Hierarchien. Dazu gibt es drei Modi, die als statische Konstanten der Klasse Context angeboten werden:

  - Context.ALL_CHILDREN – Der Context enthält alle eigenen Elemente wie auch alle Elemente aller Kind-Contexte.
  - Context.ACTIVE_CHILD – Bei diesem Modus setzen sich die Context Elemente aus den Elementen des Context Objekts wie aus den Elementen der aktiven Kind-Kontexte zusammen.
  - Context.NO_CHILDREN – dieser Modus ist geeignet, falls keine Context Hierarchie vorhanden ist oder alle Context Objekte und die zugehörigen CommandActions absolut unabhängig von anderen Context Objekten und deren Elementen agieren.

Siehe dazu auch 3.2.3.Context und die jadice API Dokumentation.

Im nächsten Schritt werden die Objekte, die von den CommandActions benötigt werden, dem Context hinzugefügt. Alle Commands in dem Beipiel benötigen lediglich den Viewer, somit wird auch nur eine Viewer Instanz in den Context eingefügt. Informationen, welche Context Objekte von welchen jadice Commands zur Ausführung erwartet werden, finden sich in der jadice API Dokumentation. Mit den folgenden Methoden könnten an dieser Stelle weitere Context Objekte erstellt und in einer Hierarchie platziert werden.

- Context#addToParentsContext()
- Context#addChildContext(Context)
- Context#removeFromParentsContext()
- Context#removeChildContext(Context)

Für dieses einfache Beispiel wird jedoch keine Context Hierarchie benötigt.

Um Änderungen innerhalb des Viewers, z.B. dass eine Seite eines Dokuments geladen wurde und angezeigt wird, zu übersetzen in Aktualisierungsereignisse der CommandActions, wird dem Viewer ein PropertyChangeListener hinzugefügt, der PropertyChangeEvents in ContextChangedEvent[81] umwandelt. Dieser Mechanismus ist eine gängige Methode, um sicher zu stellen, dass Viewer-spezifische Commands stets in ihrem Zustand mit dem Zustand des Viewers harmonisieren.

5.4.1.2. Einbinden

Nachdem im letzten Abschnitt ein Context Objekt erstellt wurde, wird in der Methode „**initGUI()**" die Benutzungsoberfläche des CommandsTestFrames zusammengesetzt.

```
private JToolBar getCommandsToolBar() {
```

81 com.levigo.swing.action.ContextChangedEvent

```
 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 Beispiel 17- Referenzen auf Menus und Toolbars

Toolbar- und Menü-Strukturen werden über die Konfiguration *menucomponents.properties* bestimmt. Detailliertere Informationen dazu entnehmen Sie bitte den Abschnitten 3.3.Die Konfigurationsdateien und 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien.

Eine jadice Toolbar und ein File Menü wurden in dieser Datei definiert und werden in diesem Beispiel zur Erstellung referenziert.

Eine Referenz auf eine definierte Struktur erstellt die Klasse DefaultMenuComponentFactory[82]. Die zu einer Konfiguration korrespondierende Instanz erhält man über die Methode „**getInstance(ConfigurationName)**", diese wiederum erstellt je nach Methoden Aufruf und Definition die gewünschte Struktur. Dazu werden die folgenden Methoden angeboten:

- **getMenu(MenuName, Context)**

  erstellt ein unter dem Namen „MenuName" definiertes Menü, der zweite Parameter ist der zur Erstellung der CommandActions benötigte Kontext.

- **getContextMenu(MenuName, Context)**

  erstellt ein unter dem Namen „MenuName" definiertes Kontext-Menü, der zweite Parameter ist der zur Erstellung der CommandActions benötigte Kontext.

- **getToolbar(ToolbarName, Context)**

  erstellt ein unter dem Namen „ToolbarName" definiertes Menü, der zweite Parameter ist der zur Erstellung der CommandActions benötigte Kontext.

Zur Nutzung einer anderen Konfiguration übergibt man der Methode „**getInstance(...)**" einfach den gewünschten Namen.

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

Bitte beachten Sie, Strukturen können qualifiziert oder unqualifiziert definiert werden. Bei unqualifizierter Definition können daraus beliebige Strukturen erstellt werden, z.B. Menü oder Toolbar. Andernfalls kann nur die Struktur erstellt werden, die in der Konfiguration angegeben wurde. Vergleichen Sie dazu bitte 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien.

Wenn keine Struktur, sondern lediglich eine Referenz auf eine bestimmte CommandAction gewünscht ist, kann dies über folgenden Aufruf ermöglicht werden.

Beispiel:

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

Als Nachfahre von AbstractAction[83] kann jede CommandAction als ausführbares Element an geeignete Komponenten, z.B. an einen Button oder einen Menüpunkt, gebunden werden.

5.4.2. Anpassen der Actions

5.4.2.1. Eigenschaften

Unter bestimmten Umständen kann es gewünscht sein, die Eigenschaften einer CommandAction zu verändern, als Beispiel eine Änderung des Tooltips oder des Menü-Textes.

Teil des Datei-Menüs ist ein Command namens „OpenDocument", das lokale Bilddokumente öffnet und in den Viewer lädt. Möchte man beispielsweise, dass der Menü Eintrag „Datei öffnen" statt nur „Öffnen" lautet, ändert man die *actions.properties* bzw. die lokalisierte Variante *actions_de.properties* Konfiguration folgendermaßen.

Beispiel:

```
#actions.properties
...
OpenDocument.ShortDescription = Öffnen
...
# ... change to
OpenDocument.ShortDescription = Datei öffnen
...
```

Eine genaue Beschreibung, welche Eigenschaften veränderlich sind und in welcher Art sie angepasst werden können, findet sich unter Abschnitt 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien.

Hinweis:

Die Konfigurationsangaben werden beim Start des Viewers einmalig gelesen und ausgewertet. Änderungen an Konfigurationsdateien werden also erst nach einem Neustart des Viewers wirksam.

Hinweis:

83 javax.swing.AbstractAction

Bitte beachten Sie, dass die Konfigurations-Dateien in internationalisierten Varianten vorliegen. Um Inkonsistenzen zu vermeiden, sollten Änderungen der Konfiguration stets in allen Varianten getätigt werden.

5.4.2.2. Anpassen der Menü- oder Toolbar-Struktur

Zur Definition eigener Menu- oder Toolbar-Strukturen oder zur Veränderung bestehender Strukturen muss die Konfiguration *menucomponents.properties* bzw. die lokalisierte Variante *menucomponents_de.properties* angepasst werden.

Ein Beispiel: Das Kontext-Menü des Viewers kann so eingeschränkt werden, dass es nur noch die Produktinformation enthält.

Dazu wird die Definition des Kontext-Menüs so gekürzt, dass es nur noch die CommandAction zur Anzeige der Produktinformation enthält.

Beispiel:

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

Eine genaue Beschreibung, welche Eigenschaften veränderlich sind und in welcher Art sie angepasst werden können, findet sich unter Abschnitt 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien.

Hinweis:

Die Konfigurationsangaben werden beim Start des Viewers einmalig gelesen und ausgewertet. Änderungen an Konfigurationsdateien werden also erst nach einem Neustart des Viewers wirksam.

Hinweis:

Bitte beachten Sie, dass die Konfigurations-Dateien in internationalisierten Varianten vorliegen. Um Inkonsistenzen zu vermeiden, sollten Änderungen der Konfiguration stets in allen Varianten getätigt werden.

5.4.3. Eigene Commands

Falls die Funktionalität der mitgelieferten Commands nicht die Bedürfnisse der integrierenden Anwendung abdeckt, können eigene Commands einfach und mit wenig Aufwand eingebunden werden.

Als Beispiel wird in diesem Kapitel ein Command erstellt, das ein lokales Bilddokument öffnet und in den Viewer lädt. In einem weiteren Schritt wird dieses Command in das Datei-Menü des CommandsTestFrame eingebunden.

