FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

ActiveDICOM – Enhancing Static Medical Images with Interaction

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Florian Mistelbauer
Matrikelnummer 0226239

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr.techn. Gabriel Mistelbauer BSc

Wien, 26.02.2017

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

ActiveDICOM – Enhancing Static Medical Images with Interaction

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Florian Mistelbauer
Registration Number 0226239

to the Faculty of Informatics
at the TU Wien, Austria

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing. Dr.techn. Gabriel Mistelbauer BSc

Vienna, 26.02.2017

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Florian Mistelbauer
Vorgartenstraße 93/16 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26.02.2017

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Digital **I**maging and **Co**mmunications in **M**edicine (DICOM) is a well-establish standard in medical imaging, consisting not only of image data, but sensitive data such as patient and examination information. Although having a large variety of advanced rendering techniques available, DICOM images are nowadays still generated and sent to the **P**icture **A**rchiving and **C**ommunication **S**ystem (PACS). These images are then fetched by the medical doctor from a workstation and used for medical reporting. The user has no other possibilities than being able to change the windowing function for displaying the DICOM images. If a certain region is of special interest, either images of the whole data set are generated or have to be specifically requested. Both approaches consume a considerable amount of time. First, one has to inspect the whole data and has to constantly focus on a particular region without being distracted. Secondly, the image generation on demand remains pending until done by the responsible assistant.

Despite supporting a broad range of features and being widely applied, DICOM images remain static. Currently, and to the best of our knowledge, there are no means to enhance these static images with interactive capabilities. Providing the opportunity to interact with an image would, inevitably, enhance the medical reporting procedure. In this thesis, we propose a visualization mapping language, called **A**ctive **DIC**OM Scrip**t** (ADICT), which enhances conventional DICOM images with interactive elements. This language acts as an abstraction layer to provide a transparent interface between heterogeneous data, interaction, and visualization. Since these images are becoming active, we refer to them as Active Digital Imaging and Communications in Medicine (ActiveDICOM). Within this thesis we describe how interactions will be encoded into the image while retaining compatibility to common DICOM viewers. Furthermore with the help of ActiveDICOM we can even introduce **Visual Analytics** to DICOM. We demonstrate our interaction-to-visualization mapping on a client-server web-based viewer, as this would aid integration within the clinical environment and on mobile devices such as tablets or smart phones.

Kurzfassung

In der medizinischen Bildgebung ist **D**igital **I**maging and **Co**mmunications in **M**edicine (DICOM) ein sehr weit verbreiteter Standard. Dieser beschreibt nicht nur Bildformate, sondern ein ganzes Ökosystem inklusive Patienteninformationen und Netzwerkkommunikation. DICOM Bilder werden auf einem **P**icture **A**rchiving and **C**ommunication **S**ystem (PACS) gespeichert und von dort dem medizinischen Personal zur Verfügung gestellt.

Nach wie vor sind Interaktionen mit diesen Bildern bis auf das Verändern der Windowing–Function limitiert. Sind gewisse Bereiche einer Darstellung oder eines Bildes von besonderer Bedeutung, muss eine spezielle Serie angefordert und erzeugt werden. Dieser Prozess ist zeitaufwändig und relevante Informationen können beim Durchsuchen mehrerer Bilder aufgrund der Fülle an Informationen verloren gehen. DICOM Bilder sind immer noch statisch, obwohl eine beachtliche Menge an Merkmalen geboten wird. Gegenwärtig ist es nicht möglich, interaktive Elemente in DICOM Bilder zu integrieren.

In dieser Diplomarbeit haben wir eine Scriptsprache entwickelt mit deren Hilfe Interaktionen und zusätzliche Inhalte wie Bilder, Videos und Dokumente gemeinsam in ein DICOM integriert werden können. Mit Hilfe dieser Sprache, **A**ctive **DIC**OM Scrip**t** (ADICT), ist es möglich komplexe, interaktive Elemente in ein DICOM einzubetten. Dieser Ansatz entspricht dem Konzept “Visual Analytics”, welches große Datenmengen geordnet und struktuiert darstellt, um nicht durch einen Informationsüberfluss wichtige Elemente zu übersehen. Unser Ansatz, welcher Interaktionen, Daten und DICOM miteinander verbindet, nennt sich Active Digital Imaging and Communications in Medicine (ActiveDICOM). Da das DICOM–Format leicht erweiterbar ist, wird auch eine Abwärtskompatibilität erhalten. Implementiert wurde unser Ansatz als webbasierte Anwendung unter der Verwendung neuester Technologien wie WebGL und moderner Java Web Technologien, wie AJAX basierter Weboberflächen und moderner Datenbankanbindungen.

Contents

CHAPTER

1 Introduction

Within this thesis we touch topics dividable into three areas. The main focus is medical imaging followed by computer science and engineering, meaning implementation of a piece of software showcasing our theoretical approaches.

Medical Imaging. This presents the main theme/topic throughout the thesis. Imaging technologies play a very important role in the medical work flow regarding diagnosis. Ranging from births, cancer, and injuries imaging technologies are very important and with the advent of powerful hardware many previously computation intensive tasks may be even performed in real time.

Computer Science. The whole thesis is part of computer science, since we designed a computer language as the basis for our approach of creating our solution. Computer language design is one of the core elements of computer science which is why it is mentioned explicitly.

Engineering. Implementing a piece of software is mainly a task of engineering. Our proposed approach, ideas and artifacts were implemented using state-of-the-art technologies. Our implementation is built upon a Java based software stack. Developing a decently sized piece of software requires one to make certain architectural choices. The implementation of our software changed significantly as we gained more fruitful knowledge throughout the creation of this thesis.

Adhering to our division into three topics this thesis' structure tries to follow it as closely as possible. Chapter 2 gives an overview of today's situation and state-of-the-art in medical terminologies, work flow and technologies. Regardless of the domain one is talking or writing about, it is important what language is spoken in the mentioned domain. We introduce notions like medical image modality, PACS, DICOM and a few more when talking about the medical imaging domain. Domain experts, such as radiologists, expect proper use of their established domain specific language. Furthermore Chapter 2 gives an introduction to the

exchange format used in medical imaging known as DICOM. Focusing especially on the imaging aspect we might call medical images as DICOMs, although strictly speaking it is only a sub set of the specification and term. Chapter 2 also gives an overview of the scientific areas of interest to our thesis. Mentioned are web technologies, scripting languages and interactions. We even briefly touch visual analytics since big data and its presentation in a structured way is an emerging topic and a challenge.

Chapter 3 gives an overview of our approach to solve the thesis' problem statement. This chapter focuses on how we engineered our solution, what technologies were used and how images can be manipulated using our software. The software we developed provides a web-based interface and thus is accessed via browsers. Using a myriad set of heterogeneous technologies, modern and established, we provide a detailed overview of our software architecture.

Chapter 4 describes the second part of our contribution, the developed scripting language. This chapter gradually becomes more formal. Beginning with a description of what the language is capable of and what not by providing examples, we then give a formal specification of the language. This chapter should act as a language reference like used for real-world, current languages such as JavaScript. The JavaScript standard is called EcmaScript standard and also defines the language in a formal way.

Chapter 5 presents our results and gives real life examples of what is possible with our approach. Chapter 6 acts as the conclusion to our thesis. It presents a future outlook and gives a summary of our presented approach, the implementation and the language.

CHAPTER

2 Related Work

2.1 Motivation

The standard for handling information and data in medical imaging is **D**igital **I**maging and **Co**mmunications in **M**edicine (DICOM) [55]. Although the DICOM standard describes many aspects such as network protocols, we are referring just to the description of medical images. Throughout this thesis we use the terms *image* and *DICOM image* interchangeably. As a widely adopted standard, DICOM images are supported by many applications and devices. Whenever medical data is acquired from a medical imaging source, such as computed tomography, the imaging data is stored in the DICOM format and sent to a server for storage, usually a **P**icture **A**rchiving and **C**ommunication **S**ystem (PACS). Once the data of a study is completely acquired, it has to be processed, usually by radiology assistants in their daily clinical routine work. After this processing, potentially a specific region of the data is segmented and tailored visualization techniques are applied to generate images that are used for medical reporting. These DICOM images are later viewed by the radiologists or other domain experts. The important aspect is that only a limited set of interaction possibilities is available. If another visualization, possibly more tailored to the desired scenario, would be required, a request has to be sent to the corresponding assistant. This is, however, a tedious and time-consuming process.

The approach/idea presented in this thesis aims to combine different visualization techniques applied to different areas of interest and interactive elements. We define a scripting language that maps a certain interaction to a specific visualization of a region within the image. We will provide a set of so-called *interactive elements*, covering common tasks of medical inspection. The most simple example is a Point-of-Interest (POI) specification, attracting the user's attention to a particular region. Extending this idea, we will propose and discuss several other, more advanced, interactive elements together with their visualizations. Since we enhance images with interaction, i.e., rendering them active, we refer to this new type of DICOM images as ActiveDICOM images.

As DICOM and PACS are well-established standards in medicine, compatibility is an

important aspect of our contribution. Common radiology workstations should be capable of showing conventional DICOM images as well as ActiveDICOM images. By encoding the necessary scripting language into special layers and tags, this information is completely hidden and transparent to the user, current viewers or even the PACS. Only by a special viewer, which is similar to conventional ones, ActiveDICOMs will be recognized and activated. By utilizing web-based technologies, such an integration would require minimal effort. The main contributions of our work are:

- A visualization mapping language, ADICT, enabling static images to become interactive. It provides an abstraction layer for various target platforms such as a web-based viewer or PDF files. The scripting language controls the execution of animations and the timely sequence of showing various interactive elements according to user interactions or other triggers.
- Interactive elements encoded into static images offering a wide spectrum of interaction possibilities. Additionally, we provide means for visual storytelling in the form of interactive menus together with visual links (similar to the work of Steinberger et al. [72]) all embedded into the image. Menus and points of interest are examples of providing a specific flow by guiding the user's attention to certain areas of interest. Providing supplementary data, such as images, videos or complete documents should further enhance a domain experts decision making process.
- A transparent integration into the current medical workflow. We propose a web-based solution rendering deployment cycles unnecessary and providing access to data even on mobile devices. By using state of the art technologies we allow the user to access the application through a wide spectrum of different devices.
- A data-centric, on-demand encoding of visual knowledge through semantic language constructs. We propose a scripting language, the interactive glue, which is supplemented with annotated data elements. This allows the creator to add any binary data, annotated with meta information, into a DICOM image. Our proposed language is divided into two parts. The first one corresponds to the first point mentioned earlier. The second part of our language is used for arbitrary data described using an extended syntax of our language.

To summarize our approach and contribution, we aim to achieve a self contained image, which incorporates animations, annotations, and documents (e.g., illustrations or external important supporting documents). All of these elements have to be glued together, which is where our domain specific language comes into play. It is used to tell which desired animation or embedded entity should be displayed and when. As future work the language might mature into a full fledged programming language far beyond the features presented within this thesis.

One question may arise immediately, i.e., why create yet another programming language? An extensive answer to this question is hopefully given by reading this thesis. Chapter 4 tries to give an in depth answer and to justify why we use/create a language.

The short answer to the above question is abstraction and of course control over a domain. With the creation of such an abstraction layer one is able to support different target/output devices more easily by providing code generation back-ends (see Figure 2.1). If running within

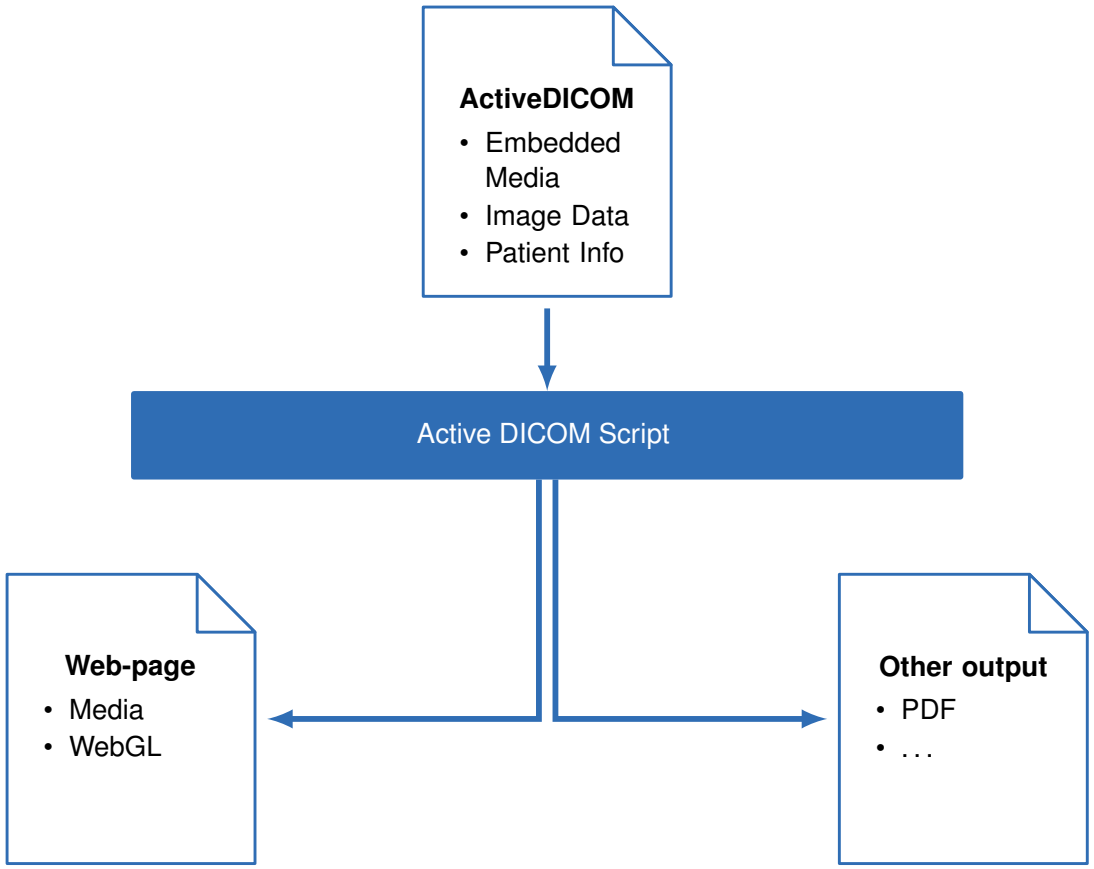


Figure 2.1: Showing an ActiveDICOM containing various supplementary elements such as embedded data. With the addition of our language, ADICT, also embedded into the DICOM, it is now possible to generate output depending on use cases and interactions triggered by user actions and needs.

a browser as a web based application, one would generate JavaScript/HTML output which is what we have done and are going to provide detailed descriptions. By creating a new language we define a domain upon which this language might operate. It is designed to deal with the proposed contributions in the most efficient way possible.

The remainder of this chapter is structured as follows. The next two sections give a short overview of medical imaging techniques and an introduction to DICOM. The last part is dedicated to survey existing ideas and techniques.

2.2 Medical Imaging Modalities

This section provides a short overview of common medical imaging modalities. DICOM images store the image modality and it is displayed by our viewer being an important information.

2.2.1 X-Ray

One of the oldest imaging techniques is based on X-Rays. It is also the most commonly used imaging technique. In the European region X-Rays are also called Röntgen radiation, named after Wilhelm Röntgen who discovered this type of radiation. It uses an ionizing radiation in order to produce images of a patient's body. An image is created as X-Ray beams pass through objects of different density. The densities of passed through materials determine the absorption of said radiation and thus can be used to produce images, such as the one shown in Figure 2.2. Typically X-Ray images are used to provide information about the following medical issues or area-of-interest:

- Broken bones
- Objects being swallowed
- Blood vessels
- Breasts (mammography)
- Lungs

Another application of X-Rays is the treatment of cancer, besides other types of radiation such as gamma rays and electron beams.

Figure 2.2: An X-Ray image of a hand. Image taken from Wikipedia [30].

2.2.2 Computed Tomography (CT)

Computed Tomography (CT) utilizes multiple X-Ray projections from various angles and combines them in order to produce a detailed cross-section image of a specific area of the human body or scanned object. It produces three dimensional datasets from a large series of images along a single axis of rotation. A more detailed and comprehensive introduction and overview is provided by Herman [37]. Figure 2.3b shows an example of a computed tomography image. It is able to generate a precise 3d-reconstruction of a scanned object and thus is a preferred method for the following areas of application:

- Organs in the pelvis, chest and abdomen
- Vascular condition/blood flow
- Pulmonary embolism (CT angiography)
- Abdominal aortic aneurysms (CT angiography)
- Bone injuries
- Cardiac tissue
- Traumatic injuries
- Cardiovascular disease

Alternative names are X-ray computed tomography (X-ray CT) or **c**omputerized **a**xial **t**omography scan (CAT scan). Such devices are mostly stationary and the patient is lying on a table and transported through tubes of various sizes. Figure 2.3a shows a common computed tomography device.

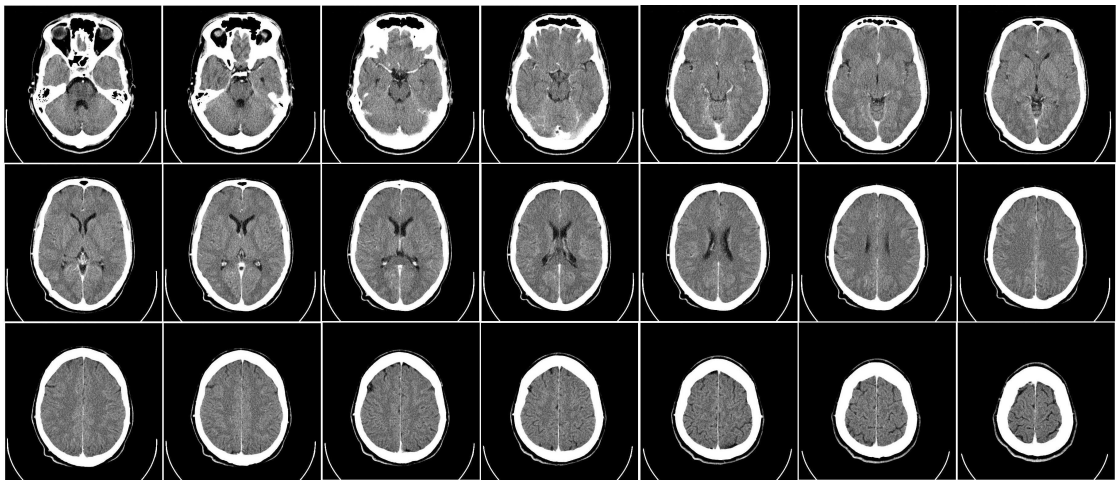
2.2.3 Magnetic Resonance Imaging (MRI)

Magnetic Resonance Imaging (MRI) is a medical imaging technology that uses radio waves and a magnetic field to create detailed images of organs and tissues. MRI has proven to be highly effective in diagnosing a number of conditions by showing the difference between normal and diseased soft tissues of the body. Figure 2.4a shows a modern MRI device. MRI is used to evaluate/examine for example the following areas of interest:

- Blood vessels
- Abnormal tissue
- Breast cancer
- Bones and joints
- Organs in the pelvis, chest and abdomen (heart, liver, kidney, spleen)
- Spinal injuries
- Tendon and ligament tears

Figure 2.4b shows example pictures produced by this imaging technique.

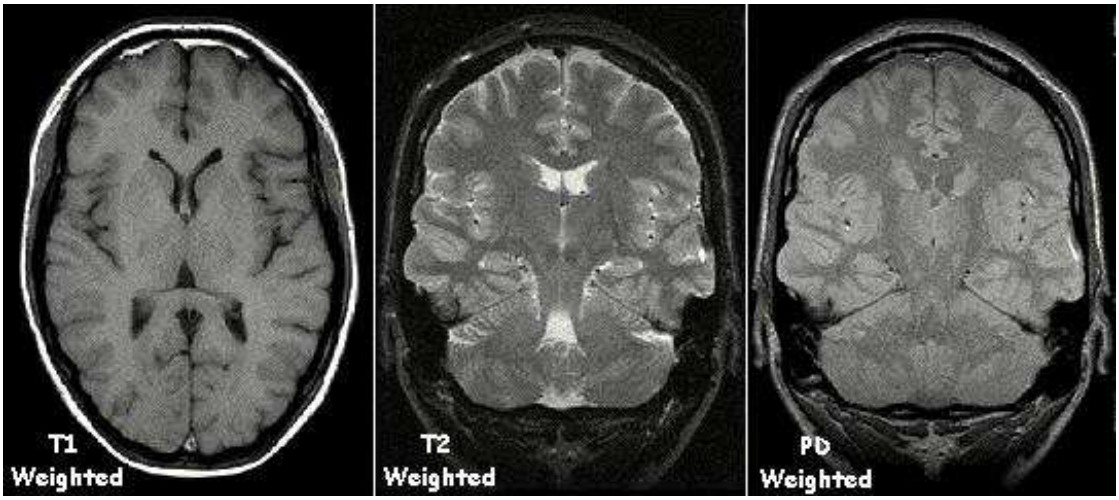
(a)



(b)

Figure 2.3: (a) shows a modern CT scanner (image taken from Wikipedia [29]) and (b) displays Computed Tomography images of a human brain (images taken from Häggström [50]).

(a)



(b)

Figure 2.4: (a) shows a medical MRI scanner (image taken from Wikipedia [39]) and (b) presents examples of T1 weighted, T2 weighted and PD weighted MRI scans (image taken from Wikipedia [44]).

2.2.4 Ultrasound (US)

Diagnostic ultrasound, also known as medical sonography or ultrasonography, uses high frequency sound waves beyond the audible limit of human hearing to create images of the inside of the body. The ultrasound machine sends sound waves into the body and is able to convert the returning sound echoes into a picture. Ultrasound technology can also produce audible sounds of blood flow, allowing medical professionals to use both sounds and visuals to assess a patient's health. Ultrasound is used in the following areas of interest:

- Pregnancy
- Abnormalities in the heart and blood vessels
- Organs in the pelvis and abdomen
- Symptoms of pain, swelling and infection

Figure 2.5 provides an example image produced by this imaging technique.

