

MASTERARBEIT

im Labor für medizinische Bildverarbeitung, Algorithmen und Krankenhaus IT

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

**Entwicklung einer modularen und erweiterbaren Anwendung
zur medizinischen Bildverarbeitung**

eingereicht von

Rudolf Franz Siegfried Korb, 790060

Erstprüfer

Prof., Dr. Holger Timinger

Zweitprüferin

Prof., Dr. Gudrun Schiedermeier

Abgabetermin

02.03.2014

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer modularen und erweiterbaren Anwendung zur medizinischen Bildverarbeitung für den Einsatz in Forschung und Lehre im „Labor für medizinische Bildverarbeitung, Algorithmen und Krankenhaus IT“.

Es wird eine Architektur entworfen, die eine Integration neuer Funktionen, sogenannter Module, erlaubt und eine dynamische Erweiterung des Systems zur Laufzeit zulässt. Die Basis der Software besteht aus der „Eclipse Rich Client Platform“ und wird mit Hilfe gängiger Architektur- und Entwurfsmuster ergänzt, um eine flexible Anwendungsstruktur zur Verfügung zu stellen. Zur Verarbeitung der medizinischen Bilder wird der DICOM-Standard eingesetzt, damit die Bilddaten angezeigt und Manipuliert werden können.

Ergebnis der Arbeit ist die Software „jMediKit - Java Medical Imaging Toolkit“ die sich durch Module und Plug-ins erweitern lässt und Funktionen zur Navigation und Orientierung in dreidimensionalen medizinischen Bildern bietet. jMediKit liefert das grundlegende System, das sich sowohl von Entwicklern durch Module, als auch von Anwendern durch Plug-ins erweitern lässt.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Der sechste Kondratieff-Zyklus	1
1.2. Wachstumsmarkt Gesundheit	3
1.3. Der Studiengang Biomedizinische Technik	4
1.4. Das Labor für medizinische Bildverarbeitung, Algorithmen und Krankenhaus IT	4
I. Anforderungen und Theoretische Grundlagen	6
2. Anforderungen an eine modulare und erweiterbare Bildverarbeitungssoftware	7
2.1. Einführung	7
2.2. Nichtfunktionale Anforderungen	7
2.3. Funktionale Anforderungen	8
2.4. Evaluierung bestehender Software	9
2.5. Slicer 3D	10
2.6. ImageJ	11
2.7. Ein Vergleich der verfügbaren Anwendung mit den Anforderungen der Stakeholder	12
3. Grundlagen medizinischer Daten- und Bildformate	14
3.1. DICOM	14
3.1.1. Die DICOM Information Object Definitionen	16
3.1.2. Von Patientendaten der realen Welt zu digitalen DICOM-Objekten	18
3.1.3. Tags in Datenelementen	20
3.1.4. VR - Value Representation	20
3.1.5. VM - Value Multiplicity	20
3.2. DICOM Pixeldaten und Bildformate	21

Inhaltsverzeichnis

3.2.1. Kodierung der Pixel im Speicherabbild einer DICOM-Objekts	21
3.2.2. Grauwertbilder	24
3.2.3. Farbbilder	26
3.3. 3D Bilddaten	27
3.4. Bilder mit zeitlicher Abhangigkeit	27
4. Grundlagen zu Entwurfsmustern der Softwareentwicklung	28
4.1. Strukturmuster	29
4.2. Verhaltensmuster	30
4.2.1. Observer	30
4.2.2. Schablonenmethode	31
4.3. Erzeugungsmuster	32
4.3.1. Fabrik Methode	32
4.3.2. Singleton	33
4.4. Das Architekturmuster Plug-in	34
II. Entwicklung des Java Medical Imaging Toolkit	36
5. Softwarearchitektur des Java Medical Imaging Toolkit	37
5.1. Die Eclipse Rich Client Platform	37
5.2. Das Eclipse Application Model	39
5.2.1. Visuelle Komponenten	39
5.2.2. Steuernde Komponenten	40
5.3. Die Benutzeroberfache von jMediKit	41
5.4. Erweiterbarkeit der Grundstruktur	43
5.5. Modulare Werkzeuge	44
5.6. Die Plug-in Architektur	45
5.6.1. Plug-in als Grundstruktur	46
5.6.2. Erweiterung mit der Schablonenmethode	47
5.6.3. Singleton als Plug-in Manager	47
5.6.4. Plug-ins mit dynamischer Parameterubergabe	47
5.7. Externe Bibliotheken	48
5.7.1. dcm4che	49
5.7.2. Simple ITK	49
5.7.3. Der Adapter zur Auflosung von Abhangigkeiten	50

Inhaltsverzeichnis

5.8.	Die Architektur der Bilddaten	51
5.8.1.	Die einfache Fabrik	52
5.8.2.	Struktur der Bilder im ImageViewPart	53
6.	Implementierung	56
6.1.	Implementierung der DICOM-Objekte	56
6.2.	Der DicomBrowser	57
6.3.	Repräsentation der Pixeldaten	59
6.4.	Räumliche Sortierung der Bilddaten	60
6.5.	Zeichnen und manipulieren der Bilddaten	63
6.5.1.	Die Rekonstruktion der Ebenen	66
6.5.2.	Berechnung der Bildkoordinaten	68
6.5.3.	Bilineare Interpolation	71
6.5.4.	Anzeige der Zusatzinformationen	73
6.5.5.	Implementierung der Werkzeuge	77
6.6.	Dynamische Parametervergabe	82
6.7.	Debugging der eigenen Plug-ins	84
6.7.1.	Dateibasiertes Debugging	85
6.7.2.	Visuelles Debugging	85
7.	Entwicklung von Erweiterungen	86
7.1.	Entwicklung von Plug-ins	86
7.1.1.	Konventionen	86
7.1.2.	Die Klassenstruktur eines Plug-ins	88
7.1.3.	Nützliche Funktionen	90
7.1.4.	Der Laplace-Operator	95
7.1.5.	Der Sobel-Operator	97
7.1.6.	Der ConnectedThresholdImageFilter aus dem SimpleITK	98
7.2.	Erweiterung der Anwendungsstruktur mit dem Eclipse Application Model .	101
7.3.	Erweiterung der Werkzeuge	106
7.3.1.	Hinzufügen eines Menüpunktes	107
7.3.2.	Hinzufügen der Funktionalität	110
7.3.3.	Hinzufügen der Werkzeuglogik	112
8.	Diskussion	114
9.	Fazit	116

Inhaltsverzeichnis

Literaturverzeichnis	I
Abbildungsverzeichnis	IV
Tabellenverzeichnis	VII
Listings	VIII
III. Anhang	IX
A. Darstellung der DICOM-Elemente im Speicher	X
A.1. Explizite VR mit [OB OW OF SQ UT UN]	X
A.2. Explizite VR	X
A.3. Implizite VR	X
B. Installation der Eclipse e4 Umgebung	XIV
B.1. Eclipse	XIV
B.2. Eclipse e4 Tools	XV
B.3. Import der jMediKit Projektdateien	XIX
B.4. Installation des Java Advanced Imaging Image I/O Tools	XXIII
B.5. Erstellung eines neuen Builds von jMediKit	XXIV
C. Ikonenverwaltung innerhalb der Anwendung	XXVII
D. Die Verwendung von Eclipse zur Plug-in-Entwicklung	XXIX

1. Einleitung

Ziel dieser Abschlussarbeit ist die Entwicklung einer Software zur medizinischen Bildverarbeitung. Eine modulare und flexible Architektur soll eine leichte Erweiterbarkeit sowohl für Entwickler (Erstellen eigener Werkzeuge und Module zur Grundfunktionserweiterung) als auch den Anwender (Integration selbst implementierter Bildverarbeitungsalgorithmen) ermöglichen.

Das Programm soll für Forschung und Lehre im Labor für medizinische Bildverarbeitung, Algorithmen und Krankenhaus IT eingesetzt werden. Nach einer Vorstellung des Labors und dem zugehörigen Studiengang der biomedizinischen Technik erfolgt die Ermittlung der Anforderungen der Software. Aufbauend werden die Grundlagen medizinischer Daten- und Bildformate dargestellt. Der Fokus dieses Grundlagenkapitels liegt auf dem DICOM-Standard und den Eigenschaften der medizinischen Bilddaten. Die folgenden Kapitel erläutern die Softwarearchitektur sowie die Implementierung. Praktische Anwendungsbeispiele schließen die technischen Aspekte der Arbeit ab. Die folgende Diskussion zeigt Erweiterungsmöglichkeiten für die zukünftige Entwicklung auf.

1.1. Der sechste Kondratieff-Zyklus

Im Jahr 1926 veröffentlichte der Wirtschaftswissenschaftler Nikolai D. Kondratieff (* 1892, †1938) die Theorie „Die Langen Wellen der Konjunktur“ [HK11]. Leo Nefiodow erweiterte 2006 die Theorie, um die Entwicklung des 20. Jahrhunderts mit einfließen lassen zu können. Kondratieff zeigte, dass sich die gesellschaftliche Wandlung nicht willkürlich vollzog. Seit der Industrialisierung Mitte des 18. Jahrhunderts stand der Wohlstand der Gesellschaft in direkter Beziehung zu besonderen Erfindungen. Er betrachtete die Phasen des Wohlstandes und die der direkt folgenden Wirtschaftskrisen und entdeckte die später nach ihm benannten „Kondratieff-Zyklen“. Wie in Abbildung 1.1 zu sehen ist, war die Dampfmaschine die erste Basisinnovation¹ und revolutionierte die Textilindustrie.[Wie12]

¹Basisinnovationen müssen nach Nefiodow vier Eigenschaften erfüllen: Entstehung eines neuen Marktes mit vielen Arbeitsplätzen; Innovation bestimmt den Zyklus; Basisinnovationen haben einen Zyklus von 40 - 60 Jahren; Sie bestimmen die Entwicklungsrichtung

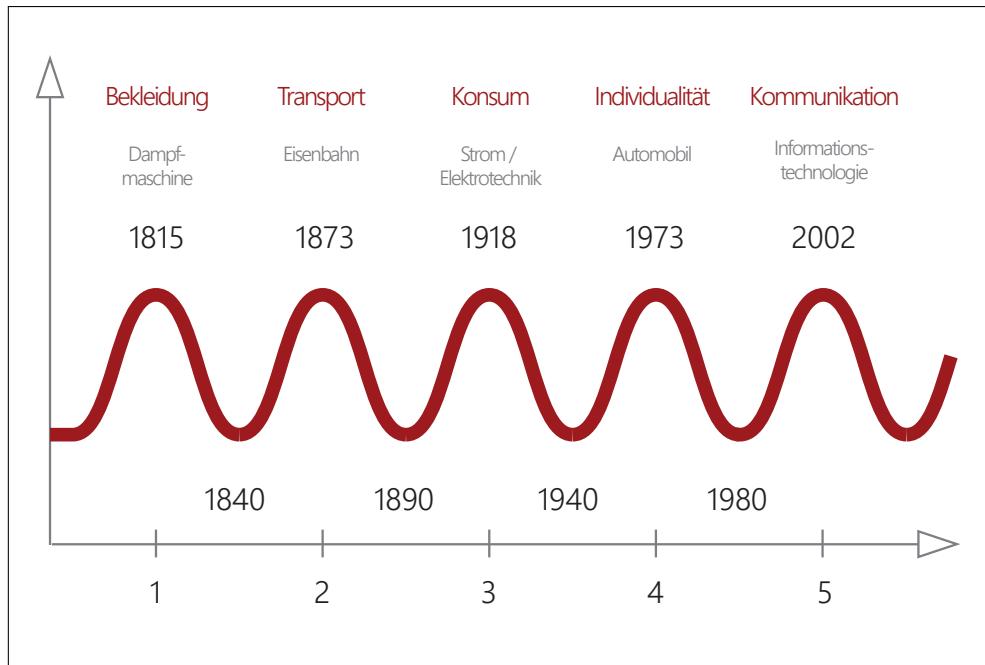


Abbildung 1.1.: Kondratieff-Zyklen [HK11, S.32]

Diese Erfindung gilt als Beginn des ersten Kondratieff-Zyklus. Vor dem maschinellen Betrieb wurden Spinnräder noch manuell bedient und Kleidung war teuer. Die dampfbetriebenen Webstühle steigerten die Effizienz um das 200-fache. In den 20er Jahren stagnierte die Branche, da die Rohstoffbeschaffung und Warenverteilung das Maximum der Effizienz erreicht hatte. Mit der Erfindung der Eisenbahn gelang der Übergang vom ersten in den zweiten Zyklus. In den folgenden Jahren konnte nun das Bedürfnis nach verbesserten Transportmöglichkeiten gestillt werden.

Dampfmaschine, Eisenbahn, Strom, Motor und der Mikrochip stehen alle für eine Basisinnovation, die zukünftige Gesellschaften geprägt haben. Im Lauf der Zeit verschwinden die Erfindungen aus dem Bewusstsein der Menschen und werden zu Gegenständen des Alltags. Motor und Mikrochip sind so stark im gesellschaftlichen Leben verankert, dass sie nicht mehr direkt wahrgenommen werden. Betrachtet man eine elektrische Zahnbürste, ist es selbstverständlich, dass die Energie aus dem Stromnetz bezogen und der Bürstenkopf von einem Motor angetrieben wird.

Das Jahr 2002 gilt als Höhepunkt des fünften Kondratieff-Zyklus und die Gesellschaft befindet sich gerade im Übergang zum Sechsten. Noch fehlt die aktuelle Basisinnovation und auch das nächste zu stillende Bedürfnis ist nach Individualität und Kommunikation

(Abbildung 1.1) noch nicht bestimmt.

Nach Nefiodow [GN11] gibt es vier Möglichkeiten welcher Markt in Zukunft den sechsten Kondratieff prägen wird:

- **Informationsmarkt**

Mobile Geräte und Soziale Netzwerke sind maßgebend für diesen Markt. So verhalf der Kurznachrichtendienst Twitter zum sogenannten „Arabischen Frühling“ durch die blitzschnelle Kommunikation über das Netz².

- **Bio - und Nanotechnologie**

Die Erfindung des Mikroskops und die Entschlüsselung der DNA im Jahr 2000 gilt als Basisinnovation. Anfangs wurden die Erkenntnisse nur in Medizin und Pharmazie angewendet. Heute profitiert auch die Landwirtschaft und Lebensmittelindustrie davon.

- **Umwelttechnologie**

Auch der Bereich Umwelttechnologie sorgte für einen Zuwachs an Arbeitsplätzen in Deutschland und Österreich. Zusätzlich siedelten sich Unternehmen besonders in ländlichen und strukturschwachen Gebieten an. [GN11, S. 107].

- **Gesundheit**

Der Gesundheitsmarkt vereint technologische Komponenten wie die Medizintechnik und psychosoziale Gesundheit. Es erfolgt ein Wechsel vom heutigen „Krankheitswesen“ zum Gesundheitswesen, angefangen von der Burnout-Prophylaxe und dem Gesundheitstourismus, bis zur Bionik und künstlichen computergesteuerten Prothesen.

1.2. Wachstumsmarkt Gesundheit

Nach Granig[GN11] ist der Gesundheitsbereich der derzeit am schnellsten wachsende Markt³. Die Bevölkerung ist gewillt in die eigene Gesundheit zu investieren und die Unternehmen positionieren sich im Gesundheitsbereich (Siemens beispielweise verstärkt sich im Bereich der Medizintechnik [GN11, S.111, 112]) Die Bio- und Nanotechnologie und die Medizintechnik ähneln sich in einigen Bereichen. Sowohl Siemon Cord [Cor07] als auch Granig [GN11, Seite 116 f] sprechen davon, dass der Markt sich nur gehemmt entwickeln

²<http://www.heise.de/tr/blog/artikel/Wie-funktioniert-die-Twitter-Revolution-1761481.html>
aufgerufen am 06.01.2014

³Gemessen am Anteil der Branche am Bruttoinlandsprodukt

kann. Grund dafür sind, sowohl für die Nano- als auch Medizintechnik, veraltete Vorschriften und auch ethische Hürden die es zu überwinden gilt. Granig nennt hierzu unter anderem die Mensch-Computer-Einheit [GN11, S.117](computergesteuerte Prothesen und künstliche Ersatzteile).

Die Entwicklung für den wachsenden Markt findet in den Unternehmen statt. Der Grundstein für Innovation wird jedoch bei den Studierenden der Hochschulen und Universitäten gelegt. Die Bildungseinrichtungen werden ein zentrales Standbein für den kommenden sechsten Kondratieff mit einem Schwerpunkt Bio-, Medizintechnik und Gesundheit sein.

1.3. Der Studiengang Biomedizinische Technik

In einem Onlineartikel vom Februar 2012⁴ veröffentlichte die Hochschule Landshut, dass ab dem Wintersemester 2012 der neue Bachelorstudiengang „Biomedizinische Technik“ angeboten wird. Der Artikel beschreibt, ähnlich wie Granig, die Medizintechnik als Wachstumsmarkt und bestätigt durch die Einführung des Studiengangs das gesellschaftlich gestiegerte Interesse am Gesundheitswesen.

Während des Studienverlaufs [Hoc13] erwerben die Studierenden vor allem im zweiten Studienabschnitt Kenntnisse im Bereich der Medizintechnik. Die Ausbildung behandelt unter anderem bildgebende Systeme, medizinische Bildverarbeitung und minimalinvasive Therapieverfahren.

Für die Ausbildung stehen Labore mit den benötigten Geräten zur Verfügung, um mit dem theoretischen Wissen praktisch zu experimentieren.

1.4. Das Labor für medizinische Bildverarbeitung, Algorithmen und Krankenhaus IT

Das Labor erfüllt zwei Aspekte. Die Ausstattung steht für Forschungs- und Entwicklungsarbeiten zur Verfügung. Gemeinsame Forschungsprojekte von Hochschule und Krankenhäusern oder Unternehmen können hier umgesetzt werden. Für die Lehre soll Studierenden die Möglichkeit geboten werden, den Prozess der medizinischen Bildverarbeitung anschaulich und praxisnah zu erleben. Mittels Doppler-Ultraschallgerät können Bilddaten

⁴<https://www.haw-landshut.de/aktuelles/news/news-archiv/news-detailansicht/article/neuer-studiengang-biomedizinische-technik-vielfältige-berufschancen.html>
abgerufen am 10.01.2014

erzeugt und anschließend an das Picture Archiving and Communication System⁵ (PACS) gesendet werden. Anschließend können Algorithmen zur Bildvorverarbeitung, Merkmalsextraktion oder auch Segmentierung implementiert und getestet werden.

Medizinische Bildformate unterscheiden sich maßgeblich von allgemein bekannten Formaten wie JPEG oder Bitmaps, daher sind zur Betrachtung sogenannte DICOM-Viewer⁶ notwendig. So enthalten medizinische Formate neben den reinen Bilddaten noch viele weitere Informationen, zum Beispiel Patientendetails oder die tatsächliche Lage des Patienten in der realen Welt zum Zeitpunkt der Aufnahme. Mit Hilfe der DICOM-Viewer lassen sich die erzeugten Bilder betrachten und grundlegende Operationen anwenden (dazu zählt beispielsweise die Skalierung oder Verschiebung des Bildes). Komplexe Bildverarbeitungsalgorithmen können allerdings nicht ausgeführt oder selbst implementiert werden.

Für Forschung und Lehre wird eine Software benötigt, die sowohl die Grundfunktionen der Betrachtung liefert, als auch eine Schnittstelle zur eigenen Erweiterung zu Verfügung stellt.

⁵Ein PACS dient als zentraler Bildspeicher, der über das Netzwerk angesprochen werden kann. Medizinische Geräte legen dort die Bilddaten ab, während die Software zur Betrachtung die Daten vom PACS holt.

⁶DICOM (Digital Imaging and Communications in Medicine) ist der heutige Standard der medizinischen Informationsverarbeitung und wird in den folgenden Kapiteln näher erläutert. Die Viewer ermöglichen die Betrachtung der Bilddaten.

Teil I.

Anforderungen und

Theoretische Grundlagen

2. Anforderungen an eine modulare und erweiterbare Bildverarbeitungssoftware

2.1. Einführung

In diesem Kapitel werden die Anforderungen an eine modulare und erweiterbare Bildverarbeitungssoftware gezeigt. In mehreren Besprechungen der Stakeholder¹ wurden die Anforderungen ermittelt. Diese werden in die beiden Bereiche *Nichtfunktionale Anforderungen* und *Funktionale Anforderungen* eingeteilt. Nach einer Evaluierung bestehender Anwendungen wird entschieden, ob eine eigene Software implementiert werden soll.

2.2. Nichtfunktionale Anforderungen

Nach Balzert [Bal11, 9] beschreiben nichtfunktionale Anforderungen, *wie* ein System arbeitet. Sie üben besonderen Einfluss auf die Softwarearchitektur eines Systems aus. Während der Besprechungen mit dem Laborleiter konnten folgende nichtfunktionale Anforderungen ermittelt werden.

- **Modularer Aufbau des Systems(1)**

Die Software soll in den Grundfunktionen erweiterbar sein. So soll es möglich sein, neue Bedienelemente der Benutzeroberfläche und neue Funktionen der Anwendung hinzuzufügen.

- **Erweiterbarkeit durch den Anwender(2)**

Anwender sollen die Möglichkeit haben, das Programm mit selbst programmierten Algorithmen und Plug-ins zu erweitern. Die eigene Implementierung von Bildverarbeitungsprozessen ist essentiell im Bereich der Lehre.

¹Ein Stakeholder beschreibt einen Teilnehmer eines Projektes.

- **Die Programmiersprache Java(3)**

Die Studierenden des Studiengangs sollen im Modul „medizinische Bildverarbeitung“ ihre Kompetenzen im Bereich der Programmierung mit Java erweitern. Dadurch soll die Anwendung in Java umgesetzt und erweiterbar sein.

- **Externe Bildverarbeitungsbibliotheken(4)**

Algorithmen in der Bildverarbeitung sind oft komplex und umfangreich. Nicht jeder benötigte Verarbeitungsprozess eignet sich zum selbst implementieren (sowohl im Lehr- als auch Forschungsbereich). Durch den Einsatz von Bibliotheken wird ein grundlegender Satz an Algorithmen vorgegeben, auf den der Benutzer zurückgreifen und in den Plug-ins verwenden kann.

- **Verarbeitung medizinischer Daten mit Hilfe des DICOM-Standards(5)**

Medizinische Bilddaten besitzen neben den rohen Pixeldaten noch eine Vielzahl zusätzlicher Information, wie Patientendaten oder Seriennummern der Aufnahmen und benötigen eine spezielle Verarbeitung. Anders als übliche Grauwertbilder besitzen DICOM-Pixeldaten nicht zwingend 255 sondern bis zu 2^{16} verschiedene Grauwerte.

2.3. Funktionale Anforderungen

Dieser Anforderungstyp legt fest, *was* das Programm leisten soll und bestimmt Funktionen und Verhalten [Bal11, 9]. Sie überschneiden sich oder stehen in Beziehung zu den nichtfunktionalen Anforderungen.

- **Dynamische Parameterübergabe bei einer Plug-in-Ausführung(6)**

Aus der Anwendererweiterbarkeit ergibt sich eine funktionale Anforderung. Nicht immer können alle Eigenschaften und Werte der Algorithmen während der Implementierung vom Anwender bestimmt werden. Durch die Abhängigkeit von den zu bearbeitenden Bilddaten und Algorithmen muss die Möglichkeit geboten werden, Parameter während der Laufzeit der Anwendung zu bestimmen.

- **Manuelle Auswahl signifikanter Punkte zur Verarbeitung in Plug-ins(7)**

Zusätzlich zu den dynamischen Parametern müssen einzelne Bildpunkte interaktiv vom Benutzer ausgewählt und später von Algorithmen benutzt werden können. Manche Bildverarbeitungsprozesse benötigen beispielsweise sogenannte Saatpunkte die vom Anwender manuell ausgewählt werden müssen.

- **Anwendung der Algorithmen im dreidimensionalen Raum(8)**

Eine Vielzahl an Bildaufnahmen liegen als dreidimensionaler Datensatz vor. Eine Bildreihe muss folglich zusätzlich zur xy-Ebene auch in z-Richtung zu bearbeiten sein.

- **Darstellung mehrerer Bilddatensätze(9)**

Das Programm soll mehrere dreidimensionale Bildreihen zur selben Zeit in verschiedenen Fenstern anzeigen können. Das erlaubt den Benutzern die Datensätze in Beziehung zueinander zu betrachten.

- **Darstellung aller Bildebenen(10)**

Ein dreidimensionaler Datensatz macht es möglich, nicht nur die Bilder der xy-Ebene (Axial) sondern auch die xz-(Coronal) sowie yz-Ebene(Sagittal) darzustellen. Zwischen den Ansichten soll der Benutzer wechseln können.

- **Dynamische Anzeige eines Punktes in allen Bildebenen(11)**

Sind drei gleiche Datensätze in drei verschiedenen Fenstern geöffnet, sollen diese dynamisch reagieren wenn der Benutzer einen Punkt eines Datensatzes auswählt. Angenommen ein Datensatz ist in axialer, coronaler und sagittaler Ansicht geöffnet. Wählt der Nutzer einen Bildpunkt aus der axialen Ansicht, sollen die beiden übrigen geöffneten Fenster aktualisiert werden und den gleichen Punkt in ihrer jeweiligen Ansicht referenzieren.

- **Anzeige der dreidimensionalen Datensätze in (x, y, z) (12)**

Damit die Darstellung der verschiedenen Ebenen möglich wird, muss der gesamte dreidimensionale Datensatz zur Anzeige zur Verfügung stehen. Der Anwender soll durch die einzelnen Schichten der Bildreihen scrollen können.

2.4. Evaluierung bestehender Software

Frei verfügbare Software im medizinischen Bereich beschränkt sich oft auf die vorgegebenen Funktionen des Programms und bietet keine Möglichkeiten der Erweiterung. Zusätzlich liegt der Fokus auf der Darstellung der Patientenbilder und weniger an den Algorithmen zur Bildverarbeitung. Abbildung 2.1 zeigt den Screenshot des DicomViewers RadiAnt². Die Bilder können einzeln, oder wie auf dem Bild zu sehen, im Bezug zueinander betrachtet werden. Die Werkzeuleiste oben ermöglicht die für DICOM-Bilder typischen

²<http://www.radiantviewer.com/de/>

Anforderungen an eine modulare und erweiterbare Bildverarbeitungssoftware

Operationen. Zwar gibt es auf dem Markt auch Open-Source Lösungen mit Schwerpunkt auf Bildverarbeitung, jedoch eignen sich diese nur bedingt für den Einsatz in der Lehre. Die Programme bieten eine Vielzahl an Funktionen, allerdings benötigt die Entwicklung von Erweiterungen einen erheblichen Zeitaufwand.

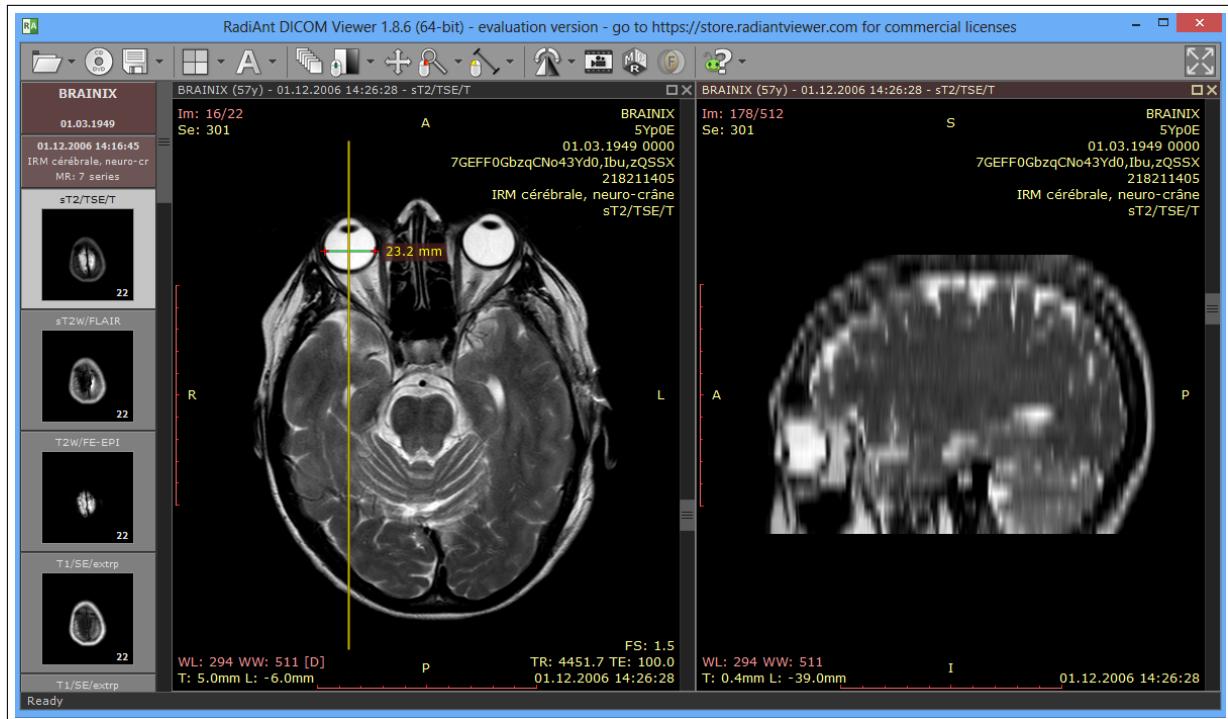


Abbildung 2.1.: RadiAnt - DicomViewer

2.5. Slicer 3D

Slicer 3D³ (Abbildung 2.2) ist ein umfassendes Werkzeug für die medizinische Bildverarbeitung im zwei- und dreidimensionalen Raum. Die quelloffene Software bietet Möglichkeiten eigene Erweiterungen zu implementieren. Slicer verwendet als Bibliotheken unter anderem das Insight Segmentation and Registration Toolkit und das Visualization Toolkit⁴. Die Erweiterungen werden in Python implementiert und dienen zur Anwendung der Bildverarbeitungsalgorithmen. Es fehlt die Möglichkeit der modularen Erweiterung

³<http://www.slicer.org>

⁴Insight Segmentation and Registration Toolkit (ITK) und Visualization Toolkit (VTK) sind umfassende in C++ entwickelte Programmbibliotheken zur medizinischen Bildverarbeitung und Visualisierung.

bezüglich neuer Funktionen. Das Labor setzt in naher Zukunft ein PACS ein und die zu entwickelnde Software soll Möglichkeiten bieten, dieses System später zu integrieren, um DICOM-Daten über das Netzwerk zu beziehen.

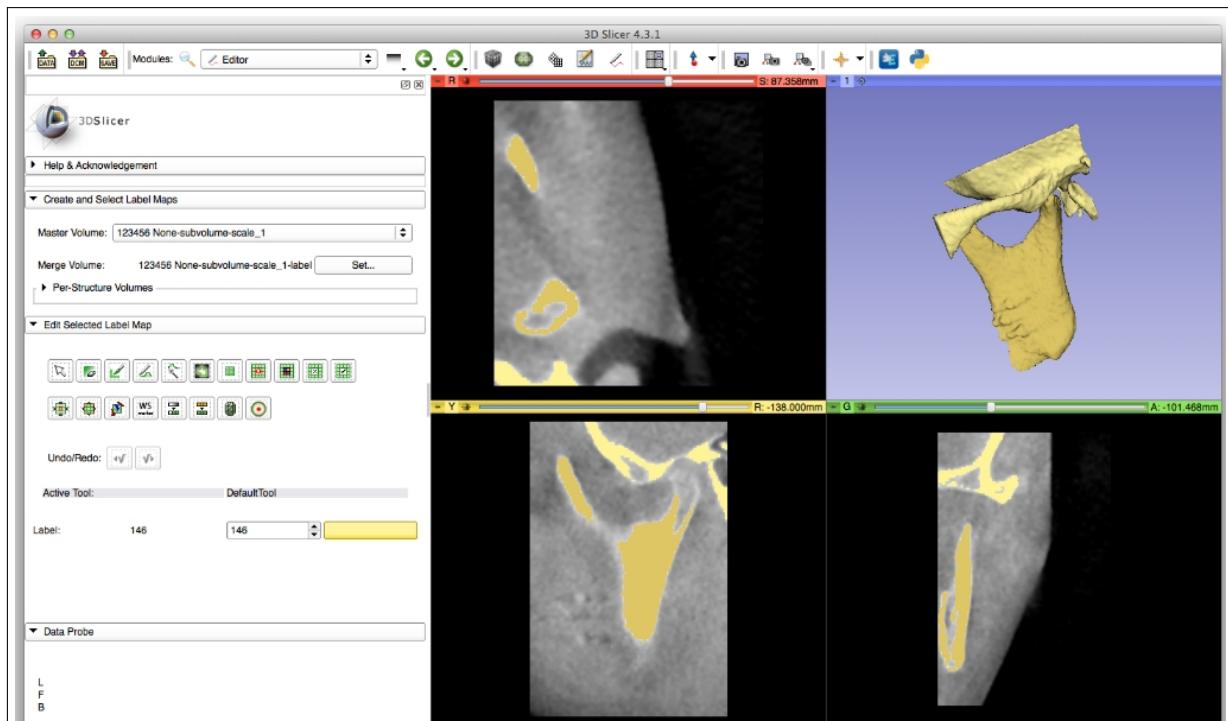


Abbildung 2.2.: Screenshot des Programms Slicer 3D [Far14]

2.6. ImageJ

ImageJ ist „State Of The Art“ im Bereich der Bildverarbeitung in Java⁵). Im Grundzustand liefert ImageJ die Standardfunktionen der Bildverarbeitung wie Abbildung 2.3 zeigt. Unter anderem kann das Bildmaterial analysiert oder mit Filtern bearbeitet werden. ImageJ verarbeitet sowohl Grauwertbilder als auch Farbbilder in den gängigen Formaten wie PNG, JPEG und vielen anderen. Mit Hilfe der „ImageStacks“ ist auch eine Bearbeitung im dreidimensionalen Bildraum möglich. Erweiterungen können schnell und zielgerichtet entwickelt werden. Im Modul „Bildverarbeitung“ der Fakultät Informatik wird ImageJ als Standard zum Bearbeiten der Übungsaufgaben verwendet. Für einen Einsatz im Studiengang Biomedizinische Technik fehlt allerdings die grundlegende Unterstützung

⁵<http://rsbweb.nih.gov/ij/>

von medizinischen Bilddaten im DICOM-Format. Die Funktionalität lässt sich über Plugins nachträglich ergänzen, allerdings fehlt eine Bibliothek die bereits implementierte Algorithmen zur Verfügung stellt, sowie eine Verknüpfung von ImageJ-Klassen wie FloatProcessor oder ByteProcessor in Bildformate der Bibliotheken. So müssten ImageJ-Bilder in fremde Bildtypen konvertierbar sein.

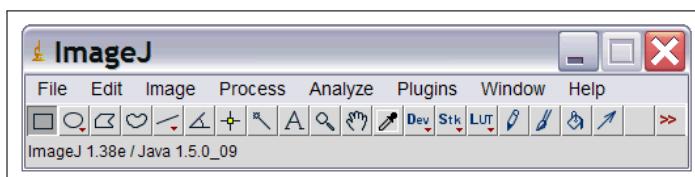


Abbildung 2.3.: Die Benutzeroberfläche von ImageJ

2.7. Ein Vergleich der verfügbaren Anwendung mit den Anforderungen der Stakeholder

Tabelle 2.1 zeigt eine Gegenüberstellung der evaluierten Software zu den Anforderungen. Slicer3D und ImageJ erfüllen einen hohen Grad davon, allerdings fehlen wichtige sogenannte harte⁶ Anforderungen. Slicer3D lässt einen einfachen Ansatz zur modularen Erweiterung der Benutzeroberfläche und Anwendungsfunktionen vermissen. ImageJ hingegen bietet diese Variabilität. Allerdings ist die Verarbeitung von DICOM-Daten nur über ein Plug-in möglich und es fehlt eine Grundlage an Algorithmen zur medizinischen Bildverarbeitung. Zusätzlich zu den vorgestellten Anwendungen bietet auch MATLAB⁷ umfangreiche Bildverarbeitungskomponenten, unter anderem auch die Verarbeitung von DICOM-Objekten an. Die hohen Lizenzkosten sprechen allerdings deutlich gegen einen Einsatz in der Lehre, da Anwendungen nur auf begrenzt vielen Maschinen installiert werden können und dadurch die Zahl der Arbeitsplätze für Studierende erheblich unnötig eingeschränkt ist. Aufgaben können nur im Labor bearbeitet werden, was voraussetzt, dass das Labor zu jeder Zeit geöffnet sein muss. Da in den vorgestellten Anwendungen keine Lösung verfügbar ist, die alle Voraussetzungen erfüllt, soll eine Software entwickelt werden, die für das Labor für medizinische Bildverarbeitung, Algorithmen und Krankenhaus IT die benötigten funktionalen und nichtfunktionalen Anforderungen umsetzt.

⁶Es wird zwischen harten und weichen Anforderungen unterschieden. Harte Anforderungen müssen erfüllt werden, während Weiche erfüllt werden können [Bal11, 9]

⁷<http://www.mathworks.de/products/matlab/>

	RadiAnt	Slicer3D	ImageJ
modularer Aufbau (1)	✗	✗	✓
Java (3)	✗	✗	✓
multiple Datensätze anzeigen (9)	✓	✓	✓
Bildanzeige in (x, y, z) (12)	✓	✓	✓
Ansicht der Ebenen (xy, xz, yz) (10)	✓	✓	✗
Dynamische Anzeige der Punkte(11)	✓	✓	✗
DICOM (5)	✓	✓	✗
Plug-ins (2)	✗	✓	✓
Bibliotheken zur Bildverarbeitung (4)	✗	✓	✗
Dynamische Parameter (6)	✗	✓	✓
Manuelle Punktauswahl (7)	✗	✓	✓
Algorithmen in (x, y, z) (8)	✗	✓	✓

Tabelle 2.1.: Gegenüberstellung der Anforderungen und verfügbarer freier Software

3. Grundlagen medizinischer Daten- und Bildformate

3.1. DICOM

Der Name DICOM steht für *Digital Imaging and Communication in Medicine*. Der Umgang mit diesem Standard ist essentieller Bestandteil der zu entwickelnden Software. Pianykh[Pia08, 1] beschreibt, dass DICOM nicht nur aus Pixel und deren zugehörigen Werten besteht. Wie der Name sagt, ist auch die Kommunikation fest im Standard verankert. Damit ist die Übertragung der Daten von medizinischen Geräten(Modalitäten) zum zentralen Speicher und deren Verteilung gemeint. Des weiteren spielt die dauerhafte Speicherung der digitalen Aufnahmen eine große Rolle. Daher wird im gleichen Zug mit DICOM immer ein PACS genannt. Das Akronym PACS bedeutet *Picture Archiving and Communication System* und besteht sowohl aus Hardware (Server, Speicherung) als auch Software(Verteilung und Kommunikation).

Abbildung 3.1 illustriert das Zusammenspiel des DICOM-Standards und dem zentralen Datenspeicher. Zuerst wird mittels der Modalitäten(z.B. mit dem Computertomographen oder einem Ultraschallgerät) die digitale Aufnahme erzeugt. Danach wird das Bild vom Gerät an das PACS gesendet. Hier werden die Aufnahmen und Patientendetails in die Datenbank und den Speicher abgelegt. Wird eine Aufnahme benötigt, können Clients Anfragen mit dem Patientennamen stellen und erhalten darauf die zugehörige Serie mit den digitalen Aufnahmen.

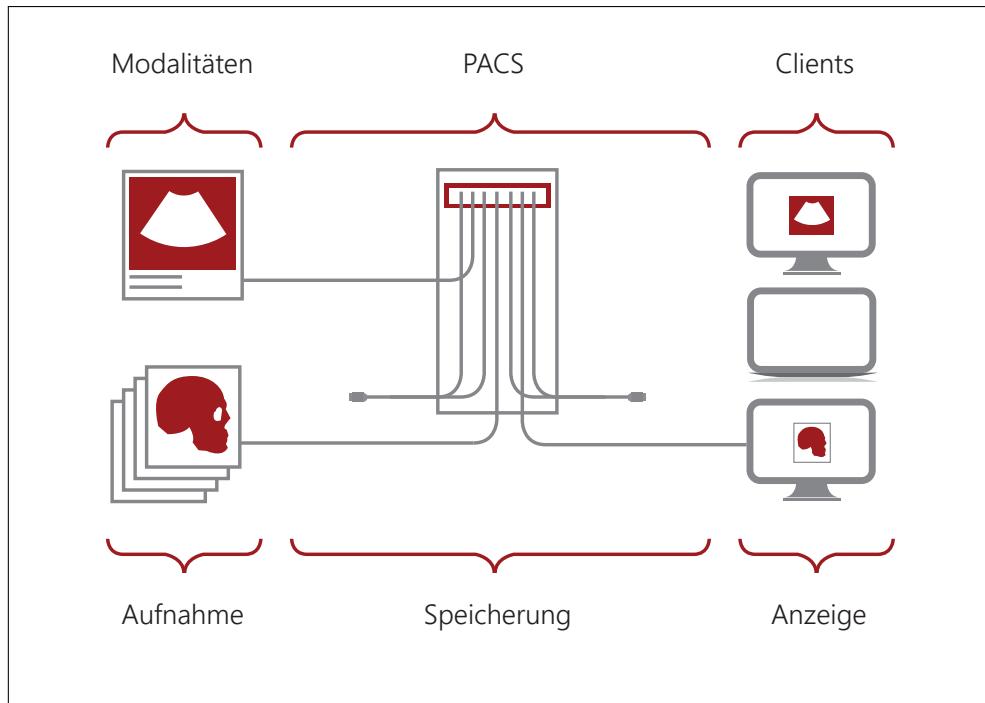


Abbildung 3.1.: Kommunikationsprozess von Aufnahme zur Verarbeitung [Pia08, S. 19]

DICOM¹ ist daher nicht nur ein einzelner Standard, sondern verknüpft die standardisierte

- Kommunikation
- Erzeugung der Bilddaten
- und Speicherung

Im Rahmen dieser Abschlussarbeit wird zur Softwareentwicklung nur der Standard der Bilddaten verwendet. Daher wird in den Grundlagen nur auf die bildspezifischen Eigenchaften von DICOM eingegangen und die Erläuterung von Kommunikation und Speicherung vernachlässigt.

¹Unter <ftp://medical.nema.org/medical/dicom/> lässt sich der aktuelle Standard abrufen. Die Kapitel befinden sich im Ordner zum jeweiligen Jahr der Veröffentlichung. Aktuell sind die Dokumente von 2011.

3.1.1. Die DICOM Information Object Definitionen

Bevor die Pixeldaten genauer betrachtet werden können, muss der prinzipielle Aufbau der DICOM-Objekte beschrieben werden. Teil 3 des Standards[Nat11a, A.1.2] zeigt den relationalen Aufbau der DICOM-Objekte. Vereinfacht können die elementaren Informationsobjekte in drei Teile aufgeteilt werden.

- **Patient**

Der Patient steht in der Hierarchie an oberster Stelle und ist die Grundlage für eine oder mehrere Studien(Study).

- **Study**

Study symbolisiert eine medizinische Studie. Eine Studie ist eine Sammlung von mehreren Serien, die von Modalitäten wie CT und MR aufgezeichnet werden. Eine Studie ist exakt einem Patient zugeordnet.

- **Series**

Eine Serie ist eine Folge von Bildern, die von einer Modalität erzeugt wird. Die Aufnahmen eines CT werden einer Serie zugeordnet. Jede Serie gehört zu nur einer Studie. Eine Serie muss allerdings nicht zwingend aus Bildern bestehen. So können Serien auch aus Messdaten oder Registrierungsdaten zusammengesetzt werden [Nat11a, A.1.2].

- **Image, Real World Values**

Auf der unteren Hierarchiestufe stehen Objekte wie Bilddaten oder die Lage des Patienten im Raum während der Aufnahme. Ein Bild wird genau einer Serie zugeordnet.

Aus diesen vier elementaren Objekten ergibt sich folgende Informationsstruktur für DICOM-objekte, die in Abbildung 3.2 als Entity-Relationship-Modell² verdeutlicht wird.

Der DICOM-Viewer OsiriX bietet auf der Herstellerseite³ die Möglichkeit Testdaten zu beziehen. Betrachtet man die Repräsentation der Daten auf der Festplatte, hält sich die Ordnerstruktur an obiges ER-Modell.

²Ein ER-Modell beschreibt die Beziehungen der Elemente zueinander. Dieser Diagrammtyp wird unter anderem häufig beim Entwickeln der Struktur einer relationalen Datenbank verwendet.

³<http://www.osirix-viewer.com/datasets/>

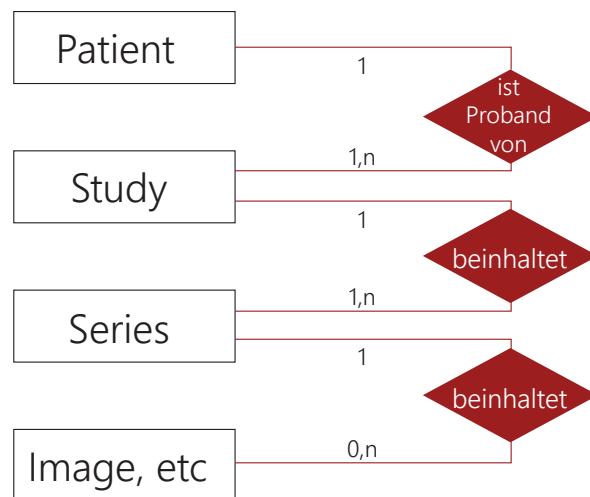


Abbildung 3.2.: Vereinfachte Darstellung der Informationsobjekthierarchie von Dicomelementen [Nat11a, A.1.2]

Abbildung 3.3 zeigt eine schematische Darstellung der Dateien. Die Beispieldaten von OsiriX bestehen aus einem Patienten, dem eine Studie sowie zwei Serien à 100 Aufnahmen zugeordnet werden. Bei näherer Betrachtung fällt auf, dass die Dateinamen beider Serien des Patienten identisch sind. Eine korrekte Zuordnung von DICOM-Dateien zur Serie ist daher nicht immer garantiert. Unabhängig von einer Repräsentation im Dateisystem oder Pfadangaben in der Datenbank eines PACS ist das Vertrauen auf Dateipfade unsicher, da durch eine einfache Dateimanipulation die Zuordnung nicht mehr hergestellt werden kann.

Um eine zuverlässige Verknüpfung zu gewährleisten besitzt jeder Patient⁴, jede Studie und jede Serie eine eindeutige Identifikationsnummer. Diese Art der Informationen wird in den einzelnen Dateien mit Hilfe von Einträgen aus dem DICOM Data Dictionary[Nat11c] hinterlegt. Die DICOM-Dateien beschreibt Pianykh [Pia08, S. 47] als eine Kopie im Speicher vom tatsächlichen DICOM-Objekt.

⁴Die Identifikationsnummern von Patienten sind meist nur innerhalb einer Institution oder Krankenhauses einzigartig, da diese manuell vergeben werden können[Pia08, 5.6.2].

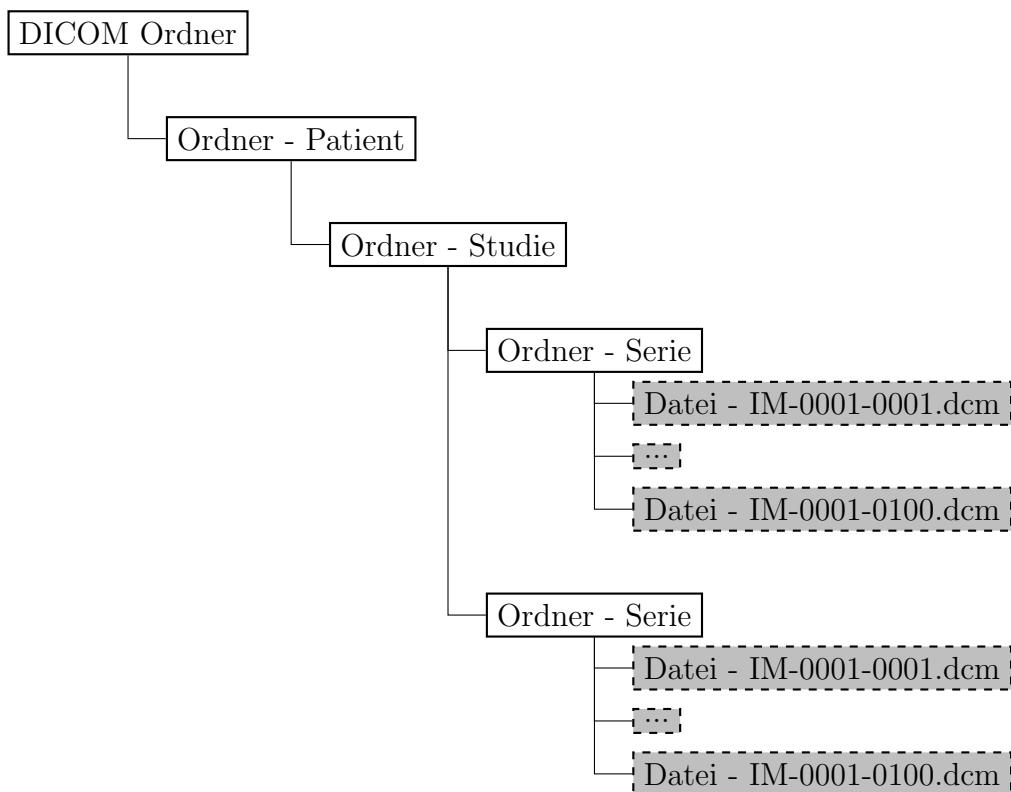


Abbildung 3.3.: Repräsentation der DICOM Informations Objects im Dateisystem

3.1.2. Von Patientendaten der realen Welt zu digitalen DICOM-Objekten

Wie der Name bereits andeutet ist das DICOM Data Dictionary vergleichbar mit einem Wörterbuch. Es enthält alle gültigen Elemente, die zur Beschreibung eines DICOM-Objekts verwendet werden können. Zusätzlich zu dem aus dem Standard bekannten Vokabular können Hersteller medizinischer Geräte ein eigenes Dictionary hinzufügen. Die proprietären Elemente können allerdings nicht standardisiert verarbeitet werden(vgl. [Pia08, S.45], da Software die diese Objekte verarbeitet, nichts von der Existenz dieser Elemente weiß). Aus diesen proprietären Elementen folgt, dass Bilddaten verschiedener Hersteller von Medizinprodukten untereinander nicht kompatibel sind und nicht ausgetauscht werden können.