```
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 Beispiel 18- Einen Command erstellen

Die abstrakte Basisklasse aller Commands ist die Klasse AbstractCommand[84]. Sie fordert zur Erweiterung die Realisierung von drei Methoden:

- **doExecute(Collection)**

  Diese Methode wird aufgerufen, um das Command auszuführen. Der Parameter enthält die zur Verfügung stehenden Kontext Elemente.

- **checkQuickly(Collection)**

  Diese Methode wird aufgerufen, wenn der Kontext sich verändert hat. Der Rückgabewert bestimmt den Zustand (aktiv, nicht aktiv) des Commands. Da diese Methode sehr oft aufgerufen wird um einen stets korrekten Status (enabled, disabled) zu haben, sollten innerhalb dieser Methode keine aufwendigen Prüfungen vollzogen werden.

- **checkDeeply(Collection)**

  Diese Methode wird ausschließlich aufgerufen, bevor das Command ausgeführt wird. Hier können auch aufwendigere Überprüfungen

84 com.levigo.util.swing.action.AbstractCommand

erfolgen. Der Rückgabewert bestimmt, ob das Command letztlich zur Ausführung kommt oder nicht.

Zur Ausführung des ADocumentOpener Commands ist eine Viewer Instanz, in dem das Dokument geladen werden soll, als Kontext Element notwendig. Die Methoden „**checkQuickly(...)**" und „**checkDeeply(...)**" nutzen zur Überprüfung dieser Voraussetzung die Methode „**isArgumentValid(...)**".

Diese und weitere nützliche Hilfsmethoden der Klasse AbstractCommand sind im Folgenden beschrieben:

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

  Prüft, ob Objekte der angegebenen Klasse entsprechend der CountCondition in den angegebenen Kontext Objekten vorhanden sind. Die verschiedenen CountConditions sind beschrieben in der jadice API Dokumentation.

- **getClassFromArguments(Collection,Class)**

  Ermittelt ein Objekt der angegebenen Klasse aus den gegebenen Kontext Objekten.

- **getClassesFromArguments(Collection,Class)**

  Ermittelt alle Objekte der angegebenen Klasse aus den gegebenen Kontext Objekten.

Die Methode „**doExecute(...)**" ermittelt zunächst die Viewer Instanz, in die ein Dokument geladen werden soll. Im nächsten Schritt nutzt sie die Klasse FileOpener[85], um eine Bilddatei auszuwählen und in den Viewer laden zu lassen. Nähere Informationen zu der Klasse FileOpener sind in Kapitel 4.26.FileOpener oder in der jadice API Dokumentation.

Nach Fertigstellung des ADocumentOpener Commands muss es zur Einbindung in den Konfigurationsdateien registriert werden. Dazu wird in den *commands.properties* ein Mapping definiert zwischen einem eindeutigen Command Namen, z.B. ADocOpener, und der Verwirklichung (my.tests.commands.ADocumentOpener).

Beispiel:

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

Achten Sie bitte auf die korrekte Angabe des Klassennamens, da Commands innerhalb der jadice Integrator API über Reflektion instantiiert werden.

Im nächsten Schritt wird eine CommandAction definiert, die später in dem Datei-Menü des CommandsTestFrames eingebunden werden soll. Dazu erweitert man die Konfiguration *actions.properties* um folgende Zeilen.

Beispiel:

```
#actions.properties
...
MyOpener.commands = ADocOpener
```

85 com.levigo.jadice.util.FileOpener

```
MyOpener.ShortDescription = Mein Opener
MyOpener.LongDescription = Mein toller Opener
MyOpener.SmallIcon = defaulticons.TB_OPEN
...
```

In der ersten Zeile wird angegeben, welche Commands bei Ausführung der CommandAction MyOpener ausgelöst werden sollen. Dabei wird der in der Datei *commands.properties* definierte Name verwendet. Die nächste Zeile gibt den Text an, der z.B. im Menü oder auf einem Button erscheinen würde. LongDescription entspricht dem Tooltip-Text und SmallIcon definiert das zu verwendende Icon. Näheres zu möglichen Angaben und Syntax findet sich unter 8.jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien.

Um abschließend MyOpener als Teil des Datei-Menüs zu definieren, wird die Konfiguration *menucomponents.properties* folgendermaßen angepasst:

Beispiel:

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

Die erste Zeile besagt, dass das Datei-Menü aus MyOpener und der CommandAction CloseDocument besteht. Die zweite Zeile wurde nicht verändert und gibt den Namen des Menüs an.

Zusammenfassend kann man sagen, dass die Registrierung eines neuen Commands in den Konfigurationsdateien in drei Schritten vollzogen wird.

- Schritt 1 – Commands.properties

  Bestimmung eines eindeutigen Namens des neuen Commands zur Verwendung in anderen Konfigurationsdateien, verbunden mit der Angabe der realisierenden Klasse.

- Schritt 2 – Action.properties

  Bestimmung der Eigenschaften der zugehörigen CommandAction, wie beispielsweise Tooltip-Text oder Icon.

- Schritt 3 – menucomponents.properties

  Anlegen von Strukturen, wie Menü-, Untermenü- oder Toolbar-Strukturen.

Abbildung 9 zeigt das neu erstellte Command eingebunden im Datei-Menü des CommandsTestFrame.

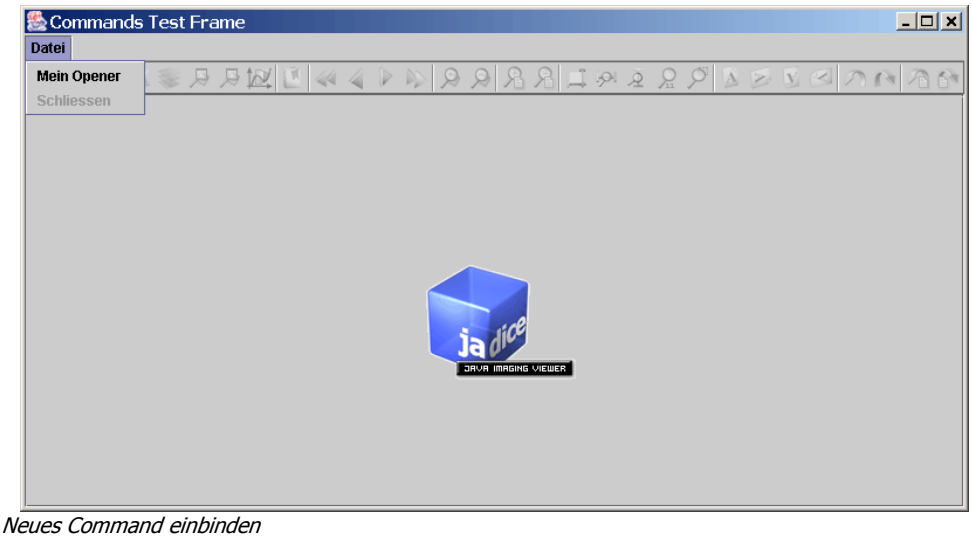

Abbildung 9 - Neues Command einbinden

Hinweis:

Die Konfigurationsangaben werden beim Start des Viewers einmalig gelesen und ausgewertet. Änderungen an Konfigurationsdateien werden also erst nach einem Neustart des Viewers wirksam.