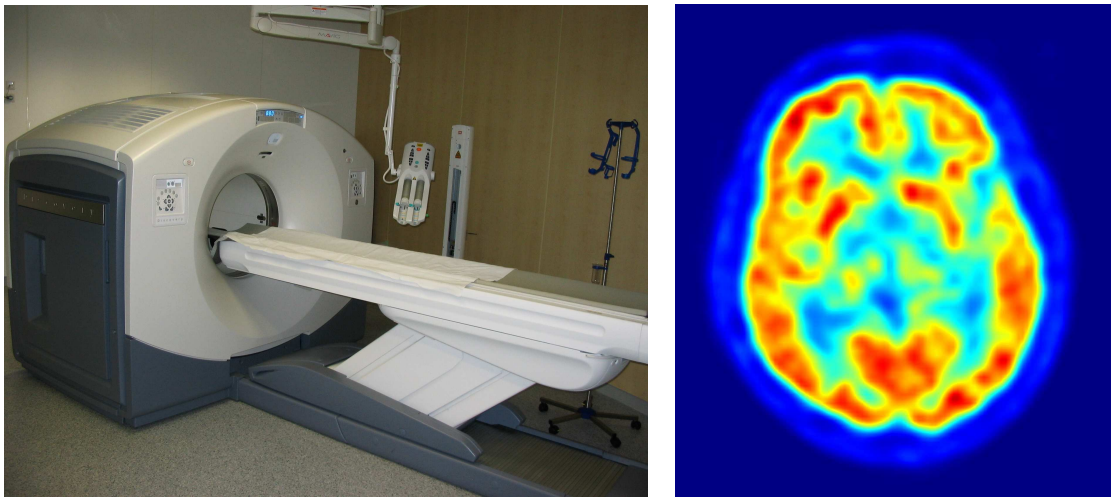
Figure 2.5: Sonogram of a fetus at 14 weeks (profile). Image taken from Wikipedia [80].

2.2.5 Positron Emission Tomography (PET)

Positron Emission Tomography (PET) is a nuclear imaging technique that provides physicians with information about how tissues and organs are functioning. PET, often used in combination with CT imaging, uses a scanner and a small amount of radiopharmaceuticals which is injected into a patient's vein to assist in making detailed, computerized pictures of areas inside the body. Figure 2.6a shows a PET-CT device with injection pumps for contrast agents. Positron Emission Tomography is used for instance to gain insight in the following fields:

- Neurological diseases such as Alzheimer's and Multiple Sclerosis
- Cancer
- Effectiveness of treatments
- Heart conditions

For added precision, physicians use a medical imaging technique that combines PET and CT. This allows images acquired from both devices to be taken sequentially and combined into a single superposed image. PET-CT serves as a prime tool in the delineation of tumor volumes, staging, and the preparation of patient treatment plans. The combination has been shown to improve oncologic care by positively impacting active treatment decisions, disease recurrence monitoring and patient outcomes, such as disease-free progression. Figure 2.6b provides an example of a PET scan of a human brain.



(a) (b)

Figure 2.6: (a) shows a PET/CT-System with a 16-slice CT. The device mounted on the ceiling is a contrast agent injection pump (image taken from Wikipedia [38]). (b) presents a PET scan of the human brain (image taken from Wikipedia [40]).

CT	MRI	PET	Ultrasound
Apply X-Ray and observe “shadow”	Apply magnetic field and observe the signal from the organ itself	Administrate isotope and observe the radiation	Apply sound wave and observe the reflection echo
Different absorportion of tissues	Different relaxation time or nuclei density	Track the radiation intensity and location	Different sound-wave propagation velocity and tissue density
High resolution and anatomical details	Relatively high resolution and differentiation of tissues with functional differences	Relatively low resolution and functional imaging (image activity)	High noise and less expensive (more widely available)

Table 2.1: Comparison of the previously mentioned medical imaging modalities.

2.2.6 Comparative Summary

Table 2.1 provides a short, comprehensive and comparative overview of the described medical image modalities.

2.3 DICOM

This section gives a short overview of what DICOM means and what in this thesis is meant by DICOM. DICOM does not only describe imaging aspects, but a vast ecosystem ranging from the mentioned imaging aspect to communication models. In this thesis we mostly focus on the imaging aspect of the DICOM standard.

2.3.1 History

DICOM is an abbreviation and stands for **D**igital **I**maging and **Co**mmunications in **M**edicine. With the rise of computed tomography (CT), followed by various other digital imaging modalities, in the 1970s and the growing use of computers in clinical environments, the American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA) realized that a standard for exchanging images between different vendors and devices is necessary. Furthermore, it was necessary to transfer images and associated data, such as image modality, patient data, and treatment/examination information. A first DICOM standard was developed in 1983 with the following goals:

- Promote communication of digital image information, regardless of device manufacturer.
- Facilitate the development and expansion of picture archiving and communication systems (PACS) that can also interface with other hospital information systems.
- Allow vendors the creation of diagnostic information databases that can be interrogated by a wide variety of devices distributed geographically.

The standard was first published in 1985. Information in this section, concerning DICOM, is based on the NEMA DICOM standard [55].

2.3.2 Scope and Goals

As a standard in medical imaging one of the main goals is to ensure the interoperability of various devices, techniques, and equipments. Adhering to the following goals, allows one to make it possible to achieve the aforementioned interoperability [55]:

- It defines a set of protocols which have to be followed for network communication.
- Furthermore, it defines syntax and semantics of commands for the above mentioned protocols.
- For media communication by specifying, a set of media storage services to be followed by devices claiming conformance to the standard, as well as a file format and a medical directory structure to facilitate access to the images and related information stored on interchange media.
- Additional information and implementation must be provided if a realization claims to adhere to the standard.
- Makes use of existing international standards wherever applicable, and conforms to established documentation guidelines for international standards.

As usual there are many points a standard would not specify, such as:

- Implementation details of features.
- A testing feature which provides means of validating whether an implementation complies to the standard or not.
- The overall set of features and functions to be expected from a system implemented by integrating a group of devices each claiming DICOM compliance.

In short, the DICOM-Standard operates within the field of medical informatics and addresses the exchange of information between medical imaging equipment and other systems. It is far more than “just” an image specification. Figure 2.7 provides a schematic overview of the DICOM communication model, based on a schematic representation of the “General Communication Model” in Chapter 1 of the DICOM-Standard [55]. An application may choose between online and offline, i.e., file based, communication:

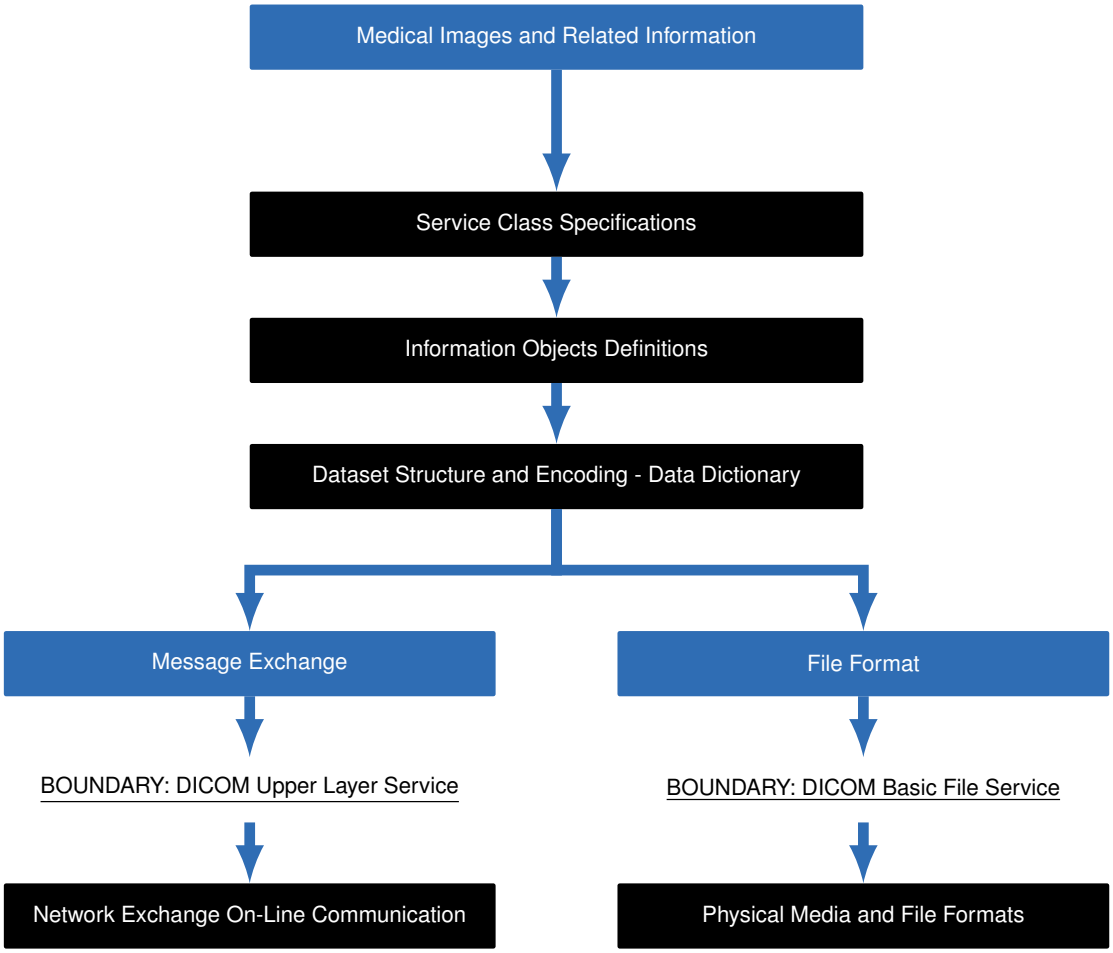


Figure 2.7: General DICOM Communication Model.

Upper Layer Service. It provides an abstraction and therefore independence from physical networking communication.

Basic File Service. It provides an abstraction of various file structures and formats.

The boundary is drawn where it is necessary to distinguish how data is retrieved. If retrieved from a file system, the boundary is named basic file service, but if it is retrieved through the use of network based communication, it is described as online communication in Figure 2.7.

2.3.3 Data Dictionary

Figure 2.7 shows an overview of various aspects of the DICOM specification and elements. One of the important elements, extensively used in this thesis, is the “Data Dictionary”. In short, the dictionary is a key-value pair based data structure allowing the creator to add arbitrary

Tag	Name	Keyword	VR	VM	Flags
(0008,0001)	Length to End	LengthToEnd	UL	1	RET
(0008,0005)	Specific Character Set	SpecificCharacterSet	CS	1–n	
(0008,0020)	Study Date	StudyDate	DA	1	
(0072,0070)	Selector UT Value	SelectorUTValue	UT	1	

Table 2.2: DICOM Data Elements, a selective sample.

additional data to a DICOM file. Being more precise, the dictionary is the central registry containing uniquely identified items. Such an entry is defined by the following elements:

- Unique tag, containing a group and an element number.
- Name of the data dictionary entry.
- Keyword for identifying an entry using alpha numeric character sequences.
- Data type (**V**alue **R**epresentation), such as a character string, integer ...
- Multiplicity for the value. It specifies how many values per attribute are permitted. Abbreviated as **V**alue **M**ultiplicity.
- Flag indicating for instance that an entry is retired.

Table 2.2 gives examples of defined data dictionary entries specified by the standard. This is only a short list and is not complete. Please refer to the DICOM-Standard [55] in order to see the full list of specified elements. The selected elements are chosen for demonstration purposes showing off various value representations and multiplicities.

Each row in Table 2.2 describes an entry of the data dictionary. As explained above each element has a tag consisting of a group number and an element number within this group. Numbers are given in hexadecimal numbers. The DICOM-Standard only defines even numbered groups. This can be used to define user defined dictionary entries, adding functionality that augments DICOM images. More about adding non-standard elements is shown in Chapter 3. The name gives a short description of the intention of the data element. The keyword specifies a unique name for the data dictionary entry in a more human readable fashion. The column labeled "VR" specifies the value representation. Compared to programming languages this can be seen as the type of a variable definition. Table 2.3 gives an overview of various value representations. Column "VM" of Table 2.2 describes the value multiplicity. In Table 2.2 one can see that all entries but one are single valued ones. The entry (0008,0005) may return more than one value, but at least one. The last column (labeled flags) shows only one entry. The entry RET stands for retired and means that such an element is not supported anymore. An implementation or piece of software may choose to support such elements in order to maintain backward compatibility. The RET flag is valid in relation to a specific version of the standard, meaning that a previous version of the standard might still see it as non retired and therefore an implementation can only adhere to a certain version of the standard. Please

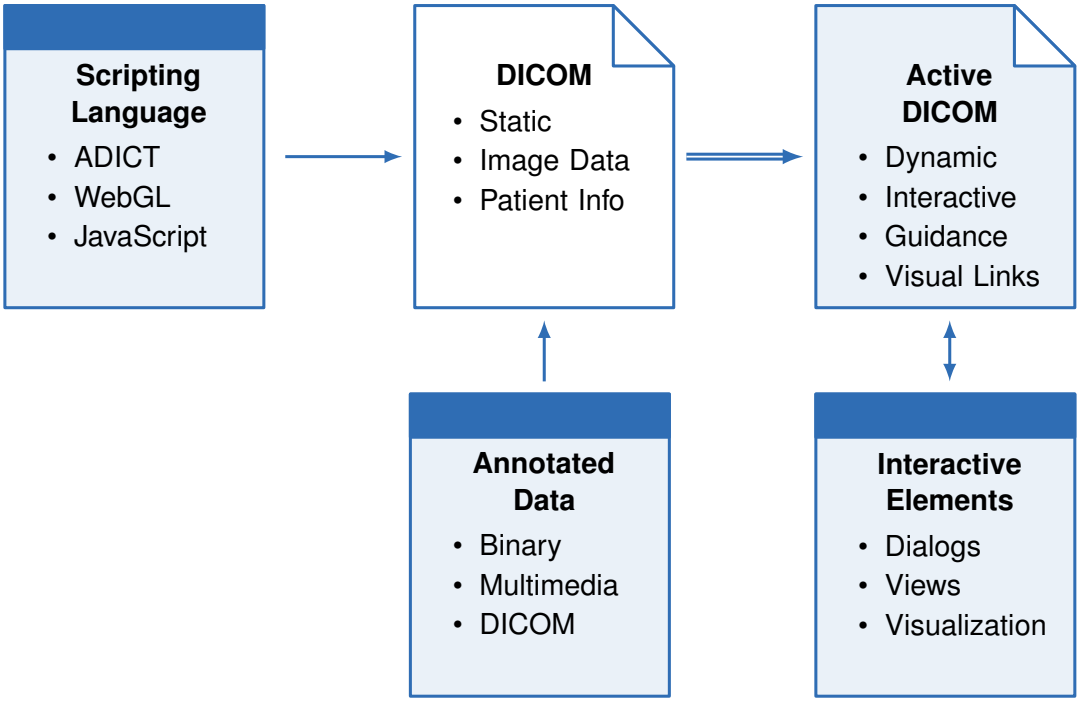


Figure 2.8: Outline of our contributions. By augmenting common DICOM images with additional features, we propose ActiveDICOM. Annotated data together with executable code are added into a DICOM image, creating an ActiveDICOM including interactive elements.

see the references which version this thesis refers to. The value representations described in Table 2.3 are in no way complete and refer only to the most important ones, for instance needed throughout the thesis. For a more detailed overview of available value representations the reader is referred to the DICOM-Standard [55].

2.4 General Overview

Our contribution combines several aspects from the field of visualization, compiler language design as well as interaction principles (see Figure 2.8). The subsequent paragraphs/sections provide an outline of the most relevant related work from these areas.

2.4.1 Medicine/Radiology

In the typical radiology workflow, the radiologist fetches processed and prepared clinical images from a PACS for medical reporting. The preparation is carried out by the referring physicians and radiology assistants [59]. Stramare et al. [74] describe a system to provide consistently structured reports for DICOM images by means of a standardized radiology database.

VR Name	Definition	Character Repertoire	Length of Value
UL Unsigned Long	Unsigned binary integer 32 bits long. Represents an integer n in the range: $0 <= n < 2^{32}$.	not applicable	4 bytes fixed
CS Code String	A string of characters with leading or trailing spaces (20H) being non-significant.	Uppercase characters, 0–9, the SPACE character, and underscore _, of the Default Character Repertoire	16 bytes maximum
DA Date	A string of the format YYYYMMDD; where YYYY shall contain year, MM shall contain the month, and DD shall contain the day.	“0“–“9“ of Default Character Repertoire	8 bytes fixed
UT Unlimited Text	It may contain Control Characters, CR, LF, FF, and ESC. It may be padded with trailing spaces, which may be ignored, but leading spaces are considered to be significant. Data Elements with this VR shall not be multi-valued.	Default Character Repertoire and/or as defined by (0008,0005) excluding Control Characters except TAB, LF, FF, CR (and ESC when used for ISO 2022 escape sequences).	$2^{32} - 2$ bytes max.

Table 2.3: DICOM **V**alue **R**epresentations referring to Table 2.2.

Several standard operations are defined for a radiology workstation to allow radiologists performing soft-copy-reading of the examination data [59]. Among these operations are changing the windowing function, browsing through slices and providing tools for quantitative image analysis. Nevertheless, most interactive aspects reside on the preparation and processing side. Several medical studies have shown that viewing an image providing only one visualization is not enough for reliable medical reporting. Northam et al. [56] investigate the reduction of the full field-of-view to a limited one for a specific Region-of-Interest (ROI) with increased spatial resolution. However, their study shows that pathologies outside the ROI are missed. With our proposed approach, we could show the limited field-of-view on-demand, encoded into the image with the field-of-view. Portugaller et al. [58] and Schertler et al. [70] assess in their studies the importance of viewing axial images together with other techniques, such as **M**aximum **I**ntensity **P**rojection (MIP) or **C**urved **P**lanar **R**eformation (CPR), discussed by Kanitsar et al. [41]. Using one technique solely, would not be sufficiently accurate for reporting. In this thesis, we describe how to embed such additional visualizations efficiently into a single DICOM image and how to present them on-demand if requested.

2.4.2 Internet Technologies

DICOM viewers are usually implemented as native client applications, as described by Zeman et al. [82] and Escott and Rubinstein [32], running on top of the computer's operating system. Example viewers are DicomWorks, as described by Puech et al. [60], and the OsiriX viewer, as mentioned in the work of Rosset et al. [68]. Teng [76] describes a system for manipulating the meta data of DICOM images that is integrated into the operating system. Native client systems require significant configuration with an increasing number of connected clients, as is the case for corporations or medical facilities with many departments. With advancements in web-based solutions, browsers became capable of supplying complex applications, such as Rich Internet Application (RIA), but straightforward to manage and maintain. Melicio Monteiro et al. [49] describe technologies for building web-based DICOM viewers. In our work, we use the widespread client-server model. A RIA offers many possibilities such as simple maintenance and access, as described by Fernàndez-Bayó et al. [33] and Kitney at al. [45]. The importance of these two aspects are increasing in our modern mobile society. Mobile devices are, as already mentioned, a rapidly growing and immersive field. However, despite being powerful, web-based technologies impose implementation restrictions, such as responsiveness, native look and feel and access to local hardware features. Technologies making it possible to access such features exist, but are mostly tied to certain browsers. Examples for such technologies are ActiveX, Java Applets and Flash (Silverlight). Bochicchio et al. [27] describe how to extend web applications with 3D features using WebGL. They mention technologies that access the graphics hardware from within the browser. A web application can be even used for high performance rendering, as described by Mobeen et al. [53]. They demonstrate that browser-based solutions are capable of performing tasks only native applications could. Mobile platforms support WebGL in certain browsers or rendering engines, as mentioned by Golubovic et al. [35].

2.4.3 Script Language Design

Becker et al. [26] describe how to add additional patient related data to DICOM images and consider the interoperability between different vendors. They discuss that adding features should be done with care and compatibility in mind. Radosevic and Klicek [62] describe a concept of a high level multimedia scripting language. Such a language provides a certain level of abstraction and yet proves to be powerful enough to let the user control the content to a sufficient degree. An object-oriented approach for DICOM manipulation, based on C++, is described by Von Land et al. [78]. Our implementation is built upon Java, i.e., we also take an object-oriented approach. Additionally, Java offers a great variety of tools and frameworks for web applications. Arguiñarena et al. [24] describe a Flash-based DICOM viewer together with various other tools. However, they do not augment the images with interactive elements.

2.4.4 Interaction

An overview and principles of Human Computer Interaction (HCI) and user interface design are provided in various previous work [28, 34, 81]. They define guidelines for the consistency of user interfaces, the usage of colors and the feedback the application should give. They also investigate several approaches for linking 2D visualization with 3D data. Kohlmann et al. [46–48] proposed LiveSync, a technique that automatically selects the most suitable 3D view from a 2D slice position. Their approach synchronizes a 2D view with a 3D view, based on local data features. Motivated by this approach, we provide interaction facilities encoded in the DICOM image itself, linking regions within the data to visualizations defined by means of a scripting language. Viola et al. [77] describe an approach where the attention is attracted to a specific region within the data, while retaining the overall context of the visualized data. In our work, we offer similar possibilities, since we embed visualizations into the image and display them later in a superimposed way. Saleem et al. [69] give a comprehensive overview of human factors for clinical information systems regarding medical workflow and task analysis, user interface design, and principles for patient safety. Motivated by them, we present novel opportunities for radiologists by augmenting DICOM images with interactive elements. This allows to create a visual comparison of different medical visualization techniques side-by-side or, as described in the work of Schmidt et al. [71], clustered and integrated into one image. Olwal et al. [57] discuss design guidelines for medical team meetings, e.g., discussion rounds, using touch devices like tablets. Rautek et al. [63] use a rule-based system for mapping data attributes to visualizations. Their approach applied fuzzy logic in order to semantically adjust the resulting visualizations. They add interaction-dependent semantics such as the viewpoint or user focus in their extended work [64]. Mistelbauer et al. [51] proposed *smart super views* where a fuzzy inference system suggests only suitable visualizations to the user.

2.4.5 Multi-View Visualization

Ropinski et al. [66] present a layout technique for interactive close-up views on multimodal data. Balabanian et al. [25] proposed a graph-based layout for integrated views to navigate

in a volume hierarchy. Various ROIs can be explored by different types of interaction and visualization capabilities. Motivated by their approach, we offer interactive elements encoded into the DICOM image. Such an image could contain many views simultaneously, displaying various regions with different visualization techniques. Roberts et al. [65] proposed linking and brushing techniques to highlight selections in all views by visually connecting them. Steinberger et al. [72] proposed visual links between various selections within multiple applications in order to preserve context. We will discuss, how we adopt this approach to guide the user through an image, i.e., to tell the story of an image.

2.4.6 Visual Analytics

Keim et al. [43] for instance mention the topic of Visual Analytics. According to them said topic combines automated analysis with interactive visualizations. Motivated by such a definition we apply it to DICOM images. They provide an elaborate definition of what Visual Analytics represents. It means the creation of tools and techniques to enable people to [23, 43]:

- Synthesize information and derive insight from massive, dynamic, ambiguous, and often conflicting data.
- Detect the expected and discover the unexpected.
- Provide timely, defensible, and understandable assessments.
- Communicate these assessment effectively for action.