Mit Hilfe des Wörterbuchs und den ca. 2000 enthaltenen Daten können nun Aussagen des wirklichen Lebens (vorausgesetzt die Aussage ist mit einem Element aus dem Wörterbuch darstellbar) in ein digitales DICOM-Objekt übersetzt werden.

Tabelle 3.1 zeigt das DICOM-Element für den Namen des Patienten aus dem Data Dictionary[Nat11c, S. 14].

Tag	Tagname	VR ^a	Wert	VM ^b
(0010,0010)	PatientName	PN	John^Doe	1

Tabelle 3.1.: Repräsentation des Patientennamen als DICOM-Element

^aValueRepresentation - beschreibt den Werttyp, zum Beispiel numerischer Wert oder Zeichenkette.^bValue Multiplicity - wieviel Werte sind im DICOM-Element enthalten.

Tag (Gruppe, Element)	Tagname	VR	Wert	VM
(0010,0010)	PatientName	PN	John^Doe	1
(0010,0030)	PatientBirthDate	DA	19700101	1
(0010,0040)	PatientSex	CS	M	1
(0010,1010)	PatientAge	AS	44	1

Tabelle 3.2.: Das erzeugte DICOM-Objekt mit den Elementen zu Patientenname, Geburtsdatum, Geschlecht und Alter

Betrachtet man den folgenden Satz(vgl. [Pia08, S.46]), kann dieser in ein DICOM-Object, wie es Tabelle 3.2 darstellt, übersetzt werden:

„John Doe, männlich, geboren am 01. Januar 1970“

Wie aus dem Beispiel von Tabelle 3.1 zu erkennen, lässt sich ein DICOM-Element nochmals in atomare Teile aufspalten. Folglich besteht ein Datenelement aus einem beschreibenden *Tag*, einer *VR*(Value Representation), einem Wert und der *VR* (Value Multiplicity). Das Element selbst nimmt eine von drei Darstellungsmöglichkeiten ein. Zusätzlich liegt im Speicherabbild des Datenelements die Länge des Wertes⁵. Abhängig von der Transfersyntax⁶ des DICOM-Objekts ist der VR-Teil optional. Die weiteren beiden Darstellungen unterscheiden sich in der Kodierung der benötigten Länge des Werts [Nat11b, 7.1]. Anhang A auf Seite X zeigt wie Datenelemente im Speicher abgelegt werden und wie viel Speicherplatz pro Element reserviert werden muss.

⁵John^Doe besitzt aufgrund der Zeichenmenge eine Länge von acht.

⁶Unter der Transfersyntax versteht man eine Menge an Kodierungsvorschriften von DICOM-Objekten[Nat11b, S.63 Section 10]. Zu diesen Vorschriften gehört zum Beispiel die Reihenfolge der Bytes im DICOM-Element oder die Komprimierung der Bilddaten.

3.1.3. Tags in Datenelementen

Ein DICOM-Element wie *PatientName* kann über ein Tag identifiziert werden. Ein Tag ist in einem DICOM-Objekt einzigartig und darf nur ein mal benutzt werden. Die numerische Darstellung hat die Form (*gggg, eeee*) wobei die hexadezimalen Ziffern *g* die Gruppe des DICOM-Elements beschreiben. Das Zeichen *e* beschreibt das Element aus der Gruppe *g*. Wie in Tabelle 3.2 zu sehen, steht 0010 für die Gruppe *Patient* und 0030 für das Element *PatientBirthDate* und ist der Patientengruppe zugehörig. Zusätzlich zu diesen Eigenschaften, kann bestimmt werden, ob der Ursprung eines DICOM-Elements im Standard- oder einem privaten Data Dictionary liegt. Eine gerade Gruppen-Ziffer zeigt, dass das Element Teil des Standards ist, während ungerade für proprietäre Elemente stehen⁷[Nat11b, 7.1]. Die Reihenfolge der Tags ist in numerischer Folge in aufsteigender Form sortiert. Fällt während des Einleseprozesses in einer Datei auf, dass die Reihenfolge nicht korrekt ist, kann die Integrität des DICOM-Objekts nicht gewährleistet werden und deutet zum Beispiel auf eine modifizierte Datei hin.

3.1.4. VR - Value Representation

Dieser Teil eines DICOM-Elements beschreibt den Typ und das Format des Wertes[Nat11b, 6.2]. Der Umfang an verschiedenen Value Representations reicht von Zeichenketten wie PersonName (PN) im Datenelement (0010,0010) PatientName über Datumsangaben (DA) bis zu numerischen Werten(FL) und Sequenzen (SQ). Eine vollständige Liste ist im Standard unter [Nat11b, Table 6.2.1] zu finden. Beziiglich der Vollständigkeit soll erwähnt werden, dass ein Datenelement mit VR-Typ SQ wiederum ein DICOM-Object enthalten kann und dadurch eine Baumstruktur entsteht.

3.1.5. VM - Value Multiplicity

Value Multiplicity bestimmt die Anzahl an Werten, die in einem DICOM-Element enthalten sind. Die Werte werden durch einen Backslash \voneinander getrennt. Der explizite Wert der Value Multiplicity kann aus dem Data Dictionary entnommen werden [Nat11b, 6.4]. Der Patientenname enthält einen Wert als Zeichenkette. Die Lage des Patienten im

⁷Ausgenommen aus dieser Regel sind folgende Gruppen: (0000, eeee), (0002, eeee), (0004, eeee), (0006, eeee), (0001, eeee), (0003, eeee), (0005, eeee), (0007, eeee), (FFFF, eeee).

Raum wird mit Hilfe von drei Winkeln beschrieben. Dieses DICOM-Element hat eine Value Multiplicity von drei. Die Werte für die Winkel α, β, γ werden als Zeichenkette `0\0\1` dargestellt.

3.2. DICOM Pixeldaten und Bildformate

Die Abschnitte 3 - 3.1.2 zeigen, dass DICOM mehr als ein Bildformat darstellt, ein essentieller Bestandteil bleiben jedoch die Pixel eines DICOM-Objekts (auch wenn diese nur optional vorhanden sein müssen, wie Abbildung 3.2 zeigt). Nach dem DICOM Data Dictionary gehören Bild-abhängige Informationen zur Gruppe `(0028, eeee)`. Die Pixeldaten liegen im Bereich `(7fe0, 0010)` am Ende eines DICOM-Objekts.

Das bedeutet, dass die konkreten Werte der Pixel im gleichen DICOM-Objekt liegen und den selben Kodierungsrichtlinien der DICOM-Tags unterliegen.

3.2.1. Kodierung der Pixel im Speicherabbild einer DICOM-Objekts

Um die grundlegende Struktur der Pixel im DICOM-Objekt zu beschreiben sind drei Datenelemente notwendig: BitsAllocated, BitsStored und HighBit. BitsAllocated beschreibt, wie viel Speicher für einen singulären Pixelwert reserviert wird. Mit Hilfe von Columns und Rows kann die Bilddimension bestimmt werden. Columns beschreibt die Breite, Rows die Höhe. Tabelle 3.3 zeigt einen Überblick dieser Elemente eines DICOM-Objekts. Die Value Representation des Datenelements PixelData `(7fe0, 0010)` kann nach DICOM Data Dictionary entweder den Wert *OB* oder *OW* annehmen. *OB* bedeutet *Other Byte String* während *OW* für *Other Word String* steht. Nach Section 8.1 des Standards[Nat11b] besteht der Unterschied zwischen den beiden VRs darin, dass OB abhängig von der Byteordnung ist. Ob die Ordnung Little Endian oder Big Endian entspricht ist abhängig von der Transfersyntax des DICOM-Objekts. Grafik 3.4 zeigt die Kodierungsreihenfolge. Little Endian kodiert vom Least Significant Bit (LSB) zum Most Significant Bit (MSB). Big Endian verarbeitet die Byte in umgekehrter Reihenfolge. Ein Element des PixelData-Arrays (sowohl mit einer VR von OB als auch OW) fasst 16 Bit, was gleichzeitig die maximale Größe an allokiertem Speicher von BitsAllocated darstellt. Nach dem Standard entspricht jede Zelle von PixelData genau einem Pixelwert [Nat11b, 8.1.1] und sowohl *OB* als auch *OW* belegen maximal 16-Bit im Speicher. Durch diese Limitierung wird deutlich, dass die Grautwertbilder eine maximale Tiefe von 16-Bit erreichen können.

BitsStored gibt darüber Auskunft, wie viel Bit pro Wert in Anspruch genommen werden. Schließlich gibt HighBit das in der Reihenfolge ranghöchste Bit von StoredBits an [Nat11b,

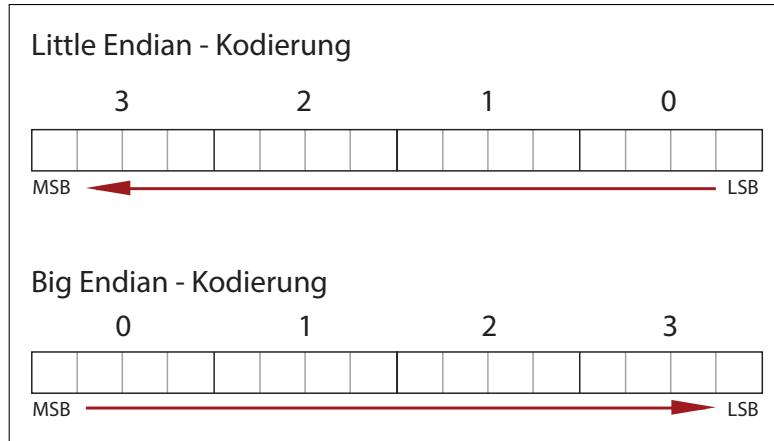


Abbildung 3.4.: Kodierungsreihenfolge von 4 Byte bei Little Endian- und Big Endian-Darstellung

8.1.1].

Abbildung 3.5(a) verdeutlicht die Repräsentation eines Pixels im Datenelement PixelData. Die Darstellung entspricht einem eindimensionalen Array. Das erste Element ist das erste Pixel in der linken oberen Ecke, das letzte Element stellt den Pixel rechts unten dar. 3.5(b) und 3.5(c) zeigen die exakte Belegung an Bit bei BitsAllocated 16 und BitsStored 12. Der graue Hintergrund zeigt die allokierten Bit, während der rote Bereich den tatsächlich benutzten Speicher markiert. Der Pixelwert in Abbildung 3.5(c) nimmt den gesamten Speicherplatz pro Pixel ein. Das schwarze Quadrat steht für das HighBit. Das Intervall der Werte ist abhängig von der Anzahl an gespeicherten Bits. Hat das Element StoredBits den Wert 12 kann ein Pixel einen Wert aus dem Bereich von $[0, 2^{12} - 1]$ annehmen. Entspricht StoredBits dem Wert 8, ist das Intervall $[0, 2^8 - 1]$. Hier spricht man von einer Grauwerttiefe von 12 beziehungsweise 8 [Han00, 2.2].

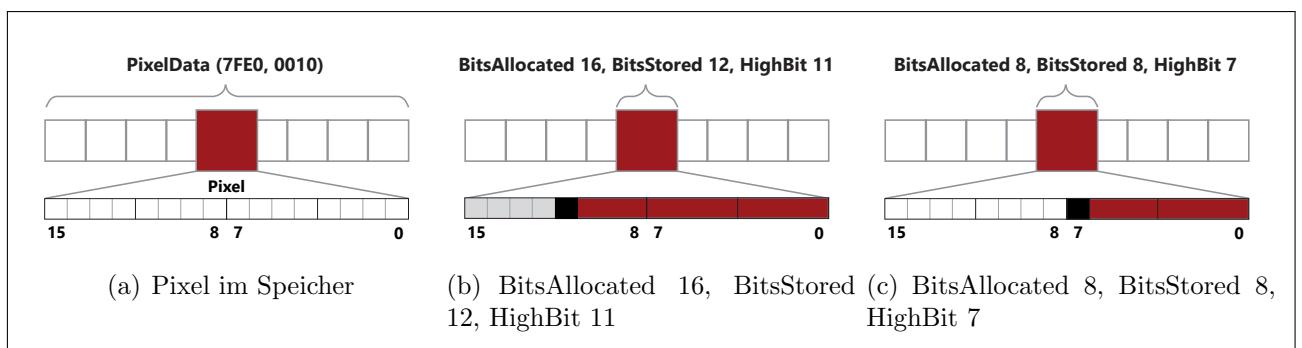


Abbildung 3.5.: Beispiele unterschiedlicher Speicherbelegung

Tag	Tagname	VR
(0028,0010)	Rows	US
(0028,0011)	Columns	US
(0028,0100)	BitsAllocated	US
(0028,0101)	BitsStored	US
(0028,0102)	HighBit	US

Tabelle 3.3.: Grundlegende Datenelemente für die digitale Repräsentation

Verschiedene Datenelemente aus dem DICOM-Standard liefern nähere Informationen zum Bildformat. So gibt das Element SamplesPerPixel (0028,0002) Auskunft darüber, wieviel Teile aus PixelData ein einzelnes Pixel repräsentieren. Hat Samples den Wert 1, so ist jedes Element aus PixelData genau ein Pixel. Daraus folgt, dass ein Grauwertbild vorliegt. 3 bedeutet, dass im DICOM-Objekt ein Farbbild mit den drei Kanälen rot, grün und blau liegt.

Die Werte der Pixel sind stark abhängig vom medizinischen Gerät, welches die Bilder aufzeichnet. Ein direkter Vergleich von Bildern, die von unterschiedlichen Geräten einer Modalität aufgezeichnet wurden ist daher nicht möglich. Um Gewebestrukturen von beispielsweise CT-Aufnahmen trotz dieser Abhängigkeiten patienten- und geräteübergreifend zu vergleichen, gibt es unter anderem die Hounsfield Skala [Han00, 2.1.3]. Mit Hilfe der Datenelemente RescaleSlope und RescaleIntercept lassen sich die ursprünglichen Werte in brauch- und lesbare Pixelwerte konvertieren. RescaleType gibt die Skala an, mit der das Ergebnis interpretiert werden kann. Ein Umrechnung erfolgt mit der Formel aus [Nat11a, C.11-1b Seite 1168]

$$Output = m * SV + b \quad (3.1)$$

mit $m = \text{RescaleSlope}$, $b = \text{RescaleIntercept}$ und $SV = \text{Pixelwert}$.

Fettgewebe zum Beispiel nimmt nach der Hounsfield-Skala Werte zwischen 0 und -100 ein [BLT98, Abbildung 1.18 Seite 15]. Ob in einem DICOM-Objekt vorzeichenbehaftete Werte vorhanden sind, sagt das Element PixelRepresentation. Eine 0 bedeutet kein Vorzeichen. Bei einem Wert von 1 können negative Werte enthalten sein.

3.2.2. Grauwertbilder

Für Bildverarbeitungsprozesse und Algorithmen bieten Grauwertbilder einige Vorteile im Vergleich zu Farbbildern. Der größte Unterschied liegt im Verhältnis zwischen Informationsgehalt zum benötigtem Speicherbedarf. Farben bieten bei Kantenübergängen oder Helligkeitsinformationen keinen Mehrwert. Das führt unter anderem dazu, dass Industrie oder auch die medizinische Bildverarbeitung vorwiegend auf dieses Format zurückgreifen. Viele medizinische Geräte liefern bei der Bildaufnahme reine Intensitätswerte und beinhalten dadurch keine Farbinformationen.

Eine Grauwerttiefe von 8 Bit wird überlicherweise in der industriellen Bildverarbeitung eingesetzt. Das entspricht dem Intervall von [0, 255] und den Werten, die mit handelsüblichen Monitoren darstellbar sind. Medizinische Bilddaten können, wie in Abschnitt 3.2.1 beschrieben, eine Tiefe von bis zu 16 Bit annehmen und Grauwerte aus dem Bereich [0, 65535] repräsentieren, da eine 8-Bit-Repräsentation eine zu geringe Genauigkeit liefert. Spezielle kostspielige Befundungsmonitore sind dazu in der Lage diese Spanne an Grauwerten darzustellen.

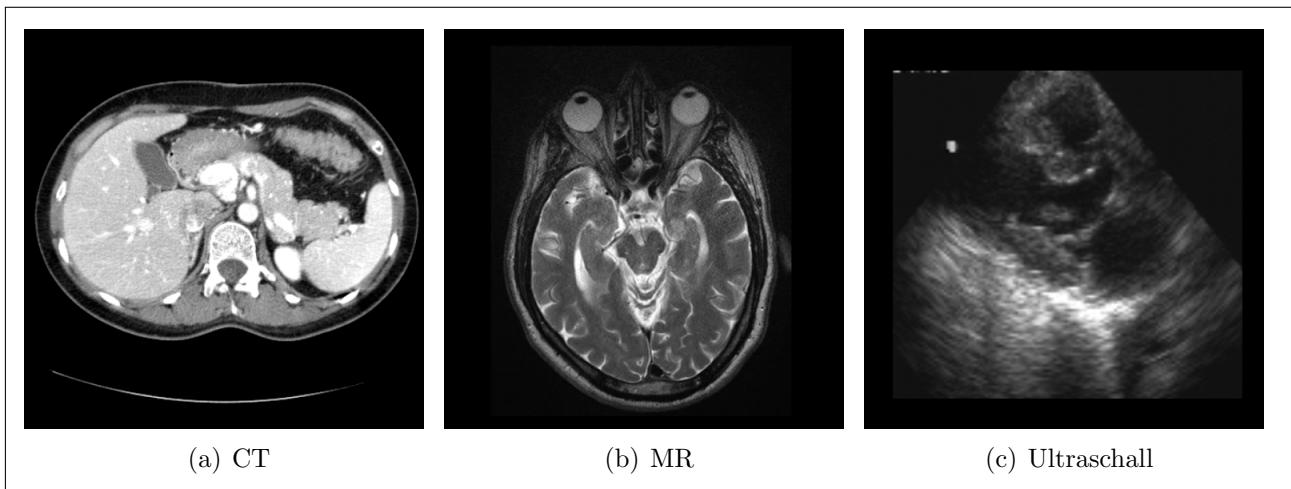


Abbildung 3.6.: Verschiedene Graustufenbilder

Fensterung von Grauwerten

Wie bereits in 3.2.2 beschrieben, lassen sich auf üblichen Monitoren nicht alle Grauwerte zur gleichen Zeit darstellen. Dadurch müssen die maximal 65535 verschiedenen Grauwerte auf 255 Werte abgebildet werden können. Dies wird durch die in der Radiologie verwendete Fensterungstechnik möglich [Han00, Kapitel 8, Seite 249]. Es wird ein Fensterzentrum und eine Fensterbreite gewählt. Alle Werte innerhalb dieses Intervalls werden zwischen

0 und 255 umgerechnet. Abbildung 3.7(a) verdeutlicht dieses Prinzip. Ein CT-Bild mit einer Tiefe von 12 Bit besitzt 4096 Grauwerte. Ein Zentrum von 2000 und eine Breite von 500 bildet alle Werte von 1750 bis 2250 auf 0 bis 255 ab. Ist ein Pixeldatum kleiner 1750 wird das Minimum 0 hinterlegt und ist der Wert größer 2250 bekommt das Pixel das Maximum 255.

Im DICOM-Standard ist ein Algorithmus gegeben, um die Pixeldaten zu konvertieren[Nat11a, C.11.2.1.2].

3.

Algorithmus 1: Berechne den Fensterungswert aus originalem Pixelwert

```

1:  $X \leftarrow \text{input}$  - tatsächlicher Pixelwert
2:  $Y \leftarrow \text{output}$  - konvertierter Wert zwischen 0 und 255
3:  $C \leftarrow \text{windowCenter}$ 
4:  $W \leftarrow \text{windowWidth}$ 
5: if  $X \leq C - 0.5 - (W - 1)/2$  then
6:    $Y = Y_{\min}$ 
7: else if  $X > C - 0.5 + (W - 1)/2$  then
8:    $Y = Y_{\max}$ 
9: else
10:   $Y = ((X - (C - 0.5))/(W - 1) + 0.5) * (Y_{\max} - Y_{\min}) + Y_{\min}$ 
11: end if
```

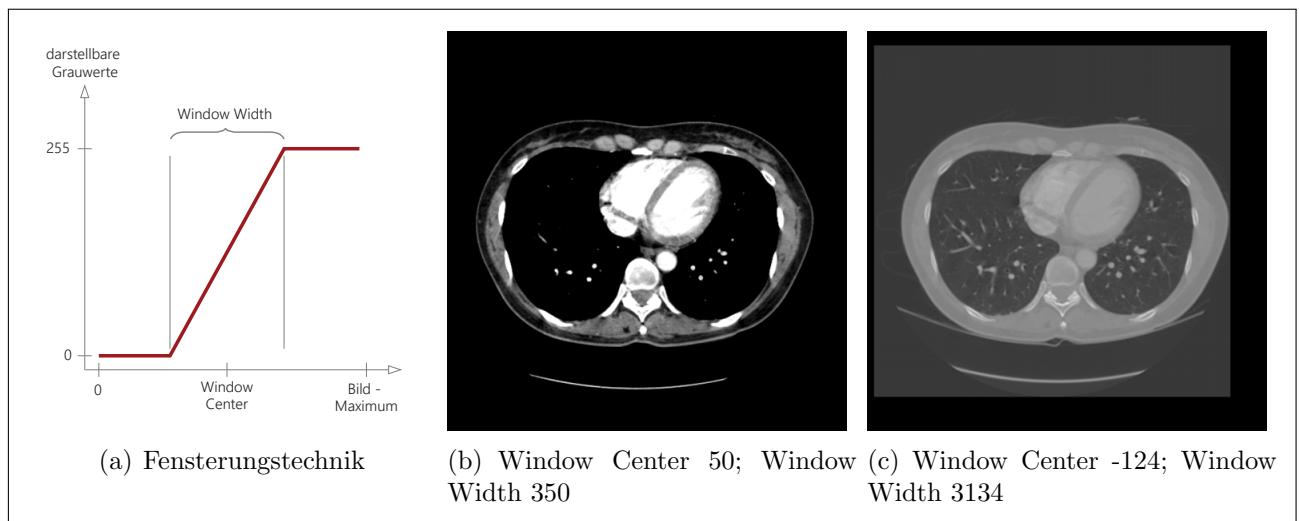


Abbildung 3.7.: Fensterungstechnik zur Darstellung medizinischer Bilddaten am handelsüblichen Monitor [Han00, S. 249]

3.2.3. Farbbilder

Obwohl Grauwertbilder das am meisten verwendete Format in der medizinischen Bildverarbeitung darstellen, haben auch Farbbilder die verschiedensten Einsatzgebiete. So wird in der Dermatologie auf Farbdarstellungen zurückgegriffen um Hauterkrankungen zu dokumentieren [Han00, 2.2.3.2]. Des weiteren kann mit Farbultraschallbildern die Fließrichtung und Geschwindigkeit des Blutes visualisiert werden und dienen zur Untersuchung von Venen und Arterien⁸.

Wie bereits beschrieben, ist das Datenelement *SamplesPerPixel* mit einem Wert von drei der Indikator, dass ein Farbbild vorliegt. Das bedeutet pro Pixel werden 3 Elemente von *PixelData* belegt mit je einem Element für den roten, grünen und blauen Farbkanal. Daraus resultiert der dreifach Speicherbedarf, $\text{BitsAllocated} * \text{SamplesPerPixel}$ [Nat11a, C.7.6.3.1.1]. Der DICOM-Standard bietet zwei Möglichkeiten, wie diese Information im Speicher hinterlegt werden. Entweder werden die Pixelwerte fortlaufend gespeichert mit R1, G1, B1; R2, G2, B2; ...RN, GN, BN; oder nach dem Kanal R1, R2, ...RN; G1, G2, ...GN; B1, B2, ...BN [Nat11a, C.7.6.3.1.3]. Das dafür verwendete Element aus dem Data Dictionary heißt *PlanarConfiguration(0028,0006)*. Abbildung 3.8 macht die beiden Schemata deutlich.

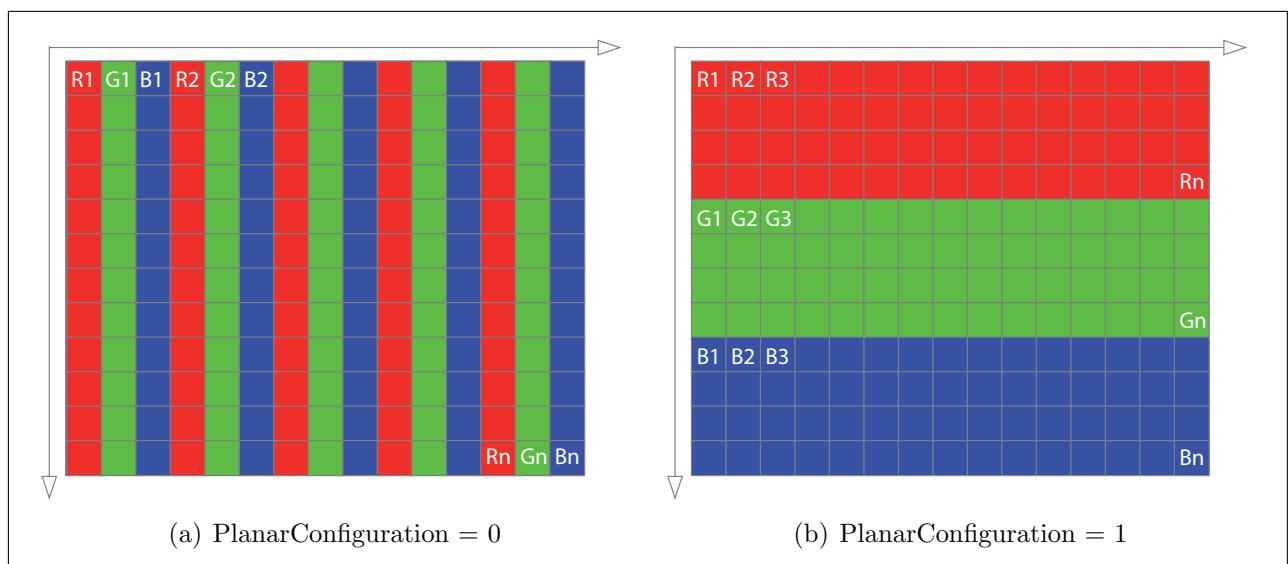


Abbildung 3.8.: Kodierung der RGB-Werte im Datenelement *PixelData* mit Hilfe der *PlanarConfiguration*

⁸<http://www.diagnostikum-berlin.de/farbkodierte-duplexsonographie-fkds> - abgerufen am 19.01.2014

3.3. 3D Bilddaten

Sowohl die Grauwertdarstellungen, als auch Farbbilder wurden bisher nur als eine zweidimensionale Abbildung behandelt. Computertomographen, Magnetresonanztomographen und auch 3D-Ultraschall sind in der Lage Schichten des menschlichen Körpers aufzunehmen. Die Menge der Schichtaufnahmen stellen eine Serie dar (vgl. Die DICOM Information Object Definitionen Seite 16). Durch die Verbindung der einzelnen DICOM-Objekte wird der Pixelraum verlassen und der Voxelraum betreten. Ein Voxel ist die dreidimensionale Repräsentation eines Pixels, mit der Tiefe als zusätzliche Dimension zu Breite und Höhe.

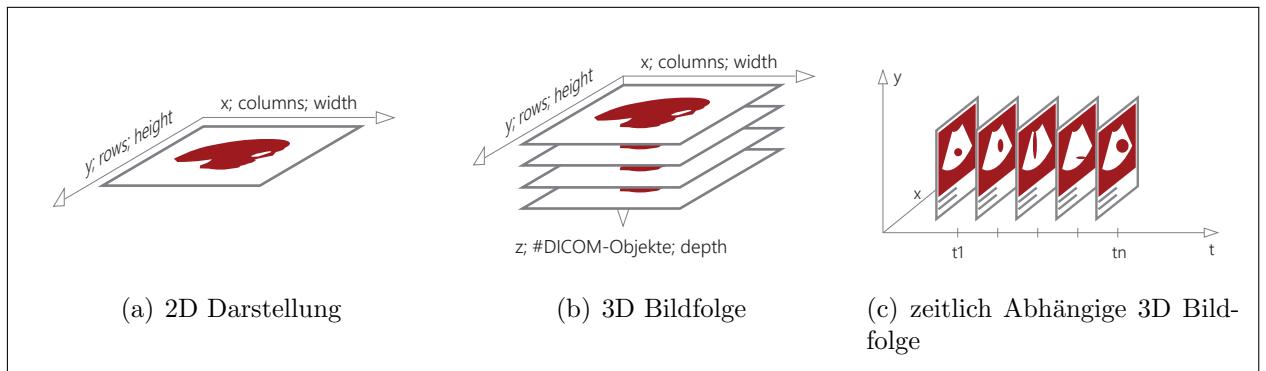


Abbildung 3.9.: Darstellung von 2- und 3-dimenionalen Bilddaten

3.4. Bilder mit zeitlicher Abhängigkeit

Auch Bildaufnahmen zu verschiedenen Zeitpunkten sind dreidimensionale Bildfolgen, wobei die Zeit die dritte Dimension darstellt [Han00, 2.2.5]. Häufige Einsatzgebiete für Bewegtbildfolgen ist die Endoskopie oder Sonographie. Die Abbildung 3.9(c) macht den Unterschied zwischen zeitlicher und räumlicher Dimension deutlich. Räumliche Darstellungen sind grundsätzlich in mehrere DICOM-Objekte und Dateien aufgeteilt. Zeitlich abhängige Daten sind in einem einzigen Objekt zusammengefasst. NumberOffFrames (0028,0008) gibt die Anzahl an unterschiedlichen Aufnahmen und Zeitpunkten an, die in PixelData enthalten sind.

4. Grundlagen zu Entwurfsmustern der Softwareentwicklung

Während der Entwicklung von Software stehen Programmierer oft vor ähnlichen Problemstellungen. Entwurfsmuster bieten hierfür verlässliche Lösungsvorschläge[GD13]. Muster werden unabhängig vom aktuellen Abschnitt des Entwicklungsprozesses eingesetzt. So gibt es Schablonen die das Softwaresystem beschreiben, oder Muster, die auf Objektebene eingesetzt werden. Hierbei wird zwischen Architekturmustern und Entwurfsmustern unterschieden. Nach Goll und Dausmann [GD13, 3.1, 3.2] ist das Ziel der Entwurfsmuster, Lösungen wiederverwendbar zu gestalten und die Flexibilität der Software zu erhöhen. Muster verbessern unter anderem die beiden Eigenschaften der Verständlichkeit und Erweiterbarkeit. Im Rahmen der Arbeit werden vor allem objektorientierte Muster folgender Klassen verwendet:

- Strukturmuster
- Verhaltensmuster
- Erzeugungsmuster

Nach dieser ersten Klassifikation kann nochmals zwischen klassenbasierten und objektbasierten Mustern unterschieden werden. Der Unterschied besteht hauptsächlich darin, dass klassenbasierte Muster die Objekttypen während der Übersetzung festlegt. Bei objektbasierten Mustern können Objekte den Typ dynamisch zur Laufzeit ändern[GD13, 4.1].

Die im folgenden dargestellten Entwurfsmuster bilden nur einen kleinen Teil der verfügbaren Muster ab. Diese Auswahl wird durch die Nutzung in der zu entwickelnden Software besonders hervorgehoben und genauer erläutert. Die konkrete Umsetzung wird im Kapitel „5. Softwarearchitektur des Java Medical Imaging Toolkit“ detailliert dargestellt.

4.1. Strukturmuster

Strukturmuster bestimmen die Zusammensetzung der Klassen oder Objekte. In dieser Abschlussarbeit wird im Speziellen auf das Muster *Adapter* eingegangen. Damit können externe inkompatible Bibliotheken mit internen Klassen verknüpft werden und ermöglichen dadurch eine Zusammenarbeit, die ohne den Adapter nicht möglich wäre.

Adapter

Der Adapter passt Schnittstellen an, um die Zusammenarbeit inkompatibler Objekte und Klassen zu ermöglichen. Er wird vorwiegend eingesetzt wenn die Schnittstelle einer Komponente nicht zu der einer benötigten Klasse passt und somit nicht gemeinsam verwendet werden können[ES13, 5.1]. Ein Adapter aus der realen Welt ist gut mit dem Muster zu vergleichen. Ein Monitor mit VGA-Eingang(Client) benötigt eine Grafikkarte mit einem VGA-Ausgang zur Kommunikation. Hat die Karte jedoch nur eine DVI-Buchse(zu adaptierende Klasse) wird ein Adapter(Adapter/Interface Adapter) benötigt, um das Signal umzuwandeln. Abbildung 4.1 zeigt das Schema des Entwurfsmusters.

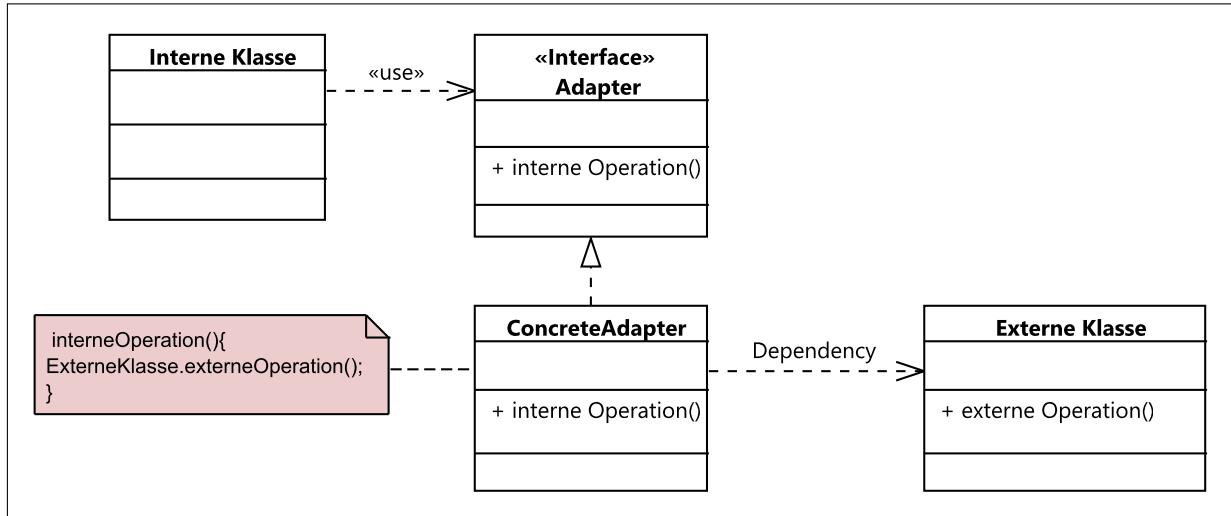


Abbildung 4.1.: UML Klassendiagramm zum Adapter [ES13, S. 78]

4.2. Verhaltensmuster

Die Interaktion zwischen Objekten kann mit Verhaltensmustern festgelegt werden. Die Anforderung aus Kapitel 2 zur Darstellungen verschiedener Ebenen erfordert eine Möglichkeit zur Kommunikation unter den Anzeigeflächen der DICOM-Objekte. Hier kommt das Beobachtermuster zum Einsatz, um Veränderungen eines Objekts an andere Objekte zu melden.

4.

4.2.1. Observer

Wenn der Zustand mehrerer Objekte von einem anderen Objektzustand abhängt, kann man das Observer-Muster verwenden. So kann der Zustand abhängig vom beobachteten Objekt angepasst werden [ES13, 4.7]. Ein Abonnement einer Zeitung(ConcreteSubject) oder eines Newsletters(ConcreteSubject) ist ein reales Beispiel dieses Muster. Alle Abonnenten¹ (Observer) erhalten(update) vom Herausgeber (Subject) automatisch (notifyObservers) die neue Ausgabe eines Magazins oder Newsletter sobald dieses erscheint. Ein Abonnent muss sich bewusst für eine Anmeldung entscheiden(register()), um neue Ausgaben zu erhalten. Das Konzept verdeutlicht Abbildung 4.2.

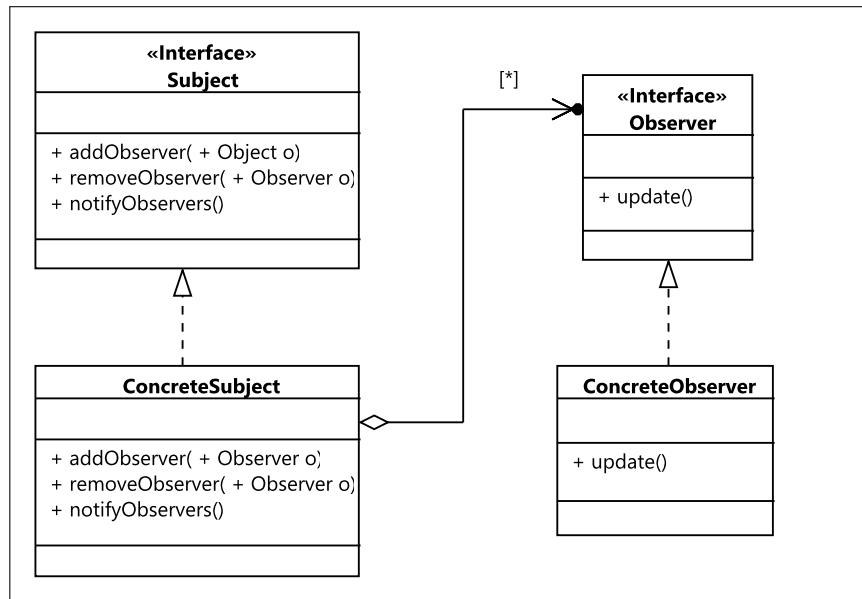


Abbildung 4.2.: UML Klassendiagramm zum Observer[ES13, S. 71]

¹Frau Müller und Herr Schmidt würden **ConcreteObserver** repräsentieren.

4.2.2. Schablonenmethode

Die Schablonenmethode wird eingesetzt, wenn Teile eines Algorithmus flexibel austauschbar sein sollen. Ein Rezept für Weihnachtsgebäck könnte zum Beispiel folgende Vorgehensweise wie in Listing 4.1 haben. Abbildung 4.3 zeigt das UML Diagramm dieses Entwurfsmusters. Eine abstrakte Klasse definiert eine Vorgehensweise *templateMethod()*. Im Beispiel ist das die Klasse *Gebaeck*. Abgeleitete Klassen definieren die genaue Vorgehensweise des Algorithmus durch die implementierten abstrakten Methoden *Operation1()* - *Operation3()*(im Beispiel sind das die Methoden *vermengeZutaten()* - *backen()*). Die Backzeiten von Spekulatius und Keksen sind unterschiedlich. Durch die spezifische Umsetzung von *backen()* wird die Backzeit individuell angepasst. Das bedeutet, dass die Vorgehensweise identisch ist und die Umsetzung spezifisch variieren kann.

```
1 | public abstract class Gebaeck {  
2 |  
3 |     public void gebaeckBacken() {  
4 |         vermengeZutaten();  
5 |         verarbeiteZutatenZuTeig();  
6 |         rolleTeigAus();  
7 |         stecheFormenAus();  
8 |         backen();  
9 |     }  
10| }  
11| }
```

Listing 4.1: Beispiel zur Schablonenmethode

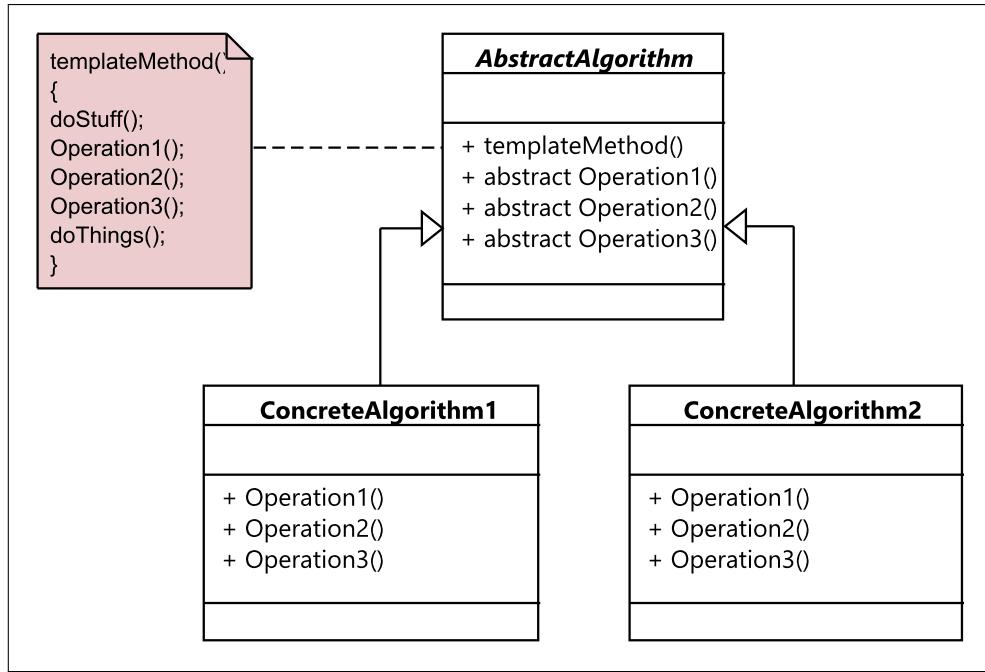


Abbildung 4.3.: UML Klassendiagramm der Schablonenmethode[ES13, S. 69]

4.3. Erzeugungsmuster

Erzeugungsmuster übernehmen die Erstellung der Objekte zur Laufzeit. Wie beschrieben, haben DICOM-Bilddaten eine Grauwerttiefe zwischen 8- und 16-Bit. Daher kann erst zur Laufzeit entschieden werden, welches Bildobjekt angelegt werden kann. Ein Fabrikmuster übernimmt diese Aufgabe.

4.3.1. Fabrik Methode

Dieses Entwurfsmuster bietet eine Schnittstelle (Fabrik) zur Erzeugung von Objekten (Produkte). Stehen die konkreten Objekttypen erst zur Laufzeit des Programms fest, kann dieses Erzeugungsmuster eingesetzt werden, um Konstruktion und Repräsentation zu trennen[ES13, 3.3].

Abbildung 4.4 zeigt den Aufbau in einem UML-Diagramm. Als Beispiel kann man einen Handwerker (Client) mit seinem Lehrling (*Factory*) betrachten, der Schrauben (*Product* → *Schraube_8mm*, *Schraube_10mm*) zureichen muss. Der Handwerker muss nicht selbst von der Leiter steigen und im Werkzeugkasten nach der passende Schraube suchen. Er bittet den Lehrling zum Beispiel eine Schraube 8mm zu bringen. Das Wissen über konkrete

Schrauben stellt eine konkrete Fabrik dar. So ist der Lehrling auch in der Lage andere Werkzeugkategorien zu verwalten. Jedes Werkzeug spezialisiert von AbstractProduct und das Wissen über die Kategorie von AbstractFactory.

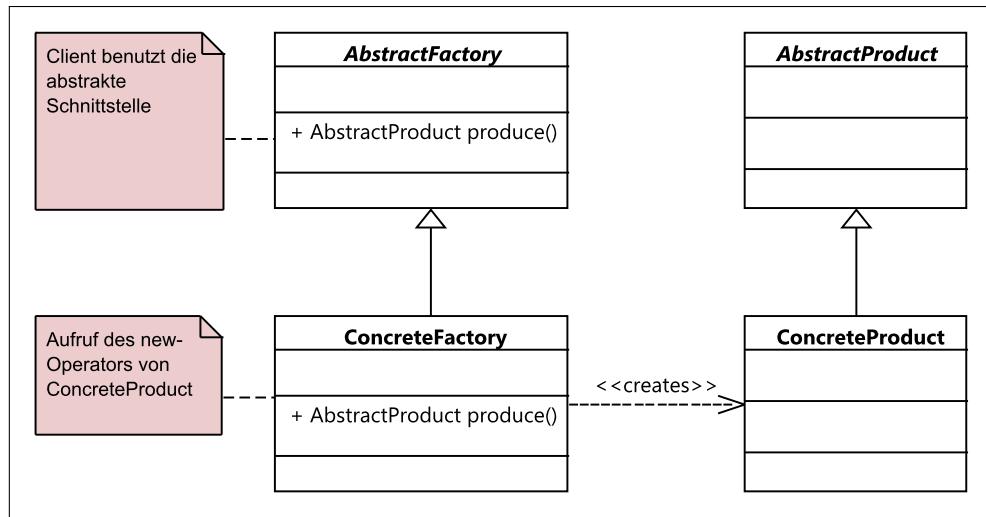


Abbildung 4.4.: UML Klassendiagramm zur Fabrik Methode [ES13, S. 35]

4.3.2. Singleton

Ein Singleton-Entwurfsmuster wird eingesetzt, wenn es nur eine Instanz einer Klasse geben darf[ES13, 3.4]. Das ist unter anderem bei Konfigurationsklassen von Software sinnvoll, da eine Manipulation von unterschiedlichen Quellen ausgeschlossen werden kann.

Das Codelisting 4.2 zeigt eine primitive Umsetzung in Java, die Intention eines Singletons wird allerdings deutlich. Durch die Sichtbarkeit *private* des Konstruktors wird eine externe Instantiierung unterbunden. Die einzige Instanz der Klasse ist das private statische Datenelement *Singleton s*. Zugriff darauf erlaubt nur die statische Methode *getInstance()*.

```
1 public class Singleton{  
2  
3     private static Singleton s = new Singleton();  
4  
5     private Singleton(){  
6         //doThings();  
7     }  
8  
9     public static Singleton getInstance(){  
10        return s;  
11    }  
12 }
```

4.

Listing 4.2: Implementierung des Singleton-Musters in Java

4.4. Das Architekturmuster Plug-in

Wird eine Software entwickelt, sollte das Open-Closed-Prinzip angewendet werden. Dieses besagt, dass Software sowohl offen, als auch geschlossen sein soll[GD13, 1.8]. Bei Objekt-orientierten Programmiersprachen ist die Vererbung ein Beispiel, die den Widerspruch erläutert. Abgeleitete Klassen *erweitern* die Fähigkeiten der Superklasse, die wiederum *nicht verändert* werden. Die Superklasse ist offen für Erweiterung und geschlossen für Veränderung.

Das Beispiel der Vererbung bezieht sich auf eine Klasse. Durch eine Erweiterung auf die Ebene der Softwarearchitektur entsteht das Plug-in-Muster. Das bedeutet, ein Plug-in stellt einem fertigen System zusätzliche Erweiterungen bereit, die zur Laufzeit eingebunden werden können und dem System vorher unbekannt sind. Die Kommunikation wird durch eine Schnittstelle möglich, die das System zur Verfügung stellt. Wird diese Schnittstelle implementiert kann die Klasse eingebunden werden. Abbildung 4.5 zeigt die Interaktion von Schnittstelle, System und konkreten Plug-ins.

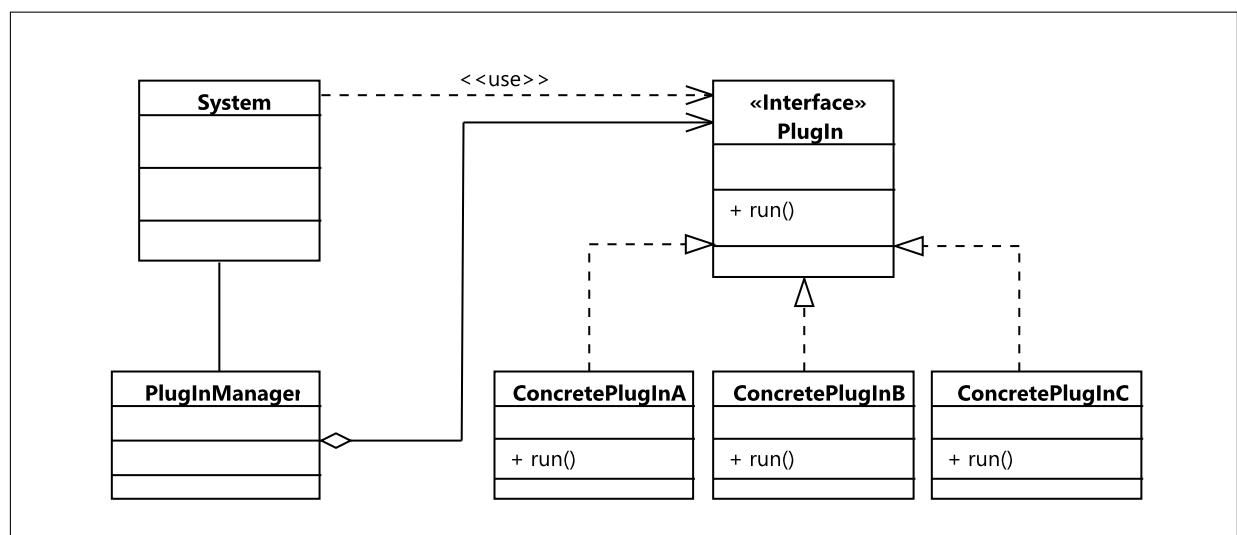


Abbildung 4.5.: UML Klassendiagramm des Architekturmusters Plug-in [GD13, S. 316]

Teil II.