Hinweis:

Bitte beachten Sie, dass die Konfigurations-Dateien in internationalisierten Varianten vorliegen. Um Inkonsistenzen zu vermeiden, sollten Änderungen der Konfiguration stets in allen Varianten getätigt werden.

5.5. Drucken

5.5.1. Einfacher Druck

PrinterJava2 ist die zentrale Klasse zum Druck von Dokumenten aus jadice. Ein einfaches Beispiel ist die Methode „**simplePrint()**", die die Klasse TestViewerFrame erweitert.

```
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 Beispiel 19 – Einfacher Druck

Zunächst wird eine Instanz der Klasse PrinterJava2 über den Default Konstruktor erstellt. Alternativ kann auch ein Konstruktor mit parametrisiertem

PrinterJob verwendet werden, sofern die integrierende Applikation einen angepassten PrinterJob verwenden möchte.

Notwendige Angaben zum Druck sind ein Dokument, die zu druckenden Seiten und ein RenderContext mit Angaben zur Darstellung auf einem Drucker-Device.

Dazu übergibt die Methode „**simplePrint()**" der PrinterJava2 Instanz das im Viewer gesetzte Dokument sowie den RenderContext des Viewers. Damit werden die Render-Einstellungen des Viewers zum Druck übernommen. Alternativ kann auch ein speziell zum Druck angepasster RenderContext gesetzt werden. Ein Beispiel dazu ist unter 5.5.3Anpassung des RenderContext zu finden.

Im nächsten Schritt wird der Druckprozess gestartet. Falls keine anderen Angaben durch den Benutzer in einem eventuell geöffneten Druckdialog gesetzt werden, werden alle Seiten des Dokuments ausgedruckt. Ob ein Drucker- oder Seitenformatdialog geöffnet wird, hängt von der gesetzten Konfiguration ab bzw. welche Angaben durch den Aufruf entsprechender Methoden der Klasse PrinterJava2 angegeben wurden. Betrachten Sie dazu auch den folgenden Abschnitt 7.Konfiguration und Einstellungen.

5.5.2. Einstellungen

Die Klasse PrinterJava2 erlaubt einige optionale Einstellungen, die im Folgenden näher erläutert werden. Alle Default-Einstellungen zum Druck entnehmen Sie bitte Abschnitt 7.Konfiguration und Einstellungen.

```
    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 Beispiel 20– Einstellungen zum Druck

Ein Seitenformat kann direkt mit der Methode „**setPageFormat(PageFormat)**" gesetzt oder vom Benutzer in einem Seitenformat-Dialog bestimmt werden. Die Anzeige eines Seitenformatdialogs wird über die Methode „**setShowPageDialog(boolean)**" gesteuert. Falls keine der beiden Möglichkeiten von der integrierenden Anwendung genutzt wird, wird das in der Konfiguration gesetzte Seitenformat verwendet.

Ähnlich verhält es sich mit der gewünschten Seitenauswahl. Sollen nur bestimmte Seiten gedruckt werden, kann dies mittels der Methode „**setPageSelection(int[])**" direkt angegeben werden. Dabei ist zu beachten, dass die Seitennummerierung, wie auch im Document oder Loader, nullbasierend ist. Alternativ kann der Benutzer die gewünschten Seiten in einem Drucker-Dialog angeben, der mittels der Methode „**setShowPrinterDialog(boolean)**" zur Ansicht gebracht werden kann. Ferner kann in diesem Dialog der Zieldrucker bestimmt werden. Standardmäßig druckt PrinterJava2 auf dem Default-Drucker des Systems. Bitte beachten Sie auch die Einstellungen zur Ansicht des Drucker Dialogs in der Konfiguration.

Für bessere Druckergebnisse können die zu druckenden Seiten optimal in den druckbaren Bereich eingepasst werden. Dazu stehen folgende Methoden zur Verfügung:

† **setEnlargePageToPaper(boolean)**

Vergrößere kleine Seiten optimal in den Druckbereich.

† **setShrinkPageToPaper(boolean)**

Verkleinere große Seiten optimal in den Druckbereich.

† **setOptimizeRotation(boolean)**

Rotiere Seiten optimal in den Druckbereich.

Ob der Druckprozess asynchron oder synchron vollzogen werden soll, kann über die Methode „**setAsynchronousPrinting(boolean)**" bestimmt werden. Die Default-Einstellung ist asynchron.

5.5.3. Anpassung des RenderContexts

Im Allgemeinen kann zum Druck der RenderContext des Viewers verwendet werden, aber auch eine durch den Integrator angepasste Instanz ist möglich. Beispielsweise durch die Angabe einer Gradationskurve für Bilddaten oder AnnotationsRenderSettings, zum Anzeigen oder Verbergen von Annotationen.

Veränderungen der Gradationskurve des Druck-RenderContexts erfolgen über eine entsprechende setter-Methode des RenderContexts.

Im folgenden Beispiel soll der RenderContext so angepasst werden, dass, unabhängig davon welche Annotationen im Viewer dargestellt werden, der Ausdruck nur das Dokument ohne Annotationen zeigt.

Dazu wird über die Methode „**getNoAnnotationsVisibleRenderContext()**" ein angepasster RenderContext erzeugt, der dann einer Instanz der Klasse PrinterJava2 übergeben werden kann.

```
public RenderContext
          getNoAnnotationsVisibleRenderContext() {

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

 annoRenderSettings.setAnnotationRenderingEnabled(false);

 return rc;
}
```

Code Beispiel 21- Anpassung des RenderContexts

Zunächst wird der RenderContext des Viewers geklont, damit die Änderungen nicht die Viewerdarstellung beeinflussen.

Der Viewer unterstützt zwei Methoden, um die Sichtbarkeit von Annotationen zu verändern. Zum einen können alle Annotationen aus-/eingeblendet werden, zum anderen können alle Annotationen eines bestimmten Typs aus-/eingeschaltet werden.

Dazu wird die Klasse AnnotationRenderSettings[86] genutzt, die in der Klasse RenderContext als ein ProcessingSetting[87] enthalten ist. Ein RenderContext unterhält verschiedene ProcessingSettings, die jeweils bestimmte Arten von Rendereigenschaften abdecken. AnnotationRenderSettings bestimmen die Sichtbarkeit von Annotationen oder nur bestimmten Annotationstypen.

Die Sichtbarkeit von Annotationen wird in obigem Beispiel mittels der Methode „**setAnnotationRenderingEnabled (boolean)**" gesetzt.

Bitte beachten Sie, dass einige Angaben des RenderContexts zum Druck ignoriert werden, da sie durch das Ausgabedevice unabänderlich vorgegeben sind. Dies umfasst Zoom, Rotation und ähnliches.

86 com.levigo.jadice.annotation.AnnotationRenderSettings
87 com.levigo.jadice.docs.ProcessingSettings

6. Logging

6.1. Die jadice® Logging Framework Facade

Bereits mit der Version 4.1 der jadice document platform wurde eine neue Logging Framework Facade eingeführt.