They also describe a process of how Visual Analytics is applied or works. Their feedback loop tightly couples human interaction and automatic visual analytics methods in order to gain insights from data. In Andrienko et al. [23] the motivation behind this topic is the term **information overload**. It is directly related to technical advances made in storage technologies as well as the whole information technology section. Information overload may lead to getting lost in data which may be according to Andrienko et al. [23]:

- irrelevant to the current task at hand,
- processed in an inappropriate way, and
- presented in an inappropriate way.

Data presentation is one of our main goals for ActiveDICOM.

2.5 DICOM Viewers

This section gives an overview of currently available software for viewing DICOMs and gives a short description of features or lack thereof. In Chapter 3 it is described why and how and what was implemented. Haak [36] provides a survey of different DICOM viewer implementations and systems on which this section is based upon. Today a vast amount of software is available to choose from. Even open source variants are present. In 1998, Honea et al. [61] started to

The Free Software Definition: Four essential freedoms
Freedom 0. The freedom to run the program as you wish, for any purpose (freedom 0).
Freedom 1. The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
Freedom 2. The freedom to redistribute copies so you can help your neighbor (freedom 2).
Freedom 3. The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

Table 2.4: The Free Software Definition: Four essential freedoms [18].

investigate commercial personal computer based viewer software. They identified a certain set of features which seemed to be important:

- Multiple windowing levels.
- Distance and angle measurements.
- Storage of annotations.

In Nagy et al. [54] an overview of open source frameworks and DICOM viewers is given. Open source applications and software/libraries play an important role. Even the implementation described later in this thesis is based on an open source framework for handling all the important DICOM interactions such as reading layers, image information and various other meta data (such as patient information and image modality, etc.). Due to the heavy use of open source software for the implementation, our focus is clearly on said software for medical image manipulation.

First it is important to specify what open source means. Basically open source means that one can access the source code freely. An experienced user then is able to even enhance or modify such software. Such improvements may then be redistributed as open source. Open source does not mean everything is available for free. Terms such as open source and free software are discussed controversially and a thorough analysis might fill an entire book on its own. Throughout this thesis open source means that the source code is available under a certain license (mostly GPL or LGPL).

The first definition is taken from the free software foundation, see Table 2.4. The next definition, described in Table 2.5, is taken from the open source “community”. As one can see it is highly controversial and the reader is referred to literature especially targeted to discuss philosophical differences. The given definitions are shortened. For the full one please refer to the full open source definition [20]. Due to their distributed and decentralized nature, open source projects are difficult to evaluate. Nagy et al. [54] describes a catalog of criteria for evaluating such projects. The following criteria are contained therein:

The Open Source Definition

Free Redistribution. The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

Source Code. The program must include source code, and must allow distribution in source code as well as compiled form.

Derived Works. The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

Integrity of The Author's Source Code. The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

No Discrimination Against Persons or Groups. The license must not discriminate against any person or group of persons.

No Discrimination Against Fields of Endeavor. The license must not restrict anyone from making use of the program in a specific field of endeavor.

Distribution of License. The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

License Must Not Be Specific to a Product. The rights attached to the program must not depend on the program's being part of a particular software distribution.

License Must Not Restrict Other Software. The license must not place restrictions on other software that is distributed along with the licensed software.

License Must Be Technology-Neutral. No provision of the license may be predicated on any individual technology or style of interface.

Table 2.5: The Open Source Definition [19].

- **Web site appearance and documentation.** Having a good and extensive web side and/or documentation is often a sign of how healthy such a project is. As documentation is often seen as being the last part project time should be spent on, it is therefore a good indicator of the health of said projects. A documentation is often needed, because of the nature of open source projects a developer is often in dire need to consult manuals as no official, centralized support is available, compared to commercial closed source projects/software.
- **Activity and utilization.** Open source projects are often initiated by volunteers and developers in their spare/free time. Although many projects have dedicated, i.e., paid developers from companies supporting the development is often not the case. Therefore a measurement of activity and last commits and how often the software was downloaded is a good indicator of how well the project is doing.
- **Ease of installation.** In Nagy [54] easy of installation is another indicator of how "finished" or well done an open source project is. The easier to install it is, the more mature it might be. An easy and user-friendly installation or being able to do something productive with the software/product the healthier and active the community is mostly.
- **Technical support forums.** Due to the nature of open source projects an active online forum is often needed for users to get help with problems. It is also another indicator for the wellbeing of said projects. A more or less direct result of active forums are FAQ sections kept up-to-date, which are also highly valued among users.

The last few paragraphs dealt with open source, its definition and how quality might be measured in such decentralized and open projects. The focus and scope of the next section is checking available open source web-based DICOM viewers. Fat client based applications are only mentioned briefly since one of the main goals was to develop a simple new web-based DICOM viewer. More about what kind of technology stack we used in the following sections. For a more comprehensive and complete overview not tailored to web based technologies the reader is referred to the above mentioned articles as they provide a huge selection of literature and insight about available rich client applications as well as web based ones.

One more short remark about why open source has to be measured in regards to how active it is. As mentioned above in an open source project there is no inherent "I have the right to..." because most of the time if one tries to phrase such a "demand" one is often confronted with "implement it yourself". This means that there is no such thing as support (not for free). Users interested in an open source software are often required to invest some time before asking for help.

In, Nagy [54], a list of web-based PACS was listed. See Table 2.6 as a selective list of open source web-based PACS software. It contains open-source software, some of it was used to implement our web-based DICOM viewer. Kaserer [42] describes the implementation of a web-based open-source DICOM viewer. The library used for our implementation, dcm4che [16], also provides a module for a web-based DICOM viewer. In our implementation we used version 2 of the library, which was updated to version 3 during the creation of our thesis. Table 2.6 shows an exemplary list of web-based DICOM image viewers. From those we derived the following features we implemented in our web-based image viewer:

Name	License	Homepage/Description
CDIMEDIC PACS Web	LGPLv3	Full featured free PACS based on dcm4chee and mysql, with remote web accession available for Linux in Debian packaging format for x86 32 and 64 bit processors. Homepage: http://sourceforge.net/projects/cdmedicpacsweb/
DIOWave	GPLv2	DIOWave Visual Storage is a Web server, which displays DICOM images. The look and feel of the Web pages are similar to imaging workstations. Homepage: http://diowave-vs.sourceforge.net/
miniwebpacs	n/a	The project develops a low cost system to provide storage, control and recovery of medical images and information in healthcare providers of small and medium capacity. Such system is based on the DICOM standard and in the actual WEB technologies. Homepage: https://sourceforge.net/projects/miniwebpacs/
cornerstone	MIT	The library is part of a web-based DICOM library. It provides a demo application with the capability to change the windowing function and cycling through various images. Homepage for the core library (Cornerstone Core): https://github.com/chafey/cornerstone
DWV (DICOM Web Viewer)	GNU GPL	DWV is an open source JavaScript and HTML5 DICOM viewer. It provides a demo application and allows the user basic interaction with medical images. Homepage: https://github.com/ivmartel/dwv

Table 2.6: Open Source web-based PACS and image viewers. The first three entries are based on Nagy [54]. This list is a short selection of viewers we examined to base our realization on.

- Changing the windowing function.
- Searching through a collection of DICOM images.
- Zooming in and out.

ClariPACS

During our realization we came across a web-based DICOM viewer with quite unique features similar to ours. The viewer we want to describe briefly is called ClariPACS [4]. It is not an open source viewer therefore we use a separate section for its introductory description/overview. Released in 2012 it was developed by two radiologist at Stanford University aimed at research and education purposes. ClariPACS features a cloud DICOM viewer usable in web-browsers and has the following advanced features, taken from [3, 4]:

- "**Works in your web browser.** No software or plugins to install. View full DICOM cases from home."
- "**Share.** Create multimedia reports and email links to full DICOM studies."
- "**Collaborate.** Discuss cases in real-time over the cloud using screen share mode."
- "**Fast.** Smart pre-loading algorithm lets you start scrolling immediately. Lossless image compression."
- "**Reliable.** Automatic backups and security updates. No maintenance. Always the latest features."
- "**HIPAA compliant.** Fully HIPAA compliant for storing PHI (requires a signed business associate agreement). For teaching file use, HIPAA-compliant anonymization is automatically performed on all cases."

One of the features most interesting to us was the ability to create "Multimedia Interactive Reports". On their page it is possible by pressing the r-key to open an interactive report view. Such a report enables one to create links to cases and studies/annotated key images. Based on their features we designed our DICOM viewer and added a more abstract and customizable, interactive, and enhancing feature set to DICOM images.

CHAPTER

3 Web-based DICOM Viewer

This chapter describes our realization and implementation to achieve our desired goals. As stated previously in the thesis the viewer, which was implemented, should run in a browser and is thus a web-based application. A connection to a PACS system is not required and thus the viewer is able to operate on its own without the need of a complicated setup or many external dependencies. The implemented viewer heavily focuses on the topic of enhancing DICOM images with interactive features and embedding various other elements, such as videos/movies and documents, into the DICOM image itself.

This chapter discusses the features of the implemented application, the technical design choices, and used technologies. At first, an introduction to the architecture of the implemented software is given being followed by a more thorough description of the many layers and components the viewer is comprised of. The last section of this chapter describes the implemented ActiveDICOM features and how they can be used, e.g., integrated into DICOM images.

One of the main decisions made during implementation and design of the application was that mostly, if not only, open source software should be used. Since Java is wide spread in the development of web–based applications and has quite an active community, it was an easy decision that Java should be the language of choice to implement the viewer and many of its components. The second most prominent language, used during development was JavaScript (or ECMAScript). Our domain specific language, ADICT, is the third most used language for the implementation. Many animations and interactive elements are implemented using ADICT.

Web development technologies change quite rapidly and therefore our chosen technology stack may be outdated already or questioned as of writing the thesis. Emerging JavaScript based technologies for instance are Angular JS or Polymer. Even our database access layer, currently using plain Hibernate instead of JPA (version 2.0 or newer), might seem outdated.

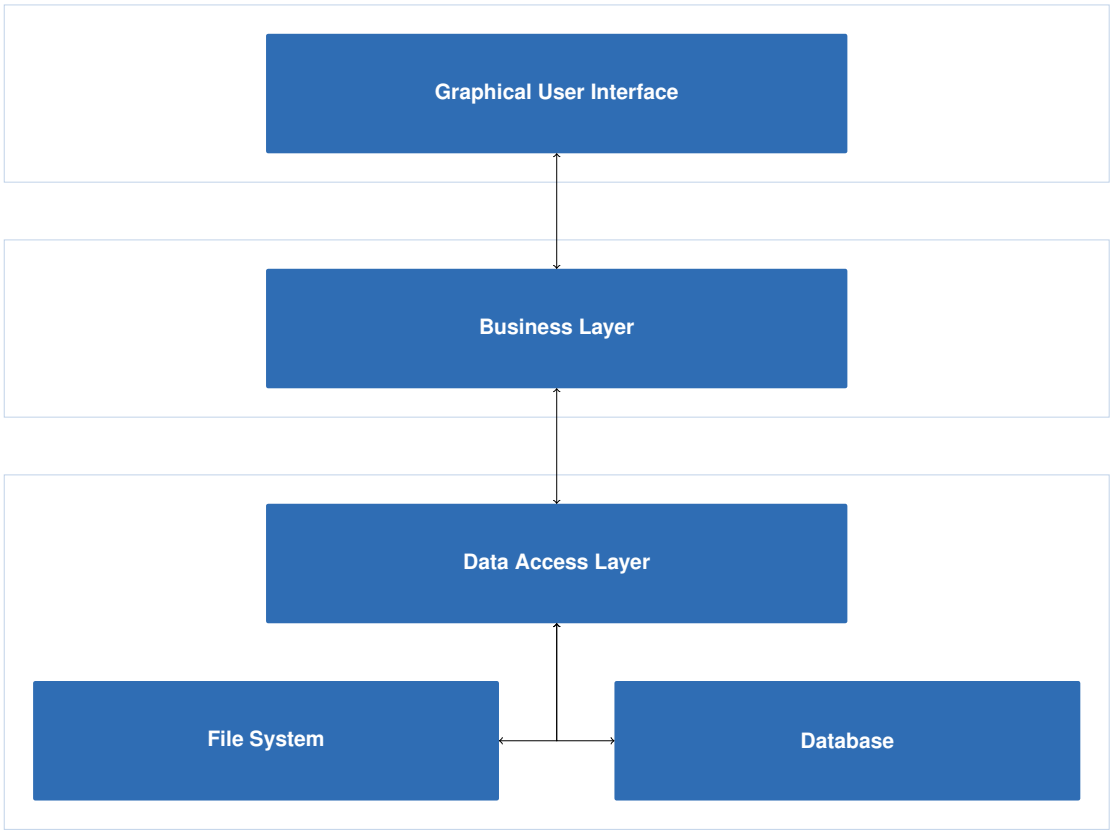


Figure 3.1: Architecture of our DICOM viewer.

3.1 Architecture

The viewer is designed as a tiered application. It consists of three tiers (shown in Figure 3.1). Boundaries are sometimes not as clear as one would believe as performance was often held higher than application design:

- **Graphical User Interface.** The graphical user interface layer implements the web interface and the navigation within the application.
- **Business Layer.** This layer is responsible for handling all aspects which define the application. This is the core layer were most of the application logic is implemented, such as ActiveDICOM features, batch jobs, user and session handling.
- **Data Access Layer.** This layer stores the data and is responsible for retrieving and managing data requests. This layer is divided into the following two parts, File System Layer and Database Layer.

Each layer of our architecture uses its own entities and is only allowed to communicate with adjacent layers. Entities from the data layer are not used in the business layer and vice versa. Figure 3.1 shows the three main layers and how they communicate with each other (black

arrows). Each black arrow requires a mapping from objects of one layer to objects from another layer. For instance, entities, i.e., objects from the data layer, are mapped to objects of the business layer if the business layer calls methods of the data layer. We chose such a tiered architecture due to the following benefits:

- **Code Reuse.** Different views using the same or similar data can reuse a huge portion of code, requiring only changes to the graphical user interface layer in optimal cases.
- **Locality.** Changes made to one layer should not impact the whole application thus keeping it local. For instance, during our implementation the GUI was redesigned several times without impacting the whole application design in a critical way.

Our approach has several shortcomings as well:

- **Complexity.** As mentioned earlier, objects have to be converted/mapped from layer to layer making code complex and may result in boilerplate code.
- **Memory.** Since each layer manages its objects data redundancy might lead to more memory usage.

3.1.1 Graphical User Interface (GUI)

Choosing a web-based GUI framework in the Java ecosystem is not an easy task, as there are plenty to choose from. Choosing a framework also defines how an application is built and structured. Therefore we defined a list of requirements for choosing a technology stack. The following sections cover the very basics of web application development and present the reasoning behind why some decisions were made in that particular way.

3.1.1.1 Single Page Application versus Classic Web Application

Normally, web applications are dissected into pages. Each page has a URL making it possible for users to create bookmarks. Navigation between pages is handled by links (this is a simplification since the GUI framework normally enhances links in order to provide browser back button support and various other hooks for developers to alter a link's behavior). Figure 3.2 gives an overview of a classic and basic web application.

In the classical way a request fetches a whole page and, thus, the browser has to render the full page. Clicking a button, a link, or triggering various other interactions might perform a round-trip to the server, which then produces the resulting page. Rendering a full page might be a limiting factor for the user experience, since updating/rendering a whole page results in flicker and an undesired interactive behavior. Modern applications, therefore, use AJAX-based technologies. AJAX stands for Asynchronous JavaScript And XML. Using AJAX-based requests, it is possible to send/receive data only needed for this particular request. The browser might then be able to perform an update only of changed regions of the web page. Figure 3.3 gives an overview of this process/workflow.
The above mentioned concepts are the basic ones for web-based applications. A single page application relies on the use of AJAX based technologies whereas a traditional web application, which is built upon the concept of distinct pages, makes rare use of AJAX based technologies

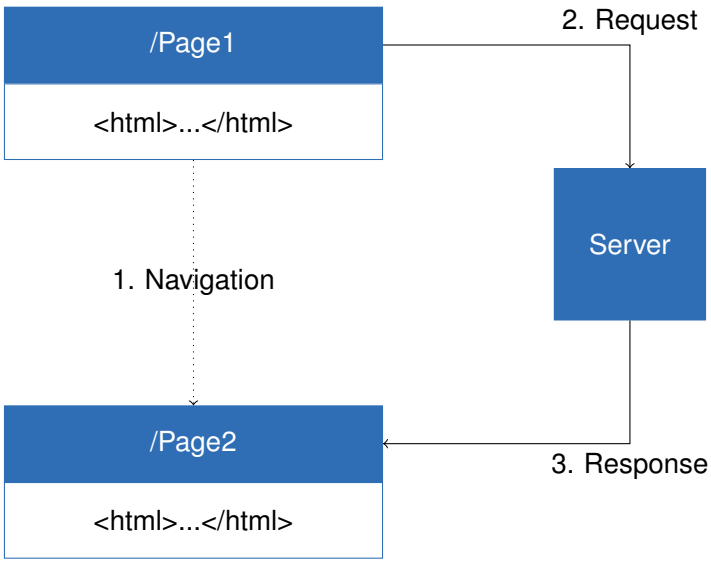


Figure 3.2: Schematic representation of a classical web application and the concept of pages. 1. Navigation. The user wants to navigate from Page 1 to Page 2, depicted as a dotted line. Form data gets submitted to the server (2. Request). The server processes the request and creates a response, i.e., a whole page (3. Response).

in general. Although AJAX is most often used with XML, in our approach (depends on the web-framework) JSON (JavaScript Object Notation) is transmitted.

3.1.1.2 Rich Internet Application (RIA)

A rich internet application aims to behave like a classic desktop application providing complex interactions such as drag and drop and complex effects, such as animations and nested dialogs. Although it is not clearly defined what a RIA application is and what a standard web-based application is, we specify it in the following way:

- A RIA should provide a desktop-like rich and interactive graphical user interface.
- It should be a web-based application deployed on a server or servlet container.
- It should facilitate complex interactions and animations such as playing multimedia, viewing embedded documents, and interacting with DICOM images.

The application has to handle user login on its own. A fine grained user privilege system is currently not implemented but work was done to add such a feature easily. RIAs might put more load onto the client (browser) side and might even introduce a longer loading time to initialize the application after the required information has been downloaded to the client, for example JavaScript files to bootstrap the application. Usually such an application heavily depends on the browser's JavaScript engine to perform well. Running the viewer on a mobile device was not part of this work since bandwidth and processing power are severely lacking.

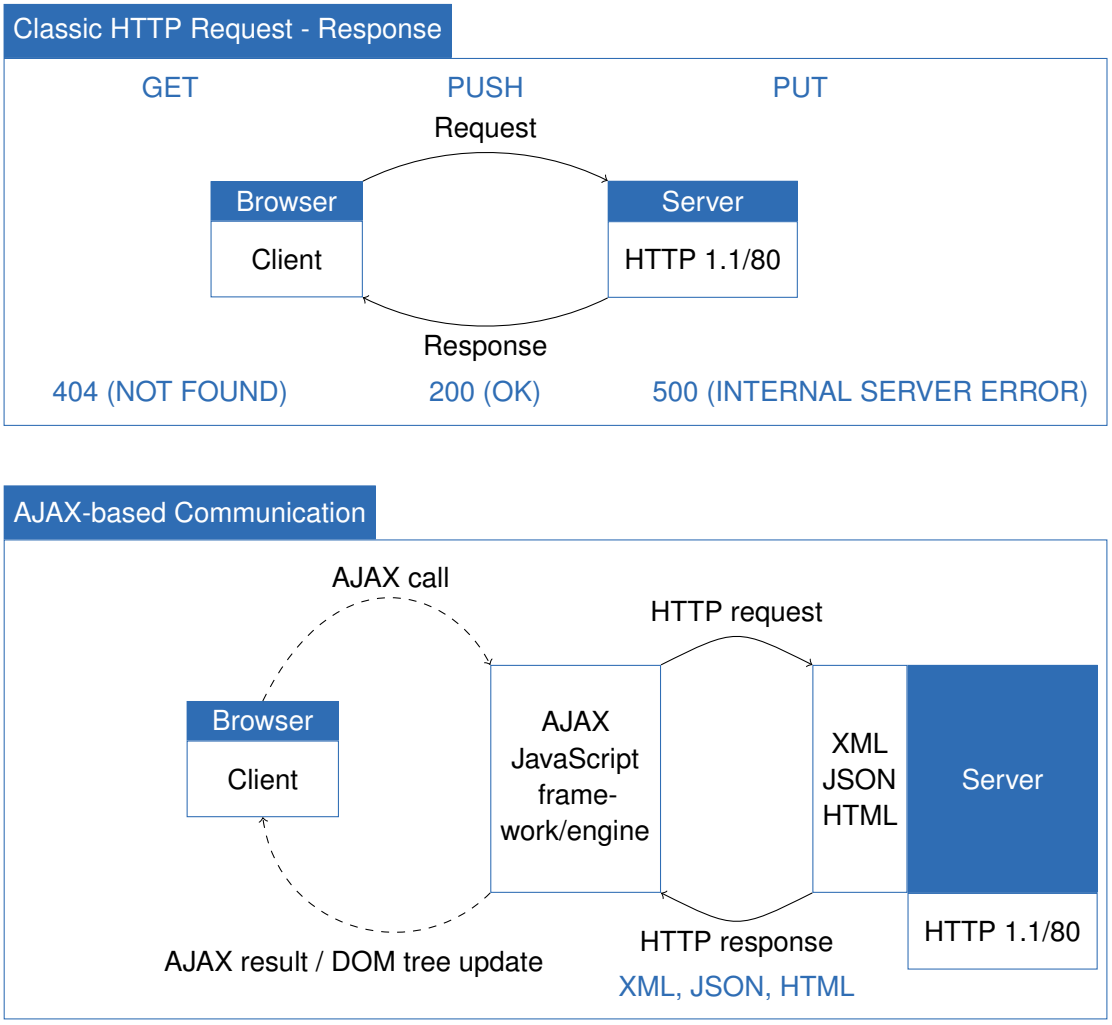


Figure 3.3: Abstract concept of a classical request-response versus an AJAX based request-response cycle. The top picture shows a server listening on port 80, and a standard HTTP port supporting protocol version 1.1. The client issues requests of a given type, such as GET which is the typical request a browser would use requesting a page. The server responds with a status code and the content. The bottom picture shows a schematic of an AJAX based communication. A JavaScript based library is acting as a middle-ware for issuing requests to the server. The server often provides XML or JSON based web services. Responses are parsed by the library. DOM tree updates are performed only for changed elements so that flickering is reduced.

Screen space might also be limited if DICOM images have to be manipulated on such a small screen.