Entwicklung des

Java Medical Imaging Toolkit -

jMediKit

5. Softwarearchitektur des Java Medical Imaging Toolkit

5.1. Die Eclipse Rich Client Platform

Die Grundlage für eine modulare Entwicklung der Software liefert die Eclipse Rich Client¹ Platform (RCP). Eclipse, basierend auf der Programmiersprache Java, wird seit 2001 von einer OpenSource-Gemeinde entwickelt[Vog13b]. Während es anfangs rein für die Java-Applikationsentwicklung entworfen wurde, ist es heute eine allgemeine erweiterbare Entwicklungsumgebung. So lässt sich zum einen das Grundprogramm mit Hilfe sogenannter Plug-ins erweitern und zum anderen eigenständige Applikationen erstellen (RCP), die auf dem Eclipse-Framework aufzubauen. Das „Aptana Studio“² ist beispielsweise ein Plug-in, das der Grundentwicklungsumgebung mehrere Funktionalitäten im Bereich der Webentwicklung (Kommunikation zum Server, Syntaxhighlighting von HTML und CSS) hinzufügt. „RSS Owl“³, ein Programm zur Verwaltung von Newsfeeds, ist ein Beispiel für eine eigenständige Rich Client Applikation auf Basis von Eclipse.

Eine Kernkomponente des Eclipse-Frameworks ist *Equinox*, eine Implementierung der OSGi-Spezifikation. OSGi bietet die Möglichkeit, modulare Java-Applikationen zu entwickeln und Softwarepakete (nach der Spezifikation „Bundles“ unter Eclipse „Plug-ins“ genannt) während der Laufzeit hinzuzufügen, zu entfernen, zu starten oder zu stoppen [Vog13a]. Das Java Medical Imaging Toolkit ist eine Implementierung eines Bundles, beziehungsweise Plug-ins.

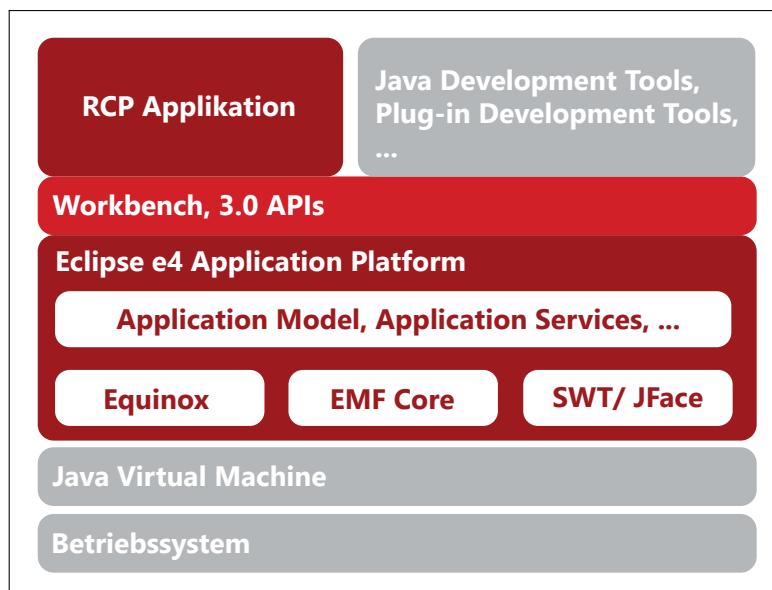
Abbildung 5.1 zeigt die Architektur der Eclipse Rich Client Platform.

Die untere Ebene bildet das Betriebssystem. Eclipse kann grundsätzlich plattformunabhängig unter Windows, verschiedener Linux Distributionen oder Mac OS eingesetzt wer-

¹Es lässt sich zwischen Rich Clients und Thin Clients unterscheiden. Rich Clients stellen sowohl die Präsentationsschicht als auch die logische Schicht auf dem Client zur Verfügung. Die Applikationsfunktionalität der Thin Clients wird komplett von einem Server bereitgestellt. Die Bedienung erfolgt meist über den Browser.

²<http://www.aptana.com/>

³<http://www.rssowl.org/>



5.

Abbildung 5.1.: Architektur der Eclipse e4 Plattform zur Entwicklung von Rich Client Applikationen [Art13]

den. Einzige Voraussetzung ist eine installierte Java Virtual Machine. Aufbauend auf Java steht der Eclipse-Kern, bestehend aus dem OSGi-Framework Equinox, dem Eclipse Modeling Framework EMF⁴ und dem Standard Widget Toolkit SWT und dient als Standardwerkzeug zur Erstellung der Benutzeroberfläche. Das SWT ist eine OpenSource Implementierung verschiedenster grafischer Bedienelemente wie Schaltflächen, Textfelder, Tabellen und vieles mehr. Der Unterschied zu den in Java integrierten grafischen Oberflächen besteht darin, dass das Standard Widget Toolkit auf die Ressourcen des Betriebssystems zugreift und sich in der Darstellung den Betriebssystemstandards anpasst[The14]. In Eclipse Rich Clients Applikationen ersetzt das SWT das Java-eigene AWT und Swing). Aufbauend auf den Grundkomponenten liegt das *Application Model*, das die Struktur der Applikation(Menüs, Fenster, etc.) beschreibt [Vog13a, Kapitel 7].

Workbench und Eclipse 3.0 APIs bieten noch die Möglichkeit Anwendungen abwärtskompatibel zu entwickeln. Die gesamte Plattform bildet die Basis für das Java Medical Imaging Toolkit. Durch das Application Model kann modular entwickelt und die Programmstruktur in zukünftigen Versionen erweitert werden.

⁴Das EMF dient beispielsweise zur Entwicklung eines Datenmodells, wird allerdings für weitere Ausführungen nicht explizit benötigt.

5.2. Das Eclipse Application Model

Damit eine Anwendung strukturiert werden kann stellt das Application Model steuernde und visuelle Elemente zur Verfügung. Das bedeutet, dass das Application Model den Aufbau der Anwendung, wie zum Beispiel die Andordnung der einzelnen Elemente, regelt und die Implementierung an anderer Stelle stattfindet.

- **Strukturen zur visuellen Beschreibung der Applikation**

Das Aussehen wird mit Hilfe von Windows, Parts, PartStacks und anderen Elementen des Application Models beschrieben. Die Elemente enthalten noch keine Logik sondern definieren nur die Struktur der Anwendung.

- **Strukturen zur Steuerung des Verhaltens**

Zu den Komponenten die das Verhalten der Applikation beeinflussen zählen zum Beispiel Tastatur-Shortcuts, Commands und Handler. Letztere dienen zur Verarbeitung von Benutzereingaben.

5.2.1. Visuelle Komponenten

Window

Windows sind einfache Repräsentationen eines Fensters der Benutzeroberfläche[The13, org.eclipse.e4.ui.model.application.ui.basic]. Sie bilden das Grundgerüst der Applikation und beinhalten Perspectives, PartStacks und andere Elemente. Ein einfaches Fenster ist in Abbildung 5.2(a) zu sehen.

Menüs

Ein Menü ist der Container für verschiedene Menü-Elemente und dient dazu, Benutzereingaben entgegenzunehmen. Einem Element können Commands hinterlegt werden, damit die Eingaben weiter verarbeitet werden können. Ein Menüpunkt kann selbst ein Menü beinhalten.

Perspective

Perspectives beinhalten eine Menge von Elementen der Benutzeroberfläche wie PartStacks und Parts. Perspectives können unabhängig vom Rest der Oberfläche gewechselt werden[The13, org.eclipse.e4.ui.model.application.ui.advanced]. So können Perspectives

beispielsweise die Anordnung der Parts definieren oder neue Parts anzeigen, die in einer anderen Perspective nicht enthalten sein sollen. Unter Eclipse hat zum Beispiel das Debug-Modul eine eigene Perspective und es kann dynamisch zwischen Debug- und Entwicklungsperspektive gewechselt werden.

PartSashContainer

Wie der Name bereits andeutet, ist ein PartSashContainer ein Container für PartStacks und Parts. Die enthaltenen Elemente werden komplett angezeigt. In Abbildung 5.2(d) ist eine solche Kombination zu sehen. Die obere Hälfte stellt einen PartStack mit den beiden Parts *TestPart A* und *TestPart B* dar. Der Untere Teil ist ein Stack-unabhängiger Part.

5.

PartStack

Auch der PartStack dient als Behälter für einzelne Parts. Der Unterschied zum PartSash-Container liegt darin, dass bei PartStacks nur der aktuell ausgewählte Part angezeigt wird. Die Darstellung des Stacks ist mit üblichen Tabs zu vergleichen, wie Abbildung 5.2(c) zeigt. Der Stack enthält die beiden Parts *TestPart A* und *TestPart B*.

Part

Ein Part ist die kleinste Einheit des Application Models und ist Kern der Benutzeroberfläche [The13, org.eclipse.e4.ui.model.application.ui.basic]. Innerhalb der Parts werden alle weiteren Bedienelemente angezeigt. Jede Abbildung von 5.2(b) - 5.2(d) enthält eine oder mehrere Parts. Betrachtet man das Application Model als Baumstruktur, symbolisieren Parts die Blattknoten. Parts können direkt einem Window unterstellt oder tief verschachtelt zwischen Perspectives und Stacks benutzt werden.

5.2.2. Steuernde Komponenten

Commands

Commands bilden die abstrakte Schicht zwischen Benutzereingabe und Verarbeitung. Commands besitzen keine eigene Implementierung. Das ermöglicht dem Entwickler das Verhalten individuell zu gestalten. So könnte ein Einfügen-Befehl im Editor ein anderes Verhalten auslösen als im Explorer-Part [The13, org.eclipse.ui.commands].

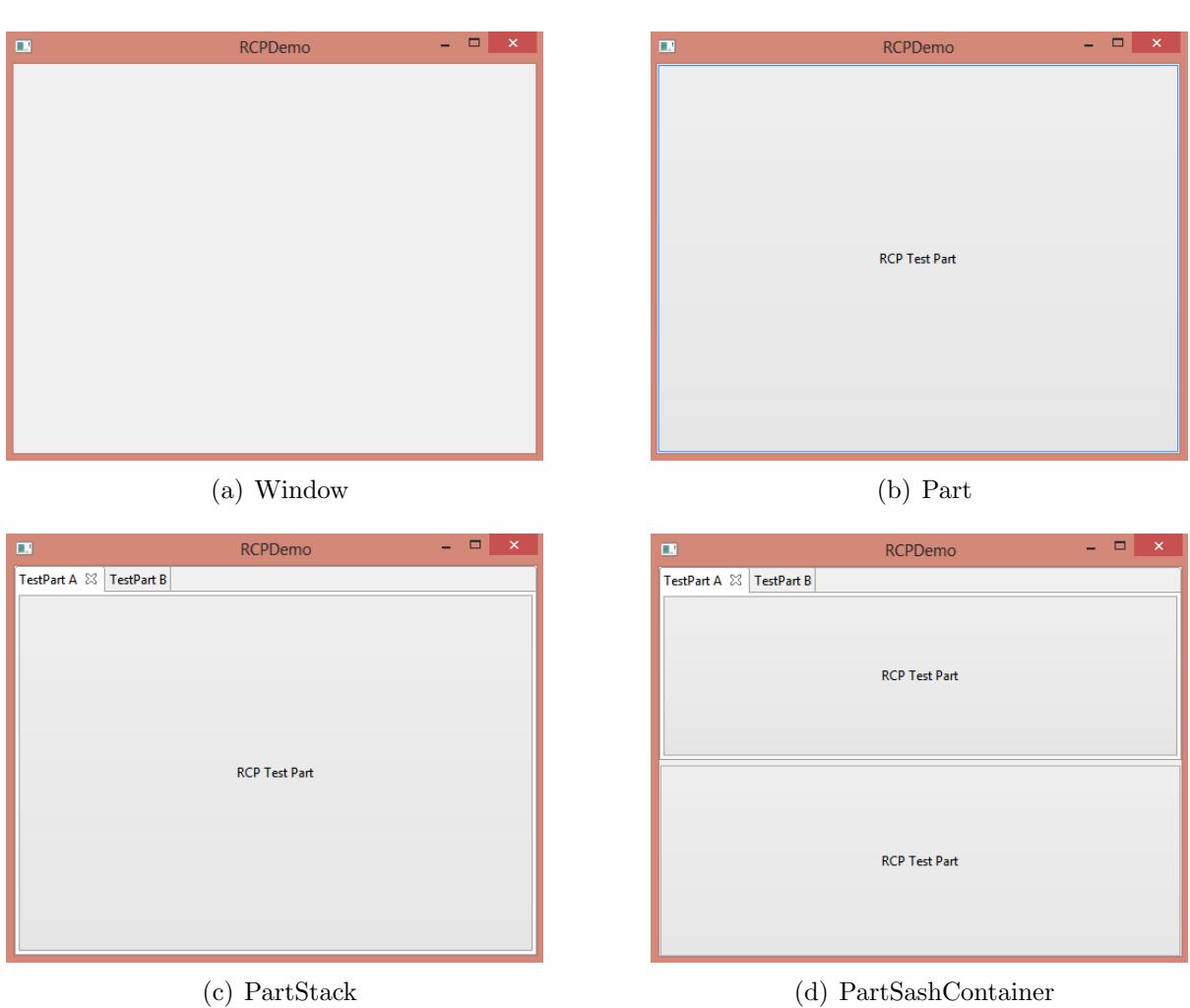


Abbildung 5.2.: Verschiedene Elemente des Application Models

Handler

Handler sind die konkreten Implementierungen der Commands und sind für die Verarbeitung der Benutzereingaben verantwortlich.

5.3. Die Benutzeroberfläche von jMediKit

Die Oberfläche des Java Medical Imaging Toolkit besteht aus sechs zentralen Elementen, um den Anforderungen zur Darstellung und Bedienung zu entsprechen.

1. Hauptmenü

Das Hauptmenü stellt globale und bildspezifische Operationen zur Verfügung. So werden unter dem Menüpunkt *Erweiterungen* die importierten Plug-ins aufgelistet.

2. Werkzeugleiste

Dieser Teil der Benutzeroberfläche stellt hauptsächlich Möglichkeiten zur Manipulation der Bilddaten zur Verfügung. Das Kapitel „6. Implementierung“ geht genau auf die verfügbaren Werkzeuge ein.

3. DicomBrowser

Dieser Part erlaubt die Navigation durch die vom Programm geladenen DICOM-Objekte. Die Anordnung entspricht der Darstellung des ER-Modells aus Kapitel „3. Grundlagen medizinischer Daten- und Bildformate“

4. ImageView

ImageView übernimmt die Anzeige der Pixeldaten der DICOM-Objekte.

5. Console

Die Konsole dient der Fehlerausgabe bei der Plug-in-Entwicklung.

6. DicomTagView

Im DicomTagView werden die Tags eines ausgewählten DICOM-Objekts dargestellt.

Die hierarchische Struktur der Anwendung wird in Abbildung 5.4 dargestellt. Die sechs Blattknoten repräsentieren den nach außen für den Benutzer sichtbaren Teil der Anwendung.

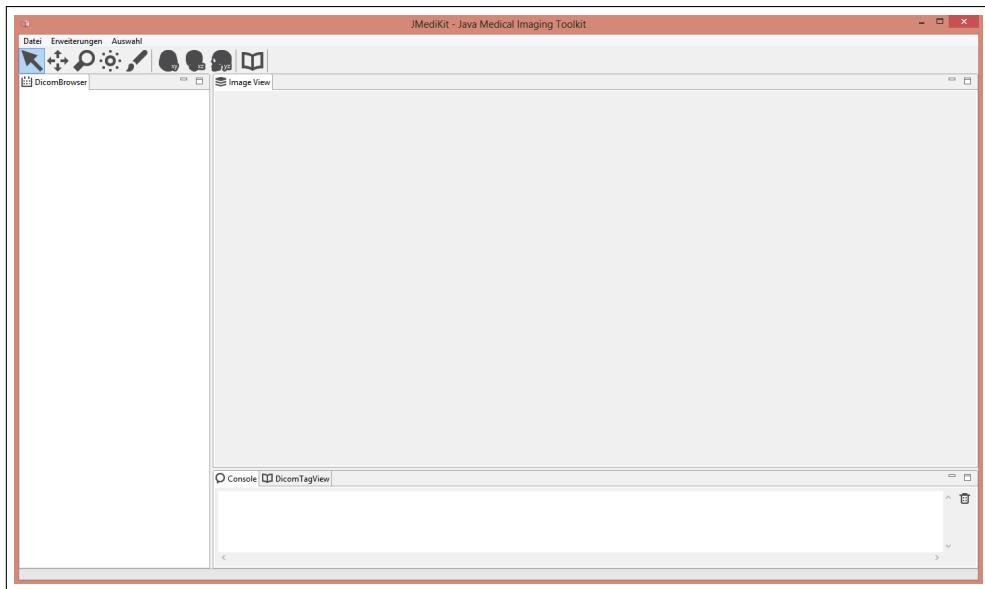


Abbildung 5.3.: Die Benutzeroberfläche von jMediKit

5.4. Erweiterbarkeit der Grundstruktur

Die flexible Struktur des Eclipse Application Models und der Rich Client Platform erlauben komfortable Erweiterungen. So können einzelne Parts den schon bestehenden Elementen zugeordnet, oder neue Perspectives eingefügt werden, die eine neue Benutzeroberfläche abbilden könnten. Beispielsweise kann ein neuer Part den FileStack (Abbildung 5.4) damit erweitern, dass DICOM-Objekte von einem PACS geladen werden. Das Kapitel „7. Entwicklung von Erweiterungen“ zeigt eine Möglichkeit, jMediKit einen neuen Part hinzuzufügen.

Durch das Hinzufügen von Elementen des Application Models ist es möglich, in die Struktur und vor allem die Benutzeroberfläche einzugreifen. Das bedeutet auch, dass Parts oder andere Strukturen nur in der Lage sind neue Funktionen einzubauen. Es fehlt die Möglichkeit bereits existierende Elemente wie das Hauptmenü (MainMenu) oder die Werkzeugelemente (Toolbar) zu erweitern.

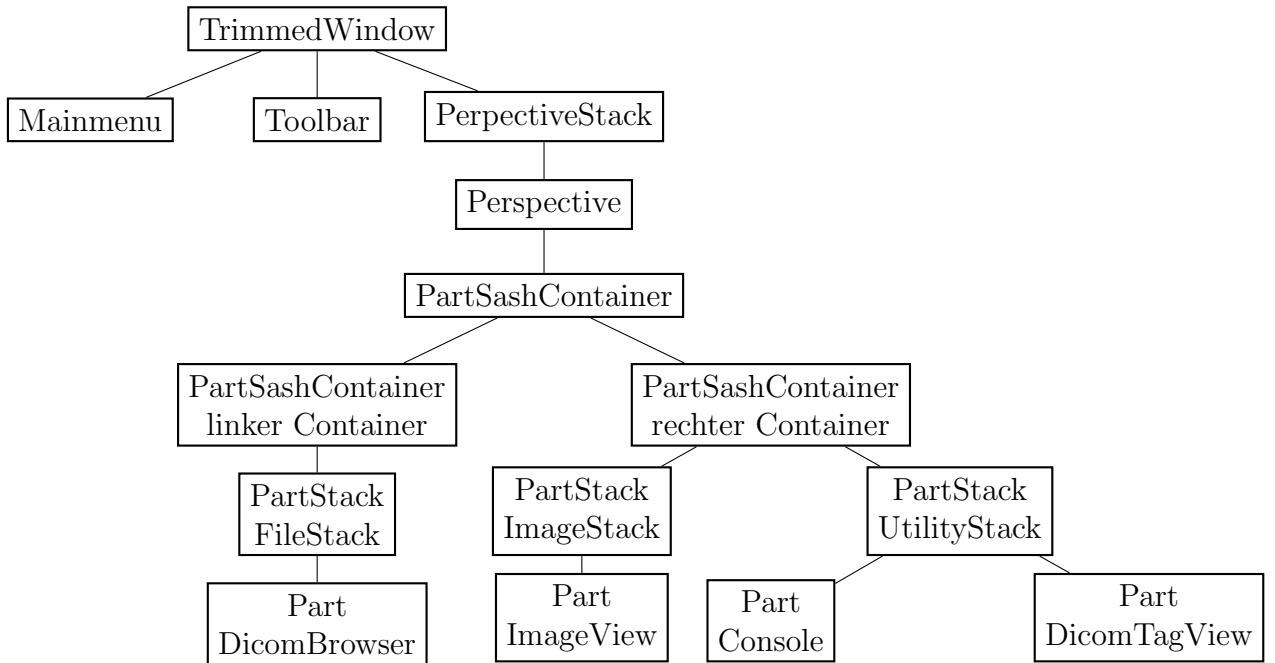


Abbildung 5.4.: jMediKit und die hierarchische Anordnung der Elemente des Application Models

5.

5.5. Modularare Werkzeuge

Neben der Applikationsstruktur und deren Implementierung liefern die Werkzeuge weitere grundlegende Funktionen der Anwendung. Damit die Werkzeuge wie die Struktur einen modularen Charakter erhalten, müssen auch diese erweiterbar sein. Ein zusätzliches Problem ist, dass während der Laufzeit nicht bekannt ist, welche Werkzeugobjekte erzeugt werden müssen. Wählt der Anwender aus der Werkzeugleiste ein Tool aus, wird das entsprechende Objekt zum Menüpunkt erzeugt. Ein Klick auf das Translationswerkzeug erzeugt zum Beispiel ein Objekt der Klasse *MoveTool*. Das bedeutet, die Objekterzeugung findet erst nach der Auswahl durch den Benutzer zur Laufzeit statt. Da die Wahl nicht vorbestimmt ist, müssen die konkreten Werkzeugobjekte dynamisch erzeugt werden können. Wie in Abschnitt 4.3.1 beschrieben, kann die Fabrik Methode für Anwendungsfälle dieser Art der Objekterzeugung eingesetzt werden. Bei den Klassen selbst wird das Open-Closed-Prinzip beachtet. Die abstrakte Klasse *ATool* gibt ein Grundwerkzeug vor und bietet eine Schnittstelle zur Erweiterung. *AToolFactory* gibt die Vorgehensweise zur Werkzeugerzeugung vor. Abgeleitete Fabriken klassifizieren die Werkzeugkategorien. Durch die Fabrik entsteht eine dynamische Objekterzeugung und die abstrakte Toolklasse liefert die nötige Schnittstelle für eine Erweiterung der Werkzeuge.

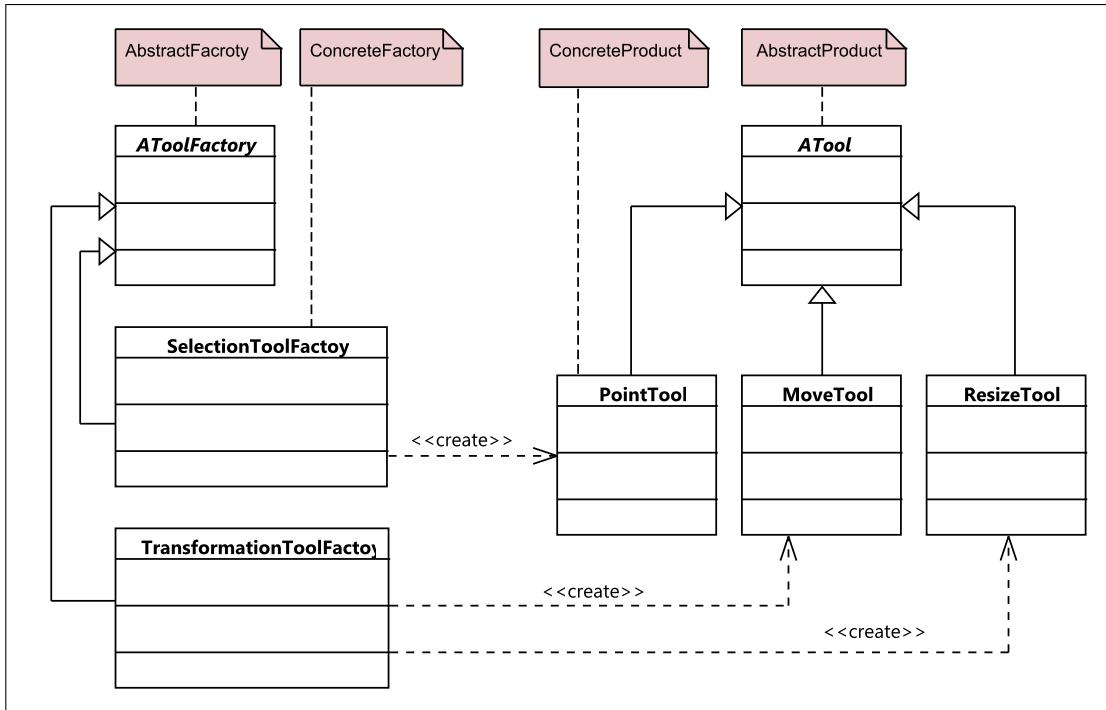


Abbildung 5.5.: Die Fabrik Methode zur Werkzeugerzeugung

Abbildung 5.5 zeigt die Umsetzung dieses Erzeugungsmusters. Die Grundversion von jMediKit enthält die zwei konkreten Fabriken *TransformationToolFactory* und *SelectionToolFactory*. Erstere beinhaltet Werkzeuge zur Bildtransformation wie die Skalierung (*ResizeTool*) und letztere enthält ein Werkzeug zur Auswahl von Punkten im Bild(*PointTool*). Sollen weitere Werkzeuge entwickelt werden, können diese von der abstrakten Klasse *ATool* abgeleitet werden und diese erweitern.

5.6. Die Plug-in Architektur

Die Entwicklung von Erweiterungen durch den Anwender spielt neben der Modularität eine weitere Rolle in dieser Abschlussarbeit. Die Entwicklung von zusätzlichen Funktionen und Bedienelementen mittels des Application Models sind erst nach einem erneuten Build-Prozess von jMediKit verfügbar, während Plug-ins auch nach einer Kompilierung des Systems integriert werden können. Es soll den Anwendern eine Möglichkeit geboten werden, eigenständig Erweiterungen zu entwickeln. Der Wirkungsbereich dieser Plug-ins beschränkt sich auf die Manipulation der Bilddaten. Dadurch ist kein explizites Wissen

über die Eclipse-Plattform nötig.

Die Grundlage der Plug-in Architektur bildet das in Kapitel 4 vorgestellte Plug-in-Muster. Die Erweiterungen arbeiten im Speziellen mit den Bilddaten. Durch die dreidimensionale Datenstruktur ist es möglich, dass Plug-ins neben der x- und y-Ebene auch in die z-Richtung arbeiten. Dadurch ist eine Anpassung des Architekturmusters notwendig, in der Plug-ins als Typen spezifiziert werden. Als Ergebnis wird eine Kombination aus Plug-in, Schablonenmethode und Singleton eingesetzt. Das UML-Diagramm in Abbildung 5.6 zeigt die Architektur und Zusammenarbeit der drei Muster.

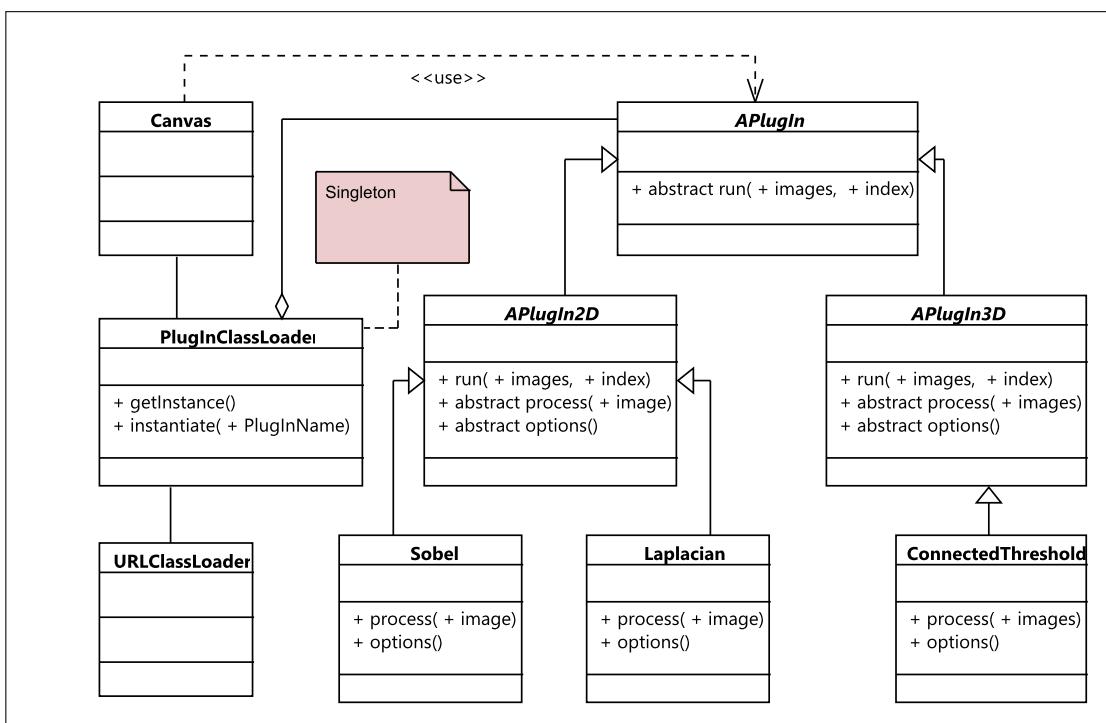


Abbildung 5.6.: Architektur der Plug-in-Struktur

5.6.1. Plug-in als Grundstruktur

Die Manager-Klasse des Architekturmusters ist *PlugInClassLoader*. Diese Klasse dient sowohl zum Laden der Plug-ins, als auch zum Instantiieren der Plug-in-Objekte. Aufgrund des eingeschränkten Wirkungsbereichs der Plug-ins ist die Zeichenfläche der Anwendung(*Canvas*) das einzige Modul, das über den Manager Plug-in-Objekte erzeugen kann. Ein Unterschied zur UML-Darstellung in Abbildung 4.5 des vorherigen Kapitels

besteht darin, dass keine Schnittstelle zur Implementierung, sondern eine abstrakte Klasse *APlugIn* zur Ableitung zur Verfügung gestellt wird. Das bringt den Vorteil, gemeinsame Methoden zu definieren die in allen Plug-ins enthalten sind.

5.6.2. Erweiterung mit der Schablonenmethode

Nach den Anforderungen in Kapitel 2 ist es nötig, Plug-in-Schnittstellen sowohl für die Entwicklung von zwei- als auch dreidimensionalen Bilddaten anzubieten. Um dies zu erfüllen, stehen dem Benutzer die beiden abstrakten Klassen *APlugIn2D* und *APlugIn3D* zur Verfügung. Beide Varianten repräsentieren die Schablonenmethode. Das Template wird von der Methode *run()* repräsentiert. Der generische Teil des Algorithmus ist *process()*. Die Schablone ist notwendig, da abhängig von 2D und 3D andere Bilddaten an den Benutzer zum Bearbeiten übergeben werden müssen. Nach dem Aufruf von *process()* wird zusätzlich die Rückgabe des Benutzers von der Templatemethode *run()* validiert. Die Schablone ermöglicht dadurch eine Vorverarbeitung und Nachbearbeitung der Bilddaten.

5.6.3. Singleton als Plug-in Manager

Der Manager *PlugInClassLoader* ist als Singleton realisiert. Dadurch wird das Problem umgangen, dass mehrere Manager die gleichen Plug-ins laden. Passiert dies, sind gleiche Klassen zueinander inkompatibel und es kann nicht sichergestellt werden, welche Klasse von welchem Manager zur Verfügung gestellt wird.

Damit neue Klassen zum Programmstart eingebunden werden können, wird der Java System Classloader über den Manager um einen URL-Classloader erweitert. So werden zu den bisherigen Klassen der Java Virtual Machine alle Plug-in-Klassen hinzugefügt.

5.6.4. Plug-ins mit dynamischer Parameterübergabe

Einen weiteren wichtigen Aspekt der Architektur zur Plug-in-Entwicklung stellt ein generischer Dialog dar. Der Anwender kann dieses Bedienfeld in sein Plug-in einbinden und so eine dynamische Parameterübergabe anstoßen.

Das Grundlegende Fenster liefert die Klasse *TitleAreaDialog* aus dem externen SWT- und JFace-Paket. jMediKit erweitert diese Klasse zu einem *PlugInDialog*. Sie enthält eine Liste mit den Schnittstellen *IPlugInDialogItem*. Dieses Interface liefert die Grundfunktionen der Parametertypen wie zum Beispiel *IntegerItem* und *StringItem*. Der Plug-in-Dialog kann beliebig viele Elemente enthalten.

Wird ein Plug-in vom Anwender entwickelt, kann der *PlugInDialog* nicht direkt verwendet

werden, da externe Bibliotheken wie SWT und JFace nicht vom Benutzer eingebunden werden und somit die Klassen und Funktionen aufgrund der Abhängigkeiten von *PlugInDialog* und *TitleAreaDialog* nicht aufgelöst werden. Um dieses Problem zu beheben, wird die weitere Generalisierung *GenericPlugInDialog* eingeführt. Dieser Dialog kann in konkreten Plug-ins instantiiert, angepasst und bearbeitet werden.

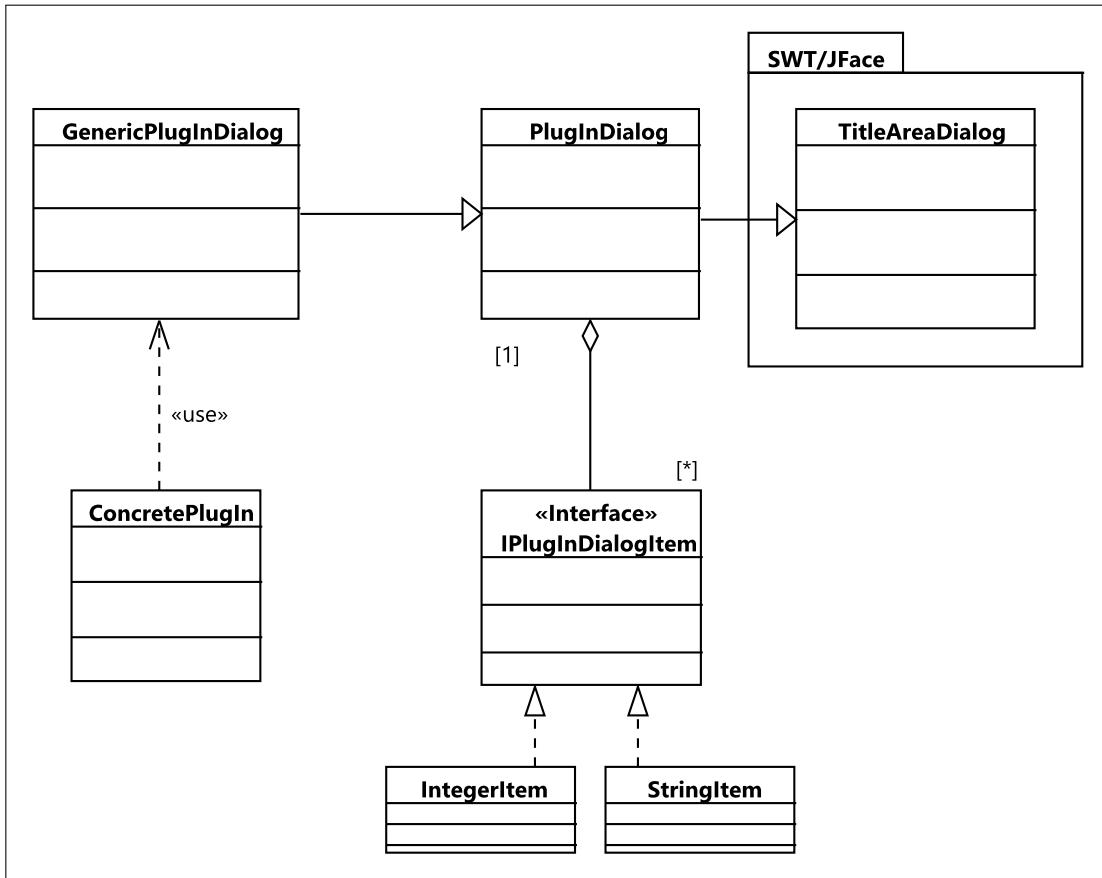


Abbildung 5.7.: Architektur des generischen Dialogs zur dynamischen Parameterbestimmung

5.7. Externe Bibliotheken

Fremde Bibliotheken werden aus verschiedenen Gründen benötigt. Zum einen für das Lesen von DICOM-Objekten und zum anderen bieten Bildverarbeitungsbibliotheken eine gute Grundlage an Funktionen. Zur Verarbeitung von DICOM-Dateien verwendet jMediKit die freie Bibliothek „dcm4che“. Für die Bildverarbeitung wird „Simple ITK“ eingesetzt.

5.7.1. dcm4che

Im Kapitel „Grundlagen medizinischer Daten- und Bildformate“ wurden die Grundlagen medizinischer Datenformate gezeigt und die Komplexität der Datenstrukturen deutlich gemacht. Eine standardgerechte Implementierung ist daher im Rahmen der Arbeit nicht zufriedenstellend umzusetzen. Dadurch wird zur Verarbeitung medizinischer Daten (vor allem Patienten- und Bilddaten) auf externe Bibliotheken zurückgegriffen. Für eine Implementierung in Java standen als Werkzeuge *Pixelmed*⁵ und *dcm4che2*⁶ zur Auswahl. Beide stellen mit einer DICOM-Objekt-Verarbeitung und Netzwerkkommunikation ähnliche Dienste bereit.

dcm4che.org bietet allerdings eine Software namens *dcm4chee* um ein PACS zu betreiben. Diese wird im Labor für medizinische Bildverarbeitung eingesetzt. Das ermöglicht eine leichtere Integration für spätere Erweiterungen von jMediKit in die bestehende Umgebung des Labors.

dcm4che2 steht unter der GNU General Public License. Dies ermöglicht die freie Verwendung der Software. JMediKit setzt die Version 2.028 ein.

Java Advanced Imaging Image I/O Tools

Um alle Bildformate (abhängig von der Transfersyntax der DICOM-Objekte werden Bilddaten komprimiert als JPEG oder JPEG200 gespeichert) lesen zu können, verwendet *dmc4che2* die Bildverarbeitungsbibliothek *Java Advanced Imaging Image I/O Tools*. Diese ist *nicht* in *dcm4che2* integriert und muss vom Anwender selbst auf dem System installiert werden. Unter <http://download.java.net/media/jai-imageio/builds/release/1.1/> - Stand 31.01.2014 können die benötigten Dateien bezogen werden. Im Anhang B.4 befindet sich die Anleitung zur Installation.

5.7.2. Simple ITK

Das *Insight Segmentation and Registration Toolkit (ITK)*⁷ ist eine umfangreiche Sammlung an Algorithmen zur Segmentierung und Registrierung. Das ITK ist plattformübergreifend einsetzbar und wurde in C++ implementiert. Zwar sind Java- oder Python-Wrapper⁸

⁵<http://www.pixelmed.com/>

⁶<http://www.dcm4che.org/>

⁷<http://www.itk.org/>

⁸Wrapper machen es möglich, Bibliotheken fremder Sprachen in aktuellen Sprachen einzubinden und zu benutzen. So wäre es möglich, das ITK, welches in C++ implementiert ist, über Wrapper in Java nutzen zu können.

verfügbar, für die aktuelle Version 4 war es zum Zeitpunkt der Erstellung dieser Arbeit allerdings nicht möglich, zuverlässig einen ITK-Build für eine Verwendung in Java zu erstellen.

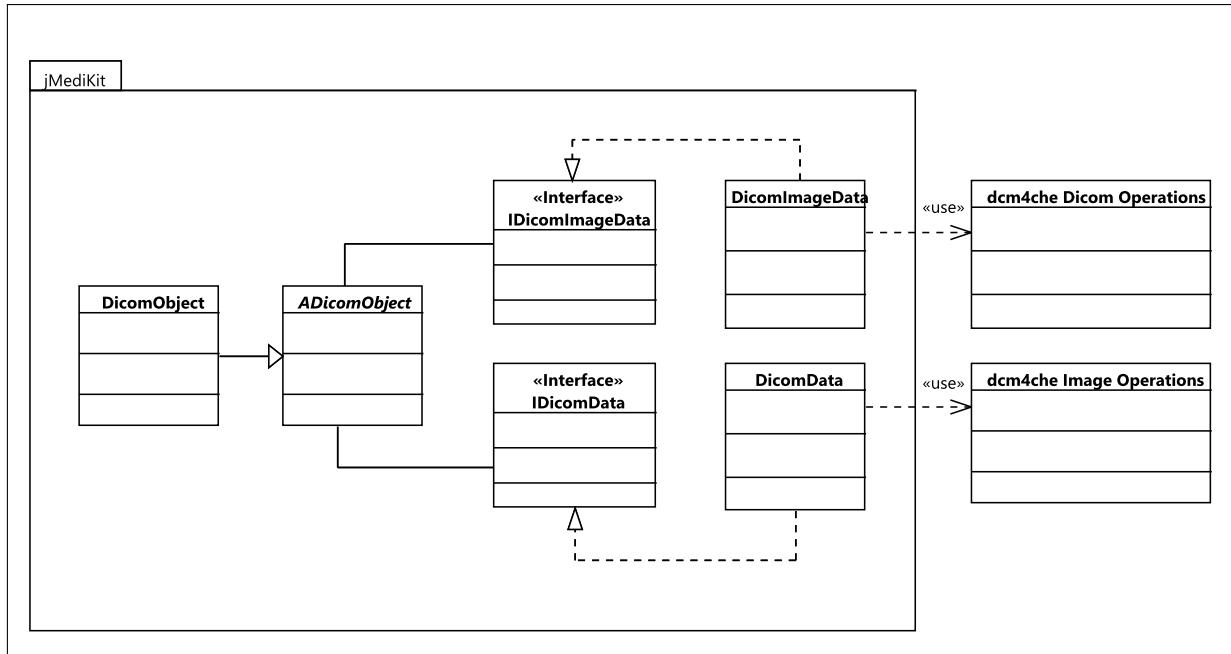
Das *Simple ITK*⁹ bietet eine kompakte Implementierung des ITK. Insgesamt sind 75% der Klassen aus dem ITK integriert[LVTN09]. Für einen Einsatz in einer Java Umgebung, kann das Projekt als Java-Bibliothek bezogen werden. Simple ITK steht unter der Apache 2.0 Lizenz und erlaubt eine uneingeschränkte Nutzung.

5.7.3. Der Adapter zur Auflösung von Abhängigkeiten

Eine Nutzung von externen Bibliotheken bedeutet gleichzeitig, dass Abhängigkeiten zwischen dem zu entwickelnden Programm und den fremden Paketen geschaffen werden. Werden bibliotheksspezifische Objekte im Quelltext erzeugt ist man abhängig von einer kontinuierlichen Weiterentwicklung der eingebundenen Projekte. Bei einem Wechsel der Bibliotheken müssen alle referenzierten Objekte angepasst werden und der Code ist nur mit einem erheblichen Mehraufwand wartbar.

Mit Hilfe des Adapters werden bei jMediKit die Abhängigkeiten aufgelöst. In Abbildung 5.8 wird das Klassendiagramm dargestellt. Wie das Plug-in Muster, wird auch der Adapter mit Anpassungen eingesetzt. Unter jMediKit wird ein DICOM-Objekt aus zwei unterschiedlichen Teilen repräsentiert. Ein Teil beinhaltet den reinen Datenteil (*IDicomData*). Das bedeutet, alle verfügbaren Tags des DICOM-Objekts sind enthalten. Der zweite Teil besteht aus den reinen Pixeldaten (*IDicomImageData*). Diese beiden Schnittstellen bilden den Adapter für die externe Bibliothek *dcm4che2*. Durch die Trennung von Daten und Bilddaten bleibt die Flexibilität bei der Bibliothekswahl erhalten. So kann zum Beispiel Simple ITK nur für das Einlesen der Bilddaten verwendet werden, denn es werden keine Methoden zum Auslesen der Tags geboten. Dadurch wäre eine zweite Bibliothek für den Datenteil notwendig. In der Implementierung des Adapters wird in beiden Teilen *dcm4ch2* eingesetzt, da diese Bibliothek sowohl Daten als auch Bilddaten verarbeiten kann. Die beiden Klassen *DicomImageData* und *DicomData* implementieren die Schnittstellen *IDicomImageData* und *IDicomData*. Keine anderen Klassen greifen auf Objekte und Methoden aus den Bibliotheken zu. Um beide Aspekte zu vereinen, stellen die Schnittstellen *IDicomData* und *IDicomImageData* die neuen zu adaptierenden Funktionen dar, die von *ADicomObject* adaptiert werden. Der konkrete Adapter ist *DicomObject*. Um die Abhängigkeiten aufzulösen und die Flexibilität in der Bibliothekswahl zu wahren, wird dieser geschachtelte Adapter eingesetzt.

⁹<http://www.simpleitk.org/>



5.

Abbildung 5.8.: Das Adapter-Muster unter jMediKit

Zusätzlich wird die Schnittstelle zur Bibliothek vereinfacht, indem nur Funktionen zur Verfügung gestellt werden, die tatsächlich eingesetzt werden. Werden zusätzliche Funktionen benötigt, kann das Interface der Adapter erweitert werden.

Soll zukünftig eine Bibliothek getauscht werden, muss nur das Interface neu implementiert werden und die Anwendung kann wie gewohnt eingesetzt werden.

5.8. Die Architektur der Bilddaten

Die Bilddaten sind ein Teil des Adapters *ADicomObject* des Java Medical Imaging Toolkits. Diese werden mit Hilfe der abstrakten Klasse *AImage* dargestellt und enthalten viele zusätzliche Informationen wie Bilddimension, Fensterungsdaten oder die Position des Patienten im Raum. *AImage* repräsentiert die Grundlage einer Bildrepräsentation von jMediKit. Wie in Kapitel 3 erläutert, besitzen medizinische Bilder unterschiedliche Grauwert-Tiefen oder Farbdarstellungen. Um die unterschiedlichen Bildtypen abzubilden, stehen die konkreten Klassen *UnsignedByteImage* → 8-Bit, *ShortImage* → 16-Bit, *UnsignedShortImage* → 16-Bit und *IntegerImage* → 32-Bit Farbbild zur Verfügung. Ein *ADicomObject* beinhaltet alle Bilddaten der entsprechenden Serie des DICOM-Objekts. Um mit *AImage* zu arbeiten gibt es zwei Möglichkeiten: Die Erste ist das Auslesen über das

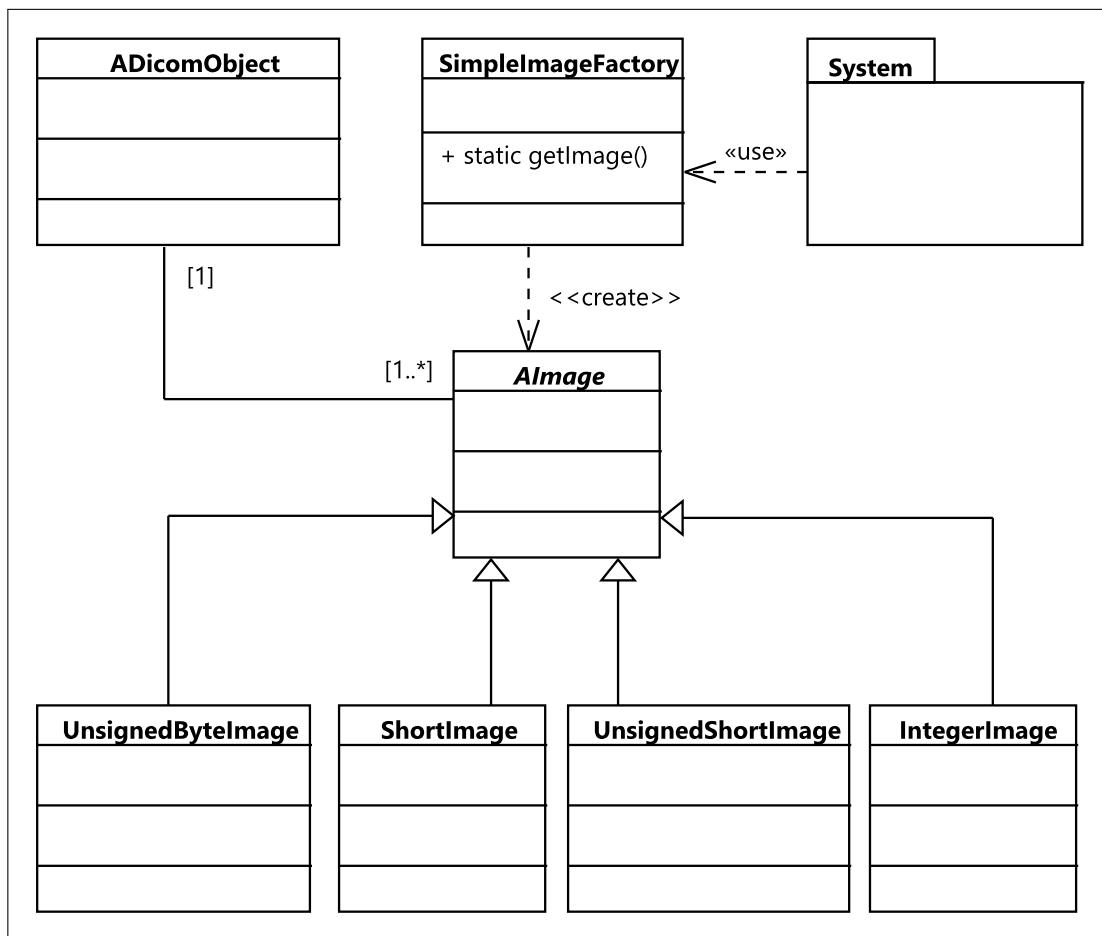


Abbildung 5.9.: Diagramm zur Klassenstruktur der Bilddaten

5.8.1. Die einfache Fabrik

Die einfache Fabrik ähnelt der Fabrikmethode, allerdings wird dieses Verfahren der Objekterzeugung nicht den Entwurfsmustern zugeordnet.