Bisher, d.h. in allen jadice Versionen vor 4.1.x, musste zur Ausgabe von jadice spezifischen Meldungen in bestehende Logging Systeme der Zielanwendung das Interface LogAdapter[88] implementiert und dem jadice Logging Mechanismus bekannt gegeben werden.

Mit der neuen jadice Logging Facade ist diese Anforderung wesentlich vereinfacht worden. Das neue Framework erlaubt ein direktes Einbinden der bekanntesten und verbreitetsten Logging Systeme und Frameworks, wie Log4J, SLF4J und via SLF4J JDK 1.4 Logging, Logback, JCL, x4Juli und viele andere mehr. Sofern Sie bereits eines der erwähnten Frameworks einsetzen, ist die Ausgabe von jadice Meldungen in eines dieser Logging Systeme oft nur eine einfache Klassenpfad Modifikation.

Falls kein bestimmtes Logging Delegate, sprich ein bestimmtes Logging Framework, durch den Klassenpfad vorgegeben und verfügbar ist, wird ein einfaches Standard Logging benutzt. Dieses einfache Logging gibt jede (non-debug) Nachricht auf `System.out` aus.

Durch diesen einfachen Standard Logger, der als Fallback zur Verfügung steht, wird die Entwicklungsphase vereinfacht. Das gewünschte Ziel Logging System wird nicht mehr zur Kompilierzeit benötigt, lediglich zur Laufzeit muss das entsprechende Framework im Klassenpfad vorhanden sein und zur Verfügung stehen. Auch wenn es nicht mehr zwingend notwendig ist, kann das Ziel Logging System zur Kompilierzeit nach wie vor integriert werden. Ebenso kann in der Entwicklungsphase ein anderes Logging System als in der Endanwendung benutzt werden.

6.2. Erste Schritte

Die Nutzung der jadice Logging Framework Facade ist recht einfach. Zunächst einmal stellt sich die Frage, welches Ziel Logging System verwendet werden soll.

Die Auslieferung der jadice document platform stellt dazu zwei Implementationen von Logging Delegates zur Verfügung. Soll Log4J verwendet werden, fügen Sie die entsprechende Log4J Implementation dem Klassenpfad der Anwendung hinzu. Zur Verwendung von einem anderen Logging Framework nehmen Sie statt dessen die entsprechende SLF4J Implementation im Klassenpfad auf.

Eine detaillierte Beschreibung, wo die gewünschte Implementation des Logging Delegates in der Auslieferung zu finden ist, ob und welche weiteren Schritte notwendig sind, entnehmen Sie bitte den folgenden Abschnitten.

Beide in der Auslieferung enthaltenen Logging Delegates benötigen keine weitere spezifische Konfiguration. Eine Anpassung des Verhaltens des Ziel Logging Systems, z.B. des Log Levels oder ähnlichem, kann über die Konfigurationsmöglichkeiten des jeweiligen Ziel Logging Systems gesteuert werden.

88 com.levigo.util.log.LogAdapter

6.2.1. Log4J

Die Implementation des Logging Delegates für Log4J entspricht folgender Namenskonvention:

- ③ **`logging-log4j-<version>.jar`**

In der Auslieferung finden Sie das entsprechende Log4J Delegate, jeweils passend zu den im Einsatz befindlichen jadice Modulen, unter folgender Verzeichnisstruktur:

- ③ **`lib-jdk15/logging`**
- ③ **`lib-all-in-one/jdk15/logging`**

Fügen Sie bitte den passenden Log4J Delegate in den Klassenpfad der Anwendung ein. Ist eine log4j Konfiguration bereits auf dem Klassenpfad verfügbar, wird keine weitere Konfiguration benötigt.

Detaillierte Informationen über die Konfiguration gibt es auf der Log4J Homepage und im Log4J Manual.

Die jadice document platform ist nicht auf eine bestimmte Logging Konfiguration angewiesen und nimmt auch keine Einstellungen am Ziel Logging System vor.

6.2.2. SLF4J

Die Namenskonvention des Logging Delegates von SLF4J entspricht folgendem Schema:

- ③ **`logging-slf4j-<version>.jar`**

Wie auch für Log4J finden Sie das entsprechende SLF4J Delegate, jeweils passend zu den im Einsatz befindlichen jadice Modulen, unter folgender Verzeichnisstruktur:

- ③ **`lib-jdk15/logging`**
- ③ **`lib-all-in-one/jdk15/logging`**

Fügen Sie das passende SLF4J Delegate in den Klassenpfad der Anwendung ein. Zusätzlich wird das **`slf4j-api-<version>.jar`** und eine Logging Implementierung benötigt. Detaillierte Informationen über slf4j und unterstützte Typen der Logging Delegates und Implementierungen finden Sie auf der SLF4J Homepage bzw. im SLF4J Manual.

6.3. Mögliche Fehler

Im Falle eines Fehlers (Laden der Framework Facade nicht möglich, etc.) finden Sie detaillierte Hilfe zu den Fehlermeldungen in der auslieferungsaktuellen HTML Dokumentation (documentation.html), die der Auslieferung der jadice document platform beiliegt.

7. Konfiguration und Einstellungen

Die jadice document platform trennt Lizenz- und Konfigurationsdaten. Die Konfigurationsdatei des jadice Pakets heißt „Jadice.properties" und befindet sich im Default Package des jadice document bzw. des jadice All-in-one Jars. Sie enthält spezifische Einstellungen wie beispielsweise:

- Einstellungen des Viewer Verhaltens
- Einstellungen des Verhaltens bestimmter AddOns
- betriebssystemspezifische Einstellungen
- Druckeinstellungen
- Einstellungen für externe AFP/MO:DCA-Ressourcen
- Cachegröße
- Lebenszeit von Temp. Dateien

etc.

Auf die einzelnen Parameter wird im Laufe dieses Kapitels näher eingegangen.

Die Konfigurationsangaben werden nur beim Start von jadice einmalig gelesen und ausgewertet. Änderungen an der Konfigurationsdatei werden also erst nach einem Neustart von jadice wirksam. jadice sucht zunächst in seinem Arbeitsverzeichnis nach der Konfiguration. Sollte jadice dort nicht fündig werden, wird im Klassenpfad gesucht. Aufgrund von Sicherheitsbeschränkungen wird bei Applets nur im Klassenpfad gesucht.

Die Konfigurationsdatei sollte stets im Default-Package verbleiben, eigene Anpassungen können direkt in dieser Datei oder in einer Kopie im Arbeitsverzeichnis oder Klassenpfad vorgenommen werden. Es ist jedoch ratsam eigene Anpassungen nur in einer Kopie vorzunehmen, um auf die Default-Einstellungen zurückgreifen zu können.

Zugriff auf die Angaben der Konfiguration erhalten Integratoren über die Klasse JadicePreferenceHolder[89].

Beispiel:

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

Dieser Aufruf gibt ein Objekt der Klasse PreferenceStore[90] zurück, das die geladenen Konfigurationsdaten beinhaltet. Ein PreferenceStore ist ähnlich einem Properties-Objekt, erlaubt jedoch qualifizierten und typsicheren Zugriff auf die Konfigurationseinstellungen.