3.1.1.3 Programming Model

As the realization of our idea, i.e., ActiveDICOM, approached the implementation phase we had to made many decisions. Among such decisions, the two most important ones have been:

1. Should the business logic be implemented on the server side?
2. Should the programming model be more like in a traditional application design?

Addressing the first point rises the question where input validation, such as user input checking, should occur. The main business logic should run on the server. Input validation and consistency checking of user input should all be done on the server side. The user interface on the other hand has to use state of the art JavaScript technologies for dynamically resizing and modern looking applications, thus mixing server- and client-side programming.

The part implementing the built in DICOM viewer is using a pure JavaScript implementation. It renders to a HTML canvas element using WebGL. This allows for smooth texture updates and zooming and translations. This part is heavily using client side processing whereas the server is just called/used via REST web-services to provide data such as DICOM images and additional embedded data (see Section 3.2.1.2 for adding data to ActiveDICOM images). In the current implementation DICOM data is processed on the server and an image is sent to the client. This was done since we encountered images ranging from a few bytes to many hundreds of mega bytes. Dealing with such a diverse size of images, a pure client based solution was not feasible. For example, whenever the windowing function is changed, the client (e.g., the browser) sends a request to the server specifying which layer with which windowing function it requires. The server processes the request returning the result as a JSON response.

One important advantage of server side processing is that all the application information might be present there, such as user data. Furthermore it is much easier to have the client and server synchronized. The downside of heavily using the server is the increased need for server hardware and processing power. Bandwidth also needs to be strong. Many requests are created and many of those send binary data like images using quite a lot of bandwidth. Client side processing may impose the risk of manipulation with simple tools built in every browser, like developer tools provided by modern browsers.

Developing on the server side using Java also proved to be more stable in regard of browser compatibility since hand written JavaScript code was scarce for the main UI. Since there was knowledge how to develop Swing based client applications, the choice whether to use such a programming model for the web or not was not a difficult one to make.

3.1.1.4 Complexity

Building a web-based solution for our ActiveDICOM idea is quite different than building a traditional desktop one. Web-based applications tend to use a wide variety of different technologies. We summarize what kind of technologies were involved in the realization of

ADICT as a web-based solution. Since our implementation also features user management and a file system/database hybrid for managing DICOM images, a data holding layer using database technologies is considered here as well:

HTML. The **H**yper **T**ext **M**arkup **L**anguage is the very basis of the world wide web. Its knowledge is required for building such solutions/realizations. Due to modern web sites heavily using JavaScript, plain HTML might not be written directly as often as years before. Understanding principles such as response, request, GET, PUT, URL and URI are still essential.

CSS, SCSS. Styling web sites is as challenging as their creation. Basically one can style a web page using inline styles attached to each HTML element. Such an approach makes it rather difficult to change the appearance later on. Therefore styles are separated into **C**ascading **S**tyle **S**heets (CSS). Such style information files are written in a domain specific language supporting nesting and complex rules when to apply which rule to what element. Plain CSS is not directly used in ADICT since it is lacking modern features such as class like inheritance, complex expressions and many other features. We use SCSS, an improved style sheet preprocessor, which generates CSS files.

JavaScript, WebGL. Creating dynamic and interactive web sites is not possible without the use of JavaScript (without installing plugins into the browser or using propriety technologies). In our viewer many elements are written in JavaScript. Even the used GUI-framework makes very heavy use of JavaScript. It is a dynamic, functional and interpreted language suited for short code snippets making it possible to create animations, windows, overlays and even writing OpenGL code for rendering textures, using shaders and other advanced graphics capabilities. WebGL heavily uses the HTML-canvas element.

Java. On the server side Java is quite popular for developing enterprise applications. Java is an object-oriented, statically typed, interpreted language. It features a rich set of libraries for nearly every problem or task. It features a quite active open source community and many of the available tools/technologies are open source. It employs garbage collection thus freeing the user of the burden of manual memory management. Structured exception handling further emphasize the creation of clean code suitable for long running server tasks/applications.

RDBMS, SQL. As a data storage solution we use a **R**elational **D**ata**b**ase **M**anagement **S**ystem. In comparison to only using a file sytem based approach it provides features like fast retrieving and searching for data and transactional reading and writing data. The language used for such operations by most RDBMS is SQL, i.e., **S**tructured **Q**uery **L**anguage.

The above short topic overview of technologies used during the development of web-based applications is only a glimpse. Every of the above mentioned elements is a complex topic by itself easily filling books. Each of the above mentioned topics uses its domain specific language and technologies thus making the technology stack a heterogeneous mixture of technologies.

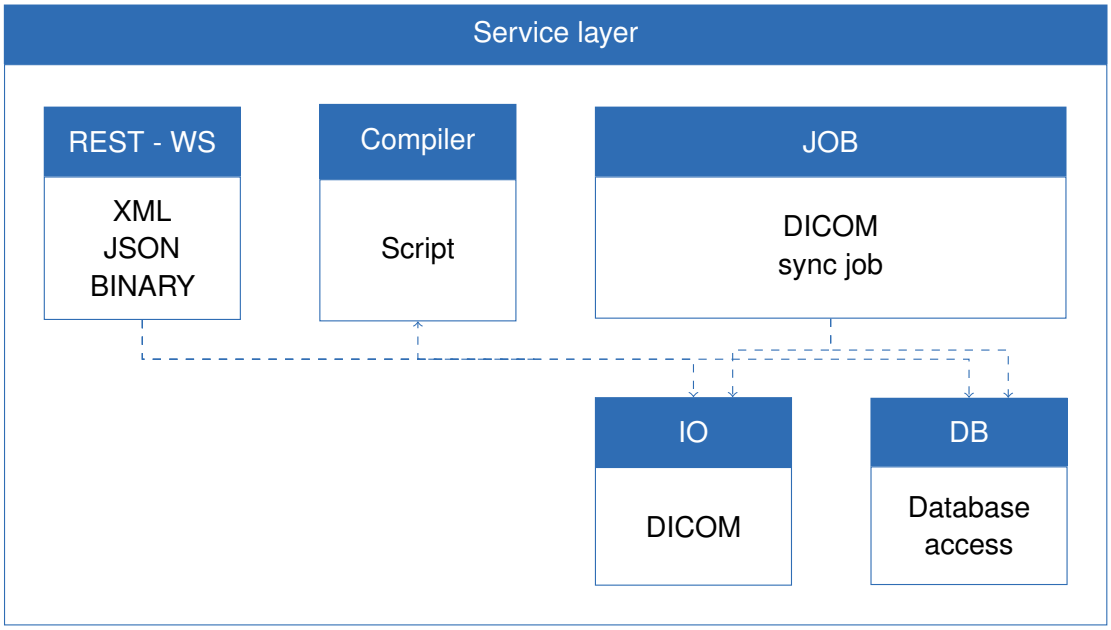


Figure 3.4: Schematics of the implemented service layer of the web based DICOM viewer. It consists of loosely coupled subsystems interacting with each other.

3.1.2 Business/Service Layer

The business/service layer is the most important one since it implements the logic of our software realization. It is the functional core responsible for user login, scanning DICOM directories and accessing the database in order to provide requested data. This layer uses many subsequently described technologies. As shown in Figure 3.4 it consists of three main parts, the ActiveDICOM Script compiler, a directory synchronization job and REST web-services for the embedded DICOM viewer.

It also consists of a layer interacting with the database and file system. It provides transactional support and logging and an important component which is used to initialize the application at startup. This component, called the startup listener, is also responsible for shutting down gracefully when the server is terminated. Localization is also handled within this layer. Each caption, label, or text presented to the user is translated by this layer to enable changing the language at runtime.

The embedded DICOM viewer, which is the main topic of Section 3.2, is implemented in JavaScript and ADICT using REST web-services in order to fetch important data from the server. These services are also part of the business layer. This layer is comprised of many loosely coupled services and interfaces, which as a whole define the so called business layer.

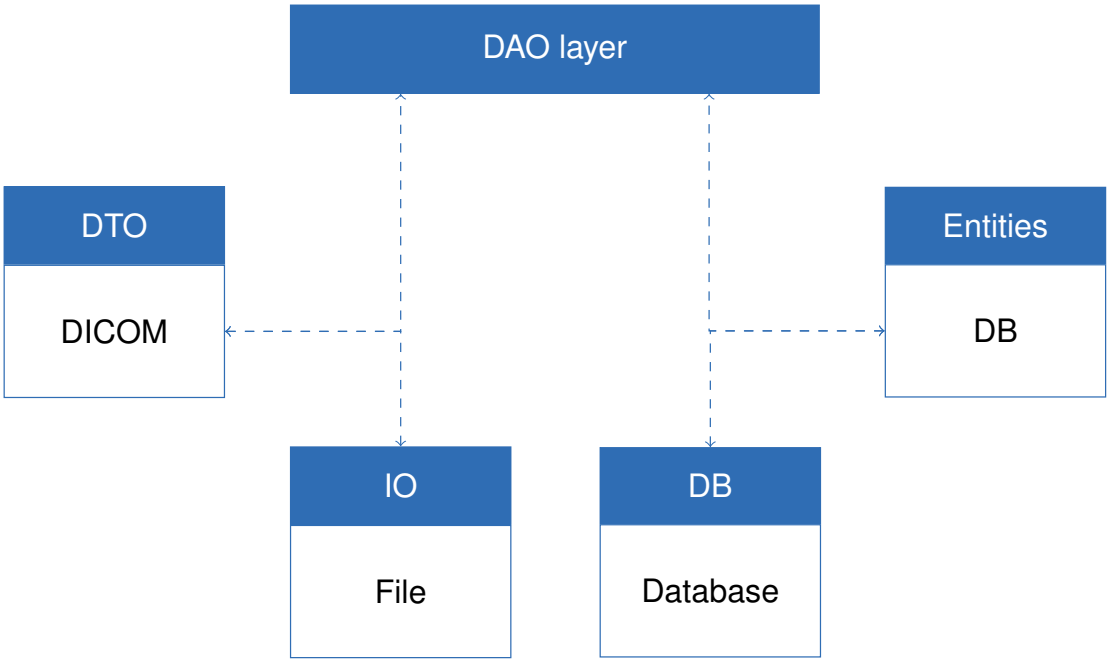


Figure 3.5: Schematic picture of the architecture on how data is accessed in our system. **D**ata **A**ccess **O**bjects (DAOs) access the file system and the database using entity objects and **D**ata **T**ransfer **O**bjects (DTOs). The service layer then uses these to further serve user or GUI requests.

3.1.3 Data Layer

In our system the data layer follows a tiered architecture. Figure 3.5 gives a high-level overview of the architecture of the data layer.

At the bottom, directly interacting or representing data, are the data access object entities, called DAO entities. Such objects are the bridge between the external data representation and the internal one. There are entities for file system access and ones for representing database tables or relations. Above that layer of entities resides the data access object layer. It implements basic input/output operations. In our case one such object is responsible for storing and retrieving DICOM related data to and from files. Other objects are responsible for searching and accessing database objects (DICOM image meta data, application users, etc.). At the top are objects providing higher level operations called from the business layer. In case of a database access such objects provide transactional support. Such objects basically wrap simple DAO operations inside a transaction combining them in order to accomplish a specific goal such as logging in a user.

Storing large binary data in a database was not an option for us which resulted in a mixed design using the file system and the database in combination. DICOM meta data is extracted periodically and synchronized with what is stored in the database. This makes it much faster to retrieve information such as the patient name, image modality and many other important

Name	Description
Vaadin [22]	Vaadin is a GUI framework which simplifies development because it allows for the development of applications in “pure” Java just like Swing applications. It is based on the Google Web Toolkit and therefore extensively uses AJAX and JSON to create an interactive RIA.
Three.js [11]	Three.js is an engine simplifying the development of WebGL based applications. The whole DICOM viewer window (see Figures 3.8 and 3.9) is implemented using this library. It provides means to render textures and perform transformations, such as zooming and rotating.
jQuery [13] and jQueryUI [14]	jQuery is a very popular JavaScipt library providing cross browser support. Furthermore combined with jQueryUI it is possible to create modal windows (popups) and eases DOM manipulation significantly.
Java 1.8 [9]	Using the latest version of the Java language provides new features like lambdas and default implementations in interfaces.

Table 3.1: Technologies and frameworks used to create the ActiveDICOM viewer.

items. Many of the applied web-services try to use the information stored in the database for fast meta data retrieval.

3.1.4 Used Frameworks and Technologies

After describing how the application is designed we focus on what technologies/ frameworks were used. Tables 3.1 and 3.2 show the technologies used to implement the viewer application.

The next chapter will describe some features of the web application created using Java based technologies. ActiveDICOM is a web based viewer for DICOMs allowing one to add interactive elements embedded into the DICOM itself.

3.1.5 Summary

Developing web applications involves many different technologies ranging from HTML, CSS, JavaScript, and Java to frameworks, databases and SQL. Coping with such a diverse set of technologies, we tried to alleviate many complexities by choosing a framework allowing to develop like traditional applications. Our system features a tiered architecture leading to easier

Name	Description
MySQL [15]	MySQL is a widely used open source relational database management system. It is supported by Oracle, the developer of one of the most used relational database systems in the world. It features a GUI management environment, called Workbench, easing maintenance and management of such a system during development.
Hibernate [8]	Hibernate is a popular object relational (OR) mapper framework used in the Java-enterprise application-development. It provides means of mapping SQL result sets to Java objects. One of the benefits of using such a framework is the provided database independence. The application therefore can easily be ported from MySQL to PostgreSQL, for instance. To achieve such an abstraction, SQL is not directly used to query the database. We use Hibernate's own HQL query language.
dcm4che [16]	This is an open source DICOM library for Java. It provides a rich set of possible ways to manipulate and extract data from DICOM images.
JAXB [9]	JAXB is a Java library, also part of the JDK, for manipulating XML files. XML files are used to configure the application and annotate Java classes for REST services producing JSON.
Jersey [12]	Jersey is a RESTful web-service implementation for Java. REST web-services are lightweight and are used to be called from JavaScript for loading textures and DICOM related meta data for the viewer.
Tomcat [2]	Tomcat is a servlet container developed by the Apache Foundation. It provides the necessary environment in which a Java web application is able to run.
Ivy [1]	Ivy is a dependency management and built tool.
Eclipse [6]	This is a very popular development environment for various languages. It is very feature-rich and can be enhanced using plugins to suit ones needs (e.g., svn, code formatter, ivy dependency management, etc.).

Table 3.2: Technologies and frameworks used to create the ActiveDICOM viewer (cont.)

extension and maintenance. This integrates well to rapidly changing requirements and/or technologies.

3.2 The ActiveDICOM Viewer

This section gives an overview of the implemented DICOM viewer. It is going to show the main features of our system and describes how some of the examples were implemented. The ActiveDICOM viewer features a simple user management meaning users have to login to the application. The login screen (Figure 3.6a) is customizable. Thus the background might be changed in order to add system announcements, see Figure 3.6b. After logging into the system, one is presented with the main screen for searching through the DICOM database, see Figure 3.7.

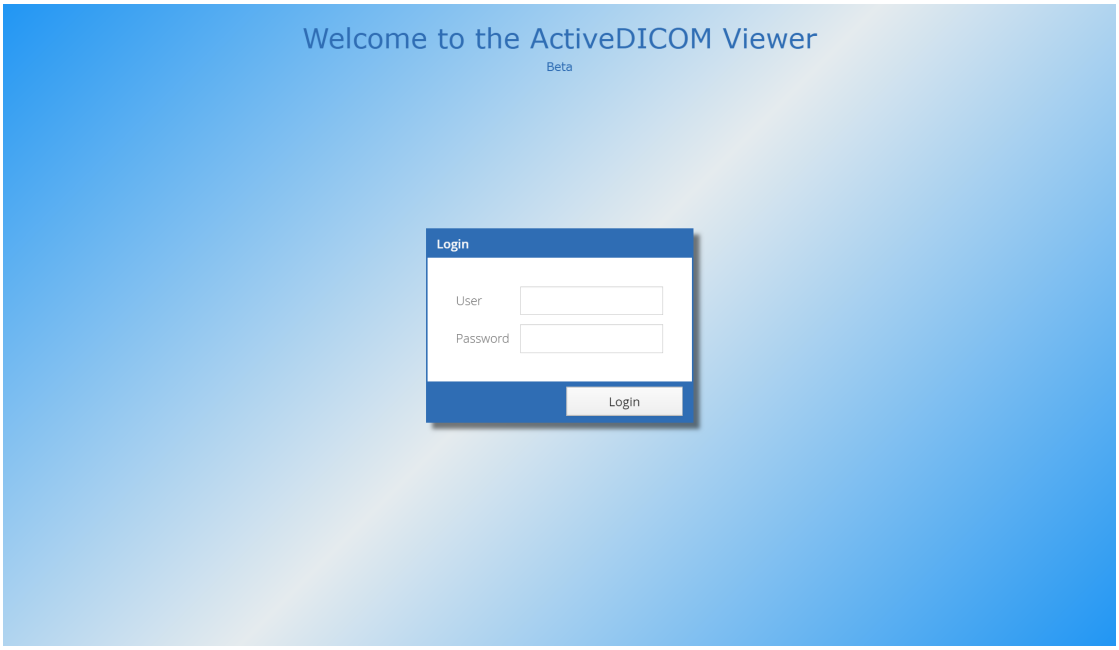
Our viewer features a streamlined interface without clutter and is divided into three parts, see Figure 3.7. On the left side of the screen resides the navigation area. It shows the application name, the user name and the available main activities. The user name is an input element allowing one to change the viewer language and log out from the system. The viewer is fully localized in German and English, with the latter as standard language. Furthermore, it is possible to change the language without logging off and on again.

On the top of the screen is the caption and below is the filter input shown. The 'x' button on the filter panel removes any filtering. By entering a patient name filtering is done by creating a specialized query for fetching data based on the submitted name.

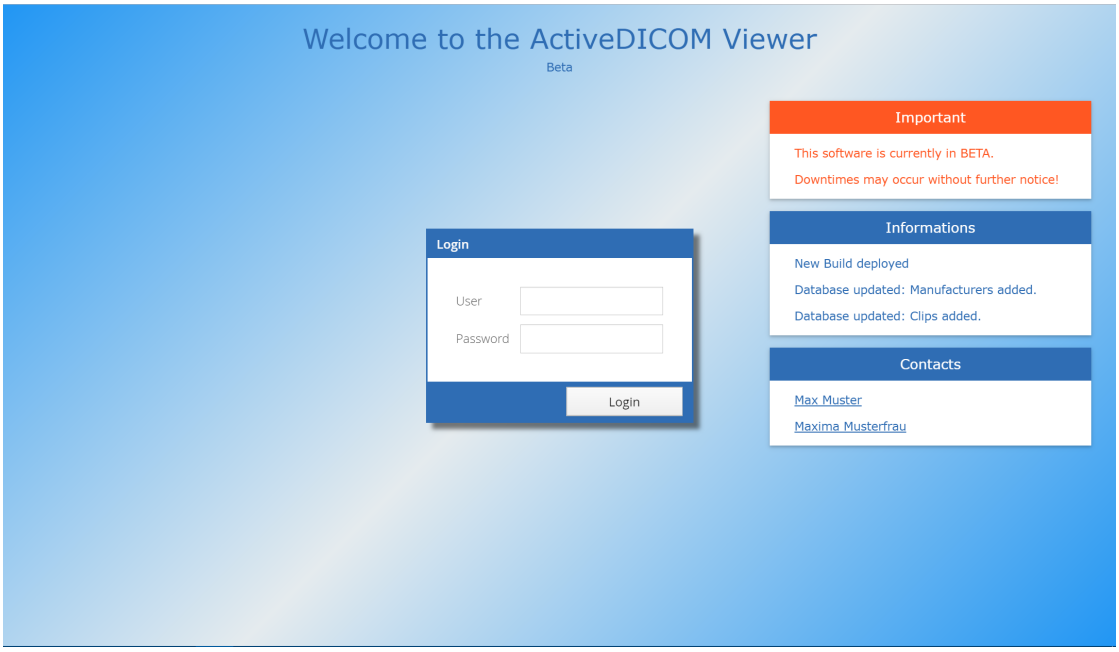
On the right side of the navigation area is the table showing the search result from the above filtering. It is a tree view giving a hierarchical overview of the DICOM images. All the images are ordered by patient, study and series. With the help of the magnifying glass on the right side of each table entry the implemented DICOM inspector is opened. It depends on the entry's type to determine the type of inspector being opened.

The standard DICOM inspector (Figure 3.8) displays to the user the selected medical image and does not allow changing images without closing the inspector. It is used to quickly view any DICOM image presented on the user interface. By selecting a node, such as a series or even the patient a more enhanced viewer is presented, which enables to quickly view various images of different scopes (see Figure 3.9).