Beim Entwickeln von Plug-ins können Anwender vor dem Problem stehen, nicht zu wissen, welche Grauwert-Tiefe das aktuell geladene Objekt besitzt. Nur ein mühsames Auslesen über die DICOM-Tags und einer Schleife zur Typ-Ermittlung würde Abhilfe schaffen. Da

die Bilder bereits in den Speicher geladen wurden, ist dem System implizit der Bildtyp bekannt, jedoch nicht explizit dem Anwender. Durch die Übergabe eines bereits geladenen Bildes, kann ein neues Bild vom richtigen Typ, ohne explizites Wissen des Nutzers, erzeugt werden.

5.8.2. Struktur der Bilder im ImageViewPart

Da die Struktur der Bilddaten dargestellt wurde, kann nun die Vorgehensweise zur Anzeige dieser vorgestellt werden.

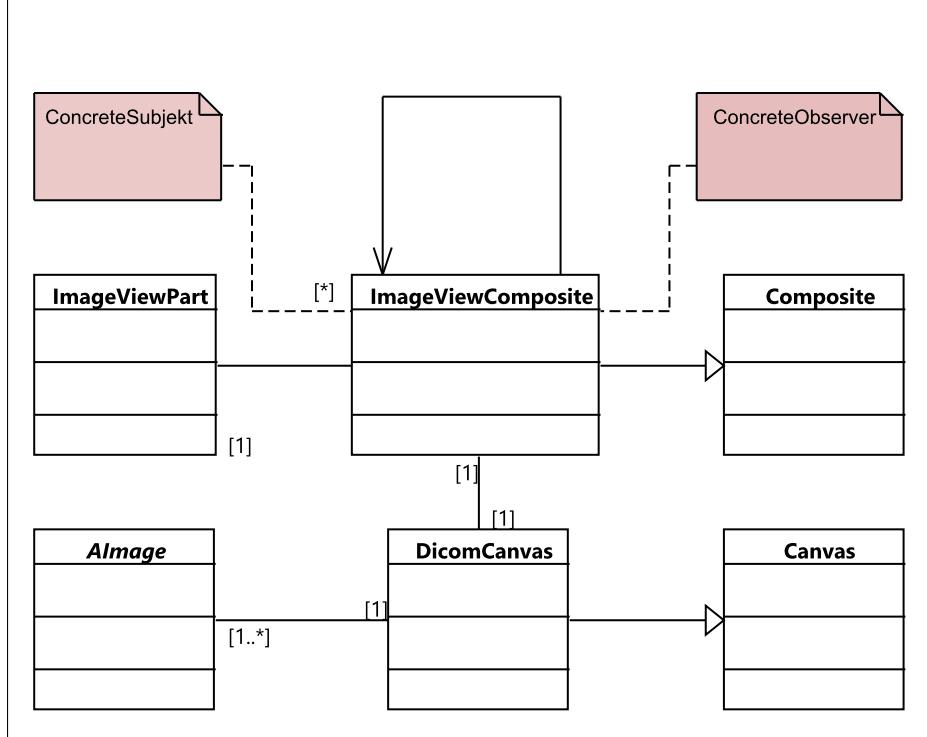


Abbildung 5.10.: Organisation der Klassen zur Anzeige der DICOM-Bilddaten

Auf der Benutzeroberfläche findet dieser Vorgang im ImageView Part statt (Abschnitt 5.4). Wie in Abbildung 5.10 zu sehen, bildet *ImageViewPart* die Grundlage und dient als Elternelement eines *ImageViewComposites*, von denen mehrere zum gleichen Zeitpunkt angezeigt werden können. *ImageViewComposites* erben von der SWT-Klasse *Composite* und enthalten weitere Bedienelemente für den Benutzer. Hierzu zählt die Scrollleiste am rechten Rand wie in Abbildung 5.11(a) zu sehen ist. Mit diesem Scrollbalken wird die

Schicht aus den dreidimensionalen Bilddaten gewählt, die vom *DicomCanvas* angezeigt werden soll. Die Superklasse von *DicomCanvas* gehört ebenfalls zum Repertoire des SWT und ist das zentrale Element zur Bildanzeige. *DicomCanvas* enthält sämtliche Bilder des DICOM-Objekts.

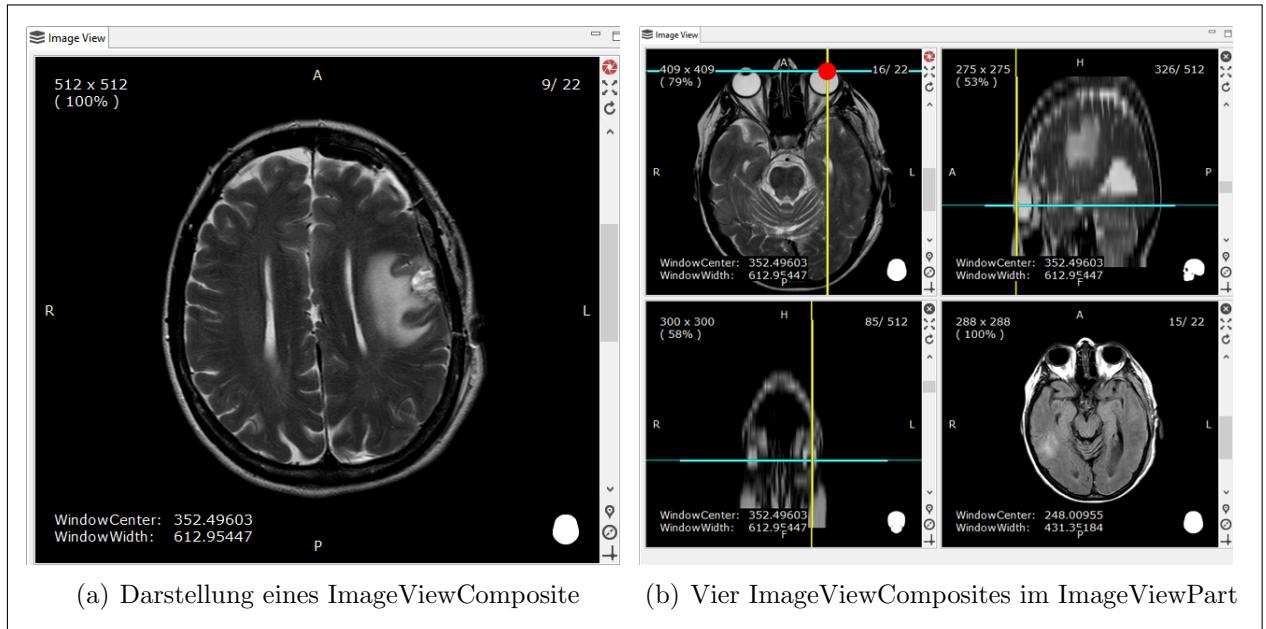


Abbildung 5.11.: Benutzeroberfläche der ImageViewComposites

Abbildung 5.11(b) zeigt vier *ImageViewComposites*, wobei alle außer Composite unten rechts die gleichen DICOM-Objekte anzeigen. Angenommen ein Benutzer führt einen Klick mit der rechten Maustaste auf den in rot markierten Punkt aus, müssen alle *ImageViewComposites* den gleichen Punkt im dreidimensionalen Raum referenzieren. Dieser wird durch den Schnittpunkt der Linien angezeigt. Eine genaue Beschreibung dazu befindet sich in Kapitel 6.

Damit dieses Verhalten von Seiten der Architektur unterstützt wird, muss sichergestellt werden, dass die *ImageViewComposites* untereinander kommunizieren können, dass eine Änderung stattgefunden hat.

Aus dem Kapitel 4 löst das Observer-Muster diese Anforderungen. Der Zyklus in Abbildung 5.10 repräsentiert das Muster und zeigt die Funktionsweise. Jedes *ImageViewComposite* ist gleichzeitig *ConcreteSubject* als auch *ConcreteObserver*. Das bedeutet, bei jeder Registrierung eines neuen DICOM-Objekts wird geprüft, ob dieses schon im *ImageViewPart* vorhanden ist. Ist dies der Fall, wird es als neuer Observer an bisherigen Subjects angemeldet. Bei diesem Einsatz besteht die Gefahr einer Endlosschleife, wenn der Zyklus nicht beachtet wird. Die Schleife tritt ein, wenn ein Subject die Observer benachrichtigt,

die Observer ändern den Status und benachrichtigen darauf wiederum ihre Beobachter und so weiter. Bei der Implementierung von jMediKit wird eine Änderung angenommen und das *DicomCanvas* neu gezeichnet und löst keine Rückmeldung einer Änderung aus. Auf dieser Architekturgrundlage findet die im folgenden Kapitel erläuterte Implementierung statt.

6. Implementierung

Dieses Kapitel erläutert den Implementierungsvorgang des Java Medical Imaging Toolkits. Besonderen Wert wird auf die Umsetzung der Blattknoten wie in Kapitel 5 Abschnitt 5.3 beschrieben gelegt, da diese Anwendungsteile direkt mit dem Anwender in Aktion treten. Nach einer Beschreibung wie die DICOM-Objekte repräsentiert werden, wird umfangreich auf die Bilddarstellung und Manipulation eingegangen.

6.1. Implementierung der DICOM-Objekte

Die beiden Schnittstellen *IDicomData* und *IDicomImageData* bilden die Grundlagen aller DICOM-Objekte die vom jMediKit erzeugt werden und sind gleichzeitig die einzige Kommunikationsmöglichkeit zu den externen Bibliotheken. Der abstrakte Adapter *ADicomObject* stellt die Funktionen beider Schnittstellen zur Verfügung. Die wichtigsten Methoden stellen das Auslesen der Pixel und den einzelnen Tags eines DICOM-Objekts dar.

Das eingebundene *dcm4che* bietet neben der Verarbeitung von DICOM-Objekten auch eine Implementierung des Kommunikations- und Speicherstandards von DICOM an. Während der aktuellen Entwicklung wurde allerdings nur ein Bruchteil der DICOM-Verarbeitung zur Erfüllung der Anforderungen benötigt und der Kommunikations- und Speicherprozess fand keine Beachtung. Bei einer Implementierung sollte dennoch eine zukünftige Erweiterung im Auge behalten werden. Dabei soll ebenso das Open-Closed-Prinzip Beachtung finden.

Hierbei werden die Schnittstellen in fachspezifische Domänen eingeteilt. Das bedeutet, *IDicomData* ist für das Verarbeiten der Tags zuständig, während *IDicomImageData* ausschließlich Bilddaten verarbeitet. Für weitere Versionen können weitere Domänen implementiert werden, wie zum Beispiel eine Schnittstelle *IDicomNetworkData*. Der Adapter vereint die Domänen zu einem vollen DICOM-Objekt.

6.2. Der DicomBrowser

Der DicomBrowser ist der zentrale Part zum Transfer der DICOM-Dateien aus dem Dateisystem zu les- und verarbeitbaren DICOM-Objekten. Abbildung 6.1 zeigt die Darstellung nach dem Einlesen eines Ordners mit DICOM-Dateien. Die Repräsentation entspricht dem ER-Modell aus Kapitel 3 Abschnitt 3.1.1. Der Knoten / symbolisiert die Wurzel. *BRAINIX* ist der für diesen Patienten eindeutige Wert seiner Patienten-Id, gefolgt von, in diesem Beispiel einer Studie, die wiederum aus sieben Serien besteht. Die einzelnen DICOM-Objekte als Blattknoten werden aufgrund der Übersichtlichkeit nicht angezeigt. Die Repräsentation der Dateien ist nicht immer geordnet wie es das ER-Modell vorgibt und man kann nicht von einer sortierten Ordnerstruktur ausgehen. Die PACS-Software *dcm4chee* ordnet die Daten beispielsweise nach dem Aufnahmedatum. Es wird eine Datenstruktur benötigt, die der DICOM Object Definiton entspricht.

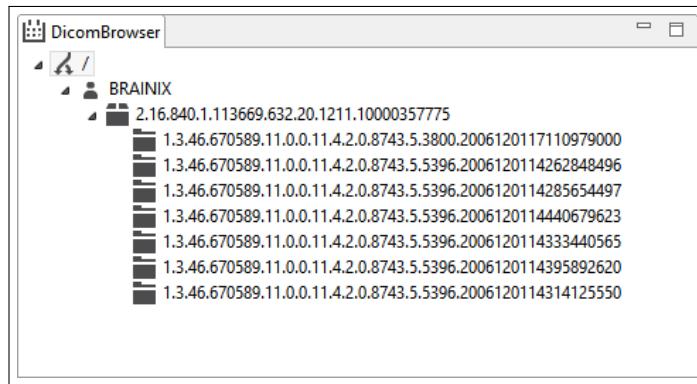


Abbildung 6.1.: Die Baumsicht des DICOM-Browsers mit geladenen Objekten

Sowohl die Anzeige, als auch die interne Behandlung der Daten soll so nah wie möglich an den DICOM-Standard angelehnt sein und unabhängig von der Auslieferung¹ der Dateien die Struktur des ER-Modells haben.

Ein Baum als Datenstruktur erfüllt die grundlegende Repräsentation mit den verschiedenen Knotentypen aus dem Patienten, Studien, Serien und den DICOM-Objekten.

Dadurch ergibt sich mit der Wurzel eine Höhe von

$$h = \max(4) \quad (6.1)$$

¹Unabhängig davon, ob Dateien von der Festplatte geladen oder über das Netzwerk bezogen werden.

und folgende konkrete Höhen der Knotentypen.

$$h_{Root} = 0 \quad h_{Patient} = 1 \quad h_{Study} = 2 \quad h_{Series} = 3 \quad h_{Object} = 4 \quad (6.2)$$

Der minimale Baum besteht nur aus der Wurzel und hat die Höhe $h_{min} = 0$. Nach der Multiplizität des Modells aus Abschnitt 3.1.1 hat der Baum, sobald ein Patientenname eingefügt wird, eine minimale Höhe von $h_{min} = 3$, da Patientenname und Study jeweils mindestens ein Kindelement enthalten. Die Breite des Baums ist unbestimmt, da Knoten eine beliebige Anzahl an Kindern besitzen können. Abbildung 6.2 zeigt einen Baum, wie er in der Anwendung repräsentiert werden könnte. Der Baum enthält alle Knotentypen von *Patientname* bis *Object*-Ebene.

Zwei Klassen des Quelltextes liefern die Basis des Baums:

- **DicomTreeRepository**

Diese Klasse repräsentiert den Baum. Sie enthält den Wurzelknoten, und einige graphentypische Operationen. Der Baum kann nach Knoten durchsucht werden und hat eine Funktion zum Einfügen neuer Knoten. Eine Löschfunktion wurde nicht implementiert, da der Baum nicht vom Benutzer manipuliert werden soll. Das Einfügen soll nur zum initialen Einlesen aufgerufen werden.

- **ADicomTreeItem**

ADicomTreeItem bilden die Knoten des Baumes. Jede Instanz besitzt eine Identifikationsnummer und kennt sowohl das Elternelement, als auch die Kindknoten.

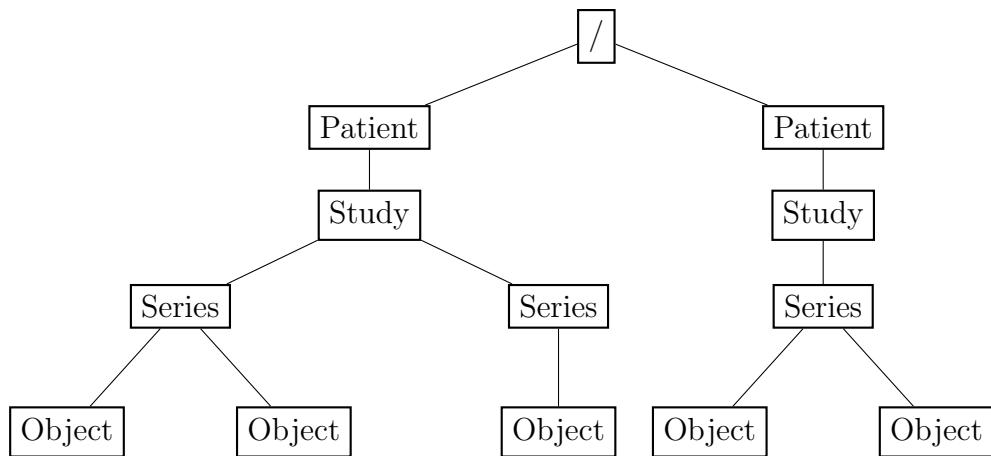


Abbildung 6.2.: Beispielhafte Darstellung eines Baumes mit $h = 4$ in der Implementierung

Da nun die Datenstruktur bestimmt ist, fehlt das Vorgehen zur Sortierung der Daten. Abbildung 3.3 in Abschnitt 3.1 zeigt eine mögliche Dateistruktur. Das liefert allerdings

keine Sicherheit, dass die Dateien immer vorsortiert zur Verfügung stehen. Liegen alle DICOM-Dateien in einem Ordner, wäre eine Einteilung in Patienten und Serien etc. nicht mehr möglich.

Wie bereits in Abschnitt 3.1 beschrieben besitzt jeder Patient, jede Studie und jede Serie eine eigene eindeutige Identifikationsnummer(UID), die eine modellgerechte Sortierung ermöglicht. Die Tags, die diese UIDs enthalten sind *Patient ID*, *Study Instace UID*, *Series Instance UID* und *SOP Instance UID*.

Jede Datei repräsentiert ein Blatt im Baum und somit ein DICOM-Objekt. Dadurch muss die abstrakte Klasse *ADicomObjekt* aus der Architekturbeschreibung aus Kapitel 5 Abschnitt 5.7.3 angepasst werden und erweitert die abstrakte Superklasse *ADicomTreeItem*. Somit erben alle DICOM-Objekte die Eigenschaften eines Knoten im Baum. Zur weiteren Klassifizierung der Knoten werden die Klassen *DicomPatientItem*, *DicomStudyItem* und *DicomSeriesItem*, die alle von *ADicomTreeItem* erben, eingesetzt. Bei der Instantiierung des DICOM-Objekts wird direkt die UID über *SOP Instance UID* zugewiesen. Als nächster Schritt wird aus dem DICOM-Objekt der Pfad von der Wurzel zum Objekt ermittelt. Dazu werden *Patient ID*, *Study Instace UID* und *Series Instance UID* des Objekts ausgelernt. Nun wird der Baum nach den entsprechenden Objekten und den UIDs durchsucht. Sind diese nicht vorhanden, wird das zugehörige Objekt erzeugt und in den Baum eingefügt, bis letztendlich das DICOM-Objekt als Blatt eingehängt werden kann.

Mittels dieser Sortierung entsprechen die Elemente im DicomBrowser der Darstellung des ER-Modells.

6.

6.3. Repräsentation der Pixeldaten

Ganzzahlige Datentypen in Java (*byte*, *short*, *int*, *long*) werden im Zweierkomplement kodiert[Ull07, S.106]]. Dadurch sind nur Werte im Bereich von $[-2^{BIT-1}, 2^{BIT-1} - 1]$ wobei *BIT* dem Speicherbedarf eines einzelnen Datums des Datentyps entspricht. Das entspricht beim Typ *short* dem Intervall von $[-2^{15}, 2^{15} - 1] \rightarrow [-32768, 32767]$.

Medizinische Grauwertbilder besitzen meist eine Tiefe von 8-, 12- und 16-Bit. Zusätzlich bestimmt der DICOM-Tag *PixelRepresentation*, ob Pixelwerte vorzeichenbehaftet sind. Durch diese variablen Eigenschaften entstehen unterschiedliche Bildtypen. Aus dem Abschnitt 3.2.2 wird deutlich, dass Grauwertbilder mit einer Tiefe von 16-Bit den Bereich von $[0, 65535]$ abdecken. Dadurch entsteht eine Diskrepanz zwischen dem 16-Bit Java Datentyp *short* und den Grauwerten. Die Pixelwerte können vom Typ *short* nicht aufgenommen werden. Das gleiche Missverhältnis entsteht bei einer Tiefe von 8-Bit. Wie in Tabelle 6.1

Implementierung

Datentyp	MIN	MAX	Unsigned
byte	-128	127	0 - 255
short	-32768	32767	0 - 65535
int	-2147483648	2147483647	0 - 4294967295
long	-9223372036854775808	9223372036854775807	0 - 18446744073709551615

Tabelle 6.1.: Ganzzahlige Datentypen in Java

Klassenname	Pixeltyp Code	Bittiefe Code	Bittiefe DICOM- Objekt	Vorzeichen
UnsignedByteImage	short	16	8	Ø
ShortImage	short	16	16	ja
UnsignedShortImage	int	32	16	Ø
IntegerImage (Farbbild)	int	32, 8-Bit je Kanal	32, 8-Bit je Kanal	Ø

Tabelle 6.2.: Von jMediKit implementierte Bildtypen

zu sehen, fasst der Datentyp *byte* maximal einen Wert von 127 während der größte Pixelwert 255 entspricht. Daraus folgt, dass Grauwertbilder ohne vorzeichenbehaftete Werte (Unsigned) mit dem nächsthöheren Datentyp repräsentiert werden.

Tabelle 6.2 zeigt eine Darstellung der implementierten Bildtypen und die zugehörigen Datentypen der Pixel im Quelltext. Haben die Pixeldaten eines DICOM-Objekts eine Tiefe von 8-Bit und sind nicht vorzeichenbehaftet, wird zur Repräsentation in der Implementierung ein Array des Typs *short* verwendet, um alle Werte aufnehmen zu können. Der Datentyp *int* könnte alle Pixelwerte eines DICOM-Objekts aufnehmen. So liegt es nahe, dass Integer durchgehend als Datentyp verwendet wird. Arbeitet man allerdings mit 8-Bittiefe ohne Vorzeichen, wird der Speicherbedarf von 16 auf 32 Bit pro Pixel verdoppelt. Daher ist es sinnvoll die Bildtypen zu kategorisieren.

6.4. Räumliche Sortierung der Bilddaten

Beim Erstellen des Baumes aus Abschnitt 6.2 wird rekursiv das Verzeichnis durchlaufen und nach lesbaren DICOM-Objekten gesucht. Ist die Suche erfolgreich, wird das Objekt dem Baum hinzugefügt. Hierbei kann das Problem auftreten, dass Dateien in der falschen Reihenfolge importiert werden. So kann es passieren, dass ein Importvorgang, abhängig

vom Dateinamen, Objekte dem Baum hinzufügt. Die räumliche Reihenfolge der einzelnen Schichten entspricht allerdings keineswegs dem Dateinamen² oder anderen dateibezogenen Reihenfolgen. Dadurch wird, wie schon bei der Sortierung nach dem ER-Modell, eine Methode benötigt, mittels DICOM-Tags die korrekte Reihenfolge der Bilddaten herzustellen.

Der Standard verfügt über mehrere Tags, welche die richtige Reihenfolge andeuten. *Instance Number* ist nach dem Standard [Nat11a, C.7.6.1] eine Nummer, die ein Bild identifiziert. Während der Entwicklung entsprach dieser Wert der Testbilder der dargestellten Reihenfolge, jedoch ist keine Information enthalten, ob diese der tatsächlichen Reihenfolge im Raum entspricht. So könnte der Wert für die Aufnahmenreihenfolge oder andere konsekutive auf- oder absteigende Folgen stehen. Dadurch ist *Instance Number* nur bedingt geeignet.

Ein Tag, der räumliche Informationen enthält ist *Slice Location*. Nach dem Standard [Nat11a, C.7.6.2] ist dieser Wert die relative Position der Bildebene in mm. Es ist allerdings keine Information enthalten, ob die Sortierung in steigender oder fallender Reihenfolge erfolgt. Da der Tag zusätzlich nur optional vorhanden ist, kann eine Nutzung zur Bestimmung der Anordnung ausgeschlossen werden.

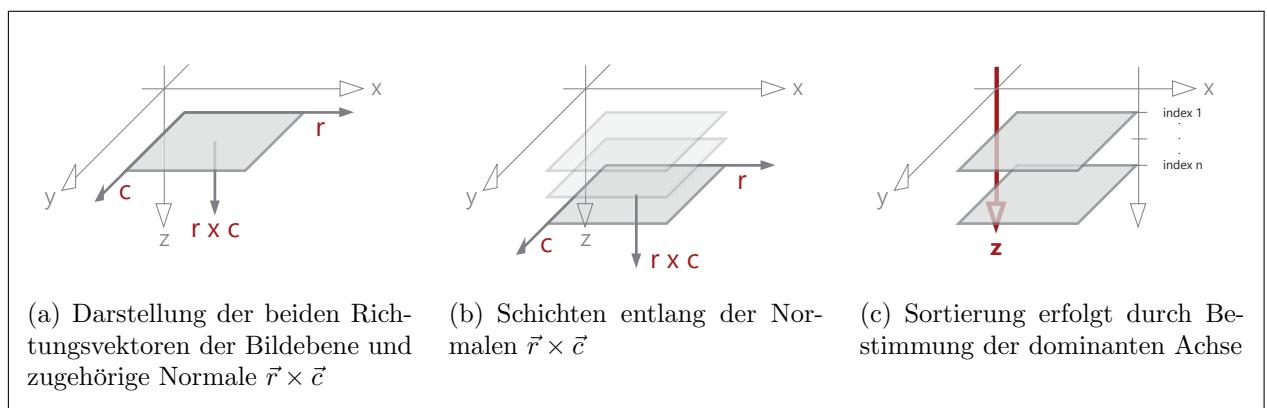


Abbildung 6.3.: Räumliche Sortierung der DICOM-Objekte

In der Mailingliste des Insight Segmentation and Registration Toolkits [Smi03] wird eine Berechnung über die Attribute *Image Position* und *Image Orientation* vorgeschlagen. *Image Position* enthält die x , y und z Koordinate in mm und *Image Orientation* den Richtungskosinus³ der ersten Reihe und Spalte des Bildes in Abhängigkeit des Patienten. Diese beiden Richtungsvektoren spannen die Bildebene auf und müssen nach [Nat11a,

²Je nach Sortierung innerhalb des Betriebssystems könnte ein Import auch nach dem Änderungsdatum erfolgen.

³Ein Richtungskosinus beschreibt die Winkel des Vektors zu den drei Koordinatenachsen.

C.7.6.2.1.1] orthogonal zueinander sein.

Die Abbildungen in 6.3 zeigen das Koordinatensystem des Patienten als rechtshändiges System[Nat11a, S.419]. Dieses ist um 180 Grad um die x-Achse gedreht. In Abbildung 6.3(a) ist die Bildebene sowie die beiden Richtungsvektoren \vec{c} der ersten Spalte und \vec{r} der ersten Reihe dargestellt. Mit Hilfe des Kreuzproduktes lässt sich nun die Normale der Ebene bestimmen. Die räumliche Anordnung der Schichten erfolgt entlang des Normalenvektors $\vec{r} \times \vec{c}$ (Abbildung 6.3(b)). Die Richtungsvektoren könnten folgende Darstellung besitzen:

$$\vec{r} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \vec{c} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \vec{n} = \vec{r} \times \vec{c} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad (6.3)$$

6.

Nachdem die Normale \vec{n} der Ebene bestimmt ist, muss der dominante Anteil des Vektors ermittelt werden. Dazu wird der maximale Betrag aus den Elementen von \vec{n} mit $|n_x|, |n_y|, |n_z|$ berechnet. Somit erhält man die Achse an der die Schichten angeordnet sind. Im Beispiel 6.3 und Abbildung 6.3(c) erfolgt die Anordnung entlang der z-Achse, da n_z den dominanten Anteil von \vec{n} darstellt. Ist $n_z < 0$ erfolgt die Wuchsrichtung der Schichten entlang des negativen Anteils der z-Achse. Wenn $n_z \geq 0$ wachsen die Ebenen in die positive Richtung.

Mit Hilfe dieser Kriterien lassen sich die Bilder über den Tag *Image Position* sortieren. Es ist bekannt, welche Achse die Reihenfolge im Raum symbolisiert. Beispiel 6.4 zeigt drei Vektoren mit möglichen Daten von *ImagePosition*. Ist der dominante Anteil des Vektors kleiner 0, verläuft die Wuchsrichtung negativ und der größte Wert ist das erste Bild der Folge. Ist der dominante Anteil positiv, hat das erste Bild den kleinsten Wert. Da im Beispiel 6.3 $n_z \geq 0$ entspricht die Sortierte Reihenfolge aus Beispiel 6.4 $\vec{b} \rightarrow \vec{a} \rightarrow \vec{c}$.

$$\vec{a}_{position} = \begin{pmatrix} 13.6 \\ 122 \\ 75 \end{pmatrix} \quad \vec{b}_{position} = \begin{pmatrix} 13.7 \\ 121.6 \\ 65 \end{pmatrix} \quad \vec{c}_{position} = \begin{pmatrix} 12.6 \\ 122.1 \\ 85 \end{pmatrix} \quad (6.4)$$

Durch eine Implementierung des Interface *Comparable<AImage>* in *AImage*, unter Berücksichtigung dieser Vorgehensweise, ist eine einfache und für diesen Zweck ausrei-

chend schnelle Sortierung der Bildebenen möglich. Somit erfolgt die Anordnung des Baumes und der Bilder unabhängig von der Dateistruktur.

Voraussetzung für diese Umsetzung ist, dass die Bildebenen parallel zu den Koordinatenachsen verlaufen. Bei Bildreihen mit Kurven oder einem schrägen Verlauf kann keine dominante Achse bestimmt werden.

6.5. Zeichnen und manipulieren der Bilddaten

Sowohl die DICOM-Objekte, als auch Bilddaten stehen im Speicher zur Verarbeitung bereit. Dieser Abschnitt befasst sich mit der Visualisierung dieser Daten. Wie aus Kapitel 5 Abschnitt 5.8.2 bekannt ist, erfolgt die Darstellung der Bilder im *ImageViewPart* der Anwendung. Innerhalb des Parts können bis zu vier *ImageViewComposites* angezeigt werden. Diese dienen vor allem zur Bedienung des enthaltenen *DicomCanvas*. So kann mit Hilfe der Scrollleiste durch den dreidimensionalen Datensatz navigiert oder Informationen wie das Koordinatensystem angezeigt und versteckt werden. Die tatsächliche Anzeige der Bilddaten findet im Canvas-Element statt. Eine direkte Manipulation der Bilder erfolgt durch die Auswahl eines Werkzeugs.

Die Breite und Höhe der Zeichenfläche ist unter anderem abhängig von der Zahl der angezeigten *ImageViewComposites*, der Bildschirmgröße und dessen Auflösung. Je mehr Composites angezeigt werden, desto kleiner wird der Bereich für das *DicomCanvas*. So kann der Fall eintreten, dass die Dimension der Zeichenfläche nicht ausreicht, um ein Bild vollständig anzuzeigen, da das Bild größer als die zur Verfügung stehende Fläche ist. Deshalb muss dem Nutzer eine Möglichkeit gegeben werden, nicht sichtbare Details in den sichtbaren Bereich schieben zu können.

Neben dem Verschieben des Bildes gibt es zwei weitere essentielle Werkzeuge zur Manipulation medizinischer Bilddaten. Bei einer Darstellung in 100% der Bildgröße sind Details nicht immer gut zu erkennen, weshalb eine Skalierung des Bildes möglich sein muss. Das zweite Werkzeug übernimmt die Fensterung der Grauwerte. Je nach Struktur oder Gewebe das betrachtet werden soll, müssen die Fensterungswerte individuell zu wählen sein. Zusätzlich zu den Werkzeugen gibt es drei verschiedene Ansichten, aus denen der Benutzer wählen kann. Damit kann bestimmt werden, welche Ebene der 3D-Bilddaten angezeigt werden soll. Abbildung 6.4 zeigt die verschiedenen Optionen. Die einzelnen Ansichten sind die (x, y) -Ebene (axial), die (x, z) -Ebene (coronal) und die (y, z) -Ebene (sagittal).

Die Faktoren DICOM-Objekt, Bedienelemente der Composites, Werkzeuge und die ver-

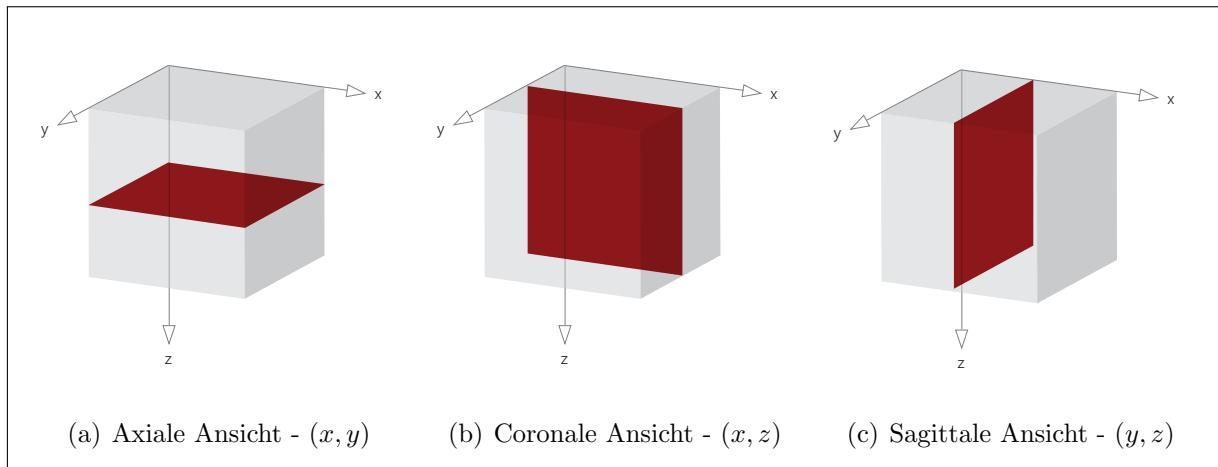


Abbildung 6.4.: Die verschiedenen Ansichten der Ebenen eines dreidimensionalen Datensatzes

6.

schiedenen Ansichten der Ebenen beeinflussen maßgeblich den Prozess der Anzeige der Bilddaten. Die Vorgehensweise wird in Abbildung 6.5 visualisiert. Der Prozess gliedert sich in sechs zentrale Verarbeitungsblöcke und zwei Einhängepunkte(Hooks) für Werkzeuge, die für eine Verarbeitung vom Öffnen der Datei bis zur Anzeige auf dem Bildschirm nötig sind.

1. Ebenenrekonstruktion

Nach Bedarf hat der Anwender die Möglichkeit eine axiale, coronale oder sagittale Ebenendarstellung zu wählen. Wird diese Option angewendet, wird der Bilddatensatz neu berechnet.

2. Bilddaten initialisieren

Abhängig vom gewählten Index der Scrollleiste des *ImageViewComposites* wird das entsprechende Bild aus dem 3D-Datensatz geladen.

3. Koordinaten des geladenen Bildes berechnen

Wird ein Bild auf der Zeichenfläche verschoben, ändern sich die Koordinaten des Bildes. Dieser Block übernimmt die Berechnung, damit die Daten korrekt angezeigt werden.

4. Bild interpolieren

Bei einer Skalierung ändert sich die Bildgröße. Die fehlenden Pixel einer Vergrößerung müssen interpoliert werden.

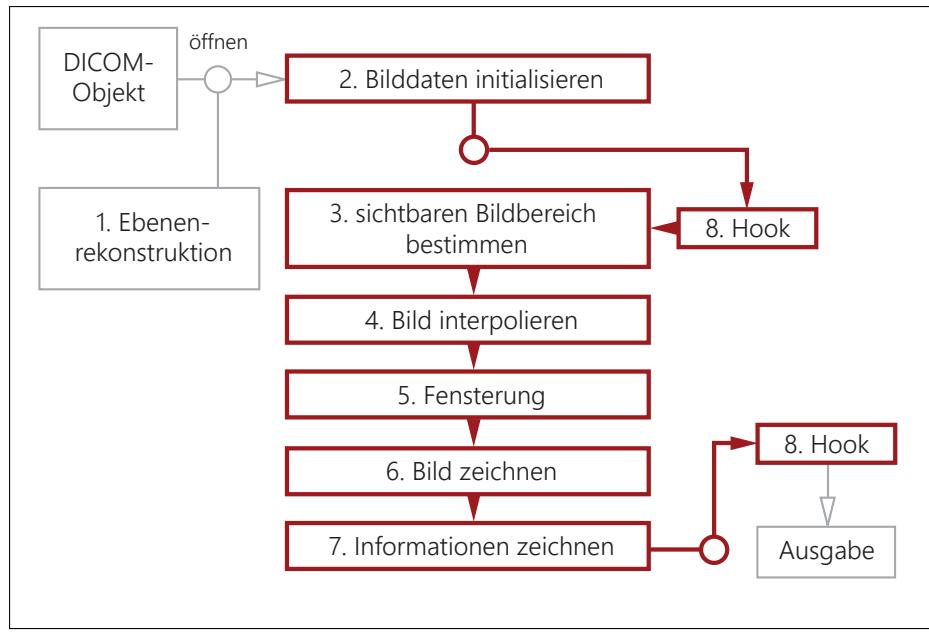


Abbildung 6.5.: Der Prozess von der Bildauswahl zur Anzeige

5. Fensterung

Mit Hilfe der Fensterung werden die Grauwerte entsprechend den Werten von Window Width und Window Center auf den Bereich von 0 - 255 abgebildet.

6. Bild zeichnen

Nachdem alle notwendigen Werte berechnet und die Grauwerte durch die Fensterung bestimmt wurden, kann das Bild auf der Zeichenfläche abgebildet werden.

7. Informationen zeichnen

Der nächste Schritt ist das Zeichnen zusätzlicher Informationen für den Anwender. Hierzu zählen zum Beispiel die Koordinatenachsen oder die Werte zu Window Width und Window Center, Bildgröße sowie die Orientierungslinien.

8. Hooks

Beide Hooks stellen Einhängepunkte für Werkzeuge dar. Das sind abstrakte Funktionen, die von konkreten Werkzeugen implementiert werden. Jeweils vor den Berechnungen und nach Abschluss des Zeichenvorgangs werden diese Funktionen aufgerufen und ausgeführt. So könnten zum Beispiel je nach Werkzeug zusätzliche Informationen eingezeichnet werden.

Die Bereiche der Rekonstruktion, Koordinatenberechnung, Interpolation und das Zeichnen der Informationen werden in den folgenden Abschnitten genauer erläutert. Die Initialisierung und das Zeichnen des Bildes sind einfache Operationen. Die Fensterung entspricht dem Algorithmus 1 aus Abschnitt 3.2.2.

6.5.1. Die Rekonstruktion der Ebenen

Mit einem dreidimensionalen Datensatz ist es möglich, eine beliebige Ebene durch die Voxel zu legen. Dadurch kann man unter anderem eine dreidimensionale Darstellung berechnen, da die Ebenen aus frei wählbaren Winkeln betrachtet werden können. Im Rahmen der Anwendung dieser Arbeit ist die axiale, coronale und sagittale Darstellung allerdings ausreichend.

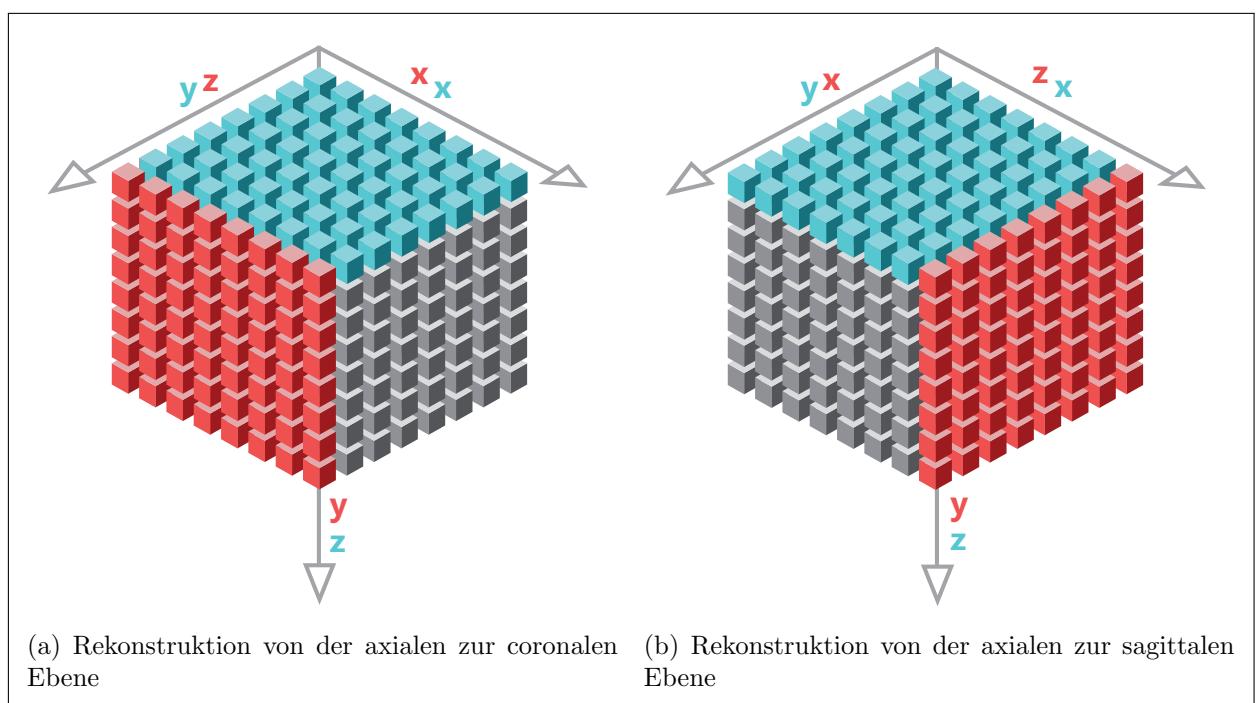


Abbildung 6.6.: Rekonstruktion der coronalen und sagittalen Ebene

Die Abbildungen 6.6 zeigen jeweils die *erste* Bildschicht der axialen Quelldarstellung als blaue und die *letzte* Schicht der Zieldarstellung als rote Voxel. Da die Quell- und Zielebene immer orthogonal zueinander sind, kann auf zeitintensive geometrische Berechnungen verzichtet werden. Die neuen Bildebene können über die Indizes der Pixel der verschiedenen Bilder schnell bestimmt werden. Die Bildgröße der rekonstruierten Ebene ist abhängig von der ursprünglichen Ebenendarstellung. So entspricht, wie Abbildung 6.6(a) zeigt, die Höhe(y) eines coronalen Bildes der Tiefe(z) einer axialen Darstellung und

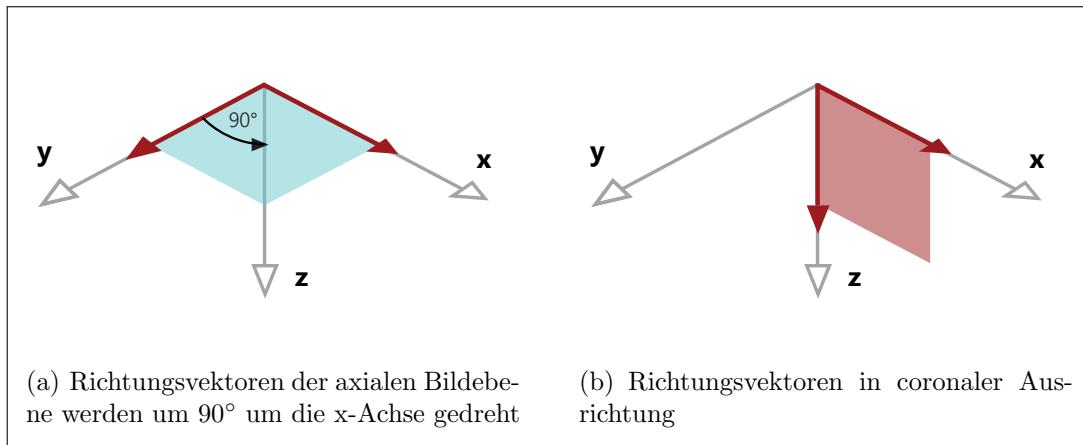


Abbildung 6.7.: Rotation der Richtungsvektoren von axialer zu coronaler Darstellung

die coronale Tiefe(z) der axialen Höhe(y). Die Bildbreite bleibt unverändert. Die erste Pixelreihe der Rekonstruktion kann mit den Werten der letzten Reihe der ersten axialen Bildschicht belegt werden. Algorithmus 2 zeigt die Möglichkeit einer Implementierung.

6.

Algorithmus 2: Berechnung der Bildebenen von axialer zu coronaler Darstellung

```

1: images  $\leftarrow$  input - Quellbilder in axialer Darstellung
2: reconstruction  $\leftarrow$  output - rekonstruierte coronale Bildebenen
3:  $X_{coronal} \leftarrow X_{axial}$ 
4:  $Y_{coronal} \leftarrow Z_{axial}$ 
5:  $Z_{coronal} \leftarrow Y_{axial}$ 
6: for all  $z \in Z_{coronal}$  do
7:   for all  $y \in Y_{coronal}$  do
8:     for all  $x \in X_{coronal}$  do
9:        $i \leftarrow$  get image( $y$ ) from images { $y$  entspricht dem  $z$ -Wert}
10:       $p \leftarrow$  get pixel( $x, z$ ) from  $i$  { $x$  entspricht dem  $x$ -Wert und  $z$  dem  $y$ -Wert}
11:      set reconstructedPixel( $xy$ ) =  $p$ 
12:    end for
13:  end for
14: end for

```

Nachdem die Bildebenen neu berechnet wurden, muss zusätzlich eine Drehung der Richtungsvektoren im DICOM-Tag *ImageOrientation* vorgenommen werden. Aufgrund der Orthogonalität wird immer in 90°-Winkeln um die x-, y- und z-Achsen gedreht. Dadurch wird gewährleistet, dass spätere Berechnungen, wie zum Beispiel die Beschriftung der Koordinatenachsen, korrekt durchgeführt werden können. Die Drehung erfolgt mit Hilfe der Drehmatrizen. Es wird zuerst um die x-, gefolgt von der y- und z-Achse rotiert.

Die Abbildungen 6.7(a) und 6.7(b) zeigen den Prozess der Drehung von der axialen in

die coronale Ebenendarstellung mit einer Rotation über 90° um die x-Achse. Folgende Drehmatrix wird zur Berechnung eingesetzt [NFPS11, 7.3.2].

$$R^x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (6.5)$$

Die Rotationsrichtung und die Drehachse ist, wie die rekonstruierten Bilder, abhängig von der Quell- und Zieldarstellung. So könnte eine sagittale Darstellung aus axialen Richtungsvektoren durch eine Drehung um die x-Achse, gefolgt von einer Rotation um die z-Achse berechnet werden.

6.

6.5.2. Berechnung der Bildkoordinaten

Nach einer eventuellen Ebenenrekonstruktion und der Initialisierung der Bilddaten können die Koordinaten und Eigenschaften des Bildes auf der Zeichenfläche bestimmt werden. Die Werkzeuge zur Translation und Skalierung beeinflussen die Koordinaten des Bildes. Während das Verschieben unproblematisch für die Performance der Anwendung ist, hat die Skalierung, im Besonderen die Vergrößerung, maßgeblichen Einfluss auf die Rechenzeit. Angenommen die Zeichenfläche hat eine Größe von 500×500 und 250.000 Pixel und das Bild mit den Maßen 1000×1000 und 1.000.000 Pixel die doppelte Größe. Daraus folgt, dass maximal $\frac{1}{4}$ der Pixel des Bildes auf der Zeichenfläche angezeigt werden können. Die sechs Blöcke des Zeichenprozesses verarbeiten auch die nicht sichtbaren Anteile des Bildes und verbrauchen dadurch unnötig Rechenzeit. In dem Beispiel bedeutet das einen Mehraufwand von 75%. Um diese zusätzliche Rechenzeit einzusparen, darf die Berechnung der Bildkoordinaten nur den aktuell sichtbaren Bereich in die Rechnungen einbeziehen.