Die jadice Familie verwaltet Eigenschaften und Einstellungen in PreferenceStores. Die Klasse PreferenceStore ist ein Interface, das, ähnlich wie die Klasse Properties[91], den Zugriff auf Daten über Key-Value Pairs verwaltet. Der Vorteil von PreferenceStores ist der gekapselte Zugriff auf die beinhalteten Daten. Als Schnittstelle können eigene Implementierungen registriert werden, die es erlauben Daten aus beliebigen Quellen, z.B. aus einer Datenbank, aus dem Datei-System, dem Intra-/Extranet o.ä., zu nutzen.

89 com.levigo.jadice.util.JadicePreferenceHolder
90 com.levigo.util.preferences.PreferenceStore
91 java.util.Properties

Integratoren ist es freigestellt eigene Implementierungen über den JadicePreferenceHolder verwalten zu lassen. Weitere Details entnehmen Sie bitte der jadice API-Dokumentation, insbesondere zu den Klassen PreferenceStore, JadicePreferenceHolder, PreferenceStoreHolder und PropertiesPreferenceStore.

7.1. Die wichtigsten Einstellungen im Einzelnen

Option	Zweck
Drucken – Vorbelegung der Printer Klasse	
jadice.viewer.show-print-dialog=true	Soll ein Druckdialog angezeigt werden?
jadice.viewer.show-pageformat-dialog=true	Soll ein Seitenformat Dialog angezeigt werden?
jadice.viewer.printer-page-format-enabled=true	Soll ein Standard-Seitenformat gesetzt sein?
jadice.viewer.printer-page-format-size-x=210 jadice.viewer.printer-page-format-size-y=297 jadice.viewer.printer-page-format-border-x-left=10 jadice.viewer.printer-page-format-border-x-right=10 jadice.viewer.printer-page-format-border-y-top=10 jadice.viewer.printer-page-format-border-y-bottom=10 jadice.viewer.printer-page-format-orientation=1	Definition des Standard-Seitenformats, wird nur verwendet, wenn die Benutzung eines Standard-Seitenformats aktiviert ist, d.h. die Eigenschaft *jadice.viewer.printer-page-format-enabled=true* gesetzt ist. Größenangaben in mm. Seitenformat: 0 = landscape; 1 = portrait
Standard Druck Commands	
jadice.viewer.printer.commands.DefaultPrintMode=PrintAll	Bestimmt den Standard Druckmodus der jadice Druck Commands. Drei Modi sind verfügbar: ˜ PrintAll - Dokument und Annotationen werden gedruckt ˜ PrintOnlyDocument - Nur das Dokument wird gedruckt ˜ PrintOnlyAnnotations - Nur die Annotationen werden gedruckt Default Wert: PrintAll
jadice.viewer.printer.commands.DefaultPrintAdjusting=FitPrint	Bestimmt bei Benutzung der jadice Druck-Commands, wie Dokumente im druckbaren Bereich eingepasst (skaliert) werden. Folgende Werte sind möglich: ˜ OrigSizePrint – Druckt das Dokument in Originalgröße. ˜ ShrinkPrint – Passt das Dokument durch Verkleinerung in den druckbaren Bereich ein. ˜ EnlargePrint - Passt das Dokument

Option	Zweck
	durch Vergrößerung in den druckbaren Bereich ein. ˜ FitPrint - Passt das Dokument stets in den druckbaren Bereich ein, d.h. entsprechend der Seitengröße wird die Dokumentansicht vergrößert bzw. verkleinert. Default Wert: FitPrint
Ziel-System spezifische Anpassungen	
jadice.viewer.printjobname.maxlength=40	Unter Windows kann es vorkommen, dass Druckaufträge mit einem zu langen Druck Job Namen nicht akzeptiert und ohne weitere Fehlermeldung verworfen werden. Diese Einstellung bestimmt die maximale Länge von Druck Job Namen, die aus dem jadice Paket initiiert werden. Mögl. Werte: Positive ganze Zahlen größer 0. Wenn ein nicht gültiger Wert angegeben oder diese Einstellung auskommentiert ist, wird keine Begrenzung des Druck Job Namens vorgenommen.
jadice.viewer.printer-transparent-fix=auto	Abhängig von der Grafikkarte, der Monitor-Einstellung und des Drucker-Treibers kann es zu Druckproblemen von transparenten Bildbereichen oder Bildelementen kommen. Im angeschalteten Zustand wird ein Workaround aktiviert, der dieses Problem vermeidet, aber einen größeren Druck-Output zur Folge hat. Mögl. Werte: true, false, auto Default Wert: auto
Hover Lens – Schwebende Lupe	
jadice.hover-lens.use-click-scaling=true	Einstellung, ob der Vergrößerungsfaktor der Lupe über Mausklicks veränderbar sein soll. Mögl. Werte: true, false Default Wert: true
jadice.hover-lens.default-scale=150	Initialer Zoom, Wert der Lupe in Prozent. Mögl. Werte: Ganze Zahlen größer als 0. Default Wert: 150, entspricht 150%
jadice.hover-lens.click-scale-step=25	Zoom-Schritt in Prozent, in dem sich der Vergrößerungsfaktor der Lupe auf

Option	Zweck
	Mausklick verändert (nur von Bedeutung, wenn *jadice.hover-lens.use-click-scaling=true* gesetzt ist). Mögl. Werte: Ganze Zahlen größer als 0 Default Wert: 25, entspricht 25%
jadice.hover-lens.shape=1	Form der schwebenden Lupe. Mögl. Werte: 1 - rechteckig 2 – rund Default Wert: 2, entspricht rund Hinweis: Integratoren können via API beliebige Shapes für die Lupe bestimmen, über die Konfigurationsdatei werden jedoch nur diese beiden Formen angeboten.
jadice.hover-lens.size.width=150 jadice.hover-lens.size.heigth=150	Größe der schwebenden Lupe in Pixel. Mögl. Werte: Ganze Zahlen größer als 0 Default Wert: Breite: 150 Höhe: 150
jadice.hover-lens.autoscroll.mode=true	Einstellung, ob die Lupe ein Autoscroll Verhalten des Viewers initiiert, wenn die Maus den Dokumenten-Bereich verlässt. Mögl. Werte: true, false Default Wert: true
Page Sorter – Seitensortierer, Miniaturansicht	
jadice.sorter.show-page-numbers=false	Bestimmt, ob der Seitensortierer Seitennummern anzeigt. Mögl. Werte: true, false Default-Wert: false
jadice.sorter.single-click-navigates=false	Einstellung, ob durch Einzel-Klick im Seitensortierer ein Seitennummerwechsel im Viewer ausgelöst werden soll oder nicht. Mögl. Werte: ˜ true Seitennavigation wird durch einfachen Klick auf eine Seite im Seitensortierer ausgelöst. ˜ false Ein einfacher Klick auf eine Seite im Sorter selektiert diese, ein Doppelklick löst ein Umblättern des Viewers auf diese Seite aus. Default Wert: false
Zoom Policy	