The ActiveDICOM viewer features three different views for managing and browsing the available set of DICOM images. The first view named "DICOM Viewer" is the standard view suitable for larger data sets as it does not display small preview images (Figure 3.7). This view presents a tree-like structure for navigating to the desired images. The second view is a more advanced one allowing to manipulate DICOM images, such as adding scripts and embedded data (Figure 3.11). The third view, shown in Figure 3.10 provides a thumbnail-like overview without displaying too much information yet, but it allows to add scripts and embedded data, making it our preferred ActiveDICOM view. Each "card" displayed in Figure 3.10 represents a DICOM image. The left side shows a preview given standard values for window width and window center. In order to preserve style and layout, images are displayed as squares having width and height of equal value. On the right side, some important patient related information

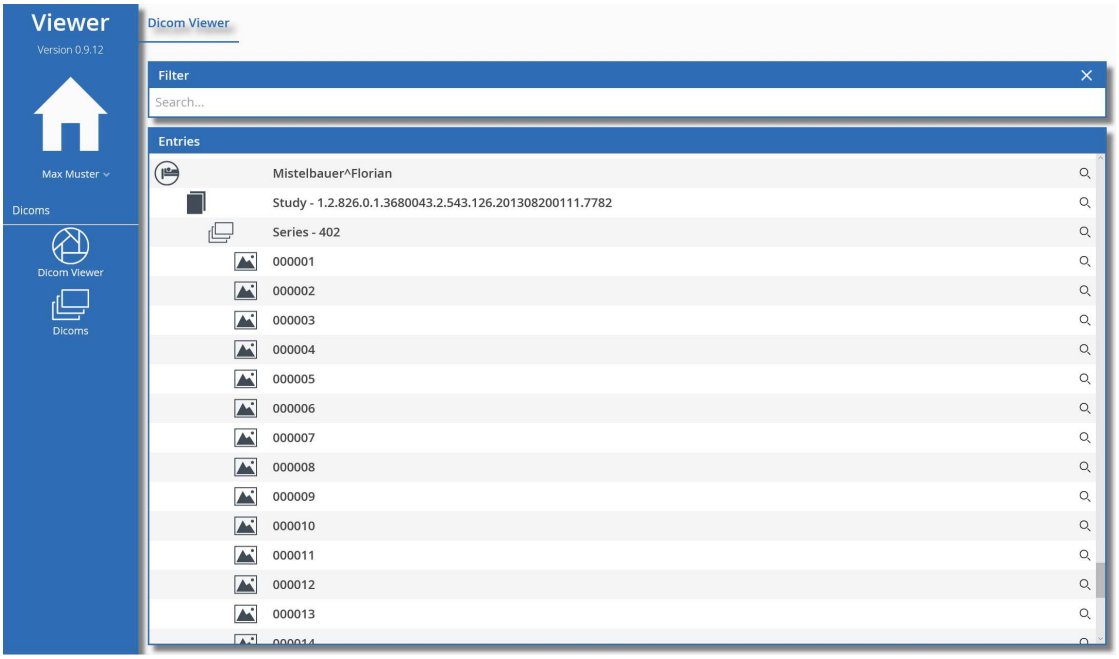


(a)

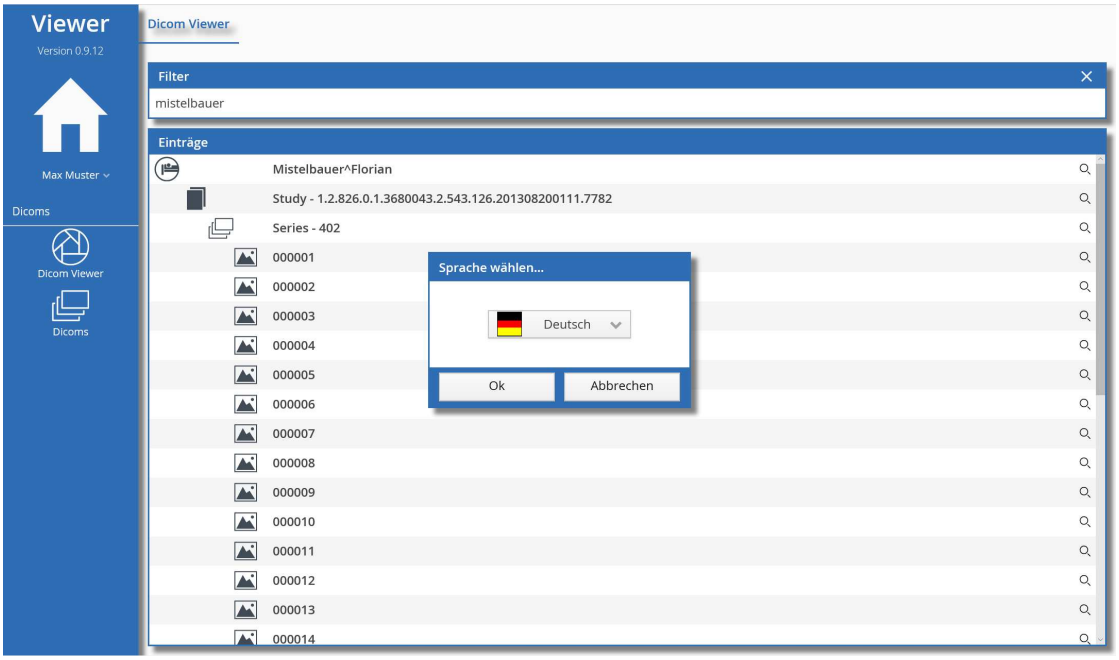


(b)

Figure 3.6: Login screen of the ActiveDICOM viewer. (a) shows the user name/password prompt without background alerts. (b) features announcements in the background.



(a) English version.



(b) German version.

Figure 3.7: Main screen for browsing through the DICOM repository. The repository is hierarchically ordered by patients, studies, and series. The German version also shows the language change dialog demonstrating the feature to change languages at any time.

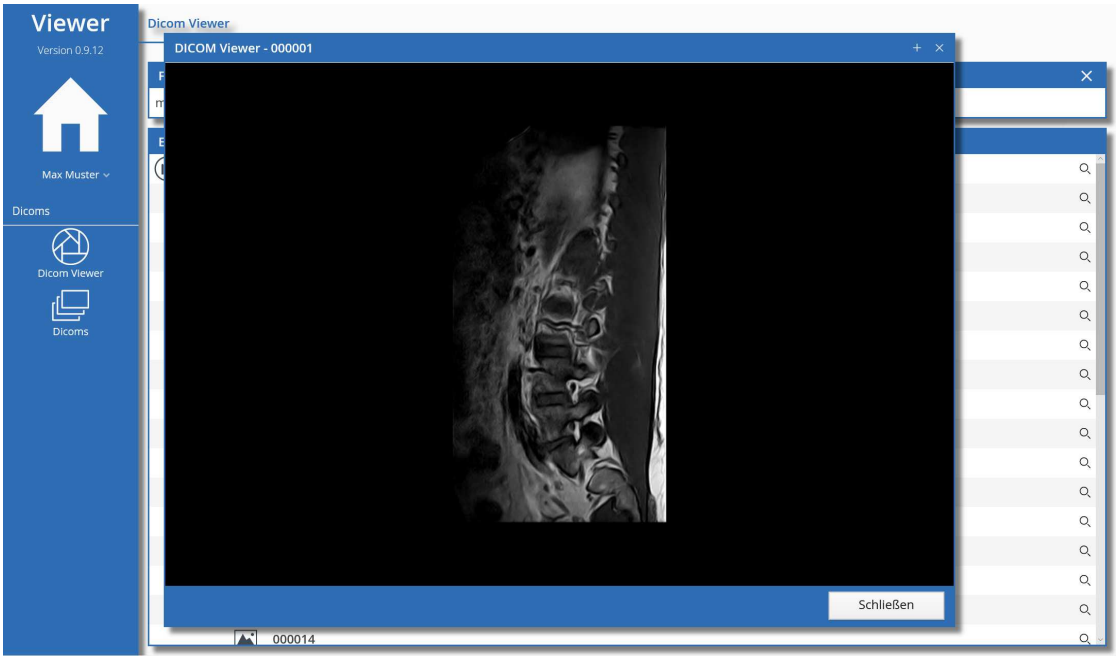


Figure 3.8: The standard viewer shown as a dialog after selecting an image. We call this view the DICOM inspector.

is displayed, such as the name, the image modality, the study UID and the series number. The last line depicts icons for advanced editing options.

All the data displayed in the mentioned views is retrieved from our database running on the server. A job running in the background keeps the database and a directory containing images synchronized. If images are deleted the job removes them from the database too. In case of an update, the job recognizes that an image was changed and replaces the corresponding one from the database with the updated image. In order to track file changes, a time stamp and a hash number based on the file are considered. The interval at which the job resynchronizes the file system content with the database can be specified in a separate and external configuration file.

3.2.1 Manipulating DICOM Images

In this section we are going to describe how images can be manipulated and enhanced with interactive features. In Figure 3.12 a single card is displayed. The icons at the bottom right side enable manipulation of DICOM images in various ways. These manipulations are our realization of ActiveDICOM.

The left most button opens the viewer as depicted in Figure 3.9. The next button, showing a note pad with a pencil, opens an editor window, which enables the user to add ADICT code. Code entered will be checked for syntactical errors prior to being written to the file. The next button, depicting a database symbol, opens the data editor. More details about the source

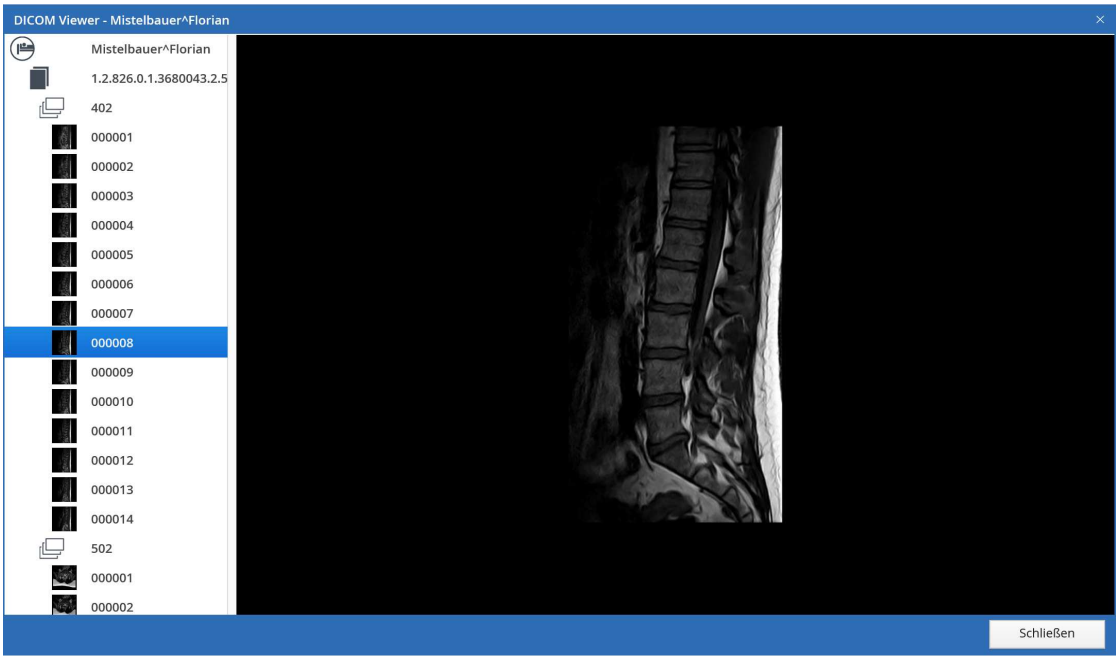


Figure 3.9: Selecting a patient, study, or series node shows a more elaborate viewer. It is now possible to quickly change images across series/studies.

editor will be given in section 3.2.1.3. To the right of the data editor button is the picture download one. Clicking it offers the possibility to convert the current DICOM image to a JPG with a certain window width and window center, see figure 3.13. The last two buttons produce high quality printed overviews of embedded scripts and data elements. These buttons are only present or enabled if such data is available. This makes it possible for the user to see whether a DICOM image is an ActiveDICOM or not.

3.2.1.1 Storing Data and Script

By using DICOM's data dictionary feature we are able to add arbitrary data, mostly in the form of unlimited text (UT), to images. It is basically a key-value based storage allowing for flexible extensions while preserving compatibility with traditional viewers. Our viewer software stores scripts in the entry $(1111_{16}, 0001_{16})$ whereas annotated data is stored in an extra "segment", $(1111_{16}, 0003_{16})$. Keeping code and data separated was implemented due to performance considerations as the size of the data section tends to grow significantly faster with the added support of large media files. Figure 3.14 gives an overview of how and where scripts and data are stored within a DICOM file.

3.2.1.2 The Data Editor

The data editor is one of the main features of our contribution. It allows the user to add arbitrary data elements to DICOM images. Such added data can be annotated and is referenced

Figure 3.10: This view organizes the DICOM library as a set of cards. Each of the cards displays a small image and only the most important information of each DICOM. It still allows for more customization than other views and enables the user to modify the medical images.

by a unique name. Script and data are interpreted by the developed domain specific language, ADICT. One of the important features is that even DICOM images might be added and then referenced using the scripting language. They can be interactively displayed and viewed by the user.

Starting with an empty set of data the editor shows a dialog with no entries, see Figure 3.15. The inner panel area will display the embedded data elements. In order to add a new entry the "+" button on the top right panel caption should be used. The "x" button to the right of the "+" button will remove all entries. After pressing the button for adding new entries another dialog will be shown, see Figure 3.16.

An entry needs a unique name and a mime type. While the mime type can be omitted, as it will be determined later on when data has been uploaded, the name is mandatory. After confirming the new entry the editor displays a line which is only a partial entry since no data has been added yet, see Figure 3.17a.

Figure 3.17b shows the successful upload of data and the determined mime type. The mime type is important to specify how the uploaded data should be treated if it is to be displayed. Finally, an example image with many embedded entries is shown in Figure 3.18a. Internally, all these data entries are annotated objects written in ADICT. The data editor parses the contained code and displays it in a convenient way. Changes made are only persisted if the "Accept" button is pressed. Until then the modifications are in memory and not written to the file. Prior to storing the data, the editor is generating code, encodes the data as a base64

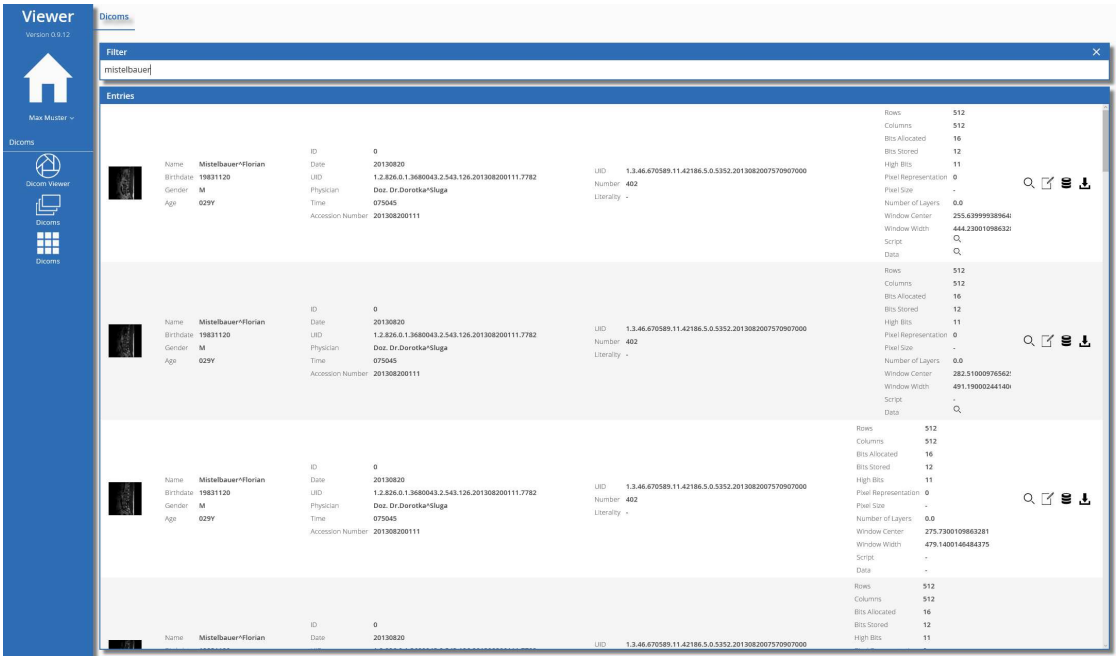


Figure 3.11: This view is used to display as much information as possible while showing a preview image of the DICOM. It also features the most options compared to the other views.

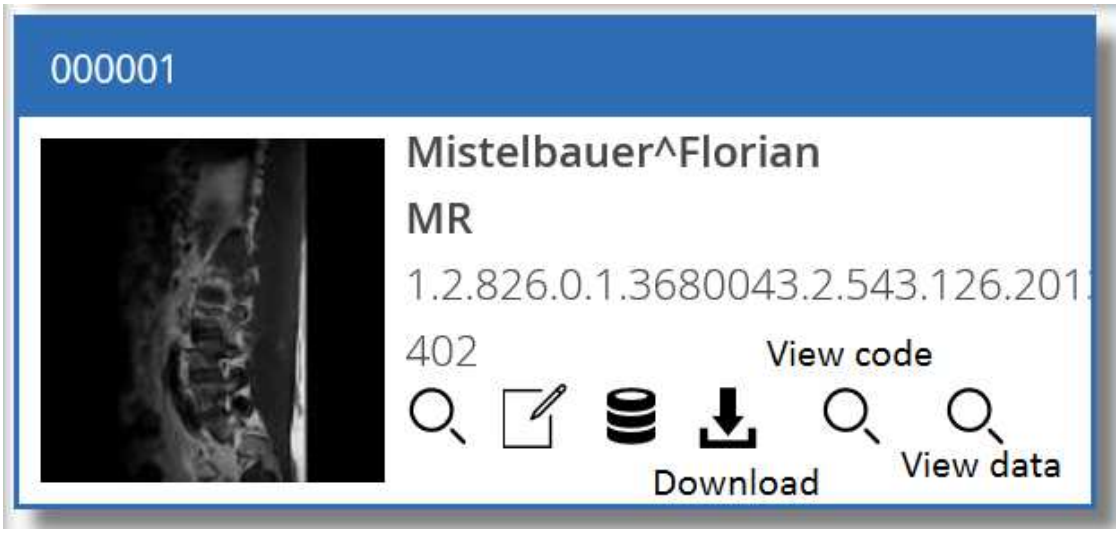


Figure 3.12: Image of a card showing all image manipulation/viewing buttons enabled.

string and writes it to the DICOM image file. Figure 3.18b shows how data looks like in the scripting language notation. In the above example an image is displayed containing other DICOM images (having the generic mime type application/octed-stream). Each entry can be edited, downloaded or removed. The next sections are going to demonstrate how such images

(a)

(b)

Figure 3.13: Downloading an image with a user defined window width and window center. (a) shows the dialog for specifying window width and center. (b) shows the exported image.

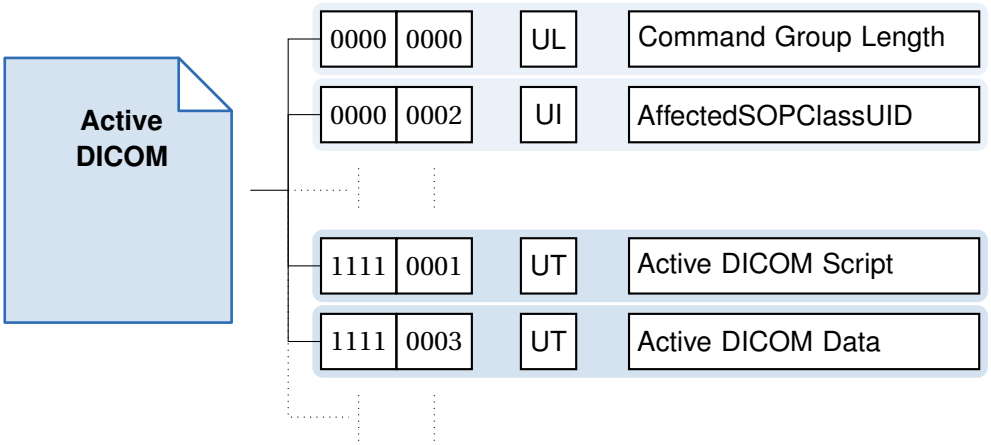


Figure 3.14: ActiveDICOM data dictionary and tags. A tag consists of a group and an element number, a value representation (*UI* for unique identifier, *UL* for unsigned long and *UT* for unlimited text) and a description. Two new odd DICOM tags are used in order to embed the code and data. Both are coded using our domain specific scripting language, mapping several interactive elements to data, such as images or other multimedia objects. By using these propriety tags the integrity to common DICOM viewers is preserved, since they will ignore them. Only if accessed with our web-based viewer, these tags will be extracted and translated into the respective output format.

are created, giving two examples, a simple one and a complex one.

3.2.1.3 Scripting - Introduction

This section gives an introduction to scripting. Figure 3.19 shows the developed code throughout this section's example. The following basic building blocks are demonstrated:

- Setting the window center and window height to a specific value once the image is displayed.
- Creating text overlays.
- Rendering geometric primitives.
- Guiding the user's attention to a specific point- or region-of-interest.

Adding scripts to DICOM images is done via the script editor. It can be opened by clicking the pencil symbol shown in Figure 3.12. The first line in a script has to be the string **#!ADICT**. This line tells the compiler, which language to use and that this script should be compiled before displaying the image on the screen (see line 1 in Figure 3.19a). The system library, **sys**, provides useful functions like **setCurrentDicomWC** and **setCurrentDicomWW** (see lines 4 and 7 in Figure 3.19a). Both functions need one parameter to change the window center or width, see lines 1 and 2 in Figure 3.19b.

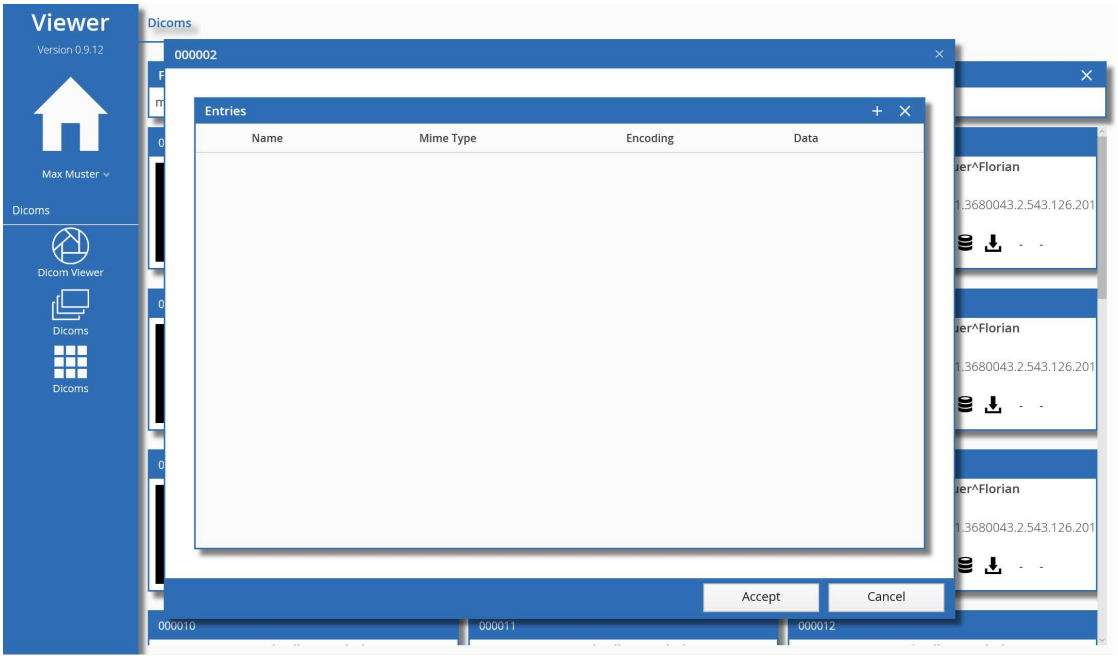


Figure 3.15: The data editor, showing that no data was added to the DICOM yet.