Wie Abbildung 6.8(a) zeigt, bestimmt *DicomCanvas* das Koordinatensystem mit der linken oberen Ecke als Ursprung. Die Position der Bilder wird durch vier Parameter bestimmt. Die Koordinaten (x, y) legen den Ursprung des Bildes fest und $(width, height)$ bestimmen mit Breite und Höhe die Dimension. Die tatsächlichen Koordinaten des Bildes sind dadurch (x, y) und $(x + width, y + height)$. Liegen diese vier Parameter außerhalb der Zeichenfläche wie in Abbildung 6.8(b), muss der Abstand (*offset*) von x , y , $width$ und $height$ zum Canvas berechnet werden.

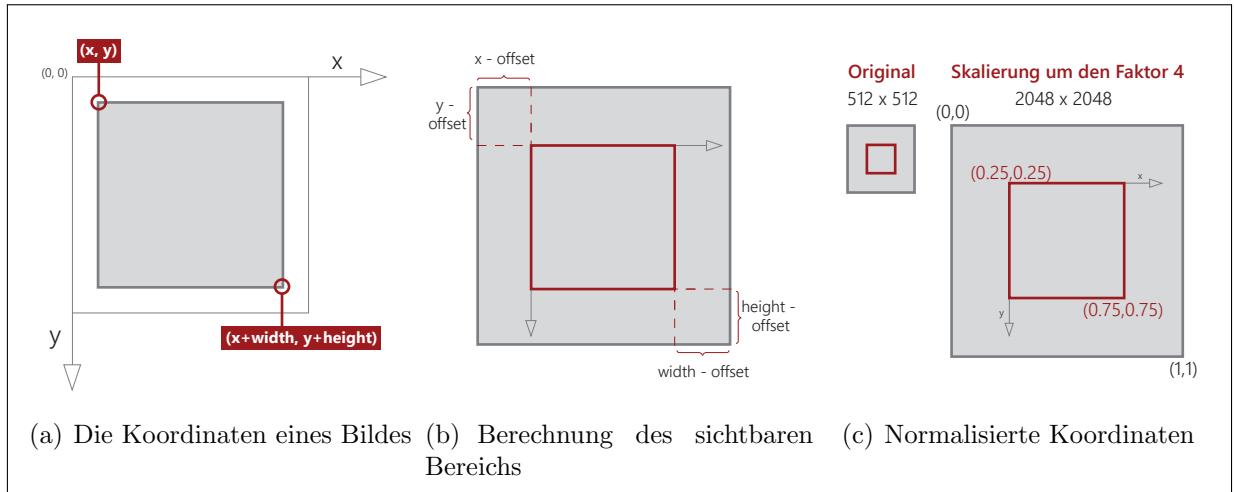


Abbildung 6.8.: Berechnung der Position und Koordinaten. Die graue Fläche symbolisiert das Bild und der rote Rahmen stellt den sichtbaren Bildausschnitt dar.

6.

Das *DicomCanvas* speichert nur das Originalbild und den aktuell sichtbaren Bildausschnitt. Da bei skalierten Bildern nur die Dimension und nicht die Pixeldaten selbst hinterlegt sind, muss der sichtbare Bereich aus dem Original interpoliert werden. Um die richtigen Pixelwerte zu finden, wird mit normalisierten Koordinaten zwischen den Werten $(0, 0)$ für (x, y) und $(1, 1)$ für $(x + width, y + height)$ gearbeitet. Abbildung 6.8(c) zeigt ein Bild mit der Dimension 512×512 , das um den Faktor vier auf 2048×2048 vergrößert wurde. Der sichtbare Bildausschnitt erstreckt sich von $(512, 512)$ bis $(1536, 1536)$. Das entspricht dem Bereich von $(128, 128)$ bis $(384, 384)$ im Originalbild. Mit Hilfe der normalisierten Koordinaten lassen sich die Bildbereiche bei gleichem Seitenverhältnis von Original und skaliertem Bild unabhängig von der Größe bestimmen. Für den Bildausschnitt im Beispiel entspricht der Bereich in normalisierter Darstellung $(0.25, 0.25)$ für (x, y) und $(0.75, 0.75)$ für $(width, height)$. Mit gegebenen Pixelwerten lassen sich die normalisierten Koordinaten wie folgt bestimmen.

$$x_{norm} = \frac{x}{width} \quad y_{norm} = \frac{y}{height} \quad (6.6)$$

Entsprechend dieser Gleichung erfolgt die Berechnung der Indizes von x und y.

$$x = x_{norm} \cdot width \quad y = y_{norm} \cdot height \quad (6.7)$$

Mit diesen Hilfsmitteln lässt sich nun der obere linke und untere rechte Abstand des Bildes zur Zeichenfläche bestimmen. Zuerst wird geprüft ob x und/oder y einen Wert < 0 haben. Ist dies der Fall folgt daraus, dass das Bild aus der oberen und/oder linken Ecke

der Zeichenfläche ragt. Die normalisierten Koordinaten lassen sich mit der Formel

$$x_{norm} = \frac{|x|}{width} \quad y_{norm} = \frac{|y|}{height} \quad (6.8)$$

ermitteln. x und y sind der Abstand vom Ursprung des Koordinatensystems zum Ursprung des Bildes und somit nicht Teil des sichtbaren Bildausschnittes. Der Betrag ist notwendig, da nur Werte zwischen 0 und 1 als normalisierte Koordinaten akzeptiert werden.

Angenommen es existiert das Bild aus dem Beispiel in Abbildung 6.8(c) mit den Koordinaten $(-512, -512), (1536, 1536)$ und einer Dimension $(width, height)$ von $(2048, 2048)$ sowie eine Zeichenfläche mit der Dimension $(0, 0)$ und $(1024, 1024)$, dann ergeben sich daraus folgende normalisierte Koordinaten:

$$x_{norm} = \frac{|-512|}{2048} = 0.25 \quad y_{norm} = \frac{|-512|}{2048} = 0.25 \quad (6.9)$$

Sind x und/oder $y >= 0$ folgt daraus, dass die obere linke Ecke des Bildes vollständig zu sehen ist. Das entspricht den normalisierten Werten $(0, 0)$.

Der nächste Schritt ist die Prüfung der unteren rechten Ecke. Bevor die Koordinaten bestimmt werden, muss der Abstand berechnet werden, da dieser nicht wie bei x und y abgelesen werden kann. Durch Subtraktion der Bildkoordinaten $(x + width, y + height)$ von der Zeichenfläche erhält man den Abstand $offset$ zwischen Bild- und Canvasrand.

6.

$$offset_{width} = x + width - width_{Canvas} \quad offset_{height} = y + height - height_{Canvas} \quad (6.10)$$

Subtrahiert man den Abstand von der Bilddimension, erhält man die tatsächlichen Koordinaten die noch im sichtbaren Bereich der Zeichenfläche liegen. Damit lassen sich die normalisierten Koordinaten berechnen.

$$x_{norm} = \frac{width - offset_{width}}{width} \quad y_{norm} = \frac{height - offset_{height}}{height} \quad (6.11)$$

Für die Daten aus dem Beispiel ergeben sich folgende Werte:

$$offset_{width} = -512 + 2048 - 1024 = 512 \quad offset_{height} = -512 + 2048 - 1024 = 512 \quad (6.12)$$

$$x_{norm} = \frac{2048 - 512}{2048} = 0.75 \quad y_{norm} = \frac{2048 - 512}{2048} = 0.75 \quad (6.13)$$

Mit den berechneten Werten der normalisierten Koordinaten $(0.25, 0.25)$ und $(0.75, 0.75)$ lässt sich der zu interpolierende Bereich aus dem Originalbild bestimmen.

$$x_1 = 0.25 \cdot 512 = 128 \quad y_1 = 0.25 \cdot 512 = 128 \quad (6.14)$$

$$x_2 = 0.75 \cdot 512 = 384 \quad y_2 = 0.75 \cdot 512 = 384 \quad (6.15)$$

6.5.3. Bilineare Interpolation

Es können verschiedene Interpolationsmethoden eingesetzt werden, um das Bild oder einen Bildausschnitt zu vergrößern oder zu verkleinern. Die Abbildungen 6.9 zeigen die Ergebnisse einer Nearest Neighbor, bilinearer und bikubischen Interpolation. Das Nearest Neighbor Verfahren stellt die schnellste Interpolationsmethode dar, liefert allerdings auch gleichzeitig die schlechtesten Ergebnisse. In Abbildung 6.9(b) erkennt man deutlich die Stufenbildung im Bild. Bessere Ergebnisse liefert die bilineare Interpolation bei einer moderaten Laufzeit. Eine nochmals verbesserte Darstellung kann durch die bikubische Interpolation erreicht werden, benötigt allerdings den größten Rechenaufwand. Die bilineare Methode liefert einen guten Kompromiss aus Qualität des skalierten Bildes und benötigter Rechenzeit.

6.

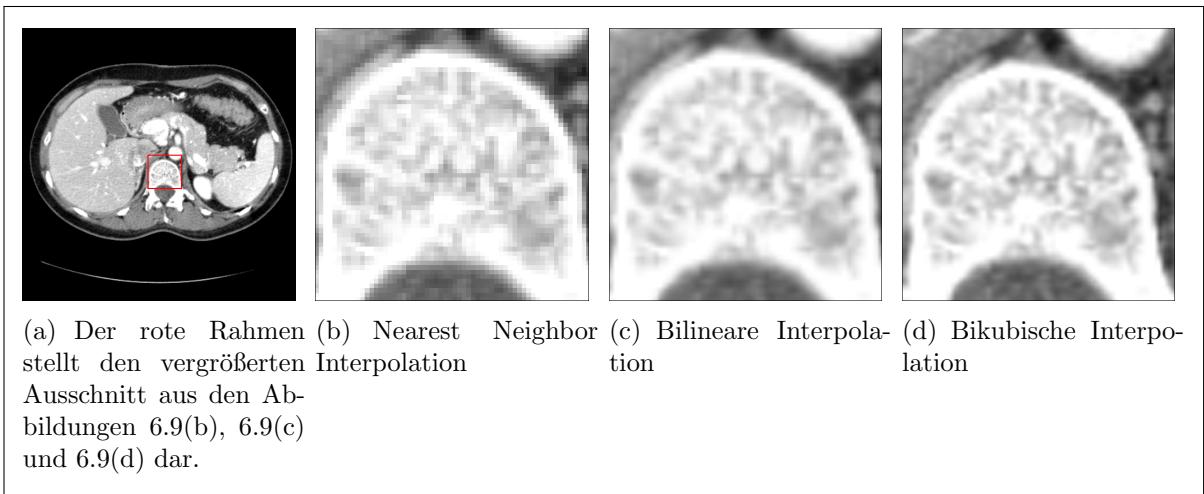
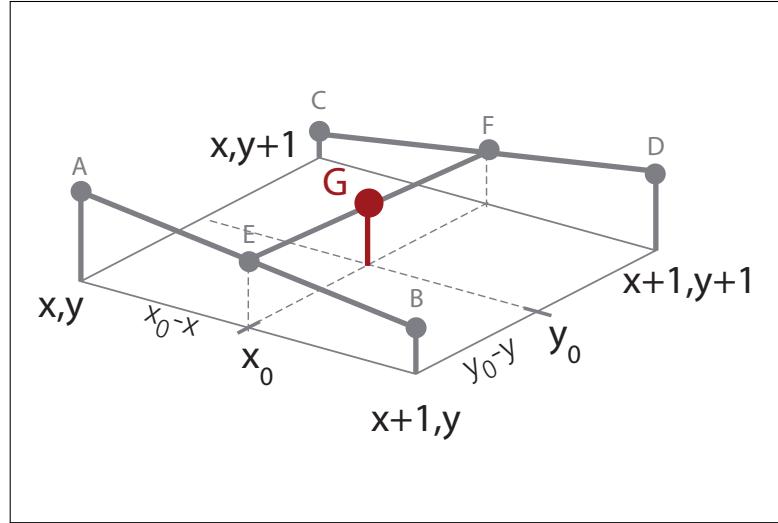


Abbildung 6.9.: Vergleich der Nearest Neighbor, bilinearen und bikubischen Interpolation

Bei der bilinearen Interpolation werden die, zu der Koordinate (x_0, y_0) nächstliegenden, vier Werte der Bildpunkte des Originalbildes zur Interpolation verwendet[BB06, S.387]. In Abbildung 6.10 sind das A , B , C und D . Zuerst findet eine Interpolation in x-Richtung

statt. Daraus ergeben sich die Werte E und F . Durch die Interpolation in y-Richtung der Werte E und F wird der finale Wert G ermittelt.



6.

Abbildung 6.10.: Bilineare Interpolation [BB06, S.388]

Für jeden Bildpunkt des Zielbildes Z wird der Referenzpunkt im Originalbild S mittels einer Normalisierung gefunden.

$$x_0 = \frac{x_Z}{width_Z} \cdot width_S \quad y_0 = \frac{y_Z}{height_Z} \cdot height_S \quad (6.16)$$

Zu den Werten von x_0 und y_0 müssen nach Burger und Burge[BB06, S. 387 ff] die nächstliegenden Bildwerte ermittelt werden.

$$A = S(x, y) \quad B = S(x + 1, y) \quad C = S(x, y + 1) \quad D = S(x + 1, y + 1) \quad (6.17)$$

wobei x und y die abgerundeten Werte von x_0 und y_0 darstellen. Die Werte E und F werden durch den Abstand von x_0 und x ermittelt.

$$E = A + (x_0 - x) \cdot (B - A) \quad (6.18)$$

$$F = C + (x_0 - x) \cdot (D - C) \quad (6.19)$$

Nach der horizontalen Interpolation wird der finale Wert G aus dem Abstand von y_0 und y berechnet.

$$G = E + (y_0 - y) \cdot (F - E) \quad (6.20)$$

Der Wert von G kann nun beim Zielbild Z an die aktuelle Position gesetzt werden. Bei Farbbildern wird jeder Kanal separat interpoliert.

6.5.4. Anzeige der Zusatzinformationen

Gefolgt von bilinearen Interpolation wird das Bild gezeichnet. Als nächster Schritt werden dem Benutzer direkt auf der Zeichenfläche wichtige Zusatzinformationen angezeigt. So kann der Anwender die aktuelle Größe des Bildes in Pixeln und das prozentuale Verhältnis zum Originalbild ablesen, sowie die gewählten Fensterungswerte einsehen. Neben den Bildeigenschaften werden auch dynamische Inhalte angezeigt, wie das Patientenkoordinatensystem und die Orientierungslinien. Orientierungslinien markieren in gleichen geöffneten 3D-Datensätzen mit unterschiedlicher Ebenenansicht einen gemeinsamen Punkt. Diese dynamischen Aspekte werden folgend erläutert.

6.

Das Koordinatensystem

Der DICOM-Standard definiert ein sogenanntes *Reference Coordinate System* [Nat11a, S. 55] und dient zur Orientierung des Patienten im Raum. Es werden sechs Richtungen der drei Achsen definiert, womit die Lage des Patienten beschrieben werden kann [Nat11a, C.7.6.1.1.1].

- **R** (right) - **L** (left) - **x**

Right bestimmt die rechte, left die linke Hand des Patienten.

- **A** (anterior) - **P** (posterior) - **y**

Die Vorderseite des Patienten wird als Anterior, die Rückseite als Posterior bezeichnet.

- **F** (foot) - **H** (head) - **z**

Foot steht für den Patientenfuß und Head für den Kopf.

Wie in Abbildung 6.11(a) dargestellt, steigt die x-Achse in Richtung der linken Hand des Patienten. Die Werte der y-Achse nehmen von der Vorderseite zur Rückseite des Patienten zu. Die z-Achse erhöht sich in Richtung der Füße zum Kopf des Patienten

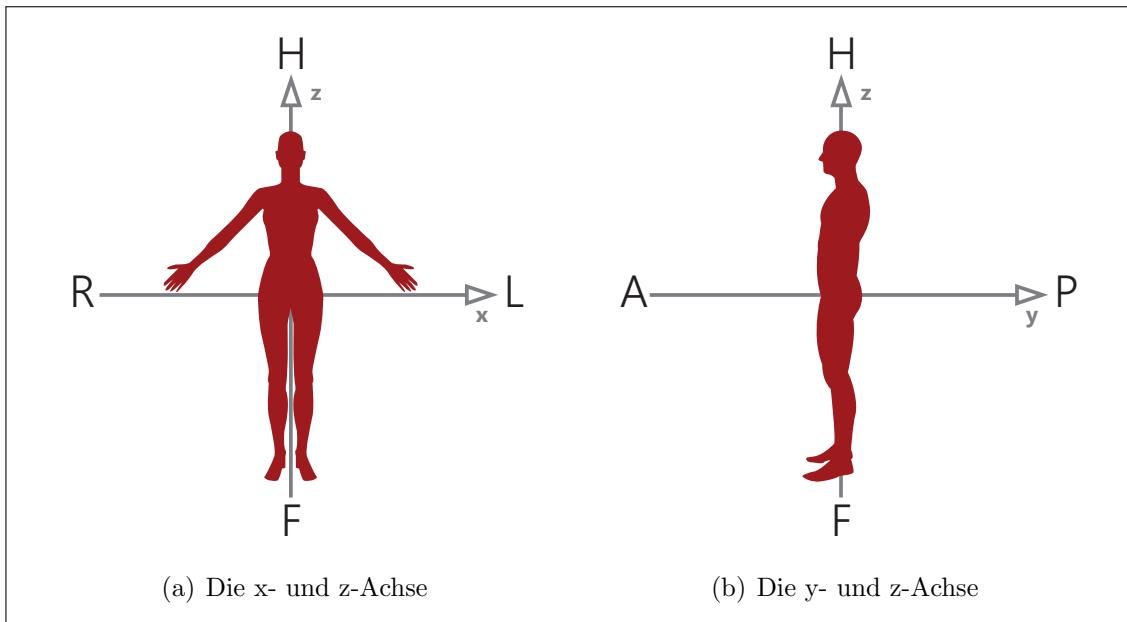


Abbildung 6.11.: Das patientenbasierte Koordinatensystem

6.

[Nat11a, C.7.6.2.1.1]. Unter Berücksichtigung des DICOM-Tags *Image Orientation* kann mit dessen Vektoren kann die Beschriftung der Koordinatenachsen bestimmt werden. Durch den maximalen Betrag der Elemente der Vektoren werden die dominanten Achsen ermittelt. Dadurch erhält man die Koordinatenachsen der Bildecke. Des weiteren kann durch die Achsen die Ebenendarstellung bestimmt werden. Ein dominanter xy-Anteil bedeutet *axiale* Ebenendarstellung, der xz-Anteil ist die *coronale* Ansicht und der yz-Anteil die *sagittale* Darstellung.

$$\vec{r} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \vec{c} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad (6.21)$$

Beispiel 6.21 zeigt zwei Vektoren von *ImageOrientation*. Die dominanten Achsen sind das x-Element aus \vec{r} mit dem Wert 1 und das y-Element aus \vec{c} ebenfalls mit 1. Durch den dominanten Anteil können die Achsen abgelesen werden, die auf die Bildecke gezeichnet werden. Die x-Achse bekommt auf der negativen Seite *R* und *L* auf der positiven Seite. Die zweite Achse der Bildecke ist *y* mit der negativen Seite *A* und positiver Seite *P*. Die z-Achse mit *H* und *F* ragt aus der Bildecke heraus beziehungsweise hinein. Sind die Elemente negativ ist die Richtung der Koordinatenachsen vertauscht. Aus den Beispielvektoren ergibt sich folgende Darstellung der Achsen wie in Abbildung 6.12 dargestellt.

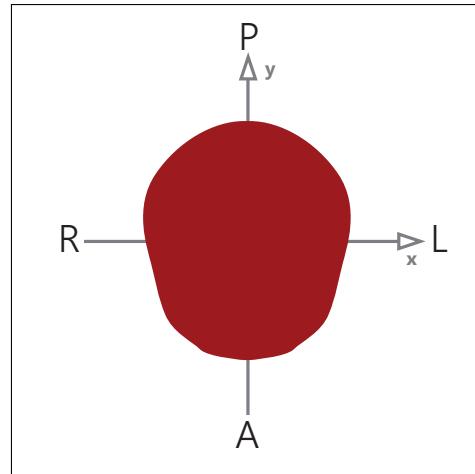


Abbildung 6.12.: Beschriftung der Koordinatenachsen bei axialer Ebenendarstellung

6.

Zeichnen der Orientierungslinien

Wenn mehrere gleiche Datensätze in unterschiedlichen Ebenendarstellungen geöffnet sind, helfen dem Anwender die Orientierungslinien bei der Navigation durch den dreidimensionalen Datensatz. Abbildung 6.13(a) zeigt eine axiale Ebenendarstellung als aktuelle Anzeige, die vom Benutzer bedient wird. Zusätzlich sind die coronale (Abbildung 6.14(b)) und sagittale Ansicht (Abbildung 6.14(c)) geöffnet. Da der Anwender in der axialen Ansicht arbeitet, werden Änderungen der Bildanzeige an die beiden Beobachter weitergegeben. Navigiert der Nutzer mit der Scrollleiste durch den axialen Datensatz, wird mit Hilfe des Oberserver-Musters bei jeder Änderung die aktuelle z-Koordinate der Bildschicht an die coronale und sagittale Darstellung übertragen. Dieser z-Wert wird an der entsprechenden Stelle der Bildschicht als, im Beispiel blaue, Orientierungslinie eingezeichnet.

Wie in den Abbildungen 6.13 zu sehen ist, entspricht die z-Koordinate in axialer Ansicht der y-Koordinate der coronalen und sagittalen Darstellung. Daraus folgt, dass die Orientierungslinie an der Stelle von $(0, y_{coronal})$ bis $(width, y_{coronal})$ sowie von $(0, y_{sagittal})$ bis $(width, y_{sagittal})$ für $y_{coronal}, y_{sagittal} = z_{axial}$

Die Koordinaten zum Einzeichnen der Orientierungslinien sind abhängig von der Quell- und Zieldarstellung. Navigiert der Anwender in der sagittalen Ebenendarstellung und aktualisiert damit eine axiale Anzeige entspricht die Koordinate $z_{sagittal} = x_{axial}$. Die Orientierungslinie wird damit durch die Punkte $(x_{axial}, 0)$ und $(x_{axial}, height)$ definiert. Diese Vorgehensweise wird bei einer Navigation mit der Scrollleiste umgesetzt. Eine weitere Möglichkeit zur Orientierung im Datensatz ist die direkte Auswahl eines Punktes.

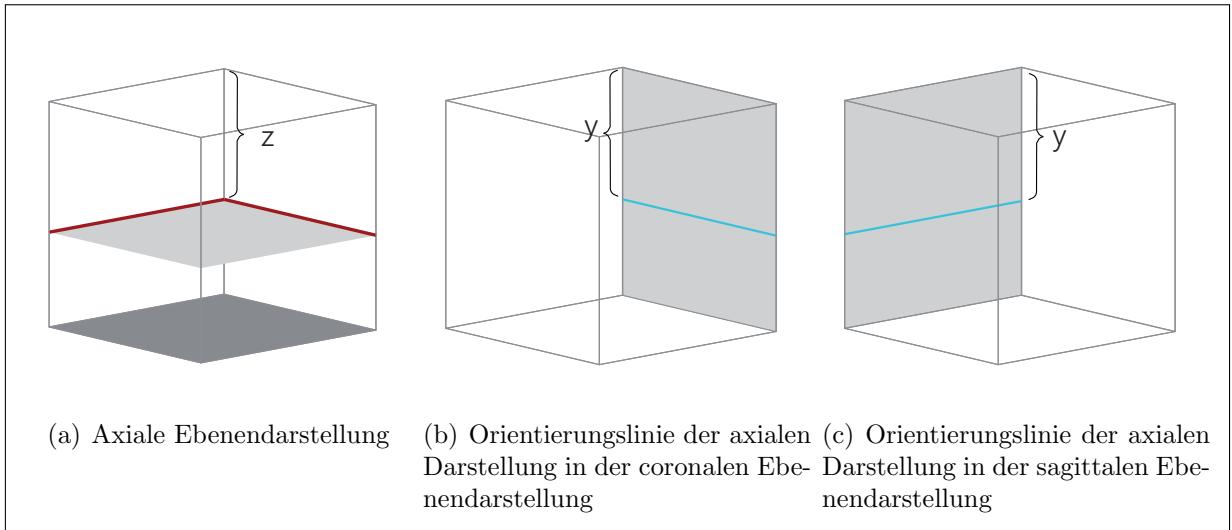


Abbildung 6.13.: Einfache Darstellung der Orientierungslinien

6.

Abbildung 6.14(a) zeigt einen vom Anwender gewählten Punkt in der axialen Darstellung. Wieder müssen alle Beobachter, also Bildanzeigen mit dem gleichen Datensatz, nach der Auswahl aktualisiert werden. Die Abbildungen 6.13 zeigen nur die aktuell vom Nutzer gewählte Bildschicht der aktiven Anzeige. Bei einer direkten Punktauswahl wird in allen Ansichten der gleiche Punkt im Raum (x, y, z) referenziert. Das bedeutet, es werden sowohl die horizontale, als auch die vertikale Orientierungslinie eingezeichnet. Zusätzlich wird die angezeigte Bildschicht der Beobachter den Koordinaten angepasst. Der gewählte Punkt wird jeweils durch den Schnittpunkt der Orientierungslinien symbolisiert. Das Beispiel der Abbildungen in 6.14 zeigt eine aktive axiale Ansicht sowie eine coronale und sagittale Ebenendarstellung als Beobachter. Der vom Anwender gewählte Punkt im Raum ist:

$$P_{axial} = \begin{pmatrix} x_{axial} \\ y_{axial} \\ z_{axial} \end{pmatrix} \quad (6.22)$$

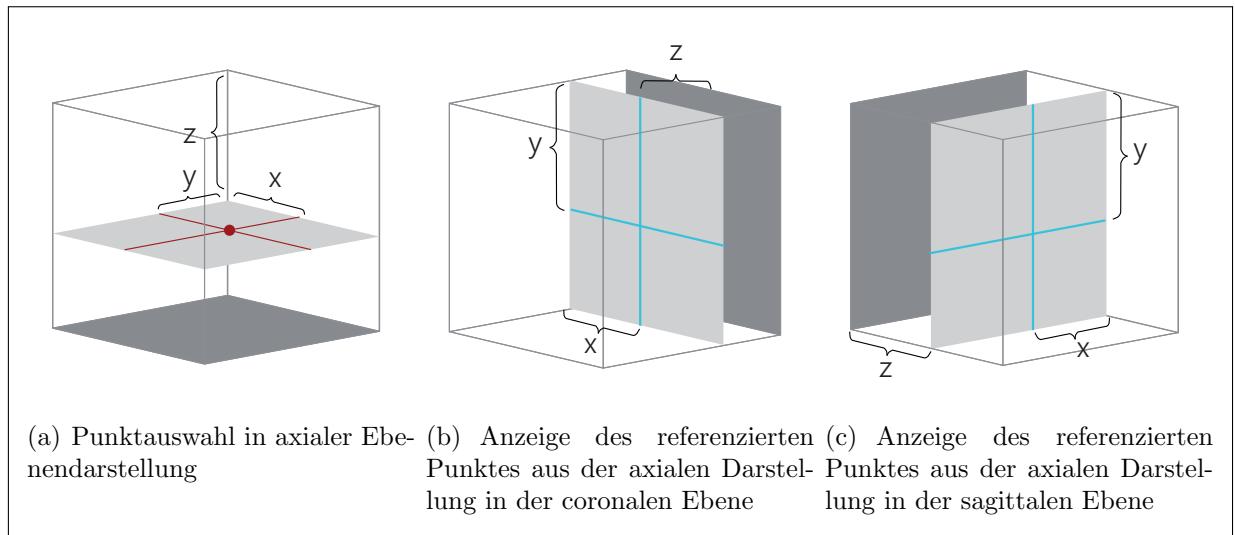


Abbildung 6.14.: Anzeige eines referenzierten Punktes in axialer Ebene mit Hilfe der Orientierungslinien

6.

Aus diesem Punkt müssen die Koordinaten für die Orientierungslinien und der Bildschicht bestimmt werden. Für die Darstellung von der axialen Ansicht zum referenzierten coronalen Punkt folgt aus Abbildung 6.14(b):

$$P_{coronal} = \begin{pmatrix} x_{coronal} \\ y_{coronal} \\ z_{coronal} \end{pmatrix} = \begin{pmatrix} x_{axial} \\ z_{axial} \\ y_{axial} \end{pmatrix} \quad (6.23)$$

Mit diesen Koordinaten ergeben sich für die coronale Ebenendarstellung die vertikale Orientierungslinie von $(x_{axial}, 0)$ bis $(x_{axial}, height)$, die horizontale Linie $(0, z_{axial})$ bis $(width, z_{axial})$ und der Index der Bildschicht y_{axial} .

Auch dieses Vorgehen ist abhängig von der Quell- und Zieldarstellung der aktiven Darstellung und Beobachtern.

6.5.5. Implementierung der Werkzeuge

Die Anzeige der Orientierungslinien komplettiert die Funktionalitäten zur Bilddarstellung, Navigation und Orientierung. Der nächste Schritt befasst sich mit der Manipulation der Bilddaten mit den zur Verfügung gestellten Werkzeugen.

Die Werkzeuge bilden die Schnittstelle zwischen Anwender und dem System. So werden Benutzereingaben entgegengenommen, interpretiert und auf der Zeichenfläche umgesetzt.

Das *DicomCanvas* reagiert mit Hilfe verschiedener *Listener* auf die eintretenden Ereignisse, wie beispielsweise einem Mausklick oder Tastendruck. Diese Ereignisse, auch *Events* genannt, werden an das Werkzeug weitergeleitet, damit das Bild und die Zeichenfläche manipuliert werden können. Das Canvas ruft die Werkzeuge bei gängigen Maus-Events auf, die in der folgenden Liste genauer dargestellt werden.

- **MouseEnter**

Dieses Event wird ausgelöst, wenn der Mauszeiger die Zeichenfläche betritt.

- **MouseExit**

Das Ereignis ist das Gegenstück zu MouseEnter. Beim Verlassen tritt dieses Event ein.

- **MouseMove**

Solange sich der Mauszeiger auf der Zeichenfläche befindet und in Bewegung ist, wird das Werkzeug mit den aktuellen Koordinaten des Cursors aktualisiert.

- **MouseDown**

Dieses Ereignis tritt ein, wenn der Mauszeiger gedrückt wird und sich über der Zeichenfläche befindet. Zusätzlich wird im Werkzeug hinterlegt, dass eine Maustaste⁴ gedrückt wurde.

- **MouseUp**

Wird die Maustaste nach dem Drücken losgelassen, tritt das MouseUp-Ereignis ein. Dies wird innerhalb des Werkzeugs übernommen.

- **MouseWheel**

Beim Bewegen des Mausrads wird unabhängig von der Richtung das MouseWheel-Event ausgelöst⁵.

Zusätzlich zu den Ereignissen definieren Werkzeuge noch zwei Einhängepunkte (Hooks), die in den Zeichenprozess eingebunden werden. Wie in Abschnitt 6.5 in Abbildung 6.5 dargestellt, werden diese Hooks vor der Berechnung der Bildkoordinaten und nach dem Zeichenprozess der Zusatzinformationen aufgerufen. Die Einhängepunkte bieten den Werkzeugen die Möglichkeit, etwas zu dem Zeichenprozess beizutragen, falls diese Funktion

⁴Das Event wird unabhängig von der Nummer der gedrückten Maustaste ausgelöst. Innerhalb der Funktion, die das Ereignis abfängt, kann ausgelesen werden, ob die erste, zweite oder dritte Maustaste betätigt wurde.

⁵Die Drehrichtung kann, wie bei den Mausklicks, innerhalb der abfangenden Funktion ausgelesen werden.

gewünscht ist. So kann beispielsweise ein Fenster am Mauszeiger gezeichnet werden, das aktuelle Informationen über Pixelwerte darstellt.

Damit die Zeichenfläche gelesen und bearbeitet werden kann, steht jedem Werkzeug das zugehörige *DicomCanvas* als Datenelement zur Verfügung. Dadurch können die Daten der Zeichenfläche oder Bildinformationen ausgelesen werden. Dazu zählen unter anderem der Bildstapel sowie die aktuelle Bildschicht und die Koordinaten auf der Zeichenfläche. Die abstrakte Klasse der Werkzeuge *ATool* liefert dem Entwickler die Informationen zum Mauszeiger und dessen Eigenschaften. Dazu zählen die aktuelle x- und y-Position des Cursors auf der Zeichenfläche und ob ein Button der Maus gedrückt ist. Wird eine der Maustasten betätigt, wird ein Startpunkt erstellt. Nach dem Loslassen des Buttons wird der Endpunkt gespeichert. Damit markieren Start und Ende die Strecke der Maus vom Drücken der Maustaste bis zum Loslassen selbiger.

Aus diesen verfügbaren Informationen lassen sich vielfältige Werkzeuge erzeugen.

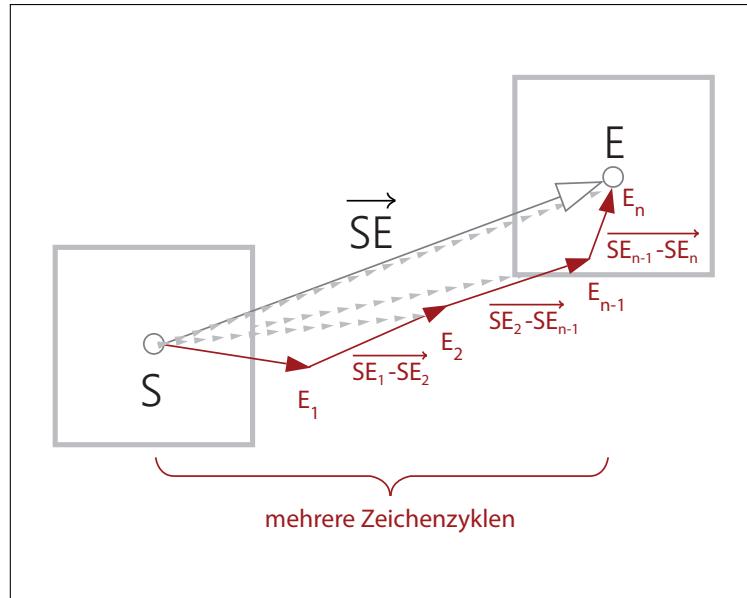
6.

Das Bild bewegen mit dem MoveTool

Mit Hilfe des *MoveTools* lässt sich die aktuelle Bildschicht über die Zeichenfläche schieben. Beim ersten Klick innerhalb des *DicomCanvas* speichert das Werkzeug die Cursorposition als Startpunkt. Solange die Maustaste gedrückt bleibt, wird der neue Mittelpunkt des Bildes aus der aktuellen Mauszeigerposition bestimmt. Die Berechnung erfolgt mit dem Verbindungsvektor aus Startpunkt S und dem Endpunkt E mit

$$\overrightarrow{SE} = \begin{pmatrix} E_x - S_x \\ E_y - S_y \end{pmatrix} \quad (6.24)$$

Die direkte Berechnung von \overrightarrow{SE} ist aufgrund mehrerer aufeinanderfolgender Zeichenzyklen nicht möglich. Abbildung 6.15 zeigt diese Zyklen durch die roten Pfeile. Bei jeder Mausbewegung wird die Zeichenfläche neu gezeichnet. Nach der ersten Translation befindet sich der Bildmittelpunkt an der Stelle $S + \overrightarrow{SE}_1$, in der folgenden an $S + \overrightarrow{SE}_2$ bis die Maus an E_n losgelassen wird und der neue Bildmittelpunkt $S + \overrightarrow{SE}_n$ ist (dargestellt durch die gestrichelten Linien). Da bei jeder Translation ein neuer Bildmittelpunkt bestimmt wird, erfolgt die richtige Berechnung aus der Differenz zwischen der vorherigen und der aktuellen Transformation. Durch Addition von Bildmittelpunkt und der Differenz erhält man das richtig translierte aktuelle Bildzentrum.



6.

Abbildung 6.15.: Translation des Bildmittelpunkts

$$E_n = E_{n-1} + (\overrightarrow{SE_{n-1}} - \overrightarrow{SE_n}) \quad (6.25)$$

E_n ist der zu bestimmende Bildmittelpunkt für die Translation. Das aktuelle Zentrum ist E_{n-1} . Die zuletzt durchgeführte Verschiebung ist $\overrightarrow{SE_{n-1}}$, während die aktuelle durch $\overrightarrow{SE_n}$ dargestellt ist.

Skalierung mit dem ResizeTool

Das *ResizeTool* ermöglicht dem Anwender das Bild zu verkleinern oder zu vergrößern. Durch das Zoomen können Details auch in kleinen Bildgrößen genauer betrachtet werden⁶. Ein Mausklick in die Zeichenfläche löst den Skalierungsprozess aus. Das Werkzeug reagiert speziell auf das *MouseMove* Ereignis. Die Bewegung in y-Richtung auf der Zeichenfläche bestimmt den Skalierungsfaktor. Der Faktor wird folgendermaßen berechnet, wobei S_y dem y-Wert des Startpunktes und E_y dem y-Wert des Endpunktes entspricht:

$$scale = \frac{E_y - S_y}{100} \quad (6.26)$$

⁶Das vergrößerte Bild ist nur eine Interpolation der originalen Pixelwerte.

Ein Endpunkt in positiver Richtung⁷ vergrößert das Bild, während ein Ziehen der Maus in die negative Richtung das Bild verkleinert. Mit einer Division durch 100 entspricht eine Mausbewegung um 10 Pixel in positiver oder negativer Richtung eine Größenänderung von 10%. Bei einem Verhältnis von 1 : 1 lässt sich die Skalierung nicht mehr kontrollieren, da die Größe rasch zu- beziehungsweise abnimmt.

Ähnlich dem *MoveTool* muss vor der Skalierung der alte Faktor abgezogen werden, damit in gleichem Verhältnis skaliert wird.

Das DefaultTool

Das *DefaultTool* stellt ein besonderes Werkzeug dar, da es selbst keine Funktionalitäten implementiert. Es nutzt die Funktionen des *MoveTools* und des *ResizeTools*. Im *DefaultTool* wird geprüft, welche Maustaste gedrückt wird. Bei einem Linksklick wird das *MoveTool* und der Translationsprozess ausgeführt. Ein Klick mit der rechten Maustaste in das Bild nimmt die Koordinaten des Cursors auf, um darauf alle weiteren Zeichenflächen, die Beobachter, zu informieren, dass ein Punkt gewählt wurde. Darauf aktualisieren die Beobachter Bildschicht und Orientierungslinien wie in Abschnitt 6.5.4 beschrieben. Das *ResizeTool* wird durch das Drehen des Mausrads angestoßen und bewirkt pro Einheit eine Größenänderung von 10%.

6.

Justierung der Fensterung mit dem WindowTool

Eine Einstellung der Fensterungswerte kann mit dem *WindowTool* vorgenommen werden. Das Werkzeug arbeitet sowohl in der x-, als auch y-Richtung. Wird der Cursor auf der x-Achse bewegt, kann der Wert des Zentrums (*WindowCenter*) angepasst werden. Die Fensterungsbreite (*WindowWidth*) wird über die y-Achse eingestellt. Der Wert um wie viel *WindowWidth* und *WindowCenter* erhöht werden, wird mit dem Startpunkt *S* und Endpunkt *E* bestimmt. \overrightarrow{SE} wird darauf auf die bestehenden Werte des Zentrums und der Fensterungsbreite addiert.

Punktauswahl mit dem PointTool

Das Werkzeug *PointTool* erfüllt zwei Aufgaben. Zum einen gibt es die Möglichkeit einzelne Punkte zu markieren und im Bild zu hinterlegen, zum anderen können zu den Pixeln mit Hilfe eines Tooltips⁸ zusätzliche Informationen angezeigt werden. Dazu zählen die Ko-

⁷Da der Ursprung der Zeichenfläche in der oberen linken Ecke liegt, ist die positive y-Richtung die untere Kante des Canvas.

⁸Tooltips sind kleine Fenster, die an der Stelle des Cursors Informationen einblenden.

ordinaten des Pixels, der tatsächliche Wert und der mit Hilfe der Fensterung interpolierte Wert. Aus der aktuellen Position und den Bildkoordinaten kann bestimmt werden, ob der Cursor innerhalb des Bildes liegt. Liegt der Mauszeiger im Bild können Punkte gewählt und Informationen zu den Pixelwerten ausgegeben werden.

- **Punktauswahl**

Ein Punkt wird gewählt nachdem eine Maustaste betätigt und wieder gelöst wird. Liegt der Punkt innerhalb des Bildes, erfolgt eine Umrechnung in normalisierte Koordinaten im Intervall $[0, 1]$ und wird dem Bild hinzugefügt.

- **Ausgabe der Information**

Das Anzeigen eines Tooltips ist ein Beispiel eines Einhängepunktes(Hook) für Werkzeuge. Ist der Zeichenprozess beendet, sollen die zusätzlichen Informationen auf die Zeichenfläche gemalt werden. Dazu wird an der Position des Cursors ein Rechteck mit den Pixeldaten gezeichnet.

6.

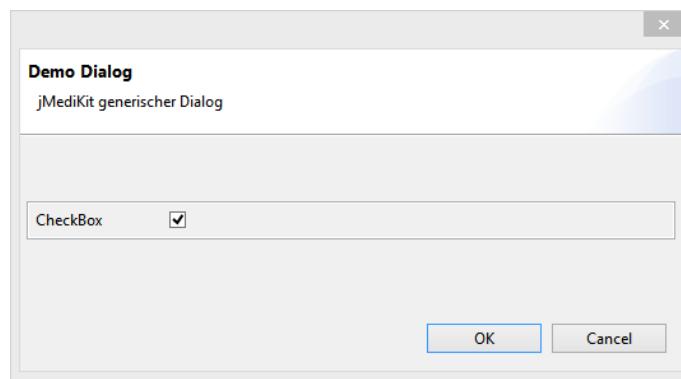
6.6. Dynamische Parametervergabe

Mit dem Punktauswahlwerkzeug erhält der Benutzer die Möglichkeit, dynamisch spezifische Bildpunkte zu wählen. Die Selektion kann anschließend innerhalb eines Plug-ins ausgelesen werden. So können zum Beispiel Start- oder Saatpunkte für Algorithmen gesetzt werden. Als weiteren dynamischen Aspekt soll der Benutzer vor dem Ausführen eines Plug-ins selbst Parameter bestimmen können. So können beispielsweise Schwellwerte im Bezug zu speziellen Pixelwerten gewählt werden. Die dynamischen Parameter bestehen allerdings nicht immer aus numerischen Werten. So müssen ebenso Zeichenketten oder Dateien übergeben werden können. Durch die Dateiauswahl können beispielsweise Textdateien ausgewählt werden, die für ein Plug-in benötigte Datensätze enthalten. Darüber hinaus besteht die Möglichkeit DICOM-Dateien zu laden, um unabhängige DICOM-Objekte innerhalb des Plug-ins als Referenzbilder zu erzeugen.

Mit dem generischen Dialogfenster aus Abschnitt 5.6.4 soll der Benutzer die Möglichkeit erhalten Parametertypen nach Wahl zusammenzustellen. Hierzu stellt die Klasse *GenericPlugInDialog* verschiedene Typen zur Verfügung.

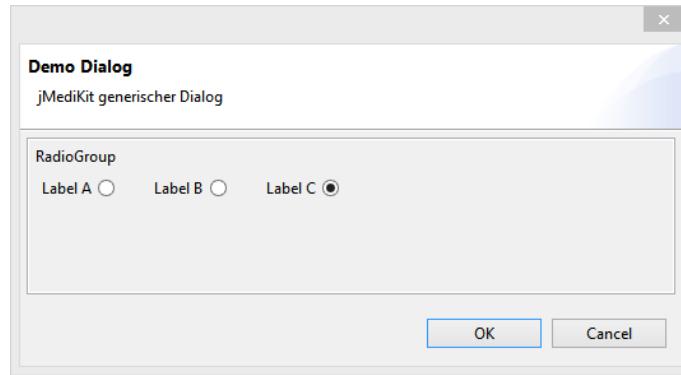
- **Checkbox**

Die Checkbox wird verwendet wenn ein bool'scher Parametertyp gewünscht wird. Markiert symbolisiert die Box den Wert *true*, ansonsten *false*.



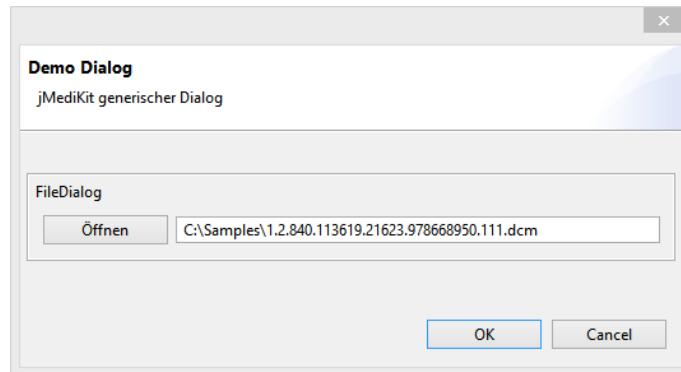
- **RadioGroup**

Mit Hilfe der Radio Group kann ein bestimmter Wert aus einer Gruppe gewählt werden. Der Parametertyp gibt jeweils die Selektion als Zeichenkette zurück.



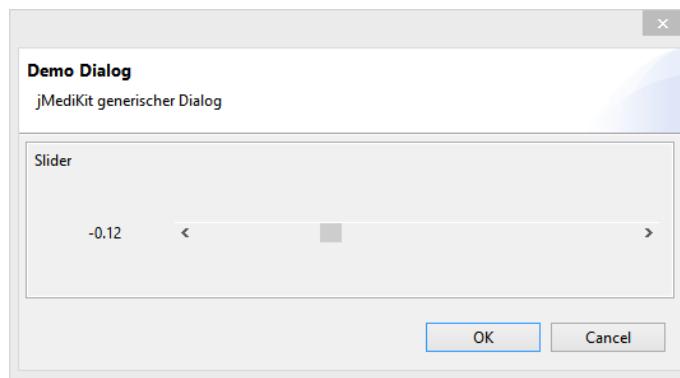
- **FileDialog**

Mit dem FileDialog kann der Anwender eine Datei in das Plug-in laden. Wird der Parameter ausgelesen, wird der Pfad als Zeichenkette zurückgegeben.



- **Slider**

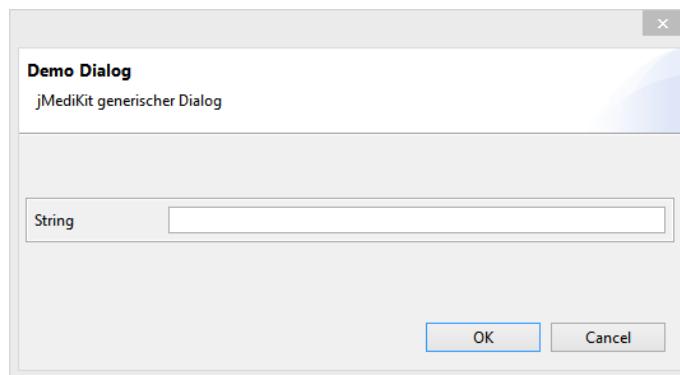
Der Slider definiert ein Intervall $[x, y]$ aus dem ein Wert gewählt werden kann. Dieser Parametertyp wird eingesetzt, wenn untere und obere Wertgrenzen existieren oder die Benutzereingabe begrenzt werden soll. Der Wert wird als Fließkommazahl zurückgegeben.



6.

- **StringItem, IntegerItem, FloatItem**

Diese drei Parametertypen verhalten sich ähnlich. Sie nehmen den jeweiligen Wert des entsprechenden Typs entgegen und geben diesen beim Auslesen wieder zurück.



Die unterschiedlichen Parametertypen können in einem einzigen Dialog beliebig miteinander kombiniert werden.

6.7. Debugging der eigenen Plug-ins

Um beispielsweise die dynamisch gesetzten Parameter in der Entwicklung von Plug-ins zu prüfen, werden Methoden für Testausgaben und zum Debugging benötigt. Das Java Medical Imaging Toolkit bietet sowohl eine dateibasierte, als auch eine visuelle Ausgabeart um das Debugging zu erleichtern und eventuelles Fehlverhalten der Plug-ins zu finden.

6.7.1. Dateibasiertes Debugging

Debugging hilft dem Benutzer Variablen zu überprüfen oder Programmabläufe zu verstehen. Da ein umfassendes Debugsystem nicht verfügbar ist, muss die Fehlersuche über Programmausgaben erfolgen. Da die Standardausgabe im Rich Client nicht einsehbar ist, wird mit Dateien gearbeitet, die Ausgaben mit *System.out* protokollieren.

In der Klasse *APlugIn* ist eine Methode *setOutput* definiert. Da sowohl *APlugIn2D* als auch *APlugIn3D* von dieser Klasse erben, ist die Methode in allen Plug-ins verfügbar. Wird diese zum Beispiel in den Optionen eines Plug-ins aufgerufen, wird die Standardausgabe *out* von *java.lang.System* auf die als Parameter spezifizierte Datei umgeleitet. Dadurch werden alle Aufrufe von *System.out.println* in dieser Datei abgelegt.