Option	Zweck
jadice.viewer.apply.zoom-policy.resize=true	Einstellung, ob die Zoom Policy auch bei Größenveränderung des Viewers Anwendung finden soll. Hinweis: ˜ Zoom-Policy Verhalten hat stets geringere Priorität als Benutzer-Einstellungen. D.h. falls der Benutzer die Einstellung des Zoomfaktors eigenständig verändert, wird die Zoom-Policy ignoriert. ˜ Ist nur in Verbindung mit Zoom Policy „fit", „fit width", „fit height" unterstützt. Mögl. Werte: true, false Default Wert: false.
jadice.viewer.zoom-policy=2	Zoom Einstellung für neu geladene Dokumente. Mögl. Werte: 1 – Behält den Zoom Wert bei. 2 - „fit"-Modus passt das Dokument in den Viewer ein. 4 - „fit width"-Modus passt das Dokument horizontal in den Viewer ein. 8 - „fit height"-Modus passt das Dokument vertikal in den Viewer ein. 16 - "100%" Modus stellt das Dokument in seiner Originalgröße dar. 32 - "page-fit" Modus passt jede Seite in den Viewer ein, sofern der User keinen anderen Seiten- oder Dokument-Zoomwert bestimmt hat. 64 – "page-fit width" Modus passt jede Seite horizontal in den Viewer ein, sofern der User keinen anderen Seiten- oder Dokument-Zoomwert bestimmt hat. 128 – "page-fit height" Modus passt jede Seite vertikal in den Viewer ein, sofern der User keinen anderen Seiten- oder Dokument-Zoomwert bestimmt hat. Default Wert: 1
Afp-spezifisch	
jadice.viewer.afp-resource-extension=ovl;300	Registrieren von speziellen Datei-Endungen für externe Afp Ressourcen.
jadice.viewer.afp-resource-path=d:\\afp_res\\	Registrieren von einem bestimmten Afp Ressourcen Verzeichnis.
Temporär Dateien (-> FileCacheInputStream)	

Option	Zweck
jadice.viewer.delete-overaged-tmps = TRUE	Automatisches Entfernen von verbliebenen temp. Dateien.
jadice.viewer.overaged-tmps-lifetime=0	Anzahl Tage, die temp. Dateien nicht gelöscht werden dürfen. Default Wert: System temp. Verzeichnis
jadice.viewer.tmps-path=c:/temp	Pfad für temp. Dateien
Annotationen	
jadice.viewer.annotation.type=vi	Gibt an, welche Art von Annotationen verarbeitet werden. Die Annotationsart bestimmt den Aufbau der Annotations-Toolbar und die Eigenschaften der Annotations-Editoren. vi = Visual Info kompatible Annotationen; fn = FileNet kompatible Annotationen; fnp8 = FileNet P8 kompatible Annotationen Default Wert: vi
jadice.viewer.annotation.creation.mode=0	Gibt an, wie sich die Tools zur Erzeugung von Annotationen verhalten. Mögliche Werte: (0) nonpermanent Mode = Das Tool wird automatisch deselektiert nachdem eine Annotation erzeugt wurde; (1) permanent Mode = Ist ein Tool ausgewählt, verbleibt es solange ausgewählt, bis der Benutzer es deselektiert; (2) both = Wurde das Tool mit einem einfachen Mausklick ausgewählt, entspricht das Verhalten dem „nonpermanent"-Mode, bei Doppelklick dem „permanent"-Mode Default Wert: 0

8. jadice Integrator API: Syntax Beschreibung der Konfigurationsdateien

Die im Auslieferungspaket beinhalteten Konfigurationsdateien der jadice document platform sind unter **com.levigo.jadice.properties.*** oder **com.levigo.jadice.graphics.*** zu finden. Sie werden in den folgenden Abschnitten detailliert beschrieben.

Hinweis:

Zur Erstellung einer eigenen Konfiguration beachten Sie bitte, dass die Konfigurationsdateien Referenzen auf korrespondierende Konfigurationen enthalten. Diese Referenzen müssen entsprechend angepasst werden.

Wenn nicht eigene Konfigurationen erstellt werden sollen, ist es angeraten die jadice Konfigurations-Dateien zu kopieren und im Klassen-Pfad vor den jadice document platform Modulen zu legen. Damit werden gemachte Änderungen erkannt, aber die ursprüngliche Konfiguration bleibt unangetastet.

Hinweis:

Bitte beachten Sie, dass die Konfigurations-Dateien in internationalisierten Varianten vorliegen. Um Inkonsistenzen zu vermeiden, sollten Änderungen der Konfiguration stets in allen Varianten getätigt werden, mindestens jedoch in der der aktuellen Locale entsprechenden Varianten.

8.1. Die Datei „commands.properties"

Diese Konfiguration erzeugt ein Mapping zwischen einem eindeutigen Command-Namen, der in den anderen Konfigurationen als Referenz verwendet wird, und dessen Verwirklichung.

Allgemein kann die Syntax wie folgt angegeben werden:

Option	Zweck
CommandName=CommandPfad.CommandKlassenName	Der Command-Bezeichner ist ein frei wählbarer, aber eindeutiger Name, der in anderen Konfigurationen als Referenz genutzt wird. Die Angabe der Command Realisierung setzt sich aus dessen Pfad- und Klassennamen zusammen. Auf Groß- und Kleinschreibung sollte an dieser Stelle geachtet werden, da Commands über Reflektion erzeugt werden.

Beispiel:

```
MeinTestCommand1=test.meineTests.AlleTestKlassen$EinTestCommand   oder
MeinTestCommand2=test.meineTests.EineTestCommandKlasse
```

In der ersten Zeile wird ein Command beschrieben, das ***MeinTestCommand1*** heißt und als innere Klasse mit dem Namen ***EinTestCommand*** der umschließenden Klasse ***AlleTestKlassen*** verwirklicht wurde. In der zweiten

Zeile wird ein weiteres Command angegeben, das MeinTestCommand2 heißt und präsentiert wird durch die Klasse EineTestCommandKlasse.

8.2. Die Datei „menucomponents.properties"

Die menucomponents-Konfiguration bestimmt die Zusammensetzung von Menüs, Sub-Menüs, Kontext-Menüs und Toolbars, im folgenden Strukturen genannt. Im folgenden Abschnitt 8.3.Die Datei "actions.properties" wird eine korrespondierende *actions.properties* bestimmt, die die Eigenschaften der Command Actions beschreibt.

Die Syntax einer Struktur kann nun wie folgt beschrieben werden, wobei der schwarz geschriebene Ausdruck „name" ein frei zu bestimmender, aber eindeutiger Strukturbezeichner ist. Rote Angaben sind feststehende Ausdrücke:

Option	Zweck
Definition einer Struktur	
name.name=TestMenuName	Name einer Struktur. Bei Menüs dient er zusätzlich als Menüname zur Anzeige in einer Menübar.
name.actions=CommandName1, CommandName2	Definition der in der Struktur beinhalteten Commands aus denen Menüeinträge oder Toolbuttons erstellt werden. Commands werden als eine durch Komma separierte Liste angegeben. Diese Art der Definition kann für alle Strukturen genutzt werden.
name.actions.toolbar=CommandName1,CommandName2	Analoge Definition der beinhalteten Commands, speziell nur für Toolbarstrukturen
name.actions.contextmenu=CommandName1, dfsdfsdf CommandName2	Analoge Definition der beinhalteten Commands, speziell für Kontext-Menüs
name.actions.menu=CommandName1, CommandName2	Analoge Definition der beinhalteten Commands, speziell nur für Menüstrukturen.
name.menuState=	Nur für Checkbox / Radiobutton Menüpunkte. Beschreibt den initialen Selektionstatus. Wert: selected – ausgewählt alle anderen Angaben – nicht ausgewählt
name.menuType=	Nur für Menü Items: Ohne Angabe wird ein normaler Menüpunkt erzeugt. Mögliche Werte: † visibilityEnabled – Item, das nur sichtbar ist, wenn es enabled ist † checkbox – Checkbox Item