Next, the viewer's attention is guided to a specific region. In this example, we show an arbitrary region that has been selected for demonstration purposes, ignoring medical relevance. Directing the attention is achieved by displaying an arrow over the DICOM image pointing to an area-of-interest, see line 3 in Figure 3.19b. The region is highlighted by drawing a circle, see line 4 in Figure 3.19b. The library **poi** (points-of-interest) implements functions like **createArrow**, **createCircle** and **textOut**. Displaying text is done using the library function **textOut**. Printing text is implemented by creating a texture, rendering text onto it, and finally rendering the texture onto the main canvas element (containing the DICOM image). All this is done using WebGL. The language's close resemblance to JavaScript should make it easy to understand the examples shown and the language as a whole. Figure 3.19c, demonstrates the final ActiveDICOM of the first example. The function for drawing circles can also be used to create arbitrary polygons.

This example should demonstrate how scripting is done and what kind of images can be created. However, the presented image is static. The next example describes how to add interactive capabilities.

3.2.1.4 Scripting - Actions

Although printing arbitrary geometric primitives is one of the many features of ActiveDICOM, having interactive elements would further improve the user's experience. Example one is extended in the following ways:

- Set a timer for the arrow to appear.

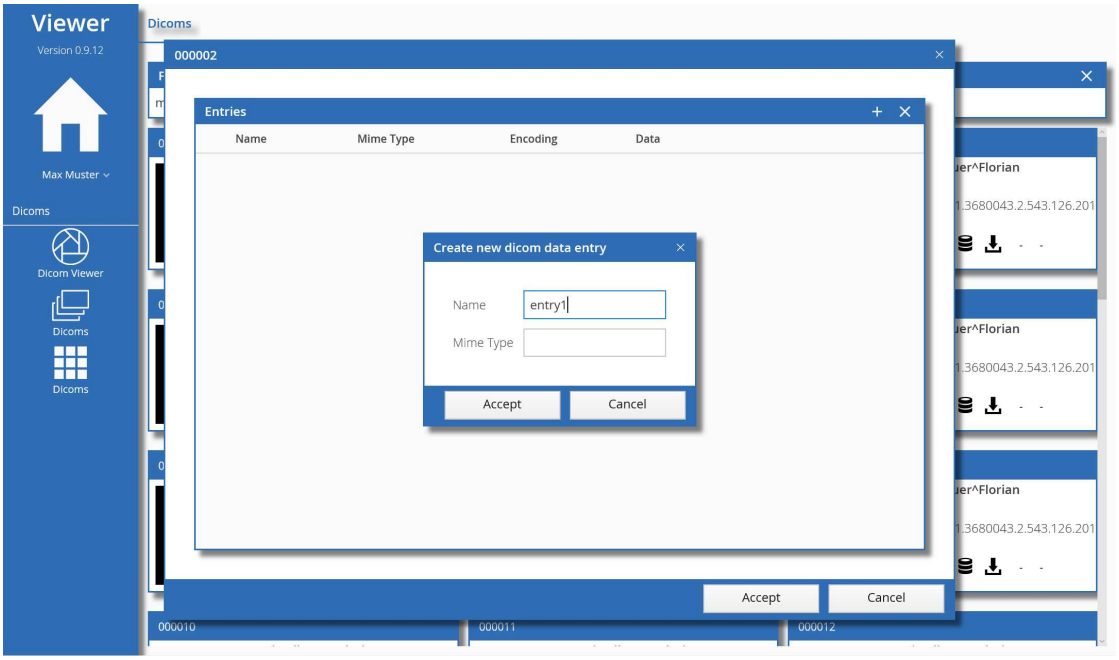
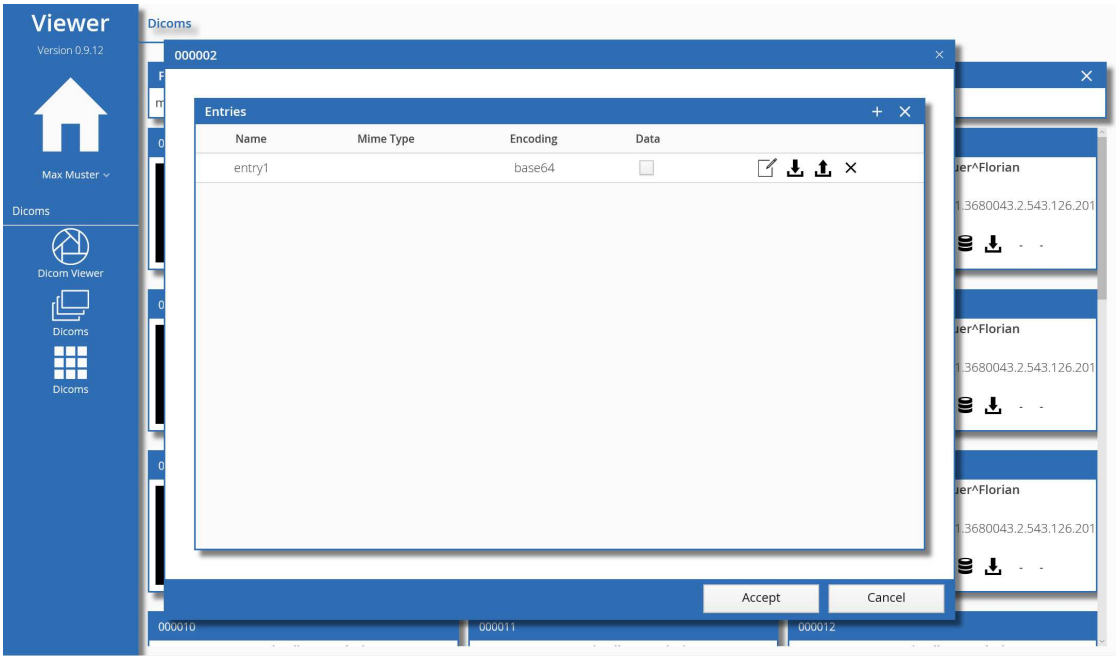


Figure 3.16: Adding a new entry to the DICOM image using the data editor.

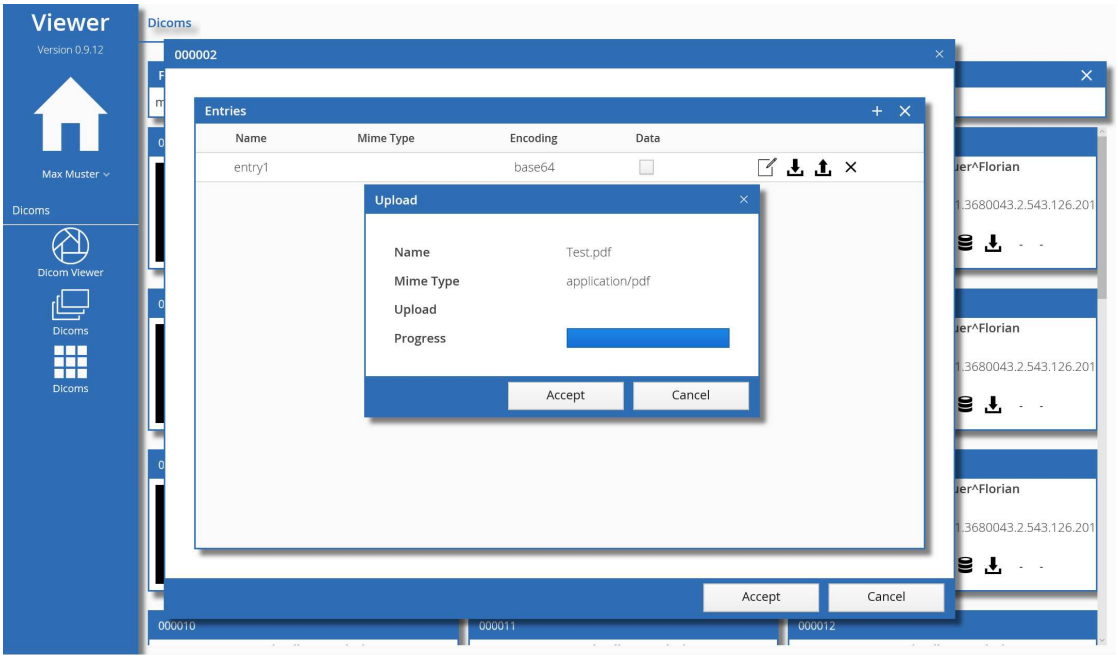
- Add an animation to the arrow.
- Add a click handler to the circle.
- Draw more circles/geometric primitives if the circle is clicked.

We take the code from the previous example (see Figure 3.19a) and add more functionality. First we add a timer to spedify when all the active/interactive elements should appear. Even the change of the windowing center and width should happen after three seconds. Since we specifically designed ADICT to offer the possibility to directly call JavaScript functions, we use the **setTimeout** function. With the help of **lambda** definitions and invocations one is able to implement event handlers. The blink animation for the arrow is done with the use of the **setInterval** function.

After three seconds(see Listing 1, line 36) the script is executed and shows a blinking arrow with the first circle. Clicking this circle, the second circle and text is displayed. Clicking the second circle shows the third one and more text to the right of the circle. This example shows how to guide the user through a sequence of points-of-interest. All this is encoded into a single DICOM image. Figure 3.20 shows the outcome of this slightly more complex example. This example demonstrates how one can direct the user's attention to certain points, called points-of-interest. A menu-like structure might be build using such primitives, which can be toggled when needed or interacted with.

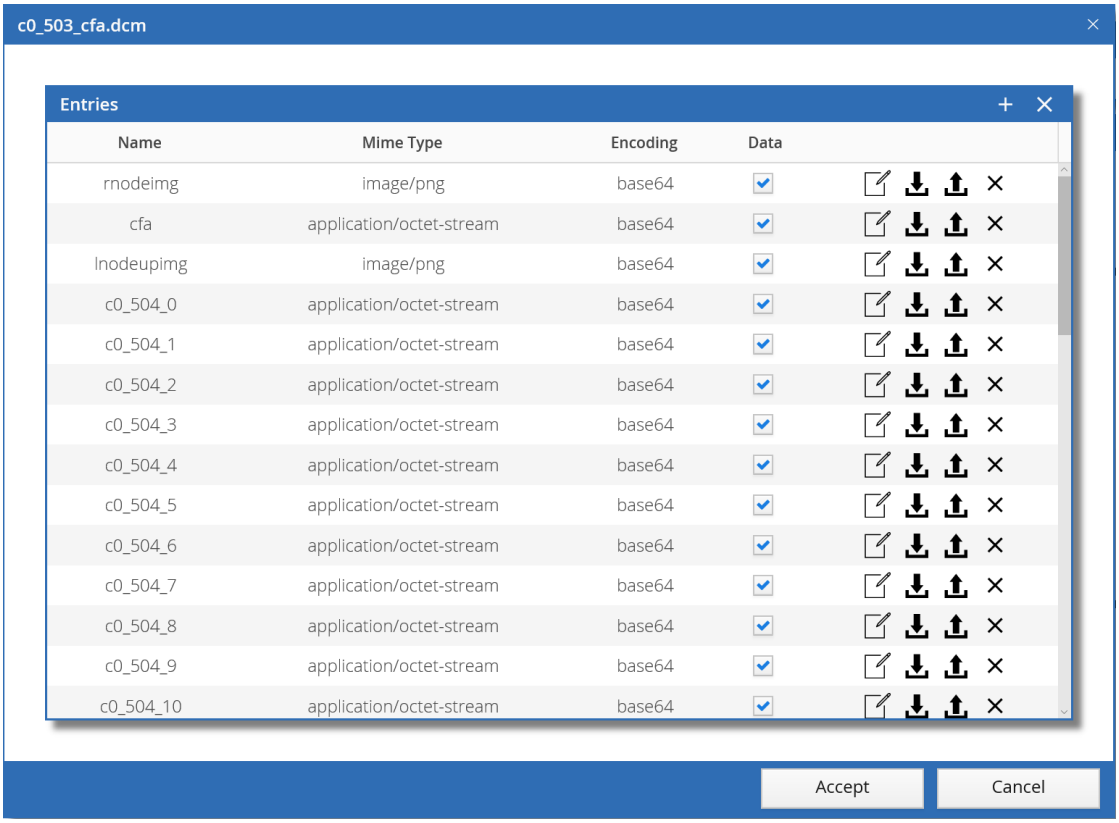


(a)

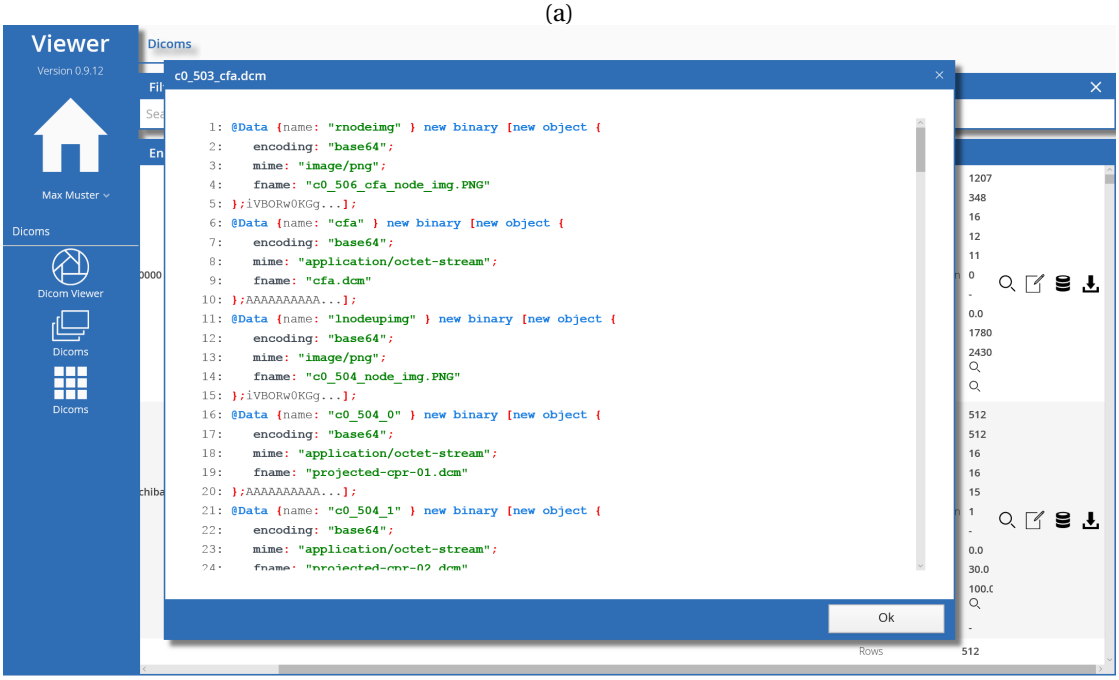


(b)

Figure 3.17: Uploading data and determining the mime type. (a) displays a partial entry. No data has been added yet and the mime type is not set yet. (b) shows that data was uploaded and the mime type was automatically determined.

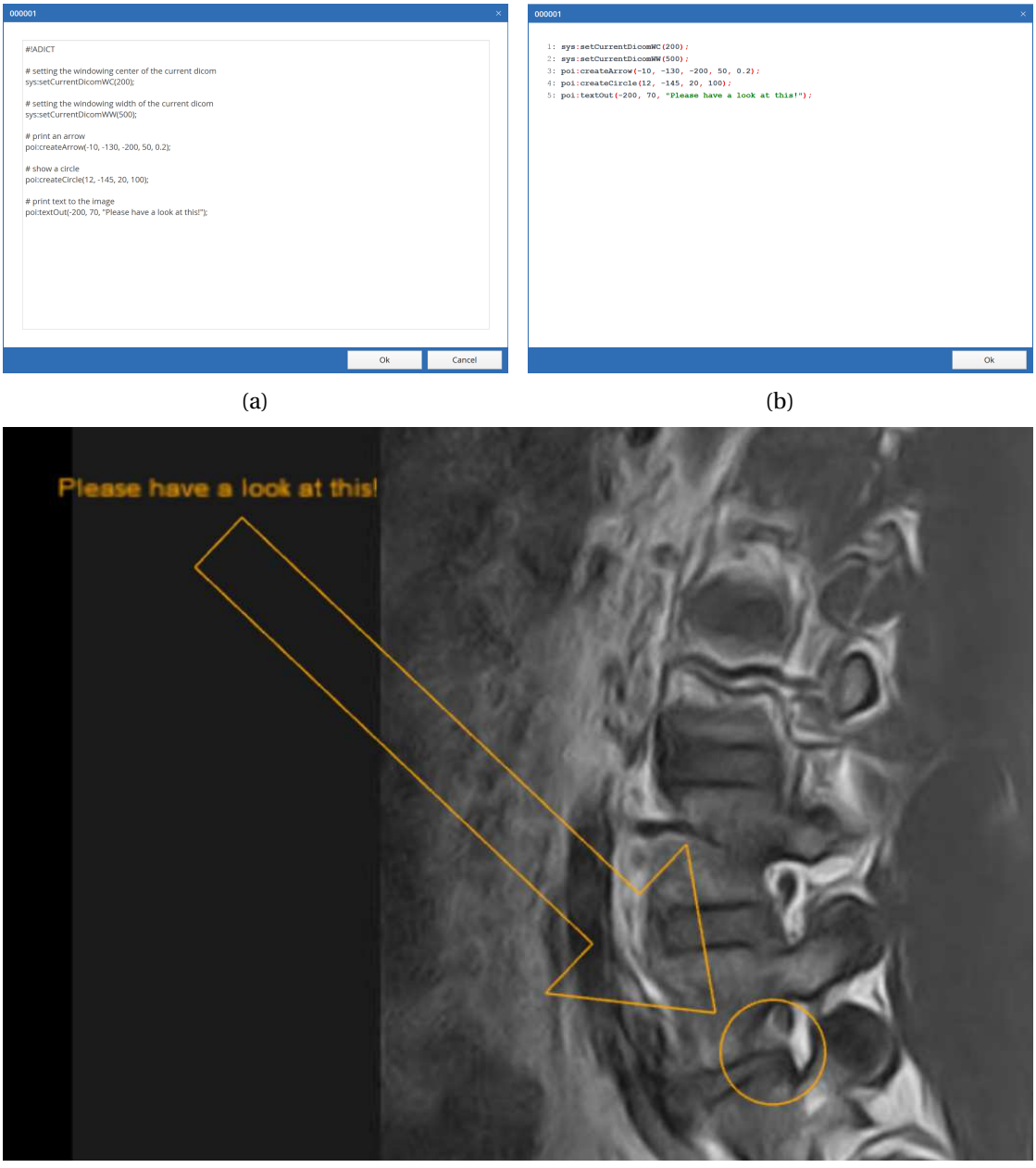


(a)



(b)

Figure 3.18: (a) shows the data editor displaying many entries embedded into a single DICOM image. (b) shows syntax highlighted data entries embedded into a DICOM image.



(a) (b)

(c)

Figure 3.19: First example of an ActiveDICOM. (a) presents the code editor showing the code of the first example. (b) shows the high quality printed and highlighted code of the first example. (c) displays the finished ActiveDICOM of the first example. It demonstrates how text and various geometric primitives are created and rendered.

Listing 1: Script of the second example.

```
#!ADICT
setTimeout(lambda() {
    sys:setCurrentDicomWC(200);
    sys:setCurrentDicomWW(500);
    local a1=poi:createArrow(-10, -130, -200, 50, 0.2);
    poi:textOut(-200, 70, "Please have a look at this!");
    local c1=poi:createCircle(12, -145, 12, 100);
    local c2=poi:createCircle(8, -95, 12, 100);
    c2:visible = false;
    local c3=poi:createCircle(8, -35, 12, 100);
    c3:visible = false;
    local l1 = poi:line(8, -95, 12, -145);
    l1:visible = false;
    local l2 = poi:line(8, -35, 8, -95);
    l2:visible = false;
    local t2 = poi:textOut(100, -95, "Please have a closer
        look at this!");
    t2:visible = false;
    local t3 = poi:textOut(100, -35, "Please have a closer
        look at this too!");
    t3:visible = false;
    setInterval(lambda() {
        a1:visible = not (a1:visible);
    }, 500);
    c1:click = lambda() {
        c2:visible = not (c2:visible);
        c3:visible = false;
        l1:visible = c2:visible;
        t2:visible = c2:visible;
        l2:visible = false;
        t3:visible = false;
    };
    c2:click = lambda() {
        c3:visible = not (c3:visible);
        l2:visible = c3:visible;
        t3:visible = c3:visible;
    };
}, 3000);
```

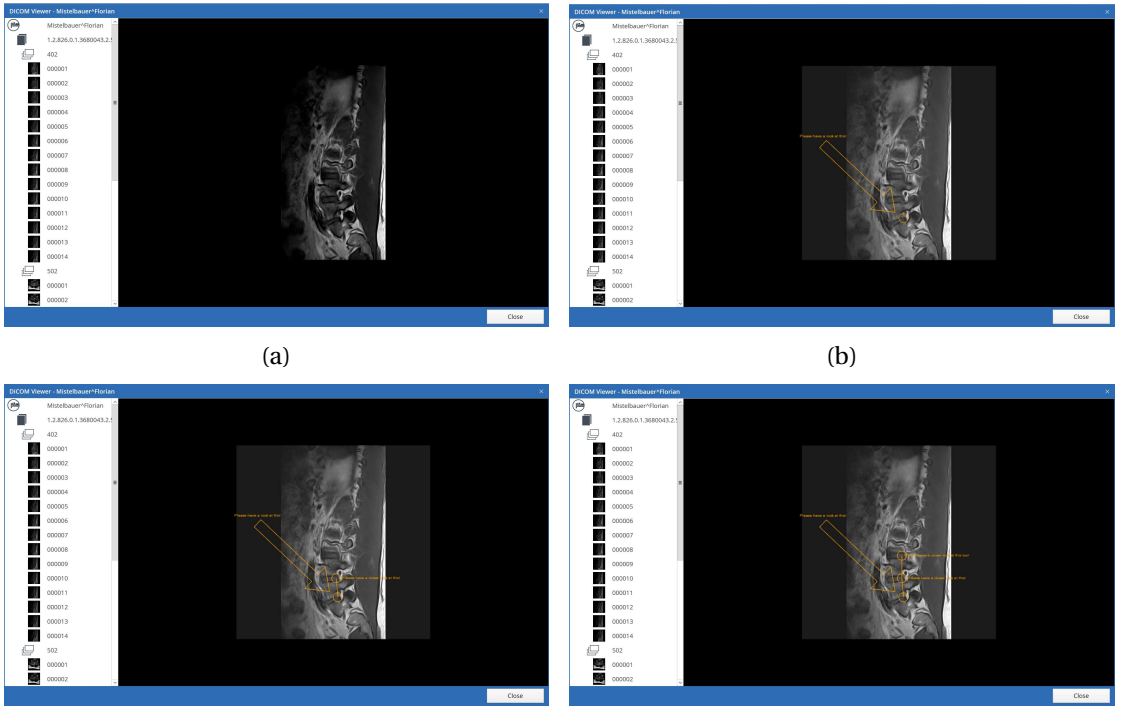
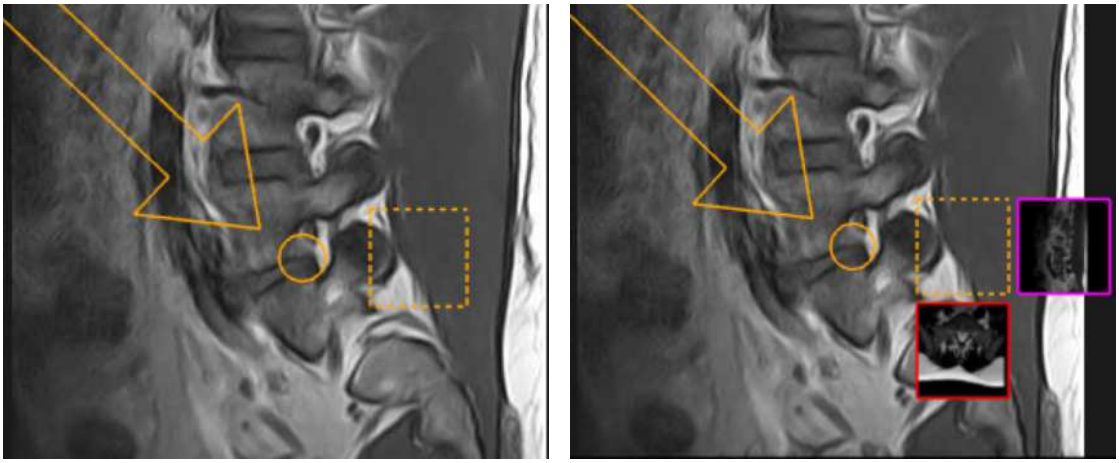


Figure 3.20: Showcase of the second example. (a) displays the original image, with standard window width and center. No geometric primitives are shown. (b) after three seconds the window width and center are changed. Text, a blinking arrow, and a circle appear. (c) clicking the first circle draws a line to the second one and prints text right next to the second circle as well. (d) the second circle behaves just like the first one. Clicking it reveals the third circle and the third text message right next to it.