6.7.2. Visuelles Debugging

In manchen Situationen ist es nützlich, spezielle Zwischenschritte der Algorithmen ausgeben zu lassen. Manche Bildverarbeitungsprozesse erwarten zum Beispiel ein vorgefiltertes Bild oder ein spezielles Bildformat wie ein Binärbild. Das visuelle Debugging erlaubt die Kontrolle dieser Zwischenschritte durch eine unabhängige Bildausgabe.

Um diese Problematik zu lösen, ist im jMediKit die Klasse *Visualizer* vorhanden. Die statische Methode *show* nimmt entweder ein Bild oder eine Liste von Bildern entgegen. Wird diese aufgerufen, werden die übergebenen Bilder in einem separaten Fenster angezeigt.

7. Entwicklung von Erweiterungen

Das Java Medical Imaging Toolkit bietet sowohl dem Entwickler, als auch dem Anwender Spielraum, die Anwendung zu erweitern. Dieses Kapitel beschreibt die Entwicklung verschiedener Plug-ins in zwei- und dreidimensionalem Raum. Zusätzlich zeigt der Entwurf eines neuen Moduls und eine Implementierung eines Werkzeugs wie die Basisfunktionen von jMediKit erweitert werden.

7.1. Entwicklung von Plug-ins

Die Entwicklung von Plug-ins ermöglicht eine Funktionserweiterung der Anwendung, ohne einen neuen Buildprozess. Bei Programmstart wird der in der Anwendung angegebene Plug-in-Ordner nach allen verfügbaren Erweiterungen durchsucht und eingebunden. Damit Plug-ins vom Programm korrekt gefunden werden können, müssen diverse Konventionen bei der Entwicklung beachtet werden, die im nächsten Abschnitt erläutert werden.

7.1.1. Konventionen

Damit der Importvorgang der Plug-ins ohne Komplikationen verläuft, ist die Anwendung auf die Konsistenz der Plug-ins angewiesen. Die folgenden Konventionen bestimmen diese Konsistenz.

- **Generalisierung**

Die Basisklasse eines Plug-ins ist entweder *APlugIn2D* oder *APlugIn3D* und muss diese erweitern.

- **Klassenname**

Der Name der Klasse des Plug-ins, die von *APlugIn2D* oder *APlugIn3D* erbt, muss mit *zwei Unterstrichen* beginnen. Zusätzlich muss das erste Zeichen ein Buchstabe oder eine Ziffer sein. Listing 7.1 zeigt den regulären Ausdruck zur Beurteilung gültiger Namen. Korrekt sind zum Beispiel *_Test* oder *_3DTest_Plug_in*. Ungültig ist beispielsweise *_Test* oder *__TestPlugIn*. Da ein Plug-in in Beziehung zu anderen

Klassen oder Paketen stehen kann, die entweder selbst implementiert oder von Bibliotheken zur Verfügung gestellt werden, wird so die Hauptklasse der Erweiterung kenntlich gemacht.

- **Ordnername**

Der Ordner, in dem ein Plug-in enthalten ist, muss den identischen Namen wie die Hauptklasse des Plug-ins haben. Ist dieser Name `_Test`, muss der *Wurzelordner* der die Erweiterung enthält ebenfalls `_Test` heißen.

- **Ordnerstruktur**

Wenn Plug-ins mit Hilfe der Eclipse-Entwicklungsumgebung entwickelt werden, entspricht der Output-Ordner mit den *.class-Dateien* eines Eclipse-Projekts der korrekten Struktur. Ist die Hauptklasse nicht in einem Package organisiert, liegt die Plug-in-Klasse direkt im Plug-in-Ordner und hätte die Struktur `/plugins/_Test/_Test.class`. Erfolgt die Entwicklung mit Packages ist die Struktur `/plugins/_Test/de/korb/_Test.class` wenn die Klasse `_Test` im Package `de.korb` liegt. `/plugins` ist der Ordner, der die einzelnen Plug-ins enthält. Dieser muss in den Einstellungen der Anwendung spezifiziert werden. Das bedeutet, dass jMediKit alle Plug-ins in `/plugins` findet, wenn die Ordner mit doppeltem Unterstrich beginnen und eine *.class*-Datei mit identischen Namen beinhalten, die APlugIn2D oder APlugIn3D generalisieren. Abbildung 7.1 zeigt die Ordnerstruktur zweier Plug-ins. `_PlugInA` ist ohne, während `_PlugInB` mit Packages organisiert wird. `_PlugInB` definiert zusätzlich eine eigenen Bibliotheksklasse. Wird der Ordner `plugins` in den Einstellungen angegeben, werden bei Programmstart die beiden Plug-ins initialisiert und können verwendet werden. Nach einer neuen Belegung der Einstellungen, muss die Anwendung neu gestartet werden.

```
1 | ^_[A-Za-z0-9].*
```

Listing 7.1: Der reguläre Ausdruck gültiger Klassennamen

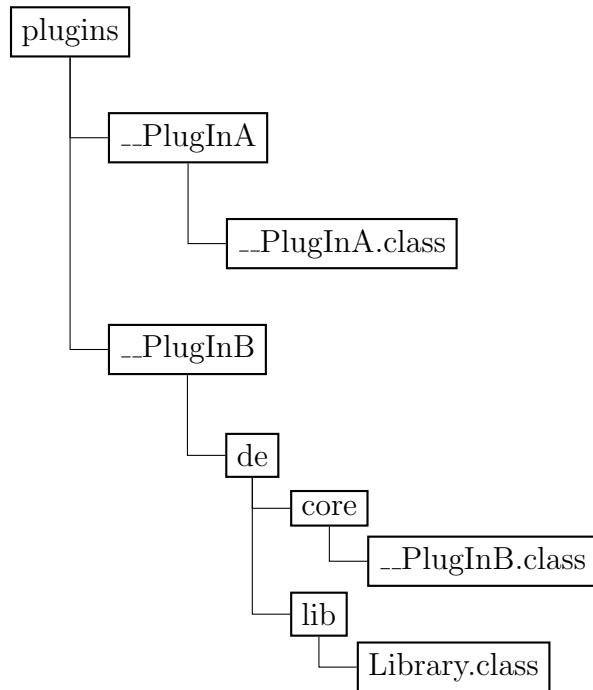


Abbildung 7.1.: Beispiel der Ordnerstruktur zweier verschiedener Plug-ins

7.1.2. Die Klassenstruktur eines Plug-ins

Das Java Medical Imaging Toolkit stellt zwei Typen von Plug-ins bereit. Der Anwender kann mit einer Erweiterung von APlugIn2D entweder im zweidimensionalen oder mit APlugIn3D im dreidimensionalen Raum arbeiten. Unabhängig vom Plug-in-Typ müssen zwei abstrakte Methoden der jeweiligen Basisklasse implementiert werden.

- **options**

Die Methode *options* wird *einmalig vor* der Ausführung des Plug-ins aufgerufen. Hier können wahlweise Parameter gesetzt werden, die für weitere Berechnungen benötigt werden.

- **process**

Das Kernelement eines Plug-ins stellt *process* dar. Als Parameter ist entweder das aktuell gewählte Bild im ImageView oder im Fall eines 3D-Plug-ins der Bildstapel als Parameter der Methode enthalten. Beim Rückgabewert muss beachtet werden, dass die Bildanzahl mit der im Parameter übereinstimmt, da sonst weitere Berechnungen, wie zum Beispiel die Ebenenrekonstruktionen, nicht korrekt ausgeführt werden können.

In Listing 7.2 ist die Basisdefinition eines zweidimensionalen Plug-ins dargestellt. Es werden die abstrakten Methoden *process* und *options* implementiert. Mit dem Parameter *img* in *process* können die Pixelwerte manipuliert werden. Das Rückgabebild wird anschließend im ImageView an Stelle des entsprechenden Index angezeigt.

```
1 | public class __2dPlugIn extends APlugIn2D{  
2 |     @Override  
3 |     public AImage process(AImage img) {  
4 |         return img;  
5 |     }  
6 |  
7 |     @Override  
8 |     public int options() {  
9 |         return 0;  
10|     }  
11| }
```

Listing 7.2: Definition des konkreten 2D-Plug-ins __2dPlugIn.java

Ein dreidimensionales Plug-in erbt die gleiche Grundstruktur wie das 2D-Plug-in. Wie in Listing 7.3 zu sehen ist, unterscheiden sich die übergebenen Parameter in der Methode *process*. Eine 3D-Erweiterung erhält den gesamten Bildstapel als Liste. Die Variable *index* enthält den Index der Bildschicht, die im aktiven ImageView angezeigt wird.

```
1 | public class __3dPlugIn extends APlugIn2D{  
2 |     @Override  
3 |     public AImage process(List<AImage> images, int index) {  
4 |         return images;  
5 |     }  
6 |  
7 |     @Override  
8 |     public int options() {  
9 |         return 0;  
10|     }  
11| }
```

Listing 7.3: Definition des konkreten 3D-Plug-ins __3dPlugIn.java

7.1.3. Nützliche Funktionen

Neben den Bilddaten stellt jMediKit weitere Funktionen für die Plug-in-Entwicklung zur Verfügung. So können DICOM-Objekte und ganze Bäume zusätzlich geladen werden und Tags und Bilder aus den Objekten zur Verarbeitung gelesen werden. Die dynamische Parameterübergabe stellt eine weitere Funktion dar, die es ermöglicht, eigenen Optionsdialoge für Plug-ins zu erstellen. In folgender Liste werden diese Features genauer erläutert.

Erstellung eines Bildes vom Typ AImage

Bilder können unter jMediKit auf zwei Arten erzeugt werden. Mit expliziter Bilderzeugung kann das Objekt entsprechend dem gewünschten Bildtyp direkt instantiiert werden. In vielen Fällen ist der Typ des Parameterbildes in *process* unbekannt und der Anwender ist von einer impliziter Objekterzeugung abhängig. Mit der statischen Methode *getAbstractImage* der Klasse *SimpleImageFactory* kann der Bildtyp eines Referenzbildes übergeben werden. Entsprechend der Referenz wird ein neues Bildobjekt mit passendem Typ zurückgegeben. Listing 7.4 zeigt jeweils ein Beispiel der Objekterzeugung.

```
1 //Explizite Bilderstellung
2 AImage simg = new ShortImage(800, 600);
3 System.out.println(simg.getImageType());
4 //prints 2
5
6 //Explizite Bilderstellung
7 AImage usimg = new UnsignedShortImage(800, 600);
8 System.out.println(usimg.getImageType());
9 //prints 3
10
11 //Implizite Bilderstellung
12 AImage i_simg = SimpleImageFactory.getAbstractImage(simg.
13     getImageType(), 800, 600);
14 System.out.println(i_simg.getImageType());
15 // prints 2
```

Listing 7.4: Erzeugung eines AImage

Erzeugung eines SimpleITK-Bildes

Bildobjekte von jMediKit und SimpleITK sind zueinander inkompatibel. Mit Hilfe von Konvertierungsfunktionen in beide Richtungen wird dieses Problem behoben. Die An-

wendung erfolgt ähnlich der *SimpleImageFactory*, mit dem Unterschied, dass als Parameter das Bild selbst übergeben wird und eine Kopie als entsprechender Typ und Format zurückgegeben wird. Listing 7.5 zeigt ein Beispiel einer Konvertierung. Die Klasse *SimpleITKFactory* stellt die Funktionen *getITKImage* und *getAImage* zu einer Konvertierung bereit.

```
1 AImage simg = new ShortImage(800, 600);
2
3 Image itk_img = SimpleITKFactory.getITKImage(simg);
4 String type = itk_img.getPixelIDTypeAsString();
5 System.out.println(type);
6 //prints 16-bit signed integer
7
8 AImage converted_img = SimpleITKFactory.getAImage(itk_img);
9 System.out.println(converted_img.getImageType());
10 //prints 2
```

Listing 7.5: Konvertierung der Bildobjekte zwischen jMediKit und SimpleITK

Saatpunkte und ROIs auslesen

Die Anwendung stellt die zwei Selektionswerkzeuge *PointTool* und *ROITool* bereit. Das ROITool wird in Abschnitt 7.3 als Beispiel der Werkzeugerweiterung entwickelt. Zum einen können Punkte gesetzt und zum anderen Bildbereiche markiert werden. Jede Bildinstanz stellt, wie in Listing 7.6 zu sehen, jeweils einen Getter bereit, der eine Liste der Datenstruktur zurück gibt.

```
1 @Override
2 public AImage process(AImage arg0) {
3     //doStuff();
4
5     ArrayList<Point2D<Float>> points = arg0.getPoints();
6     ArrayList<ROI> rois = arg0.getROIs();
7
8     //doThings();
9 }
```

Listing 7.6: Saatpunkte und Regions Of Interest aus einem AImage ermitteln

Import eines DICOM-Baums

Ist es notwendig, dass ein zusätzlicher DICOM-Baum in das Plug-in geladen werden muss, gibt es die Klasse *PlugInDicomImporter* mit der statischen Methode *recursiveImport*. Übergibt man ein Verzeichnis, wird dieses rekursiv auf der Suche nach DICOM-Objekten durchlaufen und ein Baum erstellt, der nach den Objekten durchsucht werden kann. Listing 7.7 zeigt einen Import und das Auslesen eines DICOM-Objekts.

```
1 DicomTreeRepository tree = PlugInDicomImporter.recursiveImport(  
    new File("PathToDirectory"));  
2 DicomObject obj = (DicomObject) tree.lookUpDicomTreeItem("UID");
```

Listing 7.7: Saatpunkte und Regions Of Interest aus einem AImage ermitteln

DICOM-Tags aus DICOM-Objekten auslesen

Neue DICOM-Objekte können entweder durch das Auslesen eines DICOM-Baumes, oder durch die explizite Instantiierung durch Angabe eines Dateipfads erstellt werden. Zuerst wird das Objekt entsprechend der Zeile 5 oder Zeile 9 wie in Listing 7.8 erzeugt. Die Klasse *DicomObject* stellt unter anderem die Methode *getTagData*(in Zeile 14) zur Verfügung und erhält als Parameter eine Stringrepräsentation des Tags und den Typ des Rückgabewertes. Alle möglichen Rückgabe-Typen sind in *DicomObject* als Konstanten festgelegt und müssen entsprechend der Value Representation des Tags gesetzt werden. *PatientName* hat als VR den Wert *PN* und wird als Zeichenkette behandelt. Dadurch wird als Rückgabetyp *DicomObject.RETURN_STRING* gewählt.

7.

Erstellung eines Dialogfensters

In Kapitel 6 Abschnitt 6.6 wurden alle Dialogoptionen für eine dynamische Parametervergabe gezeigt. Listing 7.9 zeigt die Anwendung eines *GenericPlugInDialog*. Innerhalb des Plug-ins wird der Dialog *dialog* als Datenelement hinterlegt, damit die Werte von *process* ausgelesen werden können. In der Methode *options* wird der Dialog erzeugt und die Dialogelemente hinzugefügt. Die RadioGroup nimmt als Parameter den Namen¹, ein Array von Strings, welche die einzelnen Elemente der Gruppe symbolisieren und einen Standartwert welches Element als ausgewählt dargestellt werden soll.

¹Der Name dient zur eindeutigen Zuordnung, um später die Werte auslesen zu können.

```

1 File d = new File("PathToDirectory");
2 File f = new File("PathToFile");
3
4 DicomTreeRepository r = PlugInDicomImporter.recursiveImport(d);
5 DicomObject obj = (DicomObject) r.lookUpDicomTreeItem("UID");
6
7 //oder
8 try {
9     DicomObject obj = new DicomObject(f);
10 } catch (IOException e) {
11     e.printStackTrace();
12 }
13
14 String name = (String) obj.getTagData("PatientName", DicomObject.
    RETURN_STRING);

```

Listing 7.8: Lesen eines Tags aus einem DICOM-Objekt

Der Slider bekommt ebenfalls einen Namen und den Standardwert, gefolgt von Anfangswert, Endwert, Schrittweite und Nachkommastellen. Im Beispiel hat der Slider als Parameter für `min = 0`, `max = 30` und `digits = 1`. Damit können im Slider Werte von 0.0 - 3.0 gewählt werden. Das Intervall berechnet sich aus $\frac{value}{10^{digit}}$. Eine Inkrementierung der Nachkommastelle erhöht die Genauigkeit. Zwei Nachkommastellen des Intervalls [0, 3] erlauben Werte zwischen 0.00 - 3.00. Ein Intervall [0, 30] mit zwei Nachkommastellen kann mit dem Aufruf `dialog.addSlider(„Name“, 1, 0, 3000, 1, 2)` erzeugt werden ($max(\frac{3000}{10^2})$).

Die Abbildung 7.2 zeigt die visuelle Darstellung des individuellen Dialogs, der in *options* zusammengesetzt wurde.

```
1 | public class __DialogExample extends APlugIn2D{  
2 |  
3 |     GenericPlugInDialog dialog;  
4 |  
5 |     @Override  
6 |     public AImage process(AImage arg0) {  
7 |  
8 |         String interpolation = (String) dialog.getItemValue("Interpolation");  
9 |         float sigma = (float) dialog.getItemValue("Sigma");  
10 |  
11 |         return arg0;  
12 |     }  
13 |  
14 |     @Override  
15 |     protected int options() {  
16 |  
17 |         dialog = new GenericPlugInDialog("Hallo(Dialog", "Ein_Beispieldialog");  
18 |         dialog.addRadioGroup("Interpolation", new String[]{"NN", "bilinear", "bicubic"}, 1);  
19 |         dialog.addSlider("Sigma", 1, 0, 30, 1, 1);  
20 |         int status = dialog.open();  
21 |         return 0;  
22 |     }  
23 | }
```

7.

Listing 7.9: Erstellung eines eigenen Optionsdialog

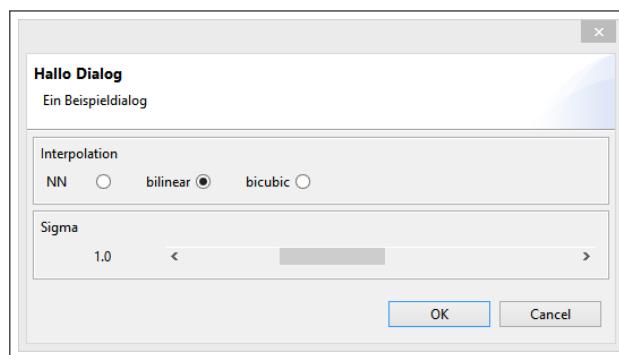


Abbildung 7.2.: Die visuelle Darstellung des Dialogs aus Listing 7.9

7.1.4. Der Laplace-Operator

Der Laplace-Operator wird zur Schärfung von Bildern eingesetzt und dient als Beispiel für die Umsetzung eines zweidimensionalen Plug-ins. Die *process*-Methode aus Listing 7.10 zeigt eine Implementierung des Filters. Der Parameter *arg0* enthält die aktuelle Bildschicht des aktiven ImageViews.

In Zeile 6 wird mit Hilfe der einfachen Fabrikfunktion das Bild für die Filterergebnisse erzeugt, da der Bildtyp von *arg0* nicht explizit bekannt ist. In Zeile 16 wird das Quellbild *arg0* durchlaufen und gefiltert. In Zeile 31 wird *copySignificantAttributes* aufgerufen. Damit werden essentielle Bildeigenschaften vom Parameterbild in das aufrufende Bild kopiert. Zu den Eigenschaften zählen zum Beispiel *WindowWidth*, *WindowCenter* oder *ImagePosition*.

Sind diese Werte vom Anwender im Rückgabewert von *process* nicht gesetzt, werden die Eigenschaften von *arg0* in das Ergebnisbild von jMediKit kopiert. Damit wird eine korrekte Anzeige im ImageView garantiert.

Nach der Rückgabe des geschärften Bildes, wird es von der Anwendung angezeigt.

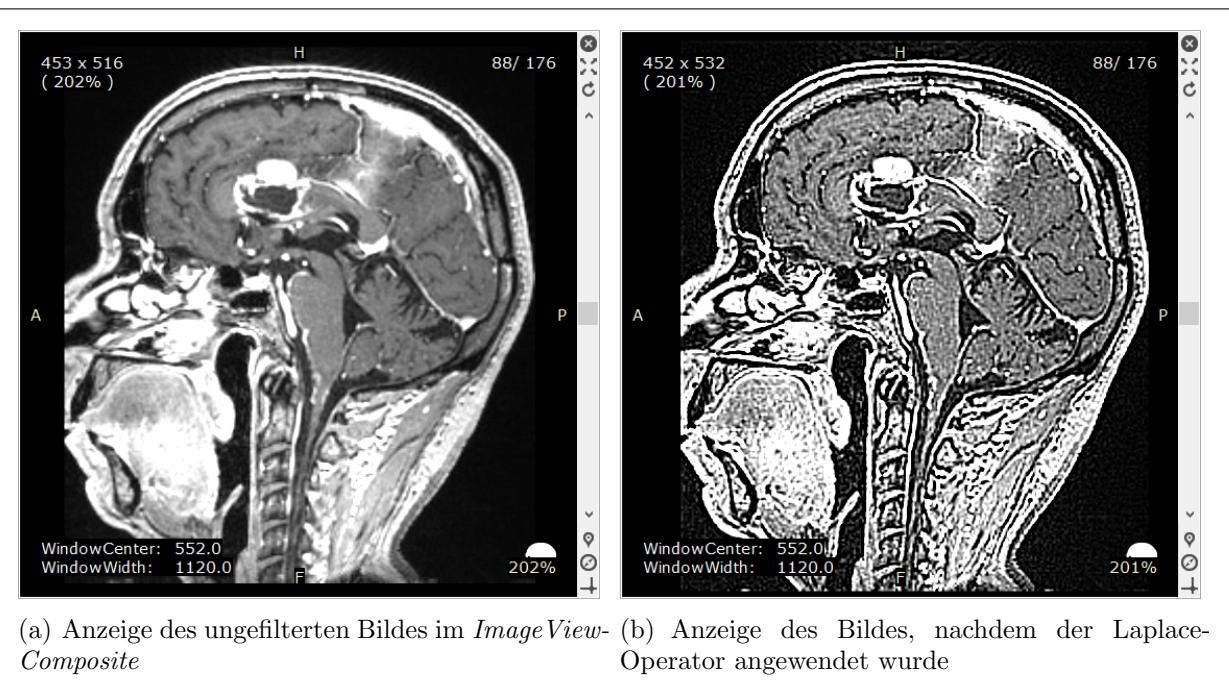


Abbildung 7.3.: Filterung eines medizinischen Bildes mit dem Laplace-Operator

```

1  @Override
2  public AImage process(AImage arg0) {
3      int width = arg0.getWidth();
4      int height = arg0.getHeight();
5      //AImage zur Speicherung des Ergebnisses
6      AImage edgeImg = SimpleImageFactory.getAbstractImage(arg0.
7          getImageType(), width, height);
8
9      int[][] filter = new int[3][3];
10     filter[0][0] = 1; filter[0][1] = 2;    filter[0][2] = 1;
11     filter[1][0] = 2; filter[1][1] = -12; filter[1][2] = 2;
12     filter[2][0] = 1; filter[2][1] = 2;    filter[2][2] = 1;
13
14     int fw = filter.length/2;
15
16     //ueber alle Pixel
17     for(int y = 1; y < height-1; y++) {
18         for(int x = 1; x < width-1; x++) {
19             int sum = 0;
20             //Filterung
21             for(int j = -fw; j <= fw; j++) {
22                 for(int i = -fw; i <= fw; i++) {
23                     int value = arg0.getPixel(x+i, y+j);
24                     int filter_value = filter[j+fw][i+fw];
25                     sum = sum + value*filter_value;
26                 }
27             }
28             //setzen des Ergebniswertes
29             edgeImg.setPixel(x, y, sum);
30         }
31     }
32     edgeImg.copySignificantAttributes(arg0);
33
34     AImage img = SimpleImageFactory.getAbstractImage(arg0.
35         getImageType(), width, height);
36
37     //Differenzbild img = arg0 - edgeImg berechnen
38     return img;
39 }
```

Listing 7.10: Implementierung der process-Methode des Laplace-Operators

7.1.5. Der Sobel-Operator

Der Sobel-Operator dient als Kantendetektor zur Visualisierung harter Intensitätsübergänge in Bildern. Listing 7.11 zeigt die Implementierung von *process*².

```
1  @Override
2  public List<AImage> process(List<AImage> arg0, int arg1) {
3
4      //Variablenbelegung
5
6      //Filter in x- und y-Richtung
7      int[][] x_dir = new int[3][3]; int[][] y_dir = new int[3][3];
8
9      for(int z = 0; z < arg0.size(); z++) {
10
11         AImage img = arg0.get(z);
12         AImage edgeImg = SimpleImageFactory.getAbstractImage(img.
13             getImageType(), width, height);
14
15         for(int y = 1; y < height-1; y++) {
16             for(int x = 1; x < width-1; x++) {
17                 for(int j = -1; j <= 1; j++) {
18                     for(int i = -1; i <= 1; i++) {
19                         //Filterung
20                     }
21                 //Kantenstaerke
22
23                 //nach Pruefung auf Ueberschreitung von min und max durch
24                 E_xy
25                 //Pixel setzen
26             }
27             edge.add(edgeImg);
28         }
29     return edge;
30 }
```

Listing 7.11: Implementierung der *process*-Methode des Sobel-Operators

Sobel wird im Beispiel als 3D-Plug-in realisiert, wie die Parameter der Methode zeigen. Der wesentliche Unterschied zur Laplace-Implementierung liegt in Zeile 9, indem der gesamte Bildstapel durchlaufen wird und der Filter auf den Ebenen x, y und z arbeitet.

²Zur Übersichtlichkeit wurde ein Teil des Quelltextes ausgelassen. Der vollständige Code befindet sich auf dem beiliegenden Datenträger.

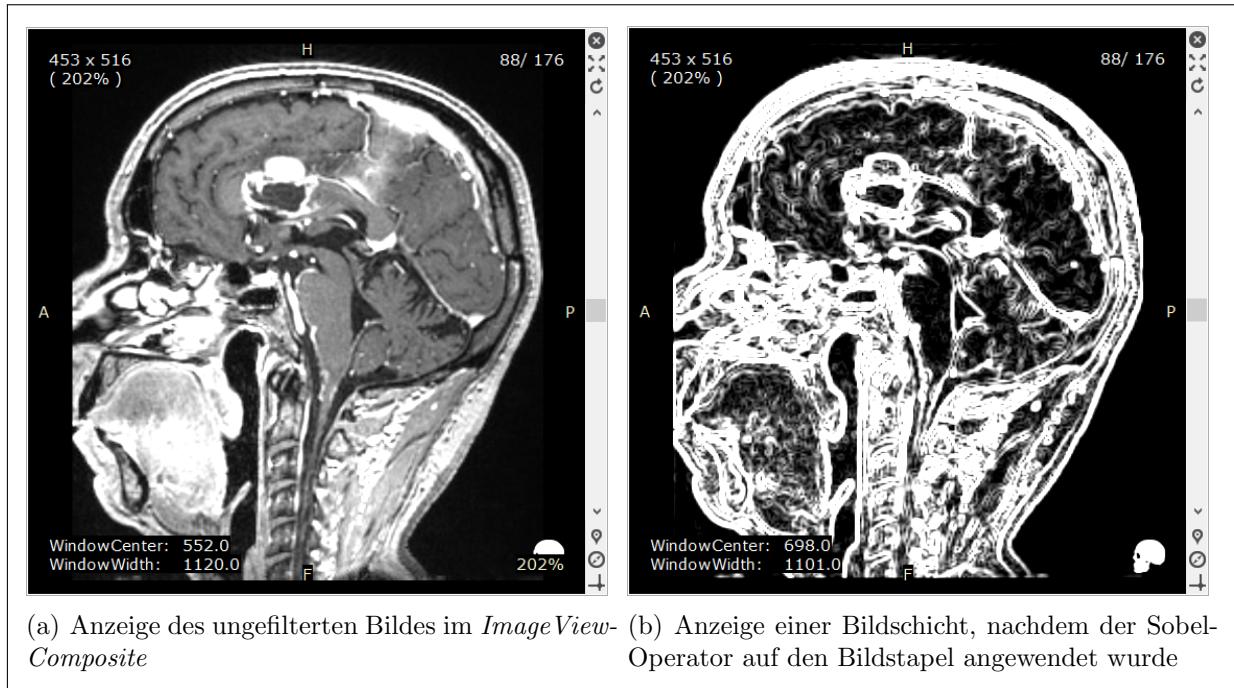


Abbildung 7.4.: Filterung eines medizinischen Bildes mit dem Sobel-Operator

7.

7.1.6. Der ConnectedThresholdImageFilter aus dem SimpleITK

Der ConnectedThresholdImageFilter ist ein Filter aus der SimpleITK-Bibliothek und wird zur Segmentierung von Bildstrukturen eingesetzt. Für eine Anwendung des Filters müssen Saatpunkte mit dem *PointTool* im Bild definiert werden. Anhand dieser Punkte wird mit einem oberen und unteren Schwellwert bestimmt, ob ein Pixel zu der zu segmentierenden Struktur gehört. Die Eingabe der Schwellwerte wird mit Hilfe eines *GenericPlugInDialogs* implementiert(Listing 7.12).Der Dialog besteht aus den zwei Integerwerten *upperThreshold* und *lowerThreshold*.

Listing 7.13 zeigt die Implementierung des Filters. Die Liste in Zeile 2 dient zur Speicherung der segmentierten Bilder. Da der ConnectedThresholdImageFilter ein 3D-Plug-in ist, wird in Zeile 4 der Bildstapel durchlaufen. Folgend werden, falls vorhanden, die Saatpunkte der aktuellen Schicht gelesen(Zeile 5) und anschließend aus dem Bild ein SimpleITK-Bild erzeugt(Zeile 7). In Zeile 8 erfolgt die Instantiierung des Filters selbst. Damit der Filter arbeiten kann, müssen einige Werte gesetzt werden. Die Zeilen 10 – 12 legen die Schwellwerte aus dem Dialog fest und es wird ein Wert bestimmt, den zu ersetzenende Pixel zugewiesen bekommen. 14 – 23 dienen dem Auslesen der Saatpunkte. Jeder definierte Punkt wird dem Filter zugewiesen. Innerhalb des SimpleITK werden die Punkte als *VectorUInt32* repräsentiert. Mit dem zweidimensionalen Vektor werden die Saatpunk-

```

1  @Override
2  protected int options() {
3      //Dialog erzeugen
4      dialog.addIntegerItem("Lower_Threshold", 0);
5      dialog.addIntegerItem("Upper_Threshold", 0);
6      dialog.open();
7
8      lowerThreshold = (int) dialog.getItemValue("Lower_Threshold");
9      upperThreshold = (int) dialog.getItemValue("Upper_Threshold");
10
11     return 0;
12 }
```

Listing 7.12: Die options-Methode von ConnectedThresholdImageFilter

te konvertiert. Zeile 23 führt den Algorithmus mit der *execute*-Methode aus. Nach einer Konvertierung in ein *AImage* in 24, wird das Ergebnisbild der Liste hinzugefügt. Ist der komplette Bildstapel abgearbeitet, werden die segmentierten Daten im aktiven ImageView angezeigt.

7.

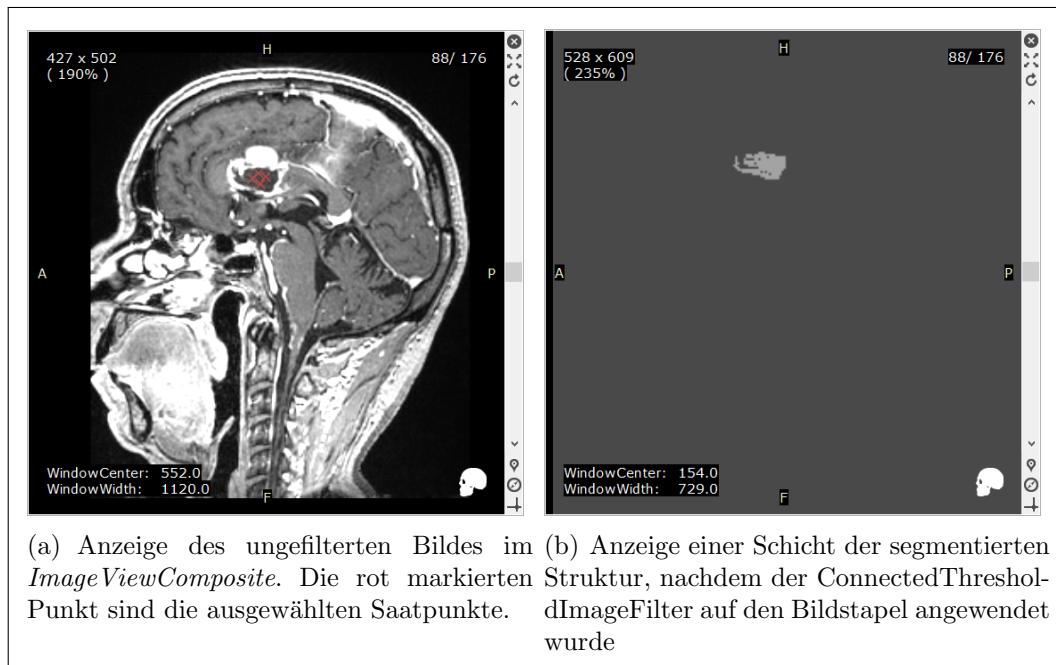


Abbildung 7.5.: Segmentierung einer Struktur mit dem ConnectedThresholdImageFilter

```

1  @Override
2  public List<AImage> process(List<AImage> arg0, int arg1) {
3      List<AImage> images = new ArrayList<AImage>(arg0.size());
4      for(int z = 0; z < arg0.size(); z++) {
5          ArrayList<Point2D<Float>> selection = arg0.get(arg1).
6              getPoints();
7
8          Image source = SimpleITKFactory.getITKImage(arg0.get(z));
9          ConnectedThresholdImageFilter segmentationFilter = new
10             ConnectedThresholdImageFilter();
11
12          segmentationFilter.setLower(lowerThreshold);
13          segmentationFilter.setUpper(upperThreshold);
14          segmentationFilter.setReplaceValue((short) 255);
15
16          for(Point2D<Float> seed : selection){
17              int x = (int) (seed.x*source.getWidth());
18              int y = (int) (seed.y*source.getHeight());
19
20              VectorUInt32 seed_v = new VectorUInt32(2);
21              seed_v.set(0, x);
22              seed_v.set(1, y);
23
24              segmentationFilter.addSeed(seed_v);
25          }
26          Image out = segmentationFilter.execute(source);
27          images.add(SimpleITKFactory.getAImage(out));
28      }
29      return images;
30  }

```

Listing 7.13: Implementierung der process-Methode des ConnectedThresholdImageFilters

7.2. Erweiterung der Anwendungsstruktur mit dem Eclipse Application Model

Anders als die Plug-ins werden Erweiterungen der Anwendung mit dem Application Model erst nach einem erneuten Buildprozess integriert. Dadurch richtet sich die Kategorie der Modularisierung an Entwickler, die jMediKit gezielt erweitern.

Als Beispiel für eine modulare Erweiterung soll im *PartStack* des *DicomBrowsers* ein neuer *Part* hinzugefügt werden, der theoretisch alle verfügbaren PACS im Netzwerk anzeigt. Implementiert wird nur eine minimale Oberfläche.

Bevor die Struktur erweitert werden kann, muss entsprechend der Anleitung in Anhang B die Eclipse-Installation vorbereitet sein. Sind die drei Projekte im *Project Explorer* vorhanden, kann mit der Erweiterung begonnen werden.

Im Projekt *org.jmedikit.plugin* befindet sich die Datei *Application.e4xmi*³. Darin sind alle strukturellen Informationen zu der Anwendungsoberfläche, Menüs, Handler, Commands und die Referenzen auf die implementierenden Klassen hinterlegt. Mit einem Doppelklick kann diese geöffnet werden und es zeigt sich ein Fenster wie in Abbildung 7.6. Die linke Spalte zeigt die Applikationsstruktur und rechts sind die entsprechenden Eigenschaften des gewählten Elements zu sehen. Ab dem Element *Windows* im Strukturbaum werden die Anwendungsteile angezeigt. Dieser Teilbaum entspricht der Abbildung 5.4 aus Kapitel 5. Da der *PartStack* in dem der *DicomBrowser* liegt erweitert werden soll, muss bis zu diesem Knoten navigiert werden. Der richtige *PartStack* hat die Id *org.jmedikit.plugin.dicombrowserStack*.

³Die XML-Datei *Application.e4xmi* kann auch direkt im Quelltext bearbeitet werden. Die Darstellung in Eclipse erhöht die Benutzerfreundlichkeit zur Anpassung der Werte

Entwicklung von Erweiterungen

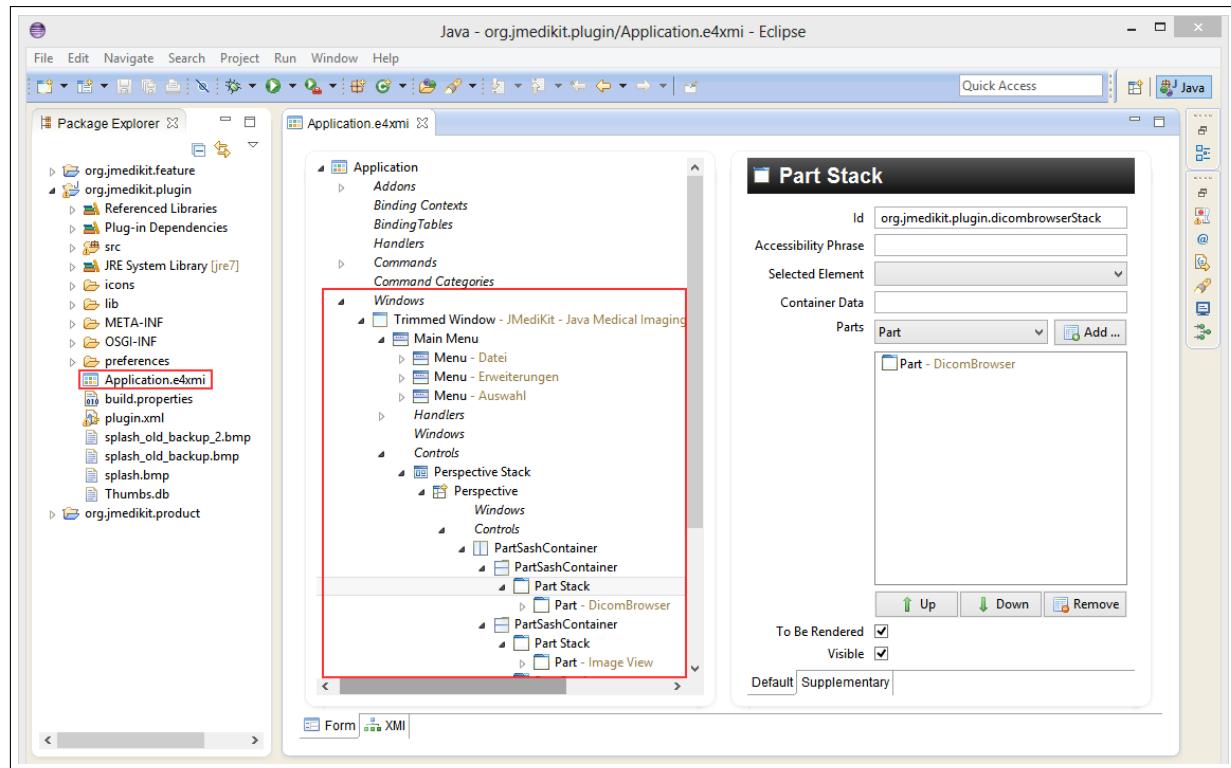


Abbildung 7.6.: Die Datei Application.e4xmi

Nach der Auswahl des *PartStack*-Elements sind rechts dessen Eigenschaften inklusive einer Liste aller enthaltenen *Parts* zu sehen (Abbildung 7.7). Über das Formularfeld *Parts* kann mit einem Klick auf *Add* ein neuer Kindknoten eingefügt werden. Im diesem Beispiel wird dem Stack ein neuer *Part* untergeordnet.

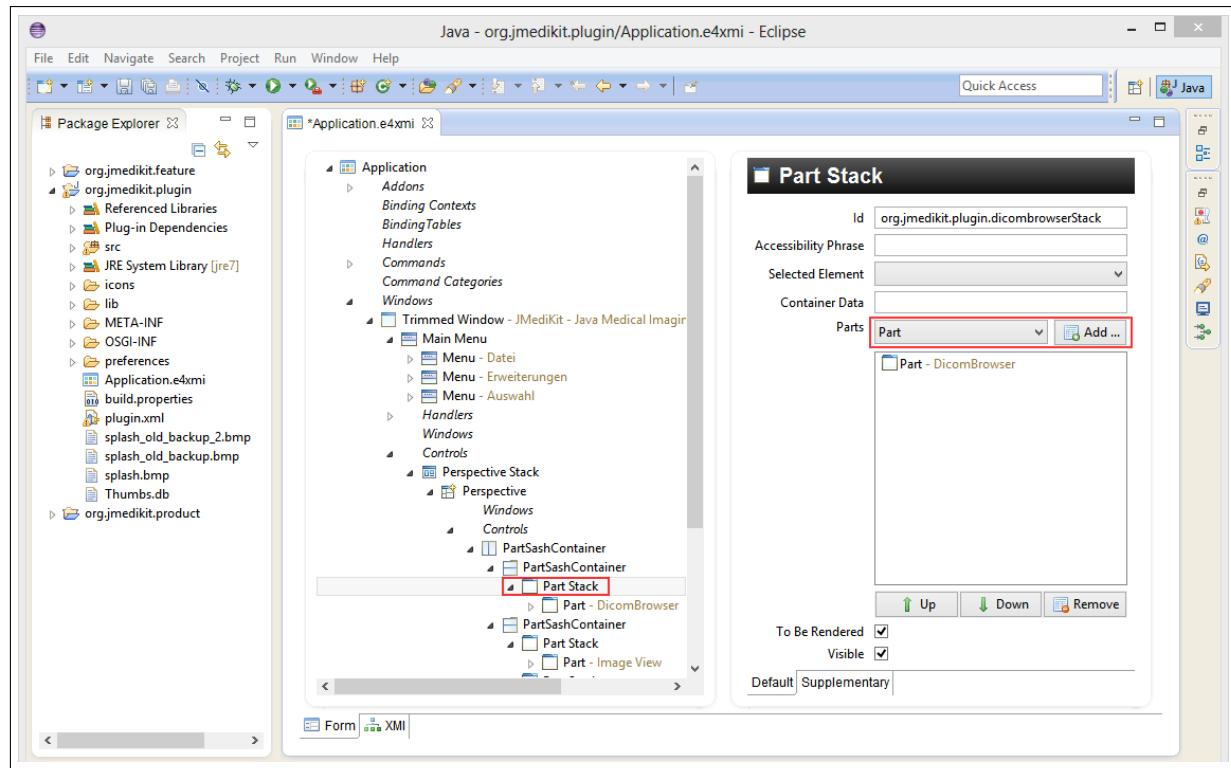
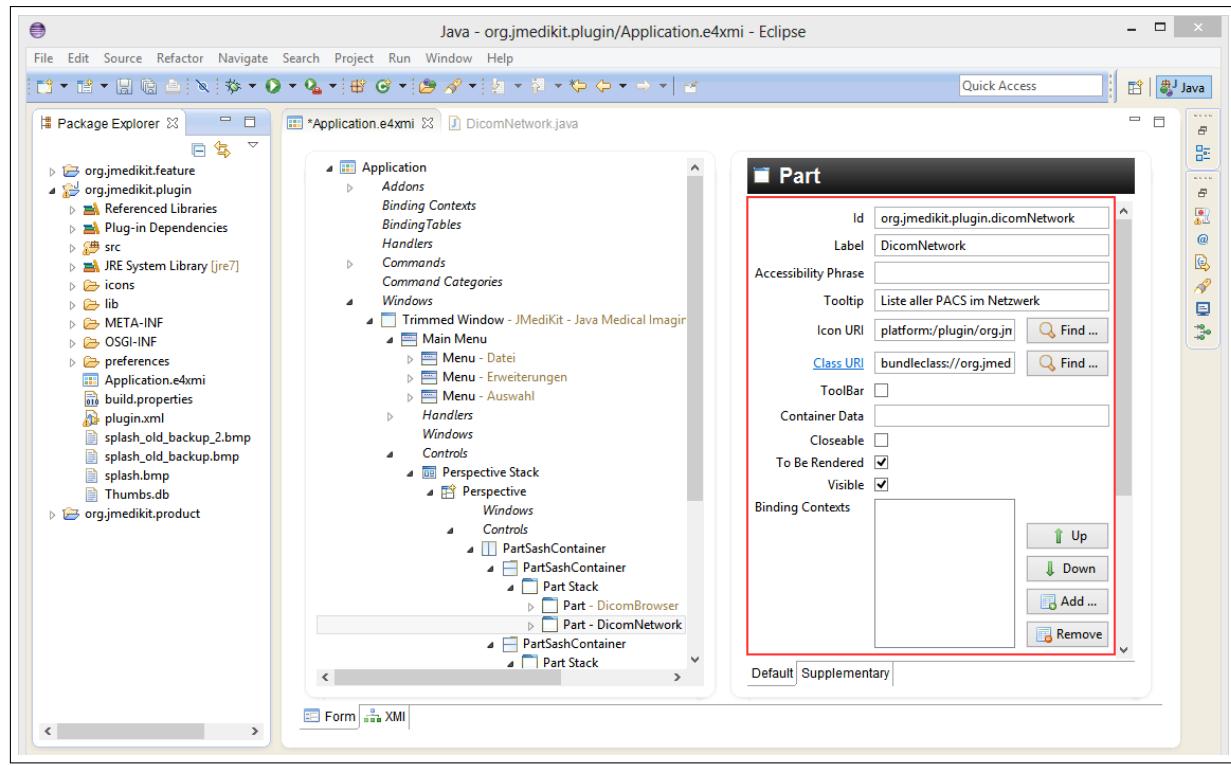


Abbildung 7.7.: Einfügen der Part-Struktur

Folgend öffnen sich rechts die Eigenschaften des neu angelegten Parts. Wichtige Einstellungen sind *Id*, *Label* und *ClassURI*. Mit Hilfe der Id kann der Part im Quelltext referenziert werden und das Label ist der sichtbare Titel des *Parts* in der Anwendung. Noch ist keine Klasse für das neue Strukturelement definiert worden. Dies kann mit einem Klick auf *ClassURI* erledigt werden. Soll ein Icon neben dem Titel erscheinen, muss zuvor eine Bilddatei nach der Anleitung in Anhang C importiert werden⁴.

⁴Grundsätzlich ist es ausreichend, die Bilddateien im *icon*-Ordner zu speichern, aufgrund einer durchgehenden Konsistenz ist es allerdings besser das Bild komplett zu importieren.

Entwicklung von Erweiterungen



7.

Abbildung 7.8.: Eigenschaften der Part-Struktur

Die zu erstellende Klasse repräsentiert neben der Benutzeroberfläche auch die Logik hinter dem *Part*. Abbildung 7.9 zeigt den Dialog zum Erstellen dieser Klasse. Alle *Part*-Implementierungen befinden sich im Package *org.jmedikit.plugin.gui*. Der Name ist frei wählbar. Neben Package und Klassename kann aus vier vordefinierten Methoden gewählt werden.

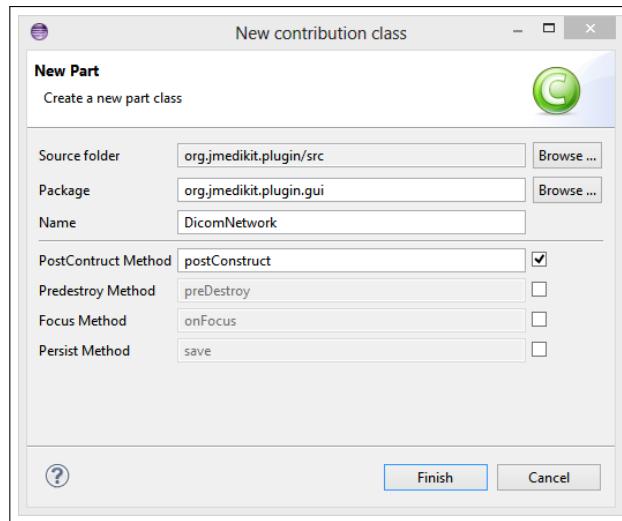


Abbildung 7.9.: Erstellen der Klasse

```
1 public class DicomNetwork {  
2  
3     @PostConstruct  
4     public void postConstruct(Composite parent) {  
5         //definiert das Layout des Elternelements  
6         //2 Spalten mit gleicher Breite  
7         GridLayout pGrid = new GridLayout(2, true);  
8         GridData parentData = new GridData(GridData.FILL_HORIZONTAL);  
9         parent.setLayout(pGrid);  
10        parent.setLayoutData(parentData);  
11  
12        //GUI  
13        //Suchfeld  
14        Text search = new Text(parent, SWT.BORDER);  
15        //Suchbutton  
16        Button button = new Button(parent, SWT.NONE);  
17        button.setText("Suche_PACS");  
18    }  
19}
```

Listing 7.14: Erweiternder Eintrag einer Konstanten in der Klasse *ImageProvider*

Listing 7.14 zeigt eine Beispielimplementierung des *Parts*. Die Annotation `@PostConstruct` sorgt dafür, dass die Methode automatisch nach dem Instantiiieren des Objekts ausgelöst wird. Diese enthält als Parameter ein *Composite* und stellt den Einhängepunkt

Entwicklung von Erweiterungen

für die weitere Benutzeroberfläche dar.