Option	Zweck
	† visibilityEnabledCheckbox – Checkbox Item, das nur sichtbar ist, wenn es enabled ist † radiobutton – Radiobutton † visibilityEnabledRadiobutton – Radiobutton Item, das nur sichtbar ist, wenn es enabled ist † iconmenu - einfacher Menüpunkt entsprechend dem gesetzten Look & Feel mit Icon Anzeige, wenn gesetzt † ohne Angabe – einfacher Menüpunkt entsprechend dem gesetzten Look & Feel, ohne Icon
Struktur bestimmende Setzungen	
{name}	Substitution Teil-Menü
{>name}	Substitution Unter-Menü
|	Separator

Dazu ein Beispiel:

```
TeilMenu.actions=Command1,Command2
SubMenu.actions=Command4,Command3
SubMenu.name=ein SubMenu
TollesMenu.actions=Command5,|,{TeilMenü},|,{>SubMenu}
TollesMenu.name=ein tolles Menü
```

Mit obiger Definition wird ein Menü erstellt, das den Titel trägt „ein tolles Menü" und aus Command5, einem Separator, Command1, Command2, einem weiteren Separator und einem Unter-Menü mit dem Namen „ein Sub Menü", zusammengesetzt aus Command4 und Command3, besteht.

Die Angabe der korrespondierenden Konfigurationsdatei *actions.properties* wird folgendermaßen bestimmt:

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

Weitere Action Konfigurationen können wie folgt angegeben werden:

```
resource.moreActions=/meine/action/definitionen/actions.pr
operties
```

Zur Unterscheidung, ob ein Command über die Default-Action Konfiguration oder eine andere Konfiguration bestimmt ist, wird der Konfigurations-Bezeichner vorangestellt.

Beispiel:

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

Hier setzt sich das Teilmenü aus dem *Command1*, definiert durch die Default Action Konfiguration, und dem *Command2*, bestimmt durch die Konfiguration mit dem Namen *moreActions*, zusammen.

8.3. Die Datei "actions.properties"

Die Datei *actions.properties* beschreibt die Eigenschaften der CommandActions. Dies umfasst bspw. welche Commands in welcher Reihenfolge zur Ausführung der Action aktiviert werden sollen, welches Icon angezeigt werden soll, ob ein Tooltip angeboten werden soll usw.

Des weiteren wird jeweils eine korrespondierende Command Konfiguration (*commands.properties*) und eine Icon Beschreibung angegeben über

```
# Icon Beschreibung
icons.defaulticons=/com/levigo/jadice/resources/graphics/jadice-viewer
resource.icons.default=defaulticons
# Commands Konfiguration
resource.commands=/com/levigo/jadice/resources/properties/commands.properties
resource.commands.default=commands
```

Wie auch in den *menucomponent.properties* weitere Action Konfigurationen definiert werden können, ist es möglich in den *action.properties* verschiedene command Konfigurationen zu bestimmen.

Beispiel:

```
resource.mycommands=/meine/command/definitionen/commands.properties
```

Vergleichen Sie dazu bitte auch 8.1.Die Datei „commands.properties" und 8.2.Die Datei „menucomponents.properties".

Die möglichen Angaben je Action sind in der folgenden Tabelle aufgelistet, wobei der schwarz geschriebene Ausdruck „name" ein frei zu bestimmender, aber eindeutiger Action Bezeichner ist. Rote Angaben sind feststehende Ausdrücke:

Option	Zweck
Definition einer Command Action	
name.SmallIcon=defaulticons.TB_OPEN	Icon Definition bezogen auf defaulticons-Referenz. Bsp: TB_OPEN – Icon Name in der Icon Referenz
name.commands=	Komma separierte Liste von Commands, die in der actionPerformed Methode aktiviert werden sollen.
name.ShortDescription=Kurzbeschreibung	Name, z.B für Menü Eintrag
name.LongDescription= Beschreibung Text	Tooltip-Text, Angaben können in HTML angegeben werden.

Option	Zweck
name.AcceleratorKey= VK_N + CTRL_MASK	Nur für Menü Items: Hot Key für Menü Items Modifier plus Taste Alt-, Shift- oder Ctrl-Mask + KeyEvent.Key Zu verwenden sind analoge Bezeichner wie es die Klasse KeyEvent[92] bzw. die Klasse InputEvent[93] vorgeben.
name.MnemonicKey= VK_N	Nur für Menü Items: Mnemonic Key zur Navigation durch Menüs Hinweis: Mnemonic Keys werden immer mit dem Modifier ALT-MASK verbunden.
name.InputMap= VK_PLUS	Eintrag in der InputMap des Context Owners im Modus: „WhenInFocused-Window". Wenn keine Komponente der verwendeten Komponentenhierachie den angegebenen Tastatur Event konsumiert, wird der Event zur Aktivierung des Commands verwendet. Vom verwendeten L&F vorbelegteTastatur Bindings priorisieren grundsätzlich Command Bindings. Tastenkombinationen können über die verschiedenen Tasten-Bezeichner verbunden mit einem Pluszeichen angegeben werden. Beispiel: SHIFT_MASK + VK_A erzeugt einen Eintrag in der InputMap, der ein Command bei dem Tastatur-Ereignis Shift-A auslöst. Hinweis: Vom System bzw. durch das verwendete L&F vordefinierte KeyBindings priorisieren grundsätzlich die hier gemachten Angaben. Soll ein KeyBinding anders belegt werden als es durch das System oder das L&F vorgeben ist, muss der Integrator sicherstellen, das die vordefinierten KeyBinding entfernt wurden. jadice verändert von sich aus gründsätzlich keine vorgegebenen Einstellungen.

8.4. Die Datei „jadice-viewer.properties"

Alle Icons der jadice document platform sind aufgrund von Übersichtlichkeit und Ressourcen-Optimierung in einer PNG-Datei (jadice-viewer.png) zusammengefasst worden. Als Web-Format kann PNG Icons sehr gut darstellen

92 java.awt.event.KeyEvent
93 java.awt.event.InputEvent

(Alpha-Kanal mit variabler Transparenz, Gamma Korrektur u.ä.) bei einer optimierten Kompression, die zwischen 5% - 25% besser als GIF ist.

Angepasste Icons müssen nicht im PNG-Format vorliegen, gleichermaßen werden auch die Formate GIF und JPEG unterstützt.

Die *jadice-viewer.properties* Konfiguration beschreibt diese Gesamtheit der zu Verfügung stehenden Icons. Auch in dieser Tabelle sind rote Angaben feststehende Ausdrücke.

Option	Zweck
extension=png	Format der Icon Datei; die Icon Datei trägt denselben Namen wie die Konfiguration, der Suffix ergibt sich aus dem Format. Mögliche Werte: png, gif, jpeg
icon.name.rectangle=0,0,24,24	Wo liegt in der Icon Datei das entsprechende Icon mit dem Namen „name"? Alle Icons werden über diese Angabe in Position und Dimension beschrieben. *name* ist eine frei wählbare, aber eindeutige Bezeichnung für dieses Icon und wird in der Konfiguration *actions.properties* als Referenz genutzt. Im Beispiel: 0,0 entspricht dem Ursprung der Icon Position 24,24 entspricht der Dimension des Icons

9. jadice Public API und internal Packages

9.1. jadice Public API

Die jadice® document platform bietet Entwicklern und Integratoren eine mächtige Public API zur Integration in spezifische Kundenanwendungen und Lösungen. Jeweils versionsaktuelle Javadoc-Informationen der Public API werden mit der Auslieferung im Verzeichnis

<Auslieferungsverzeichnis>/javadoc

zur Verfügung gestellt.