(a) (b)

Figure 3.21: Demonstrating the grid menu. (a) Initial grid, not expanded. (b) Expanded grid menu showing textured nodes using DICOM images as textures.

3.2.1.5 Scripting - Advanced Example

The subsequent example extends the second one with the following goals:

- Create a complex menu-like structure.
- Display sub-views.
- Use various embedded objects such as images, DICOMs, and PDF documents.
- Allow the user to change the window function even for embedded elements.
- Embed a video file and play it back within a sub-view.

Figure 3.20 shows the starting point for this advanced example. One can use graphic primitives to guide the user through important elements in the DICOM. The first task is to implement click callbacks for every circle, using lambda functions. Each click should act as a toggle and influence the visibility of the next or previous circle, thus providing a menu-like structure. The next step is, to place right next to the circles a clickable grid menu, using the library call **menu:create(x, y)**. This grid menu handles toggling and clicking automatically and even supports being textured with images (see Figure 3.21).

New nodes are added to the menu with **m:root:leaf(x, y)**. x and y are not screen coordinates but grid coordinates and m is a variable holding a reference to a menu created via **menu:create**.

In order to add multi-media elements such as other DICOMs, movies, images, or documents, the **media** library was implemented and can be invoked just like the system library. It provides means for creating dialogs used to display embedded elements (see Listing 2).

Listing 2: Code snipped demonstrating the use of the media library.

```
# right subview
local right_1_media = media:create();
# setting width and height of the dialog
right_1_media:width(256):height(300);
right_1_media:init();
right_1_media:title("Series 3");
right_1_media:label("MR");
right_1_media:position(72, -145);
# initially hide the dialog
right_1_media:hide();
# set the frame color of the dialog to a fancy one
right_1_media:frameColor(right_1:frameColor);
# embedding elements...
right_1_media:lib():embeddedDicom("dcm_s3_1");
right_1_media:lib():embeddedDicom("dcm_s3_2");
right_1_media:lib():embeddedDicom("dcm_s3_3");
right_1_media:lib():embeddedDicom("dcm_s3_4");
# display the first media element upon opening
right_1_media:first();
```

As mentioned above, it is possible to add arbitrary annotated data to ActiveDICOM. Each such annotated data needs a name, with which it is referenced using the media library functions. The above code example, Listing 2, embeds another series of DICOM images. The dialog, displaying the images supports to change the window width and center just like the main screen does.

Embedding a video is done using **_name:lib():embeddedVideo("<name of the video>")**, see Listing 3.

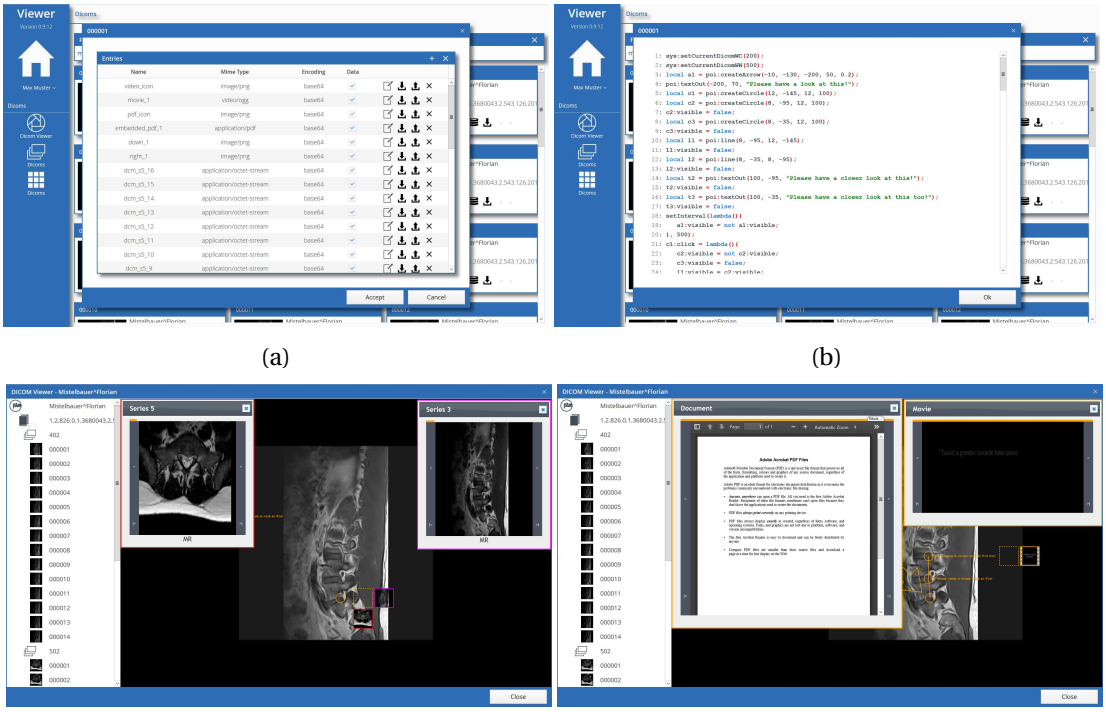
Listing 3: Code snipped showing how a video is embedded.

```
right_3_media:lib():embeddedVideo("movie_1");
```

We now summarize how an ActiveDICOM is created. First, one has to create the necessary elements which should be embedded. Then, with the use of the data editor, the data has to be added to the dicom. Then the script has to be written and added to the DICOM. Figure 3.22 shows this work-flow step-by-step and Figures 3.22c and 3.22d show the results.

3.3 Summary

In this chapter the implemented web-based viewer has been presented. We discussed what kind of technology has been used. Based on a Java stack using a server side programming



(a) (b)

(c) (d)

Figure 3.22: Creating an ActiveDICOM using our developed tools and ideas. The work-flow is as follows: (a) adding data using the data editor. (b) creating a script using the code editor. (c) viewing and interacting with the DICOM. (d) additionally, viewing multi-media data.

model we implemented the application. The language compiler was implemented in Java as well and generates JavaScript code. MySQL was used as the database system and dcm4che was used to read DICOM files. The implemented DICOM viewer was implemented using WebGL and JavaScript. RESTful web services were used to access various data elements such as embedded elements, scripts and images, or videos.

The main language used to implement the viewer was Java 1.8. It features a few new concepts, such as lambdas and default implementations for interfaces. Screenshots throughout this chapter were produced by the application running on an Apache Tomcat 8. As a graphical development environment *eclipse* was used. All Java modules were implemented and executed using Oracle JDK 1.8.

Aside from a technological overview, we demonstrated how DICOM images might be enhanced with multimedia and even other DICOM images. The embedded assets are then glued together by a scripting language. With such a language it is possible to implement complex interactive scenarios and guide the attention of medical staff to certain points or areas of interest.

CHAPTER

4 ActiveDICOM Script (ADICT)

4.1 Overview

This chapter gives a detailed description of the **A**ctive **DIC**om scrip**T** language, abbreviated as **ADICT**. We explicitly designed our language with an easy, yet powerful syntax in mind. Furthermore, the language had to meet the following design goals:

- **Powerful syntax.** The language should be quite expressive and should allow the user function definitions wherever possible. Satisfying this goal, should not make the language too complex either, meaning that there should be a balance between the complexity of the language and its expressiveness.
- **Extensibility.** ADICT should be easily extensible. Due to its close relation to ECMA-Script [31] no library or module constructs or definitions are needed. This may change in future improvements, software iterations or feature additions.
- **Modern.** The designed language has to feature modern concepts, such as lambda definitions, and should support various technologies right out of the box without the need of additional extensions.
- **Easy code generation.** This point directly implies the previous points and many other design implications of the language. During development is was necessary to make a few simplifications in order to not spend too much time implementing a compiler. The focus of our domain specific language is the augmentation of images with interactions, the ease of describing these interactions and the ease of connecting interactions and data. A complex language might increase the complexity of code generation, also making it more difficult to match source language and target language.

ADICT was heavily influenced by ECMA-Script [31]. This lead to a quite powerful syntax and an easy code generation. The prime target language is JavaScript, meaning many features present in JavaScript are present in ADICT too. Section 4.6 describes how the language was implemented and how code generation is handled. Even tough ADICT is a full featured pro-

gramming language it was first and foremost designed to add animations and user interactions to DICOM images. Section 4.8 describes limitations of ADICT and implementation choices, which may pose some restrictions on the code written in ADICT. Chapter 6 briefly describes, which future enhancements may be researched and realized concerning our ActiveDICOM system. A solid type system is only one of the areas for improvements and debate. In Section 4.2 we give a short description of the motivation leading to the development of our domain specific scripting language.

4.2 Motivation

The main goal is to add interactive features to DICOM. This can be achieved using various techniques and methods. We have chosen an approach, which might seem complex at a very first glance. In order to motivate this we look at possible advantages of the chosen approach (this is a short repetition of the above overview):

Flexibility. Flexibility is one of the most important advantages of creating a domain specific language, which does not trade-off to much power for ease of use. Combining a more general language construct to create new complex commands, animations, or interactions is much more efficient than to provide specific solutions for each possible type of animation or interaction.

Extensibility. New functions might be added using libraries. This might lead to functionality and combinations thereof not thought of. Providing the possibility to create new commands and thus extending the whole ActiveDICOM is tempting and might encourage others to create impressive new ideas built on top of our basis.

Adaptability. Changing demands and interests are a constant challenge, which might be countered with a flexible approach such as a language.

Power. Power refers to what a user can do and how much his/her creativity is restricted by the tools employed. Using a domain specific language like ADICT provides such a flexible and powerful tool, as it provides means for creating one's own tools and commands.

One advantage, not listed above, was that ADICT provides an abstraction layer. Using JavaScript directly might tie ActiveDICOM to tightly to web technologies or make it too cumbersome to implement interactions. Due to its tailored set of features, ADICT is lightweight and thus easier to implement and learn. Every idea or decision has downsides and disadvantages, some of them are the following:

Complexity. Many of the above mentioned advantages of creating a scripting language might lead to an increased complexity. A complex tool needs to be used by a very skilled user in order to produce the desired outcome. A programming language requires much more knowledge about computer science, which might not be needed for simple tasks implemented by a much easier approach. For example, a graphical user interface might be sufficient to create an interaction using predefined primitives/building blocks.

Prototyping. A scripting language without the support of an IDE might not be suited to produce a fast prototype of an idea.

Usability. Usability is a consequence of all the mentioned advantages and disadvantages. The more complex or flexible a tool is the more it tends to be difficult to handle or use. As mentioned in the description of complexity, programming knowledge might be necessary in order to achieve certain goals.

Now it is possible to answer the guiding question of this short section. A language offers flexibility, from which adaptability may be derived as well. This means that the chosen approach may be able to adapt to changing user interests and demands more easily than other approaches. The user has many possibilities to customize animations to suit his/her needs or implement totally new ones not thought off at the very start of the project. Many projects implement languages to adapt to or incorporate new features. The following short list of examples demonstrates that such an approach was chosen by many.

HTML + JS. In the world wide web and its underlying technology, HTML, JavaScript is used to add interactive elements to the otherwise quite static HTML technology. This is a quite simplified example, because with the use of plug-ins many other languages and extensions might be used. With the help of JavaScript it is possible to add client side animations, checks and interactions. For further information we refer to previous work [7].

Flash. Flash is a software used for animations and interactive elements. It is executed in the Flash Player and, if it is embedded in web pages it is executed using a Flash plug-in. The language which is used to program Flash is called ActionScript (for further information see, for example [73]). This example shows that a language is used to add extra possibilities for advanced users, more complex animations, ideas, or even whole games.

Visual Basic for Applications (VBA). The Microsoft Office program suite allows the user to write macros and very advanced scripts using the scripting language Visual Basic for Applications (VBA). The interested reader may be referred to the following literature [79].

The above examples are only a very small selection of languages created to add interactions and animations to various entities or objects. Active Dicom Script is heavily influenced by many languages. In order to create a very efficient compiler the language is kept quite simple without syntactic “sugar”. Details about the implementation are described in Section 4.6. The following sections provides a brief introduction to ADICT.

4.3 JavaScript

This section gives a brief overview of JavaScript, a programming language popular in web development. Although gaining widespread use outside of browser and web development, it

Method	Description
Number.isInteger(number)	Returns true if the given arguments is an integer.
Number.parseFloat(string)	Converts the given string into a float.
Number.parseInt(string, radix)	Convert the given string respresentation with the given radix into an integer.

Table 4.1: Number-object's functions, a selection.

is still mostly used in web browsers. This section is based on the ECMAScript Standard [31]. In Chapter 4 Section 2 of the ECMAScript standard, a short overview of the language is provided. ECMAScript and JavaScript are used interchangeably, referring to the language which will be described.

ECMAScript is a language featuring object oriented features and functional ones as well. An object is a collection of properties, each having various attributes. A property may contain objects, primitive type values and functions. Primitive values might be of the following types: Undefinded, Null, Boolean, Number, String and Symbol. Just like in Java defined objects are of type Object. Functions defined within the context of an object are called methods. Listing 1 demonstrates the definition of a function, its invocation, and how object methods (line 2) are invoked/accessed.

Listing 1: JavaScript 'Hello, World'

```
1 function helloWorld() {
2         var greeting = 'Hello, World!';
3         console.log(greeting);
4 }
5 // invoking the function
6 helloWorld();
```

The language further specifies many built-in objects and methods. If run within a browser even more predefined objects are present. Table 4.1 shows a selection of number manipulation and conversion methods defined by the ECMAScript standard within the Number object/constructor. The math object defines many useful functions or constants for performing various calculations, see Table 4.2 for a selective list.

In line 2 of Listing 1 a local variable is defined. Due to being a dynamically typed language, no type is specified. The type is automatically derived from the assignment/initialization. Using the **typeof** operator one can query the type of an expression or object. Listing 2 shows the various types of ECMAScript.

Element	Description
Math.E	The Number value for e, the base of the natural logarithms, which is approximately 2.7182818284590452354.
Math.PI	The Number value for π, which is approximately 3.1415926535897932.
Math.abs(x)	Returns the absolute value of x.
Math.cos(x)	Returns the cosine of x. The parameter is expressed in radians.
Math.sin(x)	Returns the sine of x. The parameter is expressed in radians.

Table 4.2: Math-object's functions, a selection.

Listing 2: JavaScript typeof operator.

```
1 var greeting = 'Hello, World!';
2 console.log(typeof x);                    // 'string'
3 console.log(typeof 1);                    // 'number'
4 console.log(typeof function(){});         // 'function'
5 console.log(typeof [1, 2, 3]);            // 'object'
6 console.log(typeof {a: 1});               // 'object'
```

Line 5 in Listing 2 shows a special case. Arrays are treated as objects in JavaScript. Like in Java, arrays are objects having predefined methods, such as "length" or "forEach". The last example, Listing 3, demonstrates JavaScript's expressive power and array usage.

Listing 3: JavaScript array forEach

```
1 // printing all elements of an array to the console
2 var field = [1, 2, 3, 4, 5];
3 field.forEach(function(x) {
4     console.log(x);
5 });
```

ECMAScript is quite expressive and allows for powerful and flexible scripts. Lack of types and its hard-to-understand variable scoping rules make the language a powerful tool in the hands of experienced users, yet leading to hard-to-find bugs if not used properly. Due to its massive gain of popularity in todays web development, performance and type safety are often targeted for improvements. Improved virtual machines using just-in-time compilation, are addressing performance considerations. TypeScript, for instance, tries to introduce statically

typing to ECMAScript/JavaScript. For more information about TypeScript, the reader is referred to the TypeScript page [21].

4.4 Introduction to ADICT

This section describes the basic concepts of ADICT. More advanced language features are described in section 4.5 whereas this section covers variables, statements and various loop as well as conditional ones.

4.4.1 Introductory Example

Following a "tradition", we start to describe our language with our variation of the well known "Hello, World" example.

Listing 4: Hello World

```
1 # Calling the native JavaScript function 'console:log'
2 console:log("Hello, World");
```

The above example in Listing 4, demonstrates the following concepts or elements. Line comments are written using the character symbol #, which means that the compiler ignores the rest of the line, starting from the symbol # up to the next carriage return/line feed or end of file. A comment can consist of arbitrary symbols, even special characters or operator symbols. If multiple lines of code should be commented out then each of the desired lines must start with the line comment symbol. ADICT does not support multi-line comments like those in C++ or Java, for instance. The second line is a function call, which presents a message to the user containing the text "Hello, World". Every statement, such as a function call, must be terminated by a semicolon.

Each line can contain more than one statement, but for improved readability, lines are kept short and often contain one statement only. Since each statement has to be terminated by a semicolon, one can write as many statements within one line as desired. It is even possible to write a whole program in just one single line, but this is considered bad style and practice. The following example demonstrates how code should not be written due to its bad readability (see Listing 5).

Listing 5: Badly formatted code

```
1 Statement1; Statement2; Statement3;
2 Statement4;
3 Statement5;
4 {
5 Statement6;Statement7;Statement8;
6 }
```

The following example (Listing 6) demonstrates how code will be formatted in this thesis in order to enhance readability and understanding. Each block or block statement is indented using blanks or tabs. Comments are added to further support the understanding of code snippets, also see Listing 4 line 1.

Listing 6: Pretty printed code

```
1 Statement1;
2 Statement2;
3 Statement3;
4 {
5     # Comment describing the intention of Statement4
6     Statement4;
7     Statement5;
8     Statement6;
9 }
```

4.4.2 Variables and Types

The Active DICOM Script language is dynamically typed, which means that the programmer does not have to declare types. Conversions between types are handled automatically by the system. Internally, types still exist and certain operations are only supported by certain types. The following code snippets show the different basic types recognized by ADICT and demonstrate how variables are defined. See Listing 7 for variable names and definitions and Listing 8 for illustrating different implicit types.

Listing 7: Defining variables

```
1 local a;        # valid name
2 local b;        # valid name
3 local 4;        # invalid name!
4 local _4;       # valid name
```

Defining a variable is done using the keyword **local**. A variable name has to start with a letter or _. Special characters and keywords can not be used within a variable name either. Once a variable is defined, a value can be assigned to it using the assignment operator, =. As mentioned above, the language is dynamically typed meaning that the system infers the type of the variable based on the assigned value (see Listing 8).

Listing 8: Assigning values to variables

```
1 local a = nil;              # assigning the empty reference
2 local b = 1;                # number assigned
3 local c = "Hello, World"; # string value assigned
4 local t = true;             # boolean constant true
5 local d = b;                # d has the value of b
```

Variables are used to store data, which might be needed in subsequent operations. They can be used in operations such as mathematical expressions and conditional statements. The next sections gives a short introduction into expressions.

4.4.3 Basic Expressions

Mathematical expressions use the operators +, −, ∗ and /. Many of these operators are overloaded and work for different types as well, their behavior may change accordingly. Boolean operators are **and**, **or**, and **not**. The logical **not** operator binds the strongest, followed by the mathematical operators and then by the remaining Boolean operators. This operator binding rules allow to omit parentheses. The two keywords **true** and **false** represent the respective Boolean values. The following code snippet 9, demonstrates the use of these operators (see Listing 9).

Listing 9: Expressions

```
1 console:log(true);          # true is a constant
2 console:log(false);         # false is a constant too
3 console:log(not false);     # should report true
4 console:log(1 lt 2);        # should report true, because 1
      < 2
5 console:log(not(1 lt 2)); # should report false
6 local x = true and false;
7 local y = not x;
8 local a = 10 * 10;
9 local b = 10 * a - a;
```

Parentheses can be omitted as operators have a predefined precedence. Still, for better readability, parentheses can be used. Throughout the thesis more complex expressions are going to be used, especially when providing real-world examples.

Operator	Math	ADICT	Example
Greater Than	>	gt	a gt b
Less Than	<	lt	a lt b
Greater or Equal	≥	ge	a ge b
Less or Equal	≤	le	a le b
Equal	=	eq	a eq b
Not Equal	≠	ne	a ne b

Table 4.3: Relational operators.

4.4.4 Grouping of Statements

Every command, like a variable definition or function call, is a statement. Each statement has to be terminated by a semicolon (;). Statements can be grouped together using curly brackets ({, }). Such a group of statements is called a block and is considered a statement itself, thus can be used everywhere a statement might be expected. Blocks are discussed in the advanced section in detail (see Section 4.5). A block must not be terminated by a semicolon, see Listing 10.

Listing 10: Grouping of statements (blocks)

```
1 local a = 1;
2 {       # block begin
3     local b = 2;
4     local c = a;
5 }       # block end
6 local d;
```

4.4.5 Conditional Statements

Active DICOM Script supports two different conditional statements. These statements allow the programmer to execute a statement only if a certain condition is met. A conditional expression is one which results in either true or false after being evaluated. Boolean operators can be divided into

- Relational operators, and
- Logical operators.