Abbildung 7.10 zeigt den neuen *Part* in der linken Spalte der Anwendung. Damit wurde dem *PartStack* in dem der *DicomBrowser* enthalten ist das neue *DicomNetwork* eingefügt.

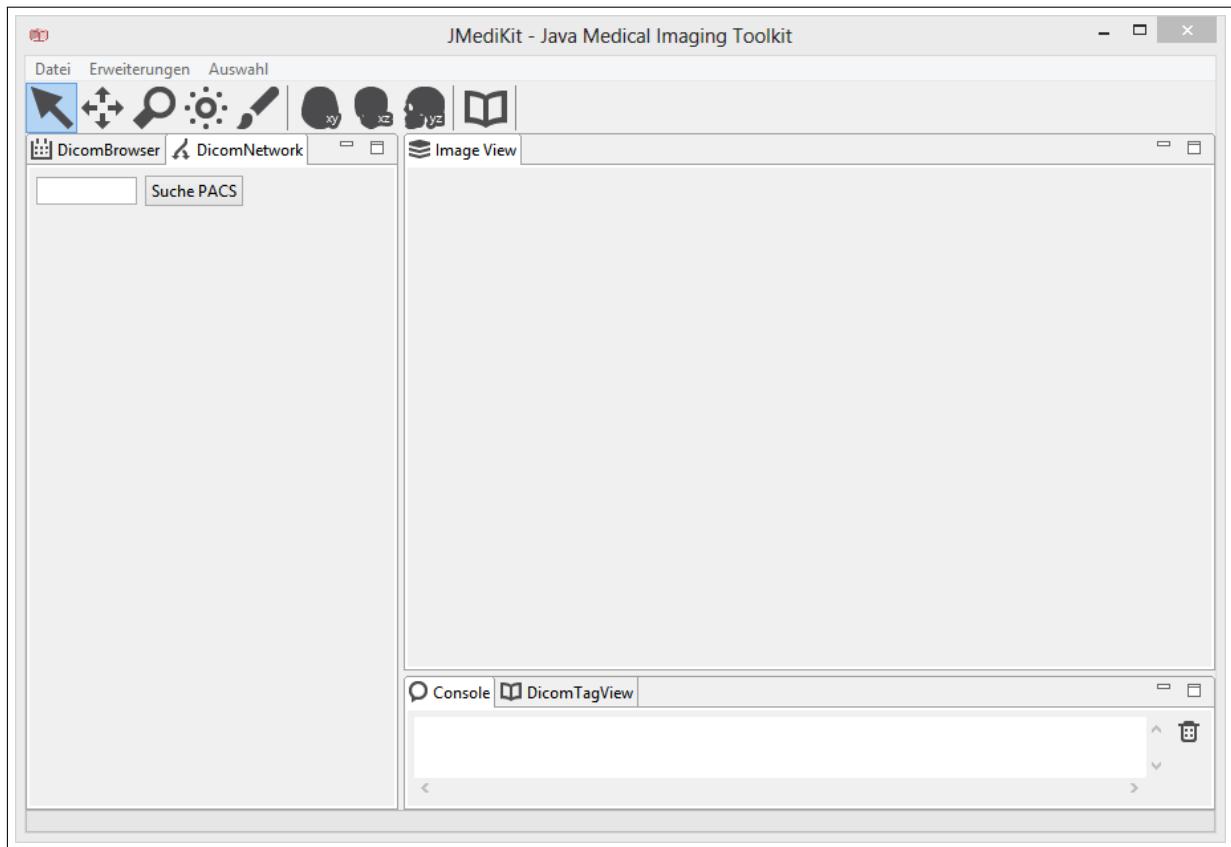


Abbildung 7.10.: Anzeige des Parts im PartStack

7.3. Erweiterung der Werkzeuge

Neben der Möglichkeit die Anwendungsstruktur zur erweitern und anzupassen, besitzen auch die Werkzeuge von jMediKit den modularen Charakter. Im folgenden Abschnitt wird der Werkzeugeleiste ein weiteres Tool hinzugefügt. Ähnlich dem *PointTool* sollen Bildbereiche ausgewählt werden. Der Unterschied besteht darin, dass keine Punkte bestimmt werden, sondern ein rechteckiger Bildbereich als *Region Of Interest*(ROI) markiert werden kann. Damit werden die Selektionswerkzeuge um das *ROITool* erweitert.

7.3.1. Hinzufügen eines Menüpunktes

Das Menü selbst befindet sich im Applikationsbaum *Window* unter dem Element *TrimBars* → *WindowTrim - Top* → *ToolBar* und enthält *HandledToolItems*-Strukturen des Application Models als Menüpunkte (Abbildung 7.11). Das bedeutet, dass zu jedem Eintrag ein *Command* mit dem zugehörigen *Handler* erstellt werden muss.

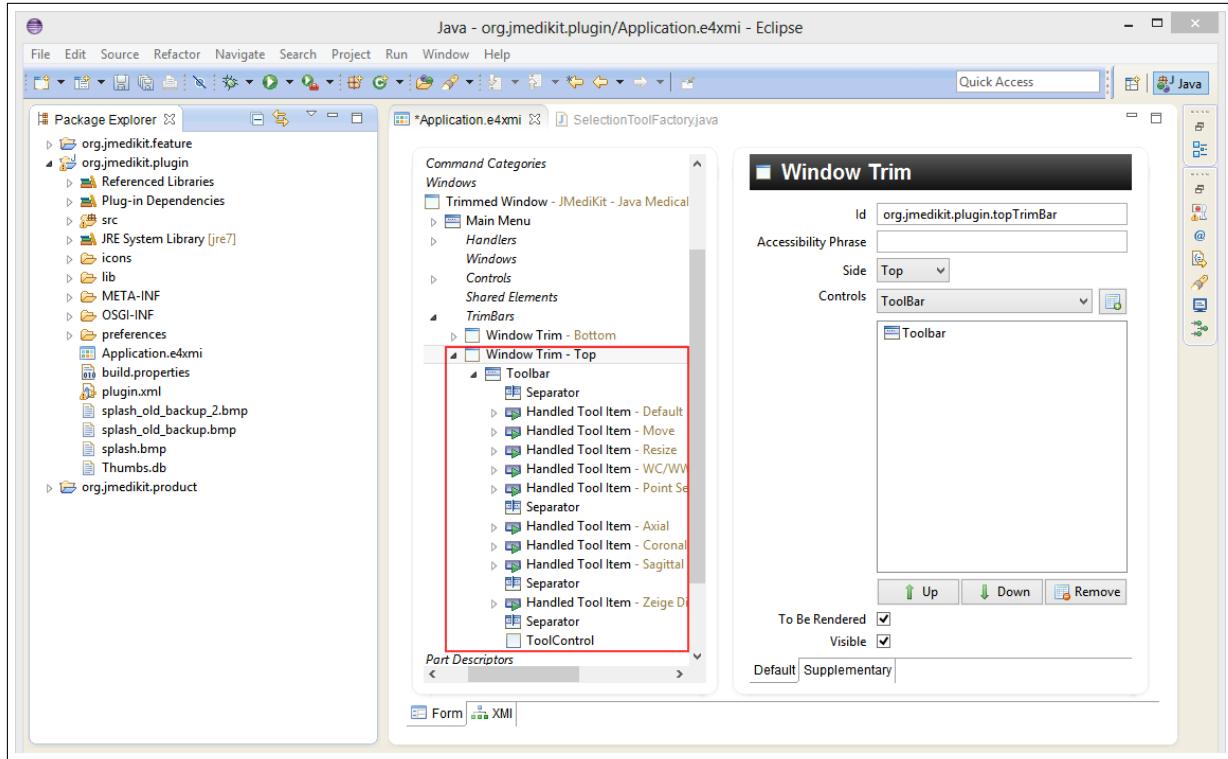


Abbildung 7.11.: Die Werkzeugleiste im Application Model

Die *Commands* sind im Strukturbau direkt unter *Application* zu finden. Wird das Element ausgewählt, erscheint auf der rechten Seite eine Liste bisher verfügbarer *Commands*. Mit einem Klick auf *Add* kann ein neuer hinzugefügt werden. In den nun angezeigten Einstellungen bekommt die *Id* den Wert *org.jmedikit.plugin.command.toolRoi* und als *Name* wird *toolRoi* eingetragen.

Die Implementierung der Befehle finden in den *Handlern* statt. Diese sind im Baum unter *Window* angesiedelt. Nach der Auswahl erscheint die Liste der erstellten *Handler*. Wie schon bei den *Commands*, wird ein neuer *Handler* hinzugefügt und *Id* sowie *Name* vergeben. Unter dem Punkt *Command* wird der zuvor erstellte Befehl angegeben. Mit einem Klick auf *ClassURI*, wie in Abbildung 7.12 dargestellt, kann die

entsprechende Klasse erstellt werden. Üblicherweise liegen Handlerklassen im Package `org.jmedikit.plugin.gui.handler` und haben den Namen `ToolNameHandler`.

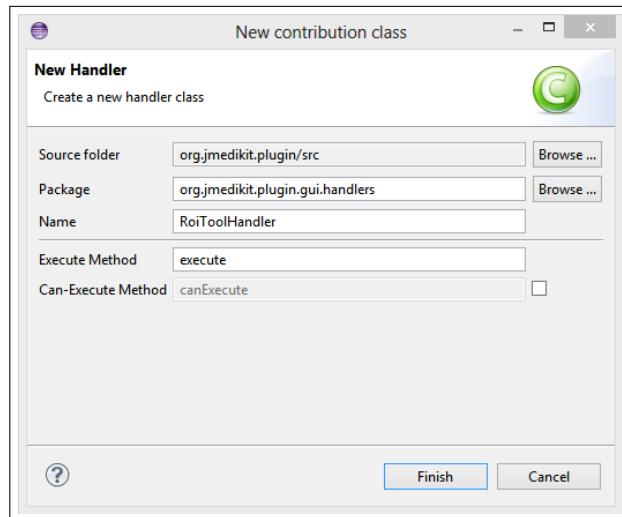
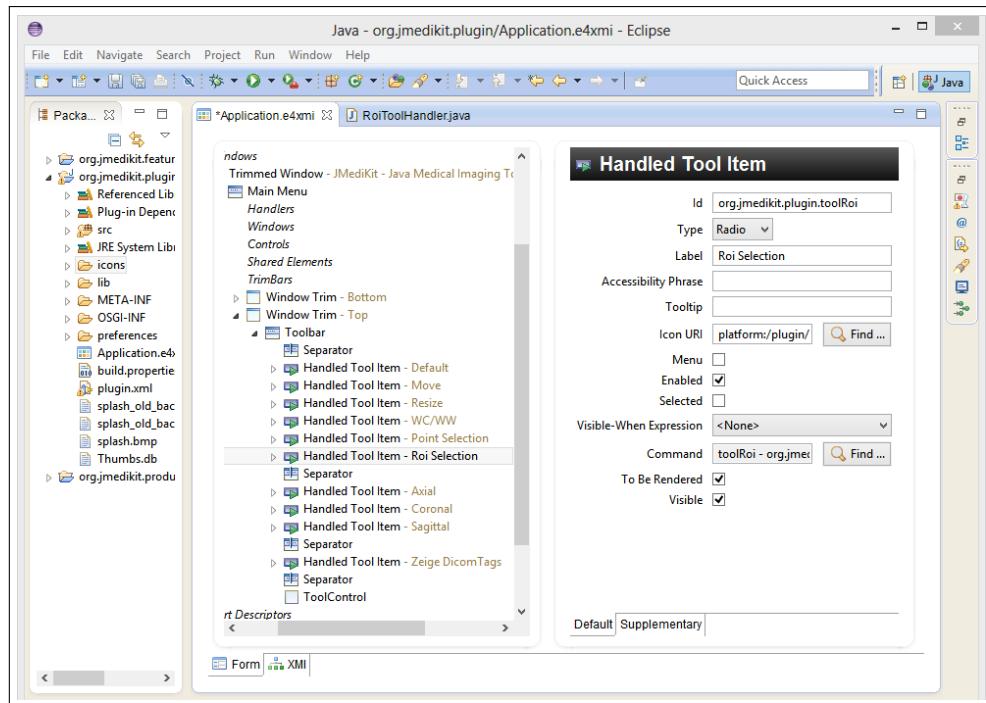


Abbildung 7.12.: Erstellung einer Klasse für die Handlerimplementierung

7.

Im nächsten Schritt erfolgt das Hinzufügen eines `HandledToolItem` zu dem Element unter `Window → TrimBars → WindowTrim - Top → ToolBar`. Die Reihenfolge der Liste entspricht der Darstellung in der Anwendung. Unter den Einstellungen muss `Id`, `Label` und `Command` mit Werten belegt werden. Als `Type` muss die Option `Radio` gewählt werden, da genau ein Werkzeug aktiv ausgewählt sein darf. Unter dem Punkt `IconURI` wird bei Bedarf ein Bild angegeben. Abbildung 7.13 zeigt die gesetzten Einstellungen und 7.14 die neue Werkzeugeleiste mit dem `ROITool` als erweiterten Eintrag.

Entwicklung von Erweiterungen



7.

Abbildung 7.13.: Einstellungen des neuen Menüpunktes

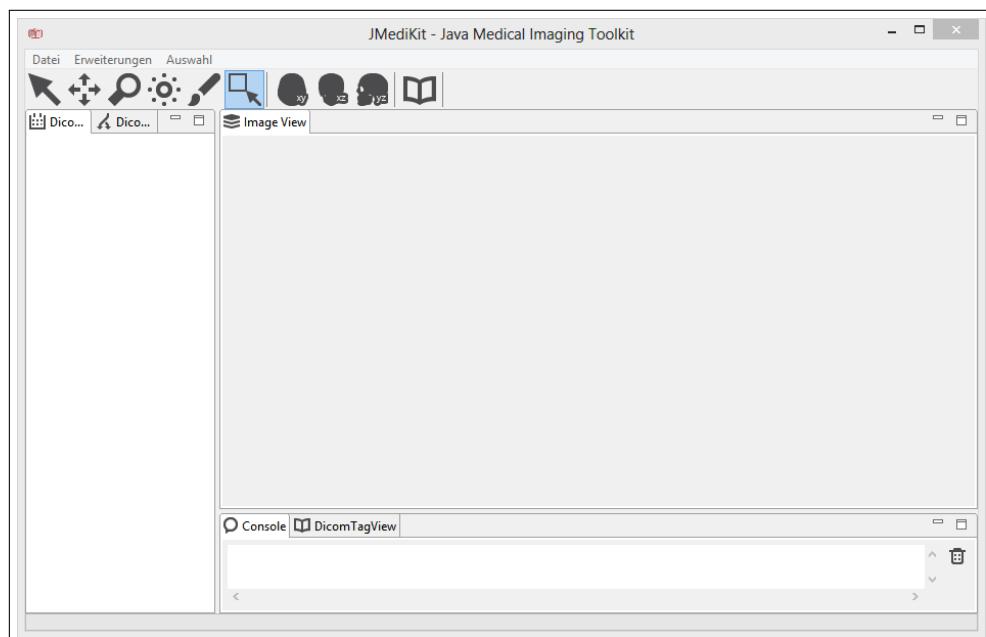


Abbildung 7.14.: Anzeige der erweiterten Werkzeugeleiste

7.3.2. Hinzufügen der Funktionalität

Nach der Erweiterung der Werkzeugleiste kann die grundlegende Klasse des Werkzeugs erstellt werden. Wichtig ist die Generalisierung von *ATool*, wie in Listing 7.15 zu sehen ist.

```
1 | public class RoiTool extends ATool{  
2 |  
3 |     public RoiTool(DicomCanvas c) {  
4 |         super(c);  
5 |         System.out.println("Hello_ROITool");  
6 |     }  
7 |  
8 |     //weitere Implementierungen abstrakter Methoden  
9 |     //der Klasse ATool  
10| }
```

Listing 7.15: Erweiterung des Basiswerkzeugs

Wird der Menüpunkt in der Werkzeugleiste betätigt, werden Events ausgelöst. In der Klasse *EventConstants* sind diese jeweils in Gruppen definiert. Die Wurzel einer Gruppe hat die in Listing 7.16 dargestellte Struktur. *TOOL_CHANGED* ist der Gruppenname der Events, die Werkzeuge betreffen. Der Zusatz */** symbolisiert die Wurzel. Ein spezifisches Werkzeug hat die Form *TOOL_CHANGED/TOOLNAME*. Durch die Gruppenmechanik ist es möglich, dass eine spezielle Funktion auf alle Werkzeug-Events lauscht und abhängig vom Event die Aufgaben delegiert.

```
1 | public final static String TOOL_CHANGED_ALL = "TOOL_CHANGED/*";  
2 | public final static String TOOL_CHANGED_ALL = "TOOL_CHANGED/POINT  
3 |         ";  
4 | public final static String TOOL_CHANGED_ALL = "TOOL_CHANGED/ROI";
```

Listing 7.16: Eventkonstanten der Klasse EventConstants

Nachdem die Tool-Klasse und das Event kreiert wurden, kann das neue Werkzeug der Fabrik, die für die Erzeugung der Objekte zuständig ist, bekannt gemacht werden. Da das *ROITool* zum Bereich der Selektion gehört, wird die Klasse *SelectionToolFactory* angepasst. In Listing 7.17 werden die Modifikationen dargestellt. Innerhalb der Fabrik werden die zugehörigen Events in einer Konstante gespeichert. Die Methode *produce* regelt die Objekterzeugung. Klickt der Anwender auf den Menüpunkt *ROITool* wird das Event

ausgelöst und der Werkzeugname an die Fabrik übergeben. Aufgrund des *toolname* wird das richtige Werkzeug erstellt.

```
1 | public final static String ROI_TOOL =
2 |     EventConstants.TOOl_CHANGED_ROI;
3 | //weitere Konstanten
4 |
5 |
6 |@Override
7 | protected ATool produce(String toolname, DicomCanvas c) {
8 |     //Pruefung vorheriger Werkzeuge
9 |     else if(toolname.equals(ROI_TOOL)) {
10|         return new RoiTool(c);
11|     }
12|     //Pruefung folgender Werkzeuge
13|     //Werkzeug wurde nicht gefunden
14|     else throw new IllegalArgumentException("ERROR");
15| }
```

Listing 7.17: Modifikation der SelectionToolFactory

Im aktuellen Zustand reagiert das Werkzeug und der neue Menüpunkt noch nicht auf die Eingaben des Benutzers. Es fehlt die Implementierung des *Handlers* und die Kommunikation mit dem *ImageView*. Listing 7.18 zeigt die Umsetzung der Klasse *ToolRoiHandler*. Das Interface *IEventBroker* regelt die Kommunikation unter den *Parts*. Die Annotation *@Execute* bedeutet, dass die Methode automatisch aufgerufen wird, sobald der Menüpunkt betätigt wird. Sowohl *@Inject* als auch *@Execute* sind Teile des *Dependency Injection Frameworks* von Eclipse. Ein umfassender Artikel ist unter [Vog13c] zu finden.

In der Methode *execute* wird dem Werkzeug entsprechend die Fabrik instantiiert. Die Variable *tool* enthält das auszulösende Event. Mit Hilfe des *eventBroker* wird die Benutzereingabe an die lauschenden Methoden der Applikation geleitet. Die Methode *post* erwartet die Event-Konstante und ein ToolEvent, welches die erzeugte Fabrik und den Werkzeugtyp als Parameter erhält.

Klickt der Benutzer auf den Menüpunkt nachdem der *Handler* implementiert wurde, erfolgt die Ausgabe „Hello ROITool“ auf der Standardausgabe. Das bedeutet, das Werkzeug wurde korrekt erzeugt und kann verwendet werden. Der nächste Schritt ist die Implementierung der Logik.

```
1 | @Inject
2 | IEventBroker eventBroker;
3 |
4 | @Execute
5 | public void execute() {
6 |
7 |     AToolFactory factory = new SelectionToolFactory();
8 |     String tool = EventConstants.TOOI_CHANGED_ROI;
9 |
10 |    eventBroker.post(EventConstants.TOOI_CHANGED_ROI, new
11 |                      SelectionToolEvent(factory, tool));
12 | }
```

Listing 7.18: Implementierung des ToolRoiHandler

7.3.3. Hinzufügen der Werkzeuglogik

Ziel des *ROITools* ist es, beliebige rechteckige Flächen innerhalb des Bildes zu markieren, um diese später in Plug-ins benutzen zu können. So können Algorithmen auf ausgewählten Bildbereichen angewendet werden. Folgende Events werden von dem neuen Werkzeug behandelt:

7.

- **MouseMove**

Bei jeder Mausbewegung wird geprüft, ob eine Maustaste gedrückt wird. Ist dies der Fall, wird die Startposition und die Aktuelle Position solange in Datenelementen der Klasse gespeichert, bis die Taste gelöst wird.

- **MouseDown**

Wird die Maustaste betätigt, werden die Koordinaten auf 0 zurückgesetzt, damit eine neue Berechnung beginnen kann. Ohne Zurücksetzung wird die zuletzt markierte ROI nochmal auf die Zeichenfläche gemalt und bleibt bis zur ersten Mausbewegung sichtbar.

- **MouseUp**

Befindet sich nach dem Loslassen der Maustaste der Start- und Endpunkt des Cursors innerhalb des Bildes, wird eine *Region Of Interest* berechnet. Dazu werden zuerst die Bildkoordinaten der beiden Punkte ermittelt. Darauf folgt eine Normalisierung der Koordinaten und ein Eintrag der im Anschluss erzeugten ROI im Bild.

- **PostCalculation**

Nach der Werteberechnung wird das Rechteck der potentiellen ROI auf der Zeichenfläche dargestellt.

Ist die Logik implementiert, kann das Werkzeug in vollem Umfang eingesetzt werden. Das Erstellen von Transformationstypen erfolgt analog zu den Selektionswerkzeugen. Eine erweiterte Implementierung des *ROI Tools* könnte zum Beispiel eine dynamische Größenanpassung der Fläche mit dem Mausrad realisieren.

8. Diskussion

Für eine erfolgreiche Umsetzung der Arbeit stehen die Anforderungen an die Modularität und Erweiterbarkeit der zu entwickelnden Software und die Methoden zur Bildanzeige und Manipulation im Fokus.

Modularität

Mit dem Einsatz der „Eclipse Rich Client Platform“ und dem zugehörigen OSGi-Framework „Equinox“ wird eine modulare Anwendungsstruktur geschaffen. Durch das Hinzufügen von neuen Strukturelementen des Application Models und deren Implementierung kann jMediKit um neue Module erweitert werden. Dadurch wird die Tiefe der Anwendung mit den zur Verfügung stehenden Grundfunktionen erweitert. So ist es zum Beispiel möglich, die Anwendung um einen Bereich zu erweitern, der aus den Voxeldaten dreidimensionale Darstellungen berechnet.

Um neue Module zu entwickeln, ist neben der Kenntnis der Struktur von jMediKit auch eine Einarbeitung in die Architektur der Eclipse-Plattform notwendig. Die Werkzeuge sind zum Beispiel fest mit der jMediKit-Architektur durch das Fabrikmuster verankert. Für eine Erweiterung benötigt ein Entwickler das Wissen zu Werkzeugleisten der Eclipse-Plattform und zur Werkzeugentwicklung von jMediKit. Zusätzlich können neue Funktionen ausschließlich nach einem erneuten Buildprozess der Anwendung hinzugefügt werden. Dadurch ist keine dynamische Modulerweiterung des aktuellen Systems möglich.

Erweiterbarkeit

Das Architekturmuster Plug-in stellt diesen dynamischen Aspekt der Anwendungserweiterung zur Verfügung. Das bedeutet, Klassen können auch nach der Kompilierung zu dem System hinzugefügt werden. Dadurch wird eine Erweiterung möglich, ohne genaue Kenntnis des Systems und seine interne Funktionsweise besitzen zu müssen. Trotz des dynamischen Charakters sind die vom Anwender entwickelten Erweiterungen auf die Aspekte der Bildverarbeitung beschränkt und können keine Beiträge zur Anwendungsstruktur leisten.

Bildanzeige und Manipulation

Mit der Verwendung der externen Bibliothek „dcm4che“ als Werkzeug zur Verarbeitung der DICOM-Daten, können medizinische Bilddaten entsprechend der Anforderungen angezeigt und manipuliert werden. Da Algorithmen sowohl in der initialen Ebenendarstellung, als auch in den rekonstruierten Darstellungen angewendet werden sollen, wird der gesamte Datensatz beim Einlesen in den Speicher geladen. Dadurch hat jMediKit einen hohen Speicherbedarf. So benötigt allein die Anzeige von vier Datensätzen mit 512 Schichten und einer Auflösung von 512×512 Pixel und einer Tiefe von 16-Bit ca. 1 Gigabyte Speicherplatz rein an Daten der Pixelwerte.

$$(312^3 \cdot 16) \cdot 4 = 8589934592\text{bit} = 1\text{GB} \quad (8.1)$$

In der initialen Ebenendarstellung könnten Bildschichten problemlos bei Bedarf nachgeladen werden. Ein dreidimensionaler Algorithmus ist zur Berechnung allerdings abhängig von den restlichen Schichten. Auch Ebenenrekonstruktionen benötigen den gesamten Datensatz. Ein dynamisches Nachladen bei der Algorithmenanwendung und Rekonstruktion verlängert die Ausführungszeit des Programms und es musste die Wahl zwischen einem hohen Speicherbedarf und schneller Bedienung oder geringem Speicherbedarf bei verzögerter Bedienung getroffen werden.

So könnten zukünftige Weiterentwicklungen einen Kompromiss zwischen Speicher und Ausführungszeit finden, um eine verbesserte Lösung zur Verfügung zu stellen. Caching-Algorithmen wären in der Lage den Ladevorgang und die Speicherung effizient zu verwalten.

9. Fazit

Die qualitativen Anforderungen der Modularität und Erweiterbarkeit an Architektur und Programmeigenschaften werden mit Hilfe einer überaus flexiblen Anwendungsstruktur umgesetzt, die eine Erweiterung an den entscheidenden Stellen zulässt. So ist sowohl eine statische Erweiterung durch Module und Werkzeuge, als auch eine dynamische durch Plug-ins in zwei- und dreidimensionalem Raum möglich.

Die funktionalen Anforderungen an die Software decken im Besonderen die Anzeige und Manipulation der Bilder sowie die dynamische Parameterübergabe ab. Die Bildanzeige kann zur Navigation und Orientierung im 3D-Datensatz verwendet werden und bietet typische Operationen wie die Translation und die Skalierung. Mit Hilfe der Selektionswerkzeuge können zusätzlich für den Anwender wichtige Bereiche markiert und in Plug-ins interpretiert werden.

JMediKit setzt die Anforderungen an die Software und deren Architektur um und liefert eine solide Grundlage für zukünftige Erweiterungen. Denkbar wäre eine Anbindung an das Picture Archiving and Commnication System des Labors, um medizinische Bilder direkt aus dem Netzwerk beziehen zu können. Zusätzlich könnte ein 3D-Renderer integriert werden, der eine dreidimensionale Darstellung des Voxelraums bietet. Das Java Medical Imaging Toolkit bietet dazu sowohl das Potential, als auch die Schnittstellen und stellt damit eine modulare und erweiterbare Anwendung zur medizinischen Bildverarbeitung dar.

Literaturverzeichnis

- [Art13] ARTHORNE, John: *Eclipse SDK 4.2.* <https://wiki.eclipse.org/Eclipse4>. Version: 2013. – Stand 05.02.2014
- [Bal11] BALZERT, Helmut: *Lehrbuch der Softwaretechnik - Entwurf, Implementierung, Installation und Betrieb*. Spektrum, 2011
- [BB06] BURGER, Wilhelm ; BURGE, Mark J.: *Digitale Bildverarbeitung - Eine Einführung mit Java und ImageJ*. Springer, 2006
- [BLT98] BÜCHELER, Egon ; LACKNER, Klaus-Jürgen ; THELEN, Manfred: *Einführung in die Radiologie: Diagnostik und Interventionen*. Georg Thieme Verlag, 1998
- [Cor07] CORD, Siemon: Innovationspolitik im 6. Kondratieff: Hinterherlaufen oder Vorausilen? In: *Wirtschaftsdienst* 87 (2007), Juli, Nr. 7, S. 450–457
- [ES13] EILEBRECHT, Karl ; STARKE, Gernot: *Patterns kompakt - Entwurfsmuster für effektive Software-Entwicklung*. Springer Vieweg, 2013
- [Far14] FARHAT, N.: *Tutorial Video 2014 January*. <http://www.slicer.org/publications/bitstream/viewbiglogo/2511/Farhat-Slicer-3D-Printing>. Version: 2014. – Stand 05.02.2014
- [GD13] GOLL, Joachim ; DAUSMANN, Manfred: *Architektur- und Entwurfsmuster der Softwaretechnik*. Springer Vieweg, 2013
- [GN11] Kapitel Der sechste Kontratieff. In: GRANIG, P. ; NEFIODOW, L. A.: *Gesundheitswirtschaft – Wachstumsmotor im 21. Jahrhundert*. Gabler Verlag, 2011
- [Han00] HANDELS, Heinz: *Medizinische Bildverarbeitung*. B.G. Teubner Stuttgart Leipzig, 2000
- [HK11] Kapitel Die gesunde Gesellschaft und ihre Ökonomie – vom Gesundheitswesen zur Gesundheitswirtschaft. In: HENSEN, P. ; KÖLZER, Christian: *Die gesunde Gesellschaft*. VS Verlag für Sozialwissenschaften, 2011

LITERATURVERZEICHNIS

- [Hoc13] HOCHSCHULE LANDSHUT: *Modulhandbuch BA BMT.* https://www.haw-landshut.de/fileadmin/hs_landshut_english/electrical_engineering/download/pdf/Modulhandb%FCcher/Modulhandbuch_BA_BMT_WS_13_14_SS_13_beschlossen_FR_2013_11_26.pdf. Version: November 2013
- [LVTN09] LEHMANN, Gaëtan ; VILVERT, Domaine de ; TONDDAST-NAVAEI, Ali: *WrapITK release page for version 0.3.0.* http://code.google.com/p/wrapitk/wiki/Release030#Class_coverage. Version: 2009
- [Nat11a] NATIONAL ELECTRICAL MANUFACTURERS ASSOCIATION: *Digital Imaging and Communications in Medicine (DICOM) - Part 3: Information Object Definitions.* ftp://medical.nema.org/medical/dicom/2011/11_03pu.pdf. Version: 2011
- [Nat11b] NATIONAL ELECTRICAL MANUFACTURERS ASSOCIATION: *Digital Imaging and Communications in Medicine (DICOM) - Part 5: Data Structures and Encoding.* ftp://medical.nema.org/medical/dicom/2011/11_05pu.pdf. Version: 2011
- [Nat11c] NATIONAL ELECTRICAL MANUFACTURERS ASSOCIATION: *Digital Imaging and Communications in Medicine (DICOM) - Part 6: Data Dictionary.* ftp://medical.nema.org/medical/dicom/2011/11_06pu.pdf. Version: 2011
- [NFPS11] NISCHWITZ, Alfred ; FISCHER, Max ; PETER, Haberäcker ; SOCHER, Gundrun: *Computergrafik und Bildverarbeitung - Band I: Computergrafik.* Vieweg+Teubner, 2011
- [Pia08] PIANYKH, Oleg S.: *Digital Imaging and Communications in Medicine (DICOM).* Springer-Verlag Berlin Heidelberg, 2008
- [Smi03] SMITH, Jolinda: *Create 3D image data by a series of 2D dicom files t.* <http://www.itk.org/pipermail/insight-users/2003-September/004762.html>. Version: 2003. – Stand 05.02.2014
- [The13] THE ECLIPSE FOUNDATION: *Eclipse documentation - Current Release.* <http://help.eclipse.org/kepler/index.jsp>. Version: 2013

LITERATURVERZEICHNIS

- [The14] THE ECLIPSE FOUNDATION: *SWT: The Standard Widget Toolkit*. <http://www.eclipse.org/swt/>. Version: 2014. – Stand 23.01.2014
- [Ull07] ULLENBOOM, Christian: *Java ist auch eine Insel - Das umfassende Handbuch*. Galileo Press, 2007
- [Vog13a] VOGEL, Lars: *Eclipse 4 RCP - Building Eclipse RCP applications based on Eclipse 4.* http://www.vogella.com/tutorials/EclipseRCP/article.html#e4overview_eclipse4. Version: 2013. – Stand 23.01.2014
- [Vog13b] VOGEL, Lars: *Eclipse IDE Tutorial*. <http://www.vogella.com/tutorials/Eclipse/article.html#eclipseoverview>. Version: 2013. – Stand 23.01.2014
- [Vog13c] VOGEL, Lars: *Using dependency injection in Java - Introduction*. <http://www.vogella.com/tutorials/DependencyInjection/article.html>. Version: 2013. – Stand 17.02.2014
- [Wie12] Kapitel Kondratieff – Von der Dampfmaschine zum Menschen. In: WIEDER, M.: *Liquid Work*. Springer Fachmedien Wiesbaden, 2012

Abbildungsverzeichnis

1.1.	Kondratieff-Zyklen [HK11, S.32]	2
2.1.	RadiAnt - DicomViewer	10
2.2.	Screenshot des Programms Slicer 3D [Far14]	11
2.3.	Die Benutzeroberfläche von ImageJ	12
3.1.	Kommunikationsprozess von Aufnahme zur Verarbeitung [Pia08, S. 19]	15
3.2.	Vereinfachte Darstellung der Informationsobjekthierarchie von Dicomelementen [Nat11a, A.1.2]	17
3.3.	Repräsentation der DICOM Informations Objects im Dateisystem	18
3.4.	Kodierungsreihenfolge von 4 Byte bei Little Endian- und Big Endian-Darstellung	22
3.5.	Beispiele unterschiedlicher Speicherbelegung	22
3.6.	Verschiedene Graustufenbilder	24
3.7.	Fensterungstechnik zur Darstellung medizinischer Bilddaten am handelsüblichen Monitor [Han00, S. 249]	25
3.8.	Kodierung der RGB-Werte im Datenelement PixelData mit Hilfe der PlanarConfiguration	26
3.9.	Darstellung von 2- und 3-dimensionalen Bilddaten	27
4.1.	UML Klassendiagramm zum Adapter [ES13, S. 78]	29
4.2.	UML Klassendiagramm zum Observer [ES13, S. 71]	30
4.3.	UML Klassendiagramm der Schablonenmethode [ES13, S. 69]	32
4.4.	UML Klassendiagramm zur Fabrik Methode [ES13, S. 35]	33
4.5.	UML Klassendiagramm des Architekturmusters Plug-in [GD13, S. 316]	35
5.1.	Architektur der Eclipse e4 Plattform zur Entwicklung von Rich Client Anwendungen [Art13]	38
5.2.	Verschiedene Elemente des Application Models	41
5.3.	Die Benutzeroberfläche von jMediKit	43

ABBILDUNGSVERZEICHNIS

5.4. jMediKit und die hierarchische Anordnung der Elemente des Application Models	44
5.5. Die Fabrik Methode zur Werkzeugerzeugung	45
5.6. Architektur der Plug-in-Struktur	46
5.7. Architektur des generischen Dialogs zur dynamischen Parameterbestimmung	48
5.8. Das Adapter-Muster unter jMediKit	51
5.9. Diagramm zur Klassenstruktur der Bilddaten	52
5.10. Organisation der Klassen zur Anzeige der DICOM-Bilddaten	53
5.11. Benutzeroberfläche der ImageViewComposites	54
 6.1. Die Baumansicht des DICOM-Browsers mit geladenen Objekten	57
6.2. Beispielhafte Darstellung eines Baumes mit $h = 4$ in der Implementierung .	58
6.3. Räumliche Sortierung der DICOM-Objekte	61
6.4. Die verschiedenen Ansichten der Ebenen eines dreidimensionalen Datensatzes	64
6.5. Der Prozess von der Bildauswahl zur Anzeige	65
6.6. Rekonstruktion der coronalen und sagittalen Ebene	66
6.7. Rotation der Richtungsvektoren von axialer zu coronaler Darstellung . . .	67
6.8. Berechnung der Position und Koordinaten. Die graue Fläche symbolisiert das Bild und der rote Rahmen stellt den sichtbaren Bildausschnitt dar. . .	69
6.9. Vergleich der Nearest Neighbor, bilinearen und bikubischen Interpolation .	71
6.10. Bilineare Interpolation [BB06, S.388]	72
6.11. Das patientenbasierte Koordinatensystem	74
6.12. Beschriftung der Koordinatenachsen bei axialer Ebenendarstellung	75
6.13. Einfache Darstellung der Orientierungslinien	76
6.14. Anzeige eines referenzierten Punktes in axialer Ebene mit Hilfe der Ori- entierungslinien	77
6.15. Translation des Bildmittelpunkts	80
 7.1. Beispiel der Ordnerstruktur zwei verschiedener Plug-ins	88
7.2. Die visuelle Darstellung des Dialogs aus Listing 7.9	94
7.3. Filterung eines medizinischen Bildes mit dem Laplace-Operator	95
7.4. Filterung eines medizinischen Bildes mit dem Sobel-Operator	98
7.5. Segmentierung einer Struktur mit dem ConnectedThresholdImageFilter .	99
7.6. Die Datei Application.e4xmi	102
7.7. Einfügen der Part-Struktur	103
7.8. Eigenschaften der Part-Struktur	104

ABBILDUNGSVERZEICHNIS

7.9. Erstellen der Klasse	105
7.10. Anzeige des Parts im PartStack	106
7.11. Die Werkzeugleiste im Application Model	107
7.12. Erstellung einer Klasse für die Handlerimplementierung	108
7.13. Einstellungen des neuen Menüpunktes	109
7.14. Anzeige der erweiterten Werkzeugleiste	109
B.1. Standardinstallation eines 32-Bit Eclipse unter Windows 8	XV
B.2. e4 Projekt Builds	XVI
B.3. e4 Repository Link	XVII
B.4. Installation neuer Software unter Eclipse	XVIII
B.5. Angabe des Repository	XVIII
B.6. Auswahl der e4 Tools zur Installation	XIX
B.7. Vorgang zum Importieren bereits bestehender Projekte	XX
B.8. Der Package Explorer nach dem Importvorgang	XXI
B.9. Erster Start der Anwendung über Eclipse	XXII
B.10. Konfigurationsfenster zu Anwendungseinstellungen und Anwendungsstart .	XXIII
B.11. Die von Eclipse verwendeten Java Runtime Environments	XXIV
B.12. Mit jmedikit.product den Build erstellen	XXV
B.13. Optionen zur Erstellung des Builds	XXVI
D.1. Dialog zum Anlegen eines neuen Javaprojekts	XXX
D.2. Auswahl des Output-Ordners	XXXI
D.3. Auswahl der externen Bibliotheken	XXXII
D.4. Der Eclipse Workspace nach Anlegen des Projekts	XXXIII
D.5. Anlegen der Hauptklasse des Plug-ins	XXXIV

Tabellenverzeichnis

2.1.	Gegenüberstellung der Anforderungen und verfügbarer freier Software . . .	13
3.1.	Repräsentation des Patientennamen als DICOM-Element	19
3.2.	Das erzeugte DICOM-Objekt mit den Elementen zu Patientenname, Ge- burtsdatum, Geschlecht und Alter	19
3.3.	Grundlegende Datenelemente für die digitale Repräsentation	23
6.1.	Ganzzahlige Datentypen in Java	60
6.2.	Von jMediKit implementierte Bildtypen	60
A.1.	Darstellung des Datenelements im Speicher wenn VR vom Typ OB, OW, OF, SQ, UT oder UN	XI
A.2.	Darstellung des Datenelements für alle anderen VR-Typen	XII
A.3.	Darstellung des Datenelements für implizite VR	XIII

Listings

4.1. Beispiel zur Schablonenmethode	31
4.2. Implementierung des Singleton-Musters in Java	34
7.1. Der reguläre Ausdruck gültiger Klassennamen	87
7.2. Definition des konkreten 2D-Plug-ins <code>_2dPlugIn.java</code>	89
7.3. Definition des konkreten 3D-Plug-ins <code>_3dPlugIn.java</code>	89
7.4. Erzeugung eines AImage	90
7.5. Konvertierung der Bildobjekte zwischen jMediKit und SimpleITK	91
7.6. Saatpunkte und Regions Of Interest aus einem AImage ermitteln	91
7.7. Saatpunkte und Regions Of Interest aus einem AImage ermitteln	92
7.8. Lesen eines Tags aus einem DICOM-Objekt	93
7.9. Erstellung eines eigenen Optionsdialog	94
7.10. Implementierung der process-Methode des Laplace-Operators	96
7.11. Implementierung der process-Methode des Sobel-Operators	97
7.12. Die options-Methode von ConnectedThresholdImageFilter	99
7.13. Implementierung der process-Methode des ConnectedThresholdImageFilters	100
7.14. Erweiternder Eintrag einer Konstanten in der Klasse <i>ImageProvider</i>	105
7.15. Erweiterung des Basiswerkzeugs	110
7.16. Eventkonstanten der Klasse EventConstants	110
7.17. Modifikation der SelectionToolFactory	111
7.18. Implementierung des ToolRoiHandler	112
C.1. <code>resource.properties</code>	XXVII
C.2. Erweiternder Eintrag einer Konstanten in der Klasse <i>ImageProvider</i>	XXVII
C.3. Verwendung des ImageProviders mit dem IResourcePool	XXVIII
D.1. Implementierung eines HalloWelt-Plug-ins	XXXV

Teil III.

Anhang

A. Darstellung der DICOM-Elemente im Speicher

A.1. Explizite VR mit [OB | OW | OF | SQ | UT | UN]

Bei expliziter VR-Struktur besteht das Element aus vier konsekutiven Feldern. Ist die VR vom Typ OB, OW, OF, SQ, UT oder UN wird das Datenelement wie in Tabelle A.1 im Speicher abgelegt. Die reservierten 2 Byte im VR-Teil sind für zukünftige Weiterentwicklungen des DICOM-Standards.[Nat11b, 7.1.2]

A.2. Explizite VR

Diese Darstellung wird gewählt, wenn VR *nicht* vom Typ OB, OW, OF, SQ, UT oder UN ist. Der Unterschied besteht im Feld „Value Length“. Bei der Form von Tabelle A.1 ist dieses Feld 32 Bit lang. Hier beträgt es lediglich 16 Bit [Nat11b, 7.1.2]. Der Grund liegt am erhöhten Speicherbedarf von A.1, da die Länge des Wertes eine undefinierte Länge haben kann.

A.3. Implizite VR

Bei einer impliziten VR Darstellung besteht das Datenelement aus den drei konsekutiven Feldern Tag, Value Length und dem Wert selbst [Nat11b, 7.1.3].

A

Tag		VR		Value Length	Value
Group# 16-bit unsigned integer	Element# 16-bit unsigned integer	VR 2-byte character String [OB — OW — OF — SQ — UT — UN]	Reservierter Bereich	32-bit un- signed in- teger	Gerade Anzahl an Byte. Enthält den Wert des Datenelements. Kodierung abhängig von VT-Typ und Transfersyntax. Wenn die Länge nicht definiert ist, wird diese auf „Se- quence Delimitation“ limitiert.
2 Byte	2 Byte	2 Byte	2 Byte	4 Byte	Anzahl an Byte entsprechend der „Value Length“ wenn von expliziter Länge

Tabelle A.1.: Darstellung des Datenelements im Speicher wenn VR vom Typ OB, OW, OF, SQ, UT oder UN

Tag	VR 2	Value Length	Value 4
Group# 16-bit unsigned integer	Element# 16-bit unsigned integer	VR 2-byte character String 2	16-bit unsi- gned inte- ger Gerade Anzahl an Byte. Enthält den Wert des Datenelements. Ko- dierung abhängig von VT-Typ und Transfersyntax.
2 Byte	2 Byte	2 Byte	2 Byte „Value Length“ Byte

Tabelle A.2.: Darstellung des Datenelements für alle anderen VR-Typen

Tag		Value	Length	Value
Group# unsigned integer	16-bit bit integer	Element# 16- bit unsigned integer	32-bit unsigned integer	Gerade Anzahl an Byte. Enthält den Wert des Datenelements. Kodierung abhängig von VT-Typ spezifiziert in [Nat11c] und Transfersyntax. Wenn die Länge nicht definiert ist wird diese auf „Sequence Delimitation“ limitiert.
2 Byte	2 Byte	2 Byte	2 Byte	„Value Length“ Byte oder undefinierte Länge

Tabelle A.3.: Darstellung des Datenelements für implizite VR.

A

B. Installation der Eclipse e4 Umgebung

B.1. Eclipse

Voraussetzung für Eclipse ist eine bereits installierte Java Virtual Machine. Werden keine Werkzeuge aus dem Java Development Kit benötigt, reicht eine Java Runtime Environment aus. Für die Entwicklung von jMediKit wurde Java 7 verwendet.

Grundlage für die Plug-in und Rich Client Entwicklung ist eine Eclipse-Installation ab Version 4. Unter <http://www.eclipse.org/downloads/> kann der aktuelle *Eclipse Standard Client* für ein beliebiges Betriebssystem bezogen werden. Bei der Wahl zwischen 32- und 64-Bit muss die Version mit der installierten Java-Variante übereinstimmen, da sonst die Native Libraries nicht geladen werden können. Nach dem Entpacken des Zip-Archivs kann der Client über *eclipse.exe* gestartet werden. Abbildung B.1 zeigt das Programmfenster von Eclipse nach dem ersten Ausführen.

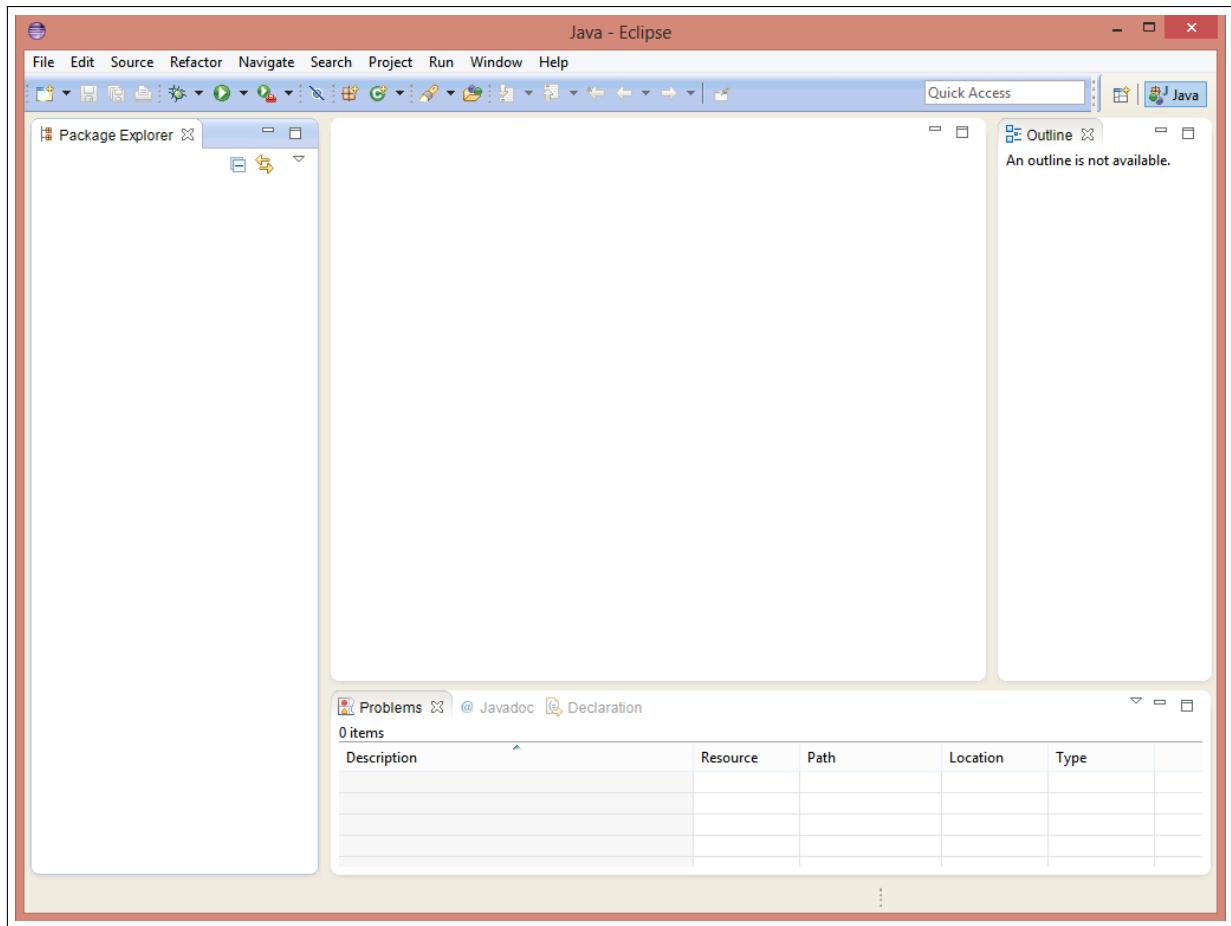


Abbildung B.1.: Standardinstallation eines 32-Bit Eclipse unter Windows 8

B.2. Eclipse e4 Tools

Eine weitere Voraussetzung ist eine Installation der e4 Tools. Das e4 Projekt ist unter <http://download.eclipse.org/e4/downloads/> mit dem Punkt *Stable Build* zu finden (Abbildung B.2).