In diesem Verzeichnis befinden sich mehrere Jar-Dateien mit Javadoc-Informationen der Public API der verschiedenen Module der jadice® document platform.

Alle Klassen und Methoden, die in diesen Javadoc Dokumentationen enthalten und beschrieben sind, sind Teil der Public jadice® API und können frei zur Einbindung von jadice® Komponenten und Funktionalitäten verwendet werden.

Die durch die Javadoc-Informationen beschriebenen Klassen, Methoden und Funktionalitäten der Public jadice® API unterliegen in vollem Umfang der von levigo solutions zugesicherten Wartung und Support.

Bei Fragen oder Problemen mit der Public jadice® API oder falls zusätzliche Informationen von weiteren Modulen benötigt werden, können sich Integratoren oder Entwickler jederzeit an levigo solutions wenden. Wir unterstützen Sie gerne.

9.2. Jadice Private API

Neben der Public API sind im Auslieferungspaket interne Klassen, Methoden, Funktionalitäten und Teilfunktionalitäten mit folgender Namensgebung ***.internal.*** enthalten.

Diese Stukturen sind Teil der internen **Private** jadice® API und dürfen nicht durch Entwickler oder Integratoren direkt genutzt werden.

levigo solutions behält sich vor sämtliche Inhalte der **Private** jadice® API, Klassen, Methoden und Funktionalitäten jederzeit und ohne Vorankündigung zu entfernen, zu verschieben, neu zu benennen oder in ihrer Funktionalität zu ändern. Es besteht keinerlei Anrecht auf Vollständigkeit der Klassen, Methoden und Funktionalitäten der **Private** jadice® API zwischen einzelnen Versionen oder Releases.

Technische Unterstützung bei Problemen durch die direkte Verwendung der Private jadice® API ist aus der Softwarepflege ausgeschlossen und wird unter keinen Umständen unterstützt.

Jegliche direkte Nutzung der internen **Private** jadice® API enthebt levigo solutions sämtlicher Gewährleistungs- und Haftungsansprüche. levigo solutions haftet in keiner Weise oder Art für jeglichen direkten, indirekten, zufälligen, speziellen, exemplarischen oder sonstigen Schaden, der entsteht oder entstehen könnte, durch die direkte Verwendung der internen **Private** jadice® API.

10. Dokumentenhistorie

Version	Datum	Autor	Änderungen
0.1	14.07.03	Oliver Suck	Als Vorlage die Entwicklerdokumentation der Vs 2.x
	16.07.03	Jörg Henne	- Neu in Version 3.x - Dokument / Layer Begrifflichkeiten
	06.08.03	Carolin Köhler	- Online Service - 2.2 Erweiterung der Begriffe und Beschreibungen - 2.3 Aktualisierung der Formate
	28.08.03	Carolin Köhler	- Updates - Viewer Teil1
	29.08.03	Carolin Köhler	- Viewer Teil2 - Document - Demonstration Klassen
	01.09.03	Carolin Köhler	- Page - PageSegment - Loader
	02.09.03	Carolin Köhler	- Loader Beschreibung: Ressourcen - ResourceLoader im Allgemeinen - Annotations Hierarchie
	08.09.03	Carolin Köhler	- RenderContext
	09.09.03	Carolin Köhler	- ResourceLoader, Hierarchie, Diagramm - ResourceFileLoader - ResourceURLLoader - ResourceGroupLoader - ResourceMultiLoader - LoadListener
	10.09.03	Carolin Köhler	- SeekableInputStreams, Diagramm
	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 Untersektionen - JadiceBookmarkHandler - Beteiligte Klassen - PageSorter
	19.09.03	Carolin Köhler	- EditPanes - PrinterJava2 - NavigatorPanel
	22.09.03	Carolin Köhler	- AddOns, Erzeugung, Aufruf, Integration - Navigator Teil 2 - Lens

Version	Datum	Autor	Änderungen
			- HoverLens - GradationCurveControl - GradationCurve - GradationCurveFileHandler - Demo Klasse überarbeitet
	23.09.03	Carolin Köhler	- Korrektur, Überarbeiten, Abschließendes zu Kapitel 1, 2, 4 - Vorbereiten Kapitel 5, 6
	24.09.03	Carolin Köhler	- Kapitel 6 fertig gestellt - BasicJadicePanel, AbstractJadicePanel - Vorbereiten Kapitel 3
	25.09.03	Carolin Köhler	- Kapitel 5.1 - Kapitel 5.2.1 - Kapitel 5.2.2 - Kapitel 5.2.3 - Kapitel 5.2.4 - Kapitel 5.2.5 - Kapitel 5.2.6 - Kapitel 5.2.7 - Kapitel 5.2.8
	26.09.03	Carolin Köhler	- Kapitel 5.5 - Kapitel 5.5.1 - Kapitel 5.5.2 - Kapitel 5.5.3
	29.09.03	Carolin Köhler	- Kapitel 5.3.1 - Kapitel 5.3.2 - Kapitel 5.3.3 - Kapitel 5.3.4 - Kapitel 7 - Kapitel 7.0.1 - Kapitel 7.0.2 - Kapitel 7.0.3 - Kapitel 7.0.4
	30.09.03	Carolin Köhler	Dokument Korrekturen, Layout - Kapitel 3 - Kapitel 3.1 - Kapitel 3.2 - Kapitel 3.3
	01.09.03	Carolin Köhler	- Überarbeiten Kapitel 4
	02.09.03	Carolin Köhler	abschließende Korrekturen, Layout - Kapitel 5.4 - Kapitel 5.4.1 - Kapitel 5.4.2 - Kapitel 5.4.3
	30.10.03	Carolin Köhler	- Kapitel 6,7 um neue Einstellungen erweitert
	04.11.03	Carolin Köhler	- Kapitel 7 um neue Einstellungen erweitert
	22.12.03	Carolin Köhler	- neues Template - Updates und Erweiterungen des jadice Pakets der Dokumentation hinzugefügt

Version	Datum	Autor	Änderungen
	19.02.04	Carolin Köhler	- verschiedene kleinere Korrekturen - DocumentSaver
	02.03.04	Carolin Köhler	- Konfigurationsparameter aktualisiert
	19.03.04	Carolin Köhler	- Querverweise aktualisiert
	26.08.04	Carolin Köhler	- Konfigurationsparameter aktualisiert
	15.09.04	Carolin Köhler	- Konfigurationsparameter aktualisiert
2.0	06.12.05	Jelkica Ćirilović Carolin Köhler	Komplett überarbeitet
	28.02.07	F. Fernandes C. Köhler J. Ćirilović	Aktualisiert für jadice version 3.1
4.1	31.03.08	C. Köhler J. Ćirilović	Aktualisiert für jadice version 4.1
4.2	18.02.09	C. Köhler J. Ćirilović	Aktualisiert für jadice version 4.2