Installation der Eclipse e4 Umgebung

The screenshot shows the 'eclipse e4 project downloads' page. At the top right is the Eclipse incubation logo. Below it is a section titled 'Latest Downloads' with a blue header bar. It contains a brief description and links to the Eclipse Foundation Software User Agreement and the e4 wiki. A note says 'All downloads are provided under the terms and conditions of the Eclipse Foundation Software User Agreement unless otherwise specified.' Below this, there's a link to 'Other eclipse.org project downloads'. The page is organized into sections for different build types:

- Checkpoint Builds:** Includes builds 0.14, 0.13, 0.12, 0.11, 0.111, 0.11, and 0.10. Each entry shows the build name and its corresponding build date.
- Stable Builds:** Contains build 0.15, which is highlighted with a red border. This entry also shows the build name and build date.
- Integration Builds:** Includes builds 001401023-2200, 001401022-2200, and 001401022-1619. Each entry shows the build name and build date.
- Maintenance Builds:** No entries are listed.

Abbildung B.2.: e4 Projekt Builds

Nach einem Klick auf den aktuellen Build öffnet sich die Seite mit dem Link zum Repository wie in Abbildung B.3 rot markiert.

Dieser Link (<http://download.eclipse.org/e4/downloads/drops/S-0.15-201401152200/repository> - Stand 24.01.2014) muss nun in die Zwischenablage kopiert werden.

B

Installation der Eclipse e4 Umgebung



Abbildung B.3.: e4 Repository Link

Nachdem der Link kopiert wurde, kann Eclipse geöffnet werden. Nach einem Klick auf den Menüpunkt *Hilfe* → *Install New Software* öffnet sich ein Fenster mit dem Titel „Available Software“ wie in Abbildung B.4 zu sehen ist.

Installation der Eclipse e4 Umgebung

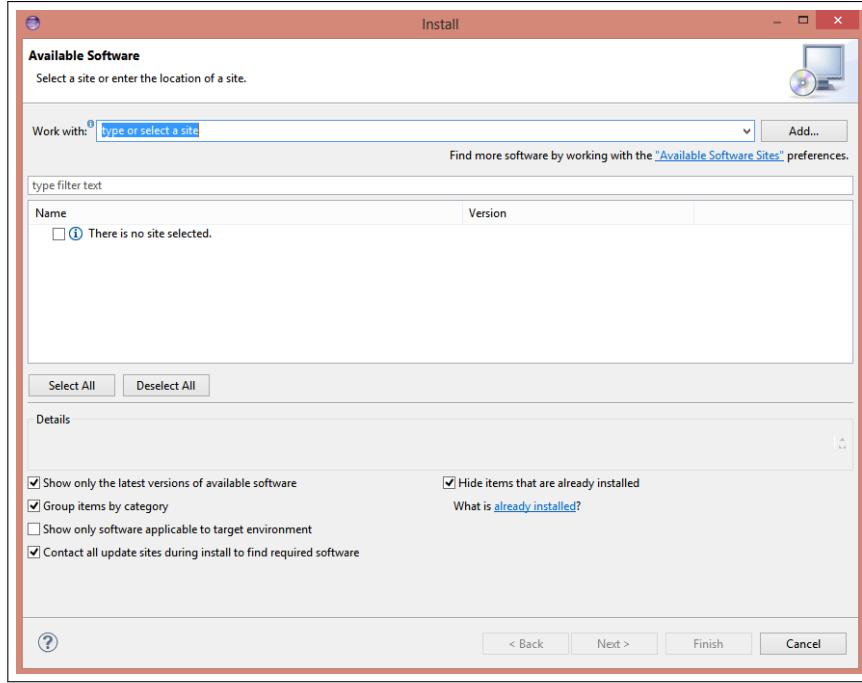


Abbildung B.4.: Installation neuer Software unter Eclipse

Mit dem Button *Add* wird nun der zuvor kopierte Link als Repository angegeben. Der *Name* kann frei vergeben werden. Unter *Location* muss man den Link zum Repository eintragen. Danach mittels *OK* die Aktion bestätigen.

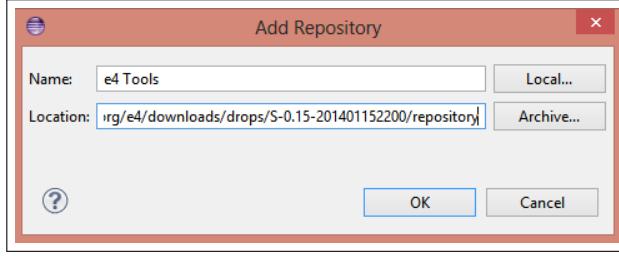


Abbildung B.5.: Angabe des Repository

Nach einem korrekten Eintrag wird die Liste der verfügbaren Software, wie in Abbildung B.6 zu sehen ist, aktualisiert. Hier muss das Paket *Eclipse 4 core tools* samt Unterpakete ausgewählt werden. Mit einem Klick auf *Next* startet die Installationsroutine. Hierbei ist den Anweisungen auf dem Bildschirm zu folgen. Nach einer erfolgreichen Installation muss Eclipse neu gestartet werden.

B

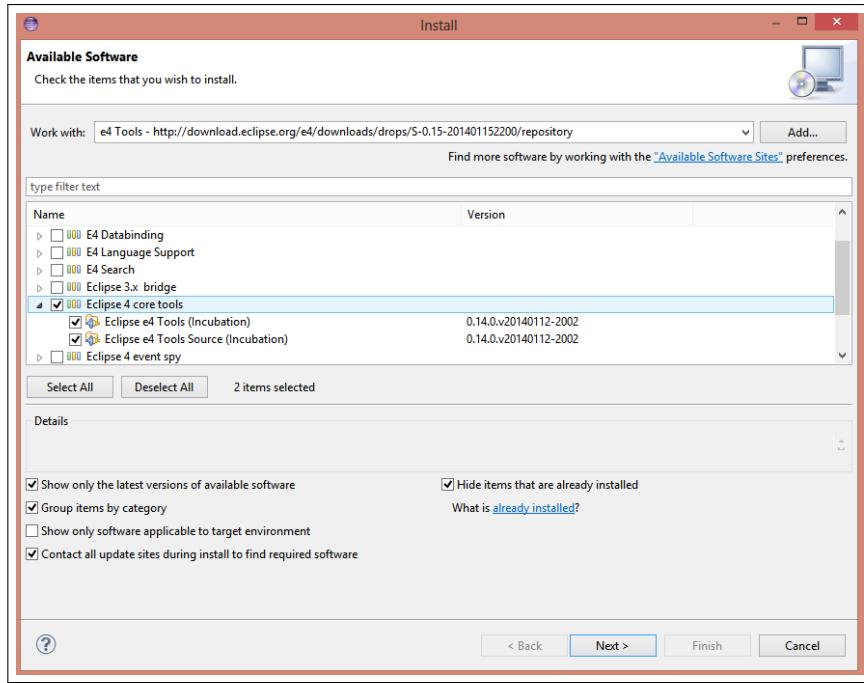


Abbildung B.6.: Auswahl der e4 Tools zur Installation

B.3. Import der jMediKit Projektdateien

Die Projektdaten befinden sich auf dem beiliegenden Datenträger. Das Wurzelverzeichnis kann als Eclipse Workspace benutzt werden. Dabei ist darauf zu achten, dass der Ordner `./metadata/.plugins` leer ist. Sollten sich Daten darin befinden, können diese gelöscht werden. Eclipse erstellt bei Bedarf die Plug-in-Daten neu. Der Workspace besteht aus den drei Projekten `org.jmedikit.product`, `org.jmedikit.feature` und `org.jmedikit.plugin`. Beim ersten Öffnen ist der Package Explorer leer und die Projekte müssen importiert werden. Nach einem Klick auf *File* → *Import* erscheint ein Dialog wie in Abbildung B.7(a) zu sehen ist. Bei der Auswahl muss der Punkt unter *General* → *Existing Projects into Workspace* markiert und mit *Next* bestätigt werden. Im folgenden Fenster (Abbildung B.7(b)) muss unter ausgewähltem *Select Root Directory* unter *Browse* das Wurzelverzeichnis von jMediKit angegeben werden. Danach können die verfügbaren Projekte hinzugefügt und mittels Klick auf *Finish* in den Eclipse Workspace importiert werden.

Installation der Eclipse e4 Umgebung

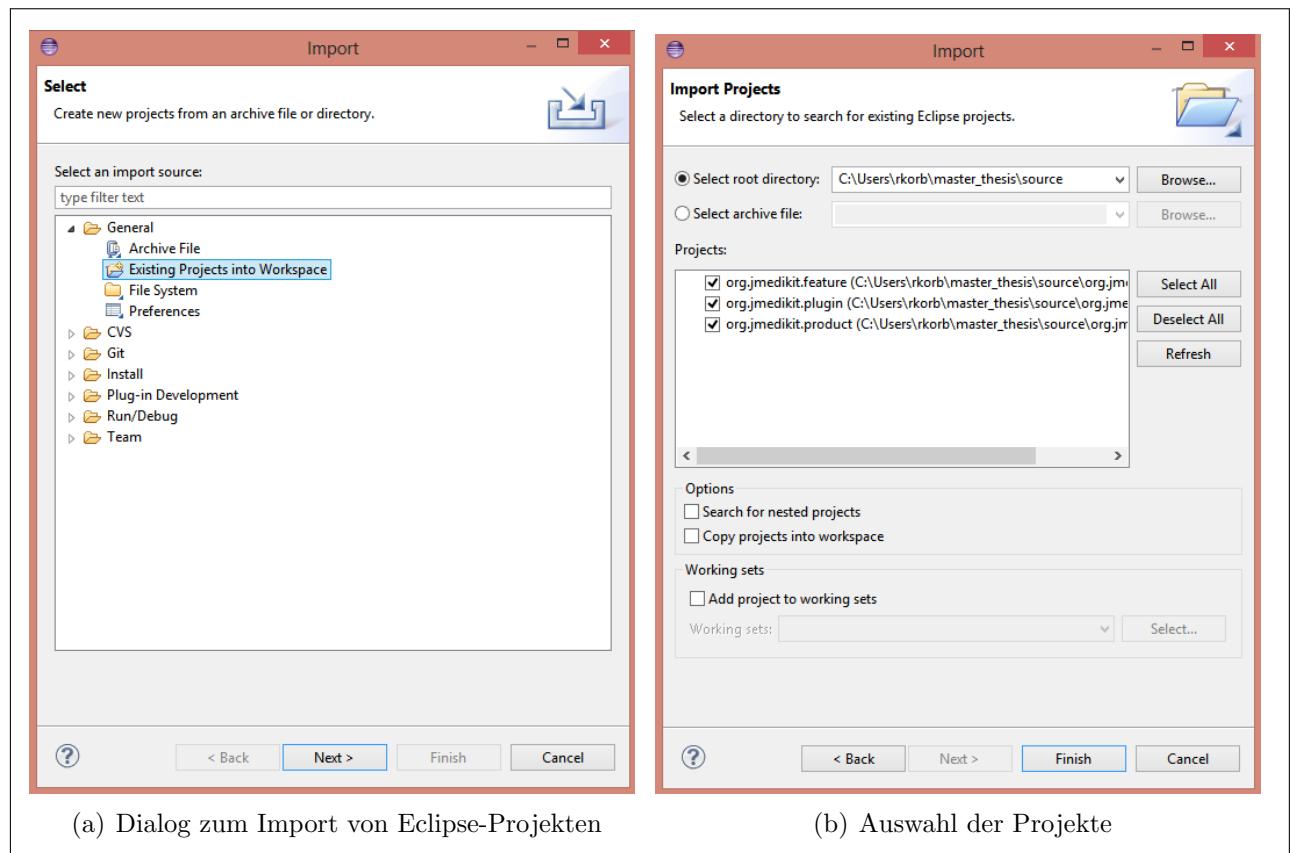


Abbildung B.7.: Vorgang zum Importieren bereits bestehender Projekte

War der Importvorgang erfolgreich, sind die drei Projekte

- *org.jmedikit.product*
- *org.jmedikit.feature*
- *org.jmedikit.plugin*

wie in Abbildung B.8 zu sehen, im Package Explorer vorhanden.

B

Installation der Eclipse e4 Umgebung

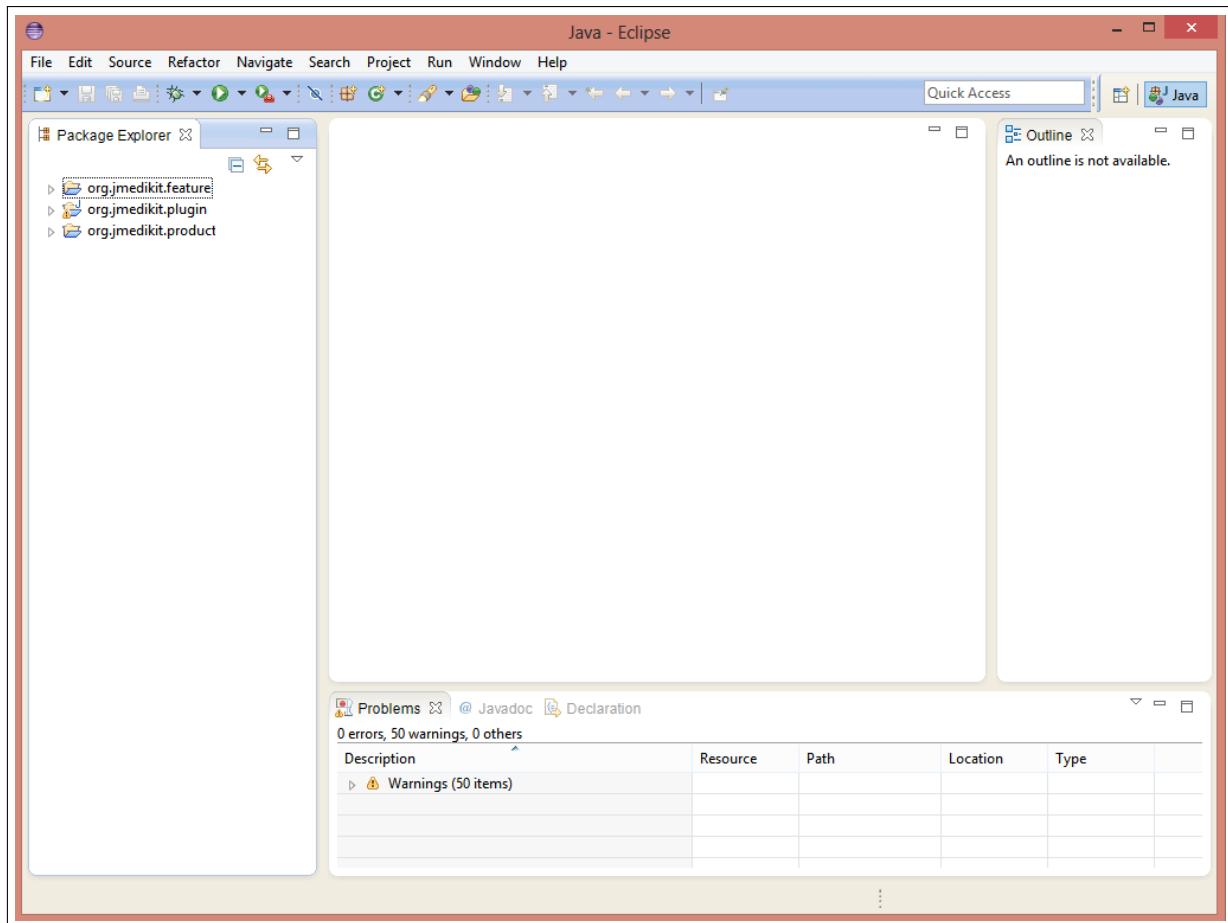


Abbildung B.8.: Der Package Explorer nach dem Importvorgang

Im Projekt *org.jmedikit.product* befindet sich die Datei *jmedikit.product*. Nach einem Doppelklick wird die Datei im Eclipse-Editor geöffnet und zeigt die Grundlegende Anwendungsdefinition von jMediKit und den zugehörigen Einstellungen. Unter dem Tab *Overview* im Bereich *Testing* kann jMediKit mit einem Klick auf *Launch an Eclipse application* gestartet werden. Abbildung B.9 hebt die Schaltfläche hervor. Bei einem ersten Start wird jMediKit mit einer Fehlermeldung geschlossen, da von Eclipse noch nicht alle zum Start notwendigen Plug-ins geladen wurden, welche von jMediKit benötigt werden.

Installation der Eclipse e4 Umgebung

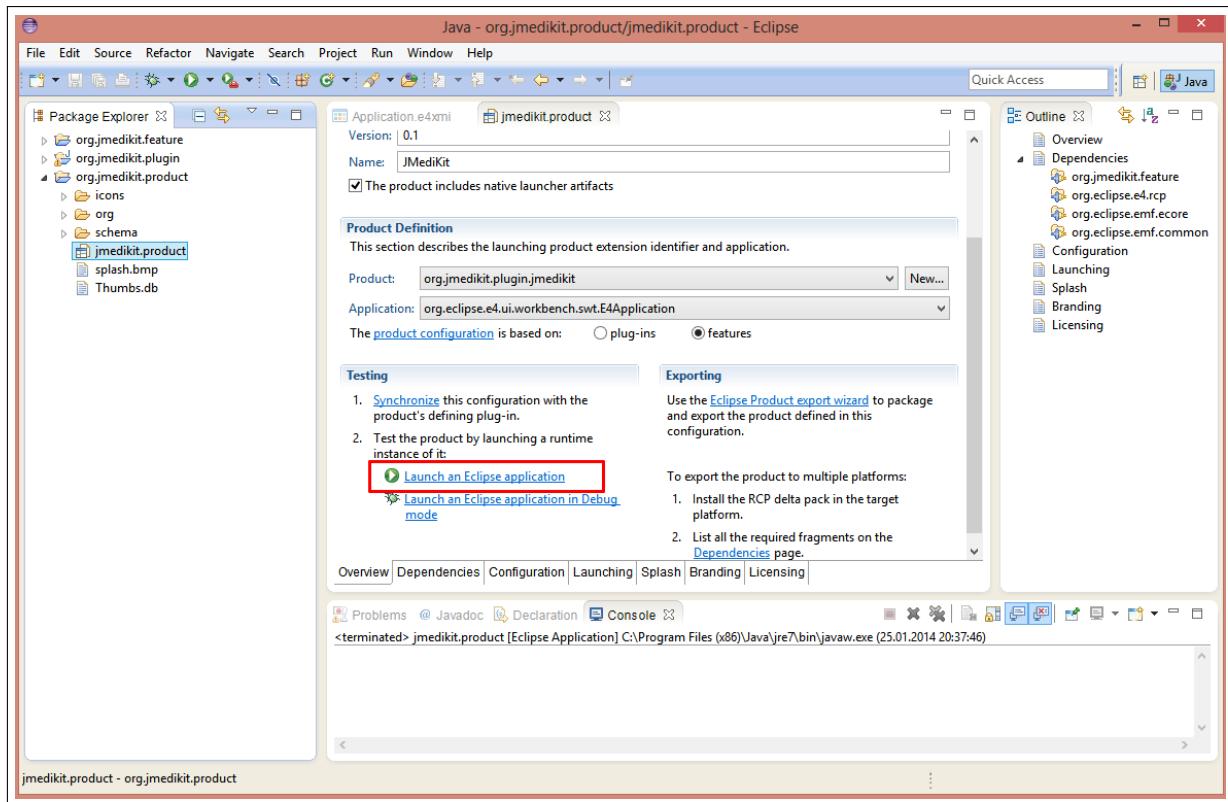


Abbildung B.9.: Erster Start der Anwendung über Eclipse

Unter *Run → Run Configuration* können die Plug-ins zur Verfügung gestellt werden. Abbildung B.10 zeigt das Konfigurationsfenster. Im linken Teil muss die Product-Datei *jmedikit.product* ausgewählt sein. Unter dem Tab *Plug-ins* befindet sich auf der rechten Seite die Schaltfläche *Add Required Plug-ins*. Nach einem Klick auf *Apply* gefolgt von *Run* kann jMediKit gestartet werden.

B

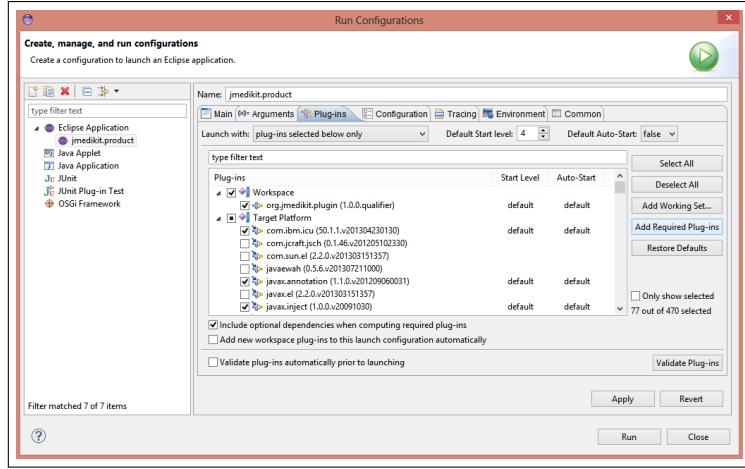


Abbildung B.10.: Konfigurationsfenster zu Anwendungseinstellungen und Anwendungsstart

B.4. Installation des Java Advanced Imaging Image I/O Tools

jMediKit benutzt die externe Bibliothek „dcm4che“, um DICOM-Objekte zu laden und die Bilder auf der Zeichenfläche anzuzeigen. „dcm4che“ selbst benötigt das Paket „Java Advanced Imaging Image I/O Tools“. Ist das Tool noch nicht installiert, kann unter der Adresse <http://download.java.net/media/jai-imageio/builds/release/1.1/>, entsprechend dem Betriebssystem und der Installationsart, die richtige Version zum Installieren gewählt werden. Ob der JDK- oder JRE-Installationstyp verwendet werden soll, hängt von der in Eclipse verwendeten Java Runtime Environment ab. Ist eine eigenständige JRE installiert, hat diese beispielsweise den Pfad `/JAVA_HOME/jre7/` und es wird die JRE-Variante installiert. Hat die JRE zum Beispiel einen Pfad der Form `/JAVA_HOME/jdk1.7.0_45/jre` ist die Java Runtime Environment des Java Development Kits im Einsatz. Um zu prüfen, welche Java Runtime Environment von Eclipse verwendet wird, muss der Einstellungsdialog unter *Window → Preferences* geöffnet und der Punkt *Java → Installed JREs* gewählt werden. Abbildung B.11 zeigt den Dialog mit einer JRE und dem Pfad `C:\Program Files (x86)\Java\jre7`.

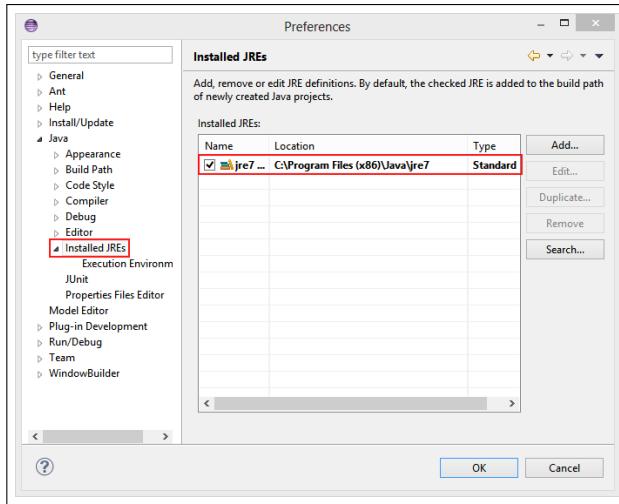


Abbildung B.11.: Die von Eclipse verwendeten Java Runtime Environments

Ob die Image I/O Tools installiert sind, kann über den Ordner der externen Bibliotheken der Java-Installation geprüft werden. Sind im Ordner */PATH_TO_JRE/lib/ext/* die .jar-Pakete *jai_imageio* und *clibwrapper_jiio* vorhanden, kann „dcm4che“ auf die Bibliothek zugreifen.

Die Java Advanced Imaging Image I/O Tools sind nur in einer 32-Bit Version verfügbar. Bei einer Installation werden native Bibliotheken in die JRE integriert. Werden die Tools mit einer 64-Bit Java und Eclipse Version eingesetzt, sind diese nativen Bibliotheken nicht verfügbar. Eine Installation für eine 64-Bit JRE erfolgt über das Verschieben der .jar-Dateien *jai_imageio* und *clibwrapper_jiio* in das Verzeichnis */PATH_TO_x64JRE/lib/ext/* der 64-Bit JRE.

Durch das Fehlen des nativen Codes einer 64-Bit Installation können nicht alle DICOM-Bilder geladen werden, da zum Beispiel eine Komprimierung im Format *JPEG2000* davon abhängig ist.

B

B.5. Erstellung eines neuen Builds von jMediKit

Um einen neuen Build von jMediKit zu veröffentlichen, muss die Datei *jmedikit.product* im Projekt *org.jmedikit.product* mit einem Doppelklick geöffnet werden. Darauf öffnet sich der *Overview* zum Produkt, wie in Abbildung B.12 zu sehen ist. Am oberen rechten Rand des *Overviews* sind vier Buttons angeordnet. Hier muss der dritte Button names *Export an Eclipse product* (in der Abbildung B.12 rot markiert) gedrückt werden.

Installation der Eclipse e4 Umgebung

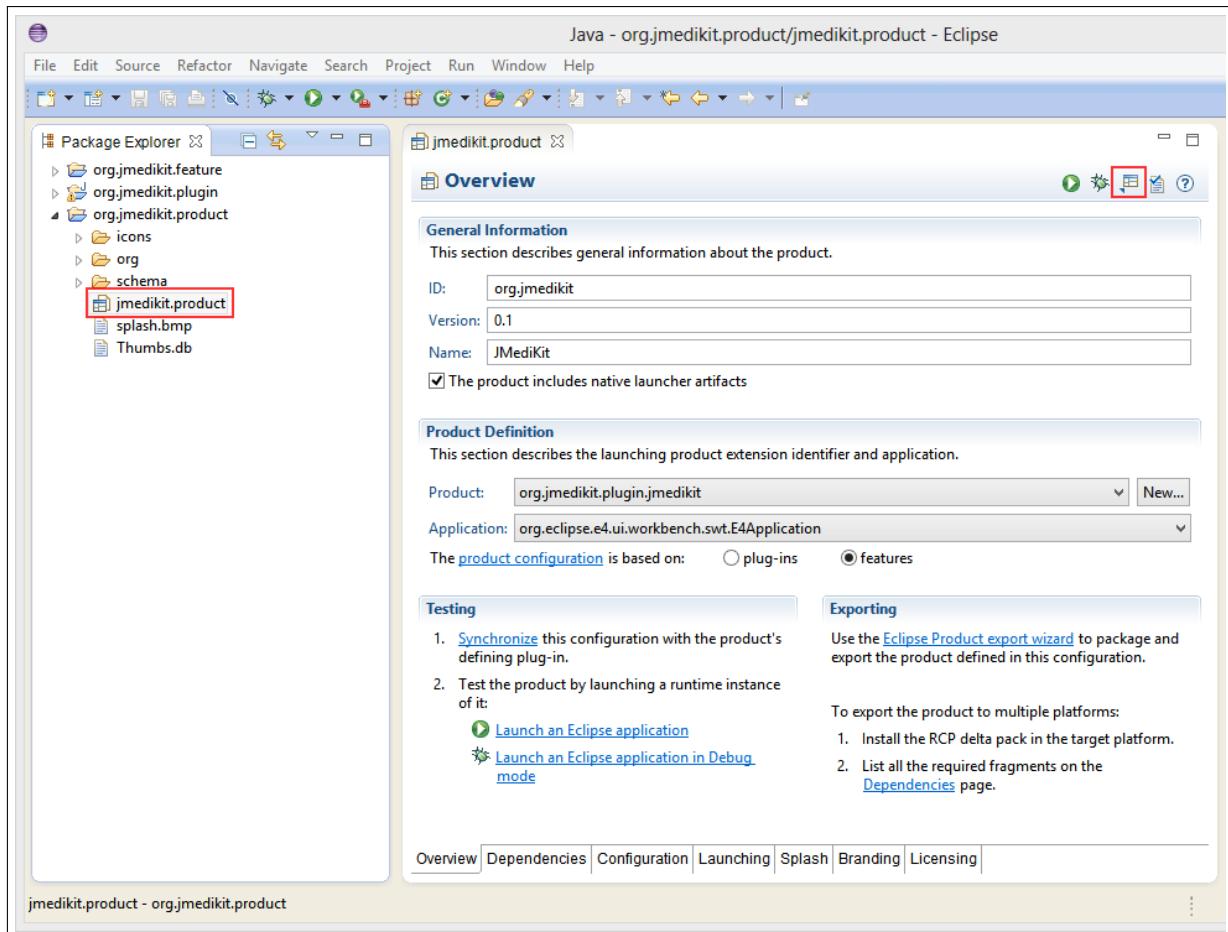


Abbildung B.12.: Mit jmedikit.product den Build erstellen

Es öffnet sich ein Fenster mit diversen Optionen (Abbildung B.13). Als *Configuration* muss die *jmedikit.product* gewählt werden. Unter *Root Directory* wird das Wurzelverzeichnis von jMediKit bestimmt. Die Option *Directory* ist das Exportverzeichnis, in dem die Anwendung nach dem Build zu finden ist. Die weiteren Optionen entsprechen der Abbildung B.13. Mit *Finish* wird der Build-Prozess gestartet.

Installation der Eclipse e4 Umgebung

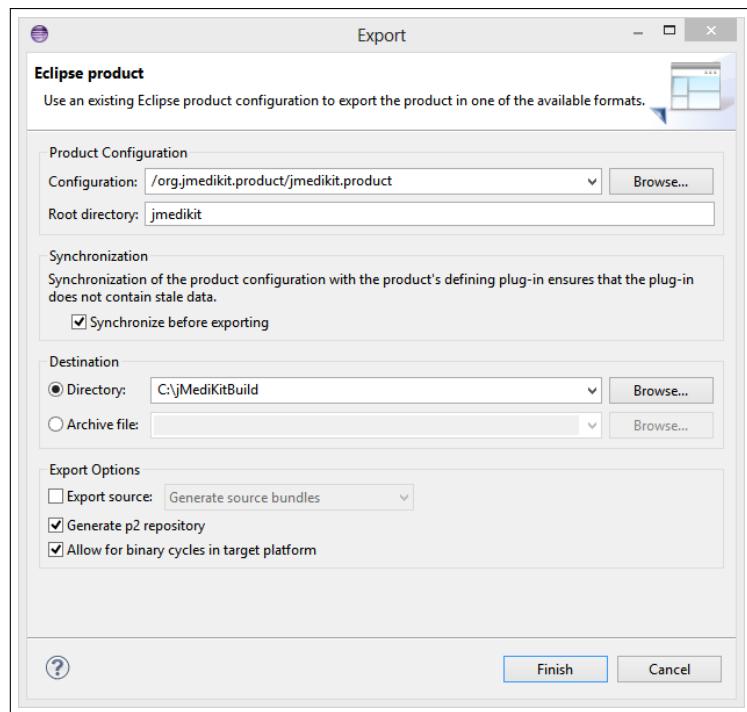


Abbildung B.13.: Optionen zur Erstellung des Builds

B

C. Iconverwaltung innerhalb der Anwendung

Die Icons der Anwendung werden mit Hilfe eines OSGi-Services verwaltet. Eclipse definiert das Interface *IResourceProviderService*, mit dessen Hilfe die Bilddaten organisiert werden können. Die Metadaten des Services befinden sich im Projekt *org.jmedikit.plugin* unter *OSGI-INF*. Hier wird der spezifische Service bekannt gemacht und die implementierende Klasse festgelegt. Zusätzlich wird die Datei *OSGI-INF/resource.properties* definiert, die alle Pfadangaben zu den Bilddateien enthält.

Um neue Bilder und Icons hinzuzufügen muss zuerst die Datei *resource.properties* um einen Bezeichner und die Pfadangabe erweitert werden. Ein neuer Eintrag kann an beliebiger Stelle, wie in Listing C.1, hinzugefügt werden.

```
1 | NEW_ICON_FILE=icons/mynewicon.png
```

Listing C.1: resource.properties

Der *ImageProvider* erweitert die von Eclipse bereitgestellte Klasse *BasicImageProvider*, die das Interface des Services implementiert. *ImageProvider* muss nun um den Bezeichner des neuen Icons mit einer Konstanten erweitert werden (Listing C.2).

```
1 | public final static String NEW_ICON_FILE = "NEW_ICON_FILE";
```

Listing C.2: Erweiternder Eintrag einer Konstanten in der Klasse *ImageProvider*

Die Bilddatei oder das Icon wird nach einem erneuten Build des Services verfügbar. Dieser kann zum Beispiel durch ein erneutes Speichern der Datei *org.jmedikit.plugin/META-INF/MANIFEST.MF* angestoßen werden. Listing C.3 zeigt die Verwendung der neuen Bilder und Icons. Innerhalb der Klassen, die Strukturen des Application Models implementieren, kann mit der in Eclipse e4 verfügbaren *Dependency Injection* gearbeitet werden¹.

¹Ein umfassender Artikel zur Dependency Injection ist unter [Vog13c] verfügbar.

Existiert innerhalb des Lebenszyklus der Anwendung ein *IResourcePool*-Objekt, kann es mit Hilfe der Annotation *@Inject* in die Klasse injiziert werden. Mit dieser Instanz wird innerhalb der Anwendung ein *SWT.Image* mit der Methode *getImageUnchecked* instantiiert.

```
1 //Erzeugen eine SWT.Image aus dem ImageProvider
2 public class PartClass{
3
4     /**
5      * Den IResourcePool mittels
6      * Dependency Injection verfuegbar machen
7      */
8     @Inject
9     IResourcePool provider;
10
11    public void someMethod(){
12        Image icon = provider.getImageUnchecked(
13            ImageProvider.NEW_ICON_FILE
14        );
15    }
16 }
```

Listing C.3: Verwendung des ImageProviders mit dem IResourcePool

D. Die Verwendung von Eclipse zur Plug-in-Entwicklung

Dieses Kapitel beschreibt das Vorbereiten eines Projekts zur Entwicklung eines Plug-ins für jMediKit. Voraussetzung für diese Anleitung ist Eclipse ab Version 4. Zusätzlich wird entweder ein Build von jMediKit oder der Quelltext und die im Plug-in verwendeten externen Bibliotheken benötigt.

Nachdem Eclipse geöffnet wurde, kann über *File → New → Java Project* ein neues Java Projekt angelegt werden. Abbildung D.1 zeigt den entsprechenden Dialog und die zu wählenden Eigenschaften. Es muss nur der Plug-in Name bestimmt werden. Die restlichen Einstellungen entsprechen den Standardwerten, können aber nach Bedarf angepasst werden.

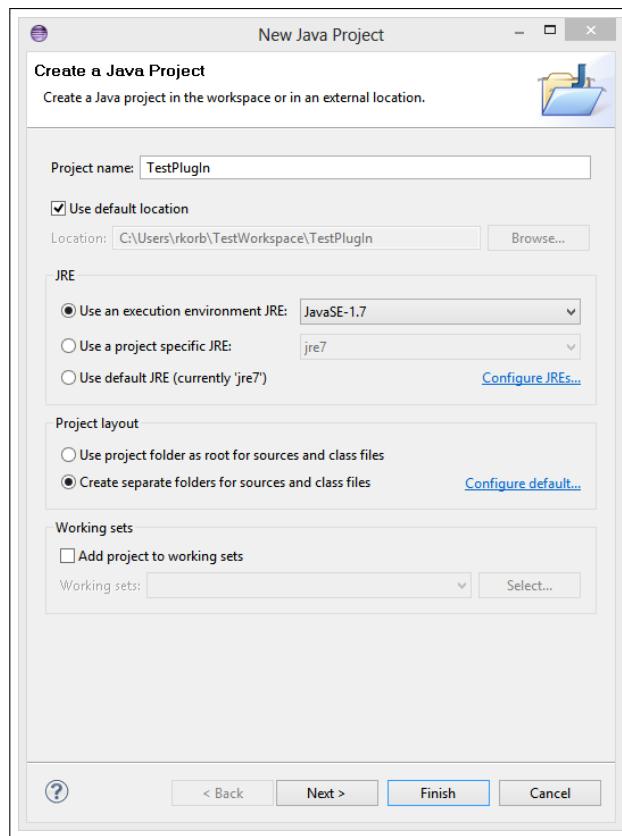


Abbildung D.1.: Dialog zum Anlegen eines neuen Javaprojekts

Mit einem Klick auf *Next* wird man zu den nächsten Projektoptionen, wie in Abbildung D.2 zu sehen, geleitet. Hier kann unter *Default Output Folder* der Ordner entsprechend der Plug-in-Konventionen für die *.class* Dateien gesetzt werden. Der Plug-in-Name im Beispiel lautet *TestPlugIn*. Entsprechend ist unter *Default Output Folder* der Wert *TestPlugIn/_TestPlugIn* einzutragen, wobei *TestPlugIn* den Projektordner darstellt und die Binärdaten unter *_TestPlugIn* gespeichert werden. Der Name des Output-Ordners für die Binärdaten kann später manuell in Eclipse mit einem *Rechtsklick auf das Projekt → Build Path → Configure Build Path → Unter dem Tab Source → Default Output Folder* ausgelesen oder geändert werden.

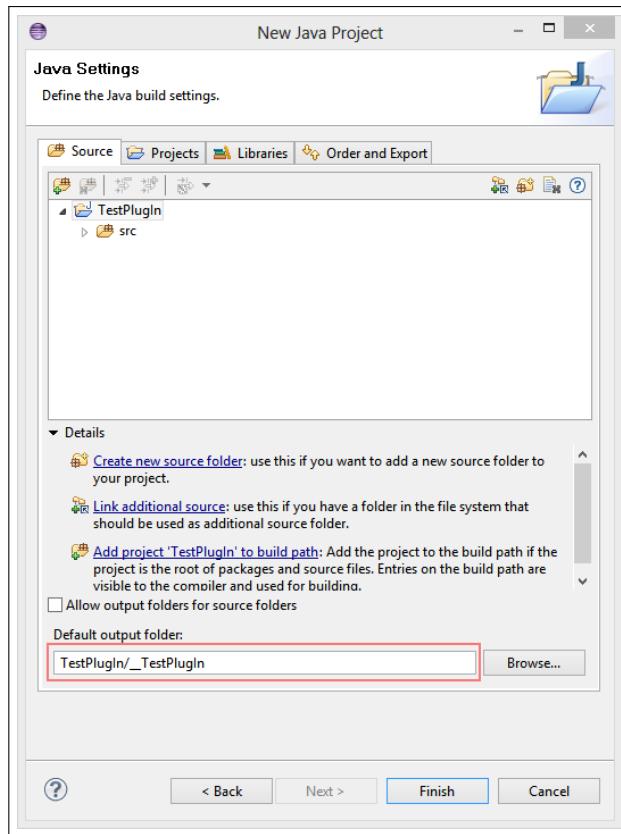


Abbildung D.2.: Auswahl des Output-Ordners

Im nächsten Schritt müssen die externen Bibliotheken dem Projekt hinzugefügt werden. Werden vom User selbst keine speziellen Bibliotheken verwendet, reicht das Inkludieren der jMediKit-Bibliothek und dem SimpleITK-Projekt.

Dazu muss im aktuellen Dialogfenster der Reiter *Libraries* ausgewählt werden. Wird mit einem jMediKit-Build gearbeitet, kann die Bibliothek mit dem Button *Add External Jar* hinzugefügt werden. Nach einem Klick öffnet sich ein Dateidialog. Der Pfad zu der jMediKit-Jar befindet sich unter

jMedikit-Installation/jmedikit/plugins/org.jmedikit.plugin_versionsnummer, wobei *versionsnummer* für die Versionsnummer des aktuellen Builds steht(zum Beispiel *org.jmedikit.plugin_1.0.0.201401220023*). Wird mit dem Quelltext von jMediKit gearbeitet, können die Projektdateien über den Button *Add External Class Folder* hinzugefügt werden. Diese liegen im Ordner *bin* im Projekt *org.jmedikit.plugin*. Der Vollständige Pfad ist */WORKSPACE/org.jmedikit.plugin/bin/*.

Um die SimpleITK.jar einzubinden gibt es zwei Möglichkeiten. Wird mit dem Build gear-

beitet, kann die jMediKit-Jar mit einem Zip-Programm geöffnet und SimpleITK aus dem Ordner *lib* entpackt werden¹. Bei der Verwendung der Quelldateien liegt die Bibliothek von SimpleITK im Projekt *org.jmedikit.plugin* ebenfalls im Ordner *lib* mit dem vollständigen Pfad */WORKSPACE/org.jmedikit.plugin/lib/SimpleITK/*. Die Datei *simpleitk-0.x.x.jar* kann nun mit dem Button *Add External Jar* hinzugefügt werden.

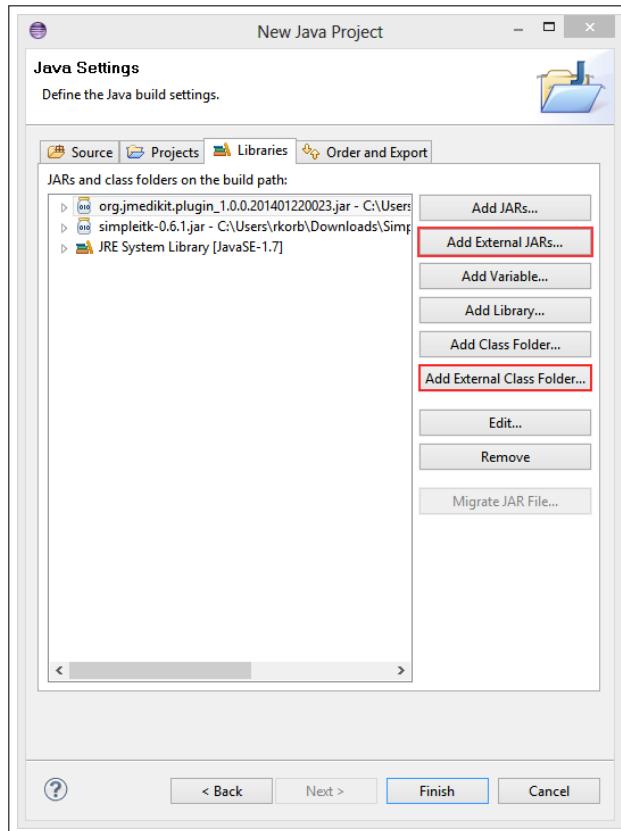


Abbildung D.3.: Auswahl der externen Bibliotheken

Mit einem Klick auf *Finish* werden die Einstellungen übernommen und das Projekt angelegt. Wurden alle Schritte korrekt ausgeführt, hat das Plug-in-Projekt eine Struktur wie in Abbildung D.4.

¹Ist kein Programm zum Entpacken von jar-Dateien vorhanden, kann SimpleITK direkt von <http://www.simpleitk.org/SimpleITK/resources/software.html> bezogen werden

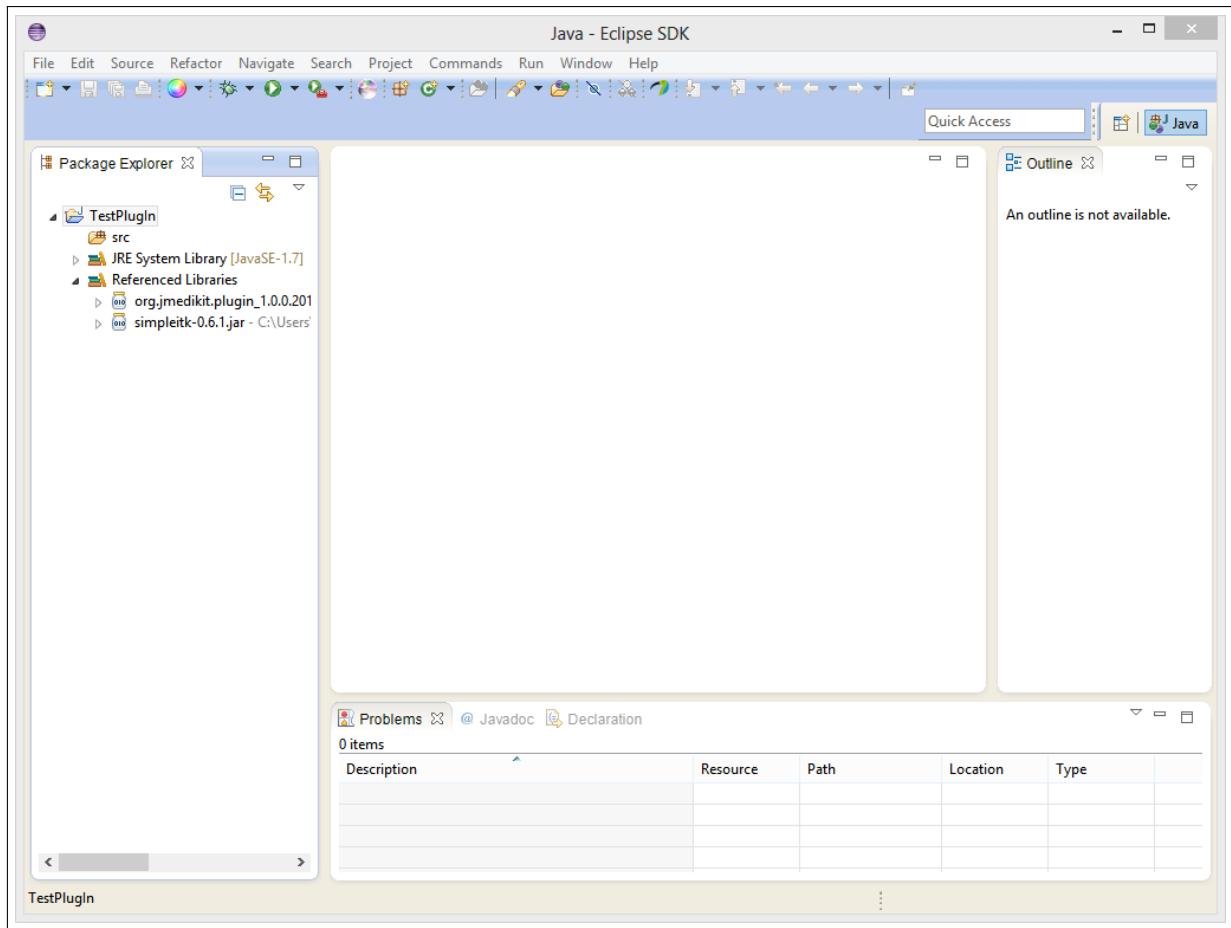


Abbildung D.4.: Der Eclipse Workspace nach Anlegen des Projekts

Mit einem Rechtsklick auf den Ordner *src* im Project Explorer, gefolgt von *New → Class*, kann die Hauptklasse des Plug-ins angelegt werden. Entsprechend der Plug-in-Konventionen wird der Name *_TestPlugIn* mit einem Klick auf *Finish* erstellt.

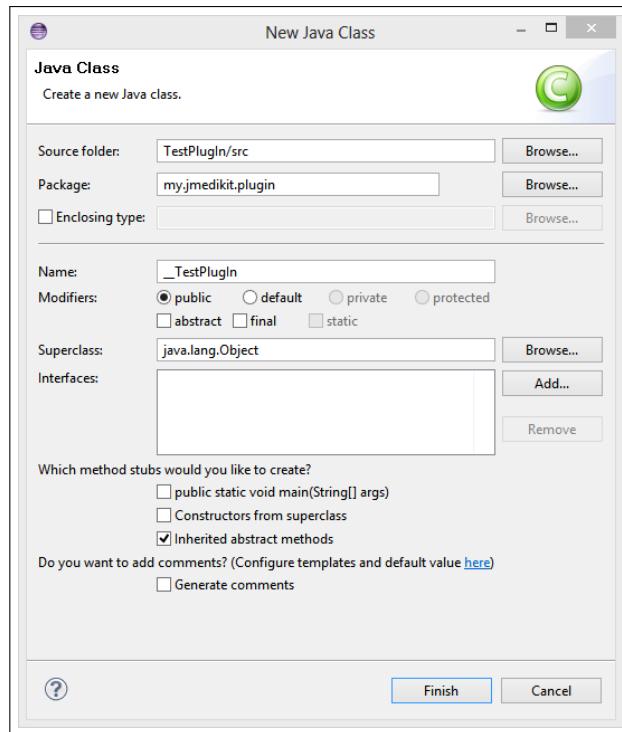


Abbildung D.5.: Anlegen der Hauptklasse des Plug-ins

Nach dem Erstellen der Klasse kann mit dem Entwickeln des Plug-ins begonnen werden.
Ein HalloWelt-Plug-in hat folgende Struktur:

```
1 | import org.jmedikit.lib.core.APlugIn2D;
2 | import org.jmedikit.lib.image.AImage;
3 |
4 |
5 | public class __TestPlugIn extends APlugIn2D{
6 |
7 |     @Override
8 |     public AImage process(AImage img) {
9 |         System.out.println("Hello_World");
10 |         return img;
11 |     }
12 |
13 |     @Override
14 |     protected int options() {
15 |         setOutput("pathToOutputFile");
16 |         return 0;
17 |     }
18 |
19 | }
```

Listing D.1: Implementierung eines HalloWelt-Plug-ins