Relational operators compare two or more operands and result in either true or false depending on the operator used. Table 4.3 gives an overview of the implemented operators. In addition

Operator	Math	ADICT	Example
And	$\wedge$	and	a and b
Or	$\vee$	or	a or b
Not	$\neg$	not	not a

Table 4.4: Logical operators.

to the above mentioned operators the following logical operators are provided, see Table 4.4. The above mentioned operators allow for the creation of complex logical expressions. This leads to the introduction of the first conditional statement.

4.4.5.1 IF Statement

In order to execute a statement only if a certain condition is met, an if-statement may be used. Listing 11 shows a simple if-statement using the less-than operator.

Listing 11: Boolean expression using relational operators

```
1  if (a lt b) {
2      # statements
3  }
```

In Listing 11 it is checked whether a is less than b. If this results in true then the statements within the block are executed. Otherwise the block of statements is skipped and not executed. The next short example (Listing 12) uses a logical expression (logical operators) for the condition expression.

Listing 12: Boolean expression using logical operators

```
1  if (a and b) {
2      # statements
3  }
```

The statements encapsulated in the block are only executed if both variables, *a* and *b*, are true. It is important to note that if-statements have to use blocks, omitting them results in an error. We explicitly implemented this to avoid badly formatted if-statements, makes code hard to read, may lead to misleading code (e.g., dangling else). If-statements can have an alternative section (else). Furthermore it is possible to nest if-else-if statements. This is shown in the last example discussing the if-statement, see Listing 13.

Listing 13: Complex/Nested If-statements

```
1  # if - then - else
2  if (1 lt 2) {
3      console:log("1 < 2");
4  } else {
5      console:log("This should never happen!");
6  }
7
8  # nested if - then - else statement
9  if (1 lt 2) {
10     console:log("1 < 2");
11 } elseif (not false) {
12     console:log("This should never happen!");
13 } else {
14     console:log("This should never happen too!");
15 }
```

4.4.5.2 Conditional Statement

Nesting if-statements might lead to complex and unreadable code. The cond-statement provides a means of writing concise conditional statements. It comes in two variations. The basic cond-statements is written as follows, see Listing 14.

Listing 14: Basic cond-statement

```
1 cond
2     1 lt 2: console:log("1 is really less than 2");
3     2 lt 3: console:log("2 is really less than 3");
4 end;
```

Each line starts with a conditional expression, resulting in a Boolean value. Depending on the condition, the statement following the double colon is executed. Each line is evaluated and without a break-statement, it is not stopped after the first successful true condition. This variation of the cond-statement corresponds to a switch-statement in languages like Java or C++ omitting the break-statement. This is hardly used which leads to the more important variation the breaking cond-statement, see Listing 15.

Listing 15: Breaking cond-statement

```
1 cond break
2     1 lt 2: console:log("1 is really less than 2");
3     2 lt 3: console:log("2 is really less than 3");
4     foo() and bar(): console:log(" foo() or/and bar() that
          is the question");
5     "hello" eq "HELLO": console:log("Which one would it be?
          ");
6 end;
```

The break-statement right after the cond keyword tells the compiler to insert break statements after each statement. For the example in Listing 15 it means that only the first line gets executed and thus the user only sees the message “1 is really less than 2”. In ADICT the cond-statement is basically a concise form of nested if-statements. Internally it is treated as such. Complex nested if-statements tend to get messy and switch-statements provided by many other programming languages are mostly allowed to be used for ordinal types. Such types are integer, character or enumeration types (e.g., enum in C++). Often strings have to be tested, which is not possible in many languages using a switch-statement, see lines 4 and 5 in Listing 15.

4.4.6 Loop Statements

The previous sections described conditions and statements without repetitions. In order to perform an operation multiple times one has to copy code as often as needed. This is error prone and not feasible in most situations. Therefore in order to repeat a statement as often as needed, ADICT offers the loop statements and many variants thereof.

Listing 16: The do-loop statement

```
1 do
2     console:log("Infinite loop!"); # semicolon needed!
3 loop;
```

The code snippet shown in Listing 16, is the most basic loop statement and, due to the absence of a breaking condition, the enclosed statement is executed indefinitely. In the given example only one statement is encapsulated within the do-loop construct. A semicolon is needed at the end of the single-line command. In the presence of a block, no semicolon is needed. Basically all loop statements are variations of the basic one, suited for different tasks.

4.4.6.1 do-for-loop Statement

The do-for-loop is mostly used for iterating over index-based data structures, such as arrays and lists (see advanced topics in Section 4.5). This loop uses a counter variable, which is

initialized with a starting value and is incremented by a certain value until a given target value is reached.

Listing 17: Bubble Sort algorithm

```
1  # data for sorting
2  local data = new array[6, 5, 4, 3, 2, 1];
3
4  # loop variables
5  local i;
6  local j;
7
8  do for i from data:length to 2 step -1 {
9     do for j from 0 to i-2 step 1 {
10       if (data[j] gt data[j+1]) {
11          # swapping the elements
12          local temp = data[j];
13          data[j] = data[j+1];
14          data[j+1] = temp;
15       }
16    } loop;
17 } loop;
18
19 # writing the sorted list to the console
20 do for i from 0 to data:length-1 step 1 {
21    console:log(data[i]);
22 } loop;
```

The counter variable has to be defined before its first usage. In the do-for-loop statement, in Listing 17 at line 20, the counter variable, i, is incremented by 1 until $data:length-1=5$ is reached. The starting value is 0. This means that six numbers are printed to the console. One can observe that the variable is not altered explicitly, because this is already done right at the end of each iteration. Using a negative step allows the programmer to write a loop counting down from a starting value. Omitting the definition of a **step** would not alter the loop-variable, resulting in an infinite loop.

Do-for-loop statements are often used to iterate over countable sets or lists. Also index-based data structures such as arrays, matrices, or hash maps are often iterated over by for-loop statements. Listing 17 describes a bubble sort algorithm using more complex for-state-ments.

4.4.6.2 While Loop

Another variation of the standard do-loop statement is the while-loop. The intention of this loop-statement is to repeat its enclosed commands as long as the loop condition is satisfied. It is possible to write the loop condition either at the beginning or at the end of the loop-

expression.

Listing 18: do-while-loop statement

```
1  local i = 0;
2  do while (i lt 10) {
3      console:log(i);
4      i = i + 1;
5  } loop;
```

The while-loop (Listing 18) condition is checked at the beginning of the loop and the enclosed statements are repeated as long as the condition is satisfied, i.e., evaluates to true. With the while-loop being an enhanced loop, it is possible to write the loop condition at the end as well, shown in Listing 19.

Listing 19: do-loop-while statement

```
1  local i = 0;
2  do {
3      console:log(i);
4      i = i + 1;
5  } loop while (i lt 10);
```

The difference between the above while-loop examples is the time of checking the loop condition. The first example prints all numbers ranging from 0 to 9 to the console. The last example also prints the same numbers with the only difference that the body of the loop is executed at **least once**.

4.4.6.3 Until Loop

The notion of how to interpret the condition of a loop is often subject to personal preferences or the context of the loop. Sometimes one wishes to express not to repeat a certain set of statements as long as a condition is satisfied rather than until it is satisfied. Basically it is the logically negated condition of the while-loop condition, meaning that it is possible to use, the while-loop and the until-loop interchangeably by adapting the loop condition.

Listing 20: do-until-loop statement

```
1  local i = 0;
2  do until (i ge 10) {
3      console:log(i);
4      i = i + 1;
5  } loop;
```

The meaning of the loop now is that statements should be repeated until a condition is satisfied, see Listing 20. Once again it is possible to write the loop condition at the end of the loop, shown in Listing 21. This results in the more commonly used repeat-until (Pascal) or do-while loop(C/C++).

Listing 21: do-loop-until statement

```
1 local i = 0;
2 do {
3     console:log(i);
4     i = i + 1;
5 } loop until (i ge 10);
```

All the presented statements are the building blocks for larger applications. Every statement can be nested and used where statements are allowed. The next section is going to introduce some more advanced language constructs, which are used in the later, much more, complex examples. One of the quite often used advanced statements are lambdas. These are function definitions making it possible to create new commands.

4.5 Advanced Topics

In this section advanced language features are described. Within this category are functions and function definitions, arrays, and objects. The last part of this section is going to show some advanced expressions such as inline function definitions and calls.

4.5.1 Functions

Functions allow the user/developer to create new commands and let one structure the code in reusable units. Code reuse makes a program more readable, reduces redundant code and therefore may reduce the possibility for errors/bugs. A function definition is an expression. This allows for such definitions to appear within arithmetic or boolean expression. More precisely, function definitions may occur on the right hand side of an assignment and thus can be called within a single statement, meaning they are treated as expressions, see Listing 27 lines 9 to 14 as an example for advanced language concepts.

The following script, i.e., Listing 22, shows how a function is defined in ADICT. Being an expression it has to be assigned to a variable in order be referenced later on.

Listing 22: Function definition

```
1 def calcSumOfAandB = lambda() {
2     local a = 5;
3     local b = 7;
4     console:log(a + b);
5 };
6
7 calcSumOfAandB();
```

This code snippet demonstrates a few concepts. Functions are defined using the keyword **lambda()** followed by curly brackets. Lambdas are an expression and thus can be assigned to a variable. A function call is performed using the ()–operator. Both variables, a and b, are called local variables. Such variables are only accessible inside the body of the function and outside they are undefined, hence the keyword **local** for their definitions. In order to emphases that this is a function definition and not an ordinary variable definition, one can use the keyword **def** instead of **local**, but this is not mandatory.

Many programming languages distinguish between two types of functions. One type returns a value and one that does not. The latter ones are often called procedures. The function in Listing 22 would be called procedure as no return value is specified. The next example in Listing 23 is a modification, turning the previous "procedure" into a function.

Listing 23: Function returning a value

```
1 local calcSumOfAandB = lambda() {
2     local a = 5;
3     local b = 7;
4     return a + b;
5 };
6 console:log(calcSumOfAandB());
```

In the example above a new statement was introduced using the keyword **return**. Such a statement may only appear within a function definition and serves two purposes:

- **return** assigns a value to the result of the function, and
- execution is stopped and the control is returned to the caller.

The second point implies that statements after **return** are ignored and are not executed. Functions can also call themselves, defining a recursion. This is a powerful and elegant tool allowing the programmer to create compact and concise algorithms. An important aspect of recursions is the stopping condition in order to avoid an infinite recursion, resulting in a stack overflow. The following equation is taken as guiding example for the remainder of this section:

$$sum(n) = \sum_{i=0}^{n} i \tag{4.1}$$

The first approach is be an iterative solution, shown in Listing 24, using a loop instead of a recursion.

Listing 24: Iterative version of $sum(n)$

```
1  local n = 3;
2  local sum = lambda() {
3     local i;
4     local result = 0;
5     do for i from 0 to n step 1 {
6        result = result + i;
7     } loop;
8     return result;
9  };
10 local sum_result = sum(); # 6
11 console:log(sum_result);
```

The above listing is just a the first step constructing a more complex and proper function. The example is considered bad practice since its use of a "global" variable, named **n**. The next version of the sum-function is going to be a recursive one.

Listing 25: Recursive version of $sum(n)$

```
1  local n = 3;
2  local sum = lambda() {
3     local orig = n;
4     n = n - 1;
5     # breaking condition
6     if (n lt 0) {
7        return 0;
8     }
9     # recursive step
10    return orig + sum();
11 };
12 local sum_result = sum(); # 6
```

The first version of the recursive implementation is even worse than the first version of the iterative one. Again it uses a "global" variable, but even alters the value of it. This is called a side effect and might be dangerous in large programs. Such alterations of global variables might be overlooked and may result in errors that are difficult to find and correct.

Using global variables for passing values to functions is considered a bad practice and if using many functions will result in unreadable or maintainable code. Parameters are normally used to pass values to functions. The next example is an improved version of the iterative one.

See Listing 26 for the improved iterative version of $sum(n)$.

Listing 26: Iterative version of $sum(n)$ using parameters

```
1 local sum = lambda(n) {
2     local i;
3     local result = 0;
4     do for i from 0 to n step 1 {
5         result = result + i;
6     } loop;
7     return result;
8 };
9 local sum_result = sum(3); # 6
```

The first line defines a parameter, basically a local variable within the function body. Such a parameter must be set by the caller, see line 9 of Listing 26. With the use of parameters we can finally provide a recursive implementation, which is short, simple, and concise like many recursive algorithms (see Listing 27).

Listing 27: Recursive version of $sum(n)$ using parameters

```
1  local sum = lambda(n) {
2      cond break
3          (n le 0): return 0;
4          true: return n + sum(n - 1);
5      end;
6  };
7  local sum_result = sum(3); # 6
8  # defining a function and calling it within a single
       expression
9  local sum_result2 = (lambda(n) {
10     cond break
11         (n le 0): return 0;
12         true: return n + _(n - 1); # _ references the
               current function
13     end;
14 })(3); # 6
```

As mentioned above, parameters are local variables within a function. It is possible to assign values to parameters, but such changes are not passed to the caller of the function. This is not true for passed constructs, such as objects and arrays. These exceptions are discussed further down in this section.

4.5.2 Objects

The topic of this section is how to group data and functions so that they form one unit. ADICT's object implementation does not allow for complex object constructs like, e.g., JavaScript does. The notion of an object means a composition of data and functions operating on this data. Functions defined within an object are referred to as methods.

Listing 28: Object definition

```
1 local obj = new object {
2     member_variable_1: 1;
3     member_variable_2: 2 # no semicolon
4 };
5 local value = obj:member_variable_1; # value = 1
```

Listing 28 uses a new operator, called **new**. It is used to create new objects and arrays (see next section). Each line within an object definition must start with an identifier, followed by a colon and then an expression is expected. Such a line is then terminated by a semicolon. It is important to note that the last line must not be terminated by a semicolon. A variable defined within the scope of an object is referred to as member variable in most programming languages. Line 5 shows how a member variable is accessed again using the colon operator.

The next example illustrates how methods are defined and how member variables are accessed from within methods (see Listing 29).

Listing 29: Nested objects and methods

```
1  local obj = new object {
2      point: new object {
3          x: 0;
4          y: 0
5      };
6      setX: lambda(n) {
7          this:point:x = n;
8          return this;
9      };
10     getX: lambda() {
11         return this:point:x;
12     }
13 };
14 local xValue = obj:setX(100):getX(); # xValue = 100
```

Object definitions may also be nested. The keyword **this** can be used to access the current object. Since method *setX* returns **this**, meaning a reference to the current object, line 14

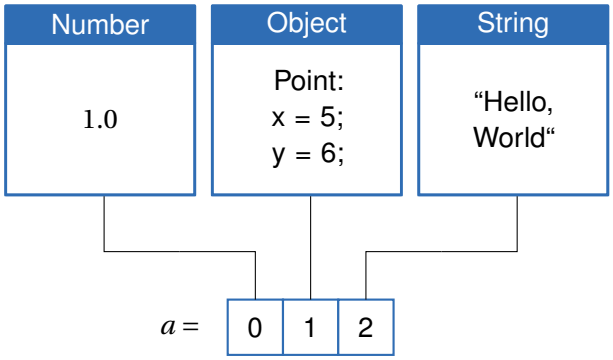


Figure 4.1: An array *a*, containing differently typed elements.

makes it possible to chain the call of *setX* with *getX*. Line 2 defines a member variable which is an object containing the coordinates of an example 2D point.

4.5.3 Arrays

An array defines a sequence of objects. In most programming languages arrays are only allowed to contain values of the same type. For instance in *C*/*C*++ one can create an array of integers. Such an array only contains integer numbers. Since **ADICT** does not have types explicitly, one can create a single array with entries of different types, even objects (see Figure 4.1).

The array defined in Figure 4.1 is of length three and contains a number, an object, and a string entry. Accessing an array element at a given position is done using edged brackets. An array starts with index zero and has length – 1 as its maximum index. The following code example creates the array from Figure 4.1. It also shows how arrays can be accessed. In order to create a new array the **new** operator is used. This time the keyword **array** has to be used to determine what to create (see Listing 30).

Listing 30: Array definition of figure 4.1

```
1  local a = new array [
2      0,
3      new object {
4          x: 5;
5          y: 6
6      },
7      "Hello, World"
8  ];
9  local a1x = a[1]:x; # 5
10 a[2] = "Hello"; # changing the third entry
```

The examples in these sections are just for demonstration purposes and illustrate that even

Function name	Description	ADICT
length	Returns the number of elements in an array.	a:length()
indexOf	Searches the array for a specific element and returns its position.	a:indexOf(0)
lastIndexOf	Same functionality as indexOf but starting from the end of the array.	a:lastIndexOf(0)
pop	Removes the last element of an array and returns it.	a:pop()
push	Adds a new element to the end of an array. Returns the length of the new array.	a:push(10)
reverse	Reverses the order of the elements in an array.	a:reverse()
sort	Sorts the elements of an array. The algorithm used is implementation-dependent (using merge or quick sort).	a:sort()
toString	Converts an array to a string and returns the string as the result.	a:toString()

Table 4.5: Predefined array functions (selective list).

with simple constructs complex data structures can be created. Arrays include, due to their relationship to JavaScript, a set of built in functions (see Table 4.5).

4.5.4 Annotated Data

With annotations it is possible to add meta information to expressions or statements. One such annotation is **@Data**. It is used to define data section entries in ActiveDICOM. As mentioned in previous sections, the data section contains arbitrary data embedded in the image, which has to be referenced later when needed. Such an annotation is basically an object definition beginning with **@**.

Listing 31: Annotated binary data.

```
1  @Data {
2          name: "MRI_predefined"
3  } new binary [
4          new object {
5                  mime: "image/png";
6                  encoding: "base64";
7                  fname: "00000001.png"
8          };
9          # base64 encoded binary data
10         00000000000000000000000000
11 ];
```

Annotated data, as shown in Listing 31, can be accessed via its name *MRI_predefined*. Keys such as **encoding** provide information about how to extract the contained data from the stream (see line 10). Listing 31 contains a **base64** encoded data stream. Using the **new** operator followed by the keyword **binary** defines a binary data element. The first entry of a binary data object is an object definition containing useful header information. The second part contains the data as a **base64** encoded string.

This section gave a short overview of the features of the implemented programming language. The following section gives an overview of the implementation, which should motivate how some features were implemented.

4.6 Implementation of ADICT

The compiler written for ADICT is implemented in Java as a library, which is used in the web application to compile embedded code to JavaScript. The implemented library provides the following components:

- **Compiler.** This is the compiler for ADICT. It is described in the current section.
- **Pretty Printer (PP).** The pretty printer provides functionality for formatting given ADICT code in a structured and easily readable way.
- **Syntax Highlighter.** This component generates HTML output with syntax highlight in color.

The pretty printer and the syntax highlighter are used in the web application for displaying embedded code in a readable, formatted, and printable way. Line breaks and indentations are inserted to provide better readability of script code.

The main part of the compiler library is the compiler implementation itself. The following three main parts of the compiler can be identified, as shown in Figure 4.2:

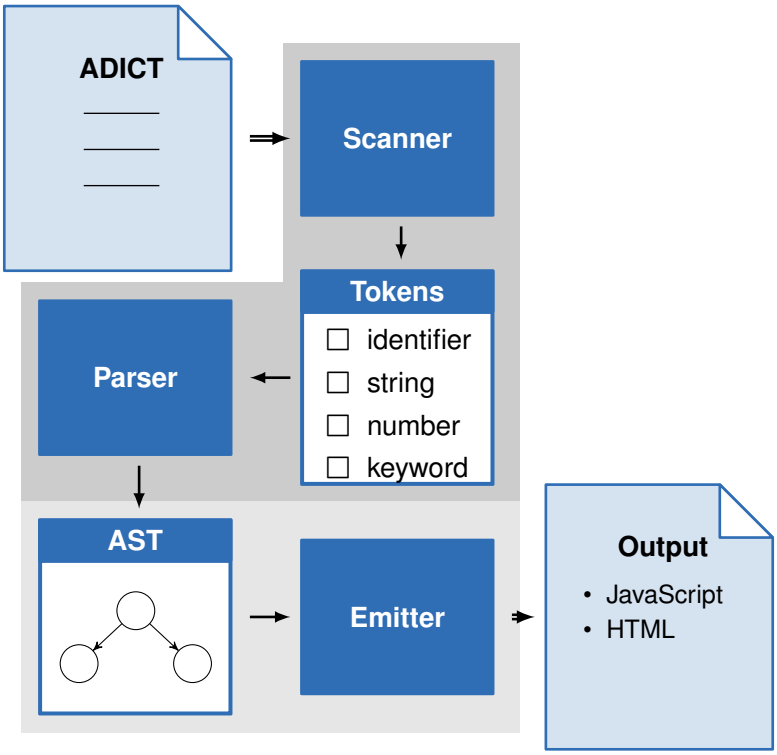


Figure 4.2: ADICT compiler: Multi-tiered compiler architecture consisting of two tiers, i.e., a front-end (dark gray area) and a back-end (gray area). The front-end deals with tokens and its result is an **A**bstract **S**yntax **T**ree (AST). The back-end produces code depending on various use cases, such as HTML or JavaScript.

- **Scanner.** This is the first part of the front-end. The scanner provides a stream of tokens for the parser. It filters comments and parses strings and numbers. Whenever a token cannot be successfully parsed, an error is reported and parsing is stopped.
- **Parser.** This is the second part of the front-end. It receives a stream of tokens from the scanner and creates an **A**bstract **S**yntax **T**ree (AST). Syntactical checks are performed in this phase/stage and every time an error occurs, this is reported to the user and further processing is stopped. This parser is implemented using a recursive descend approach and is not generated by automatic tools. For more details about LL parsers see Rosenkrantz [67].
- **Emitters.** We call this part back-end. All components considered in this package operate on an AST. There are two major components. The first one is the JavaScript emitter, which converts ADICT into JavaScript. The JavaScript output is embedded into the browser and creates the interactive elements designed by the developer of the ActiveDICOM. The second emitter is able to produce HTML output and is used to generate well structured source code with syntax highlighting (see Figure 4.2).

In order to keep the compiler and execution environment (interpreter) simple, no semantic checking is currently implemented. This was not necessary as the primary target language is JavaScript which lacks types and even allows the programmer to omit declaring variables. The downside is that many errors, made by the programmer are encountered at run time making it harder to track errors. Variable scopes are also not present, which is correlated to not having to declare variables prior to their first use. The compiler just checks for syntactic integrity and then compiles the source language to the target one, i.e., JavaScript. Section 4.8 gives on overview of limitations of our approach and which future aspects of ActiveDICOM are planned.

Annotated data is treated as a program with only variable definitions. Annotations were added to the compiler to provide extra information, i.e., meta information, to language elements such as variable definitions. Basically, it is used as a serialization of data objects using Active Dicom Script. As for data serialization there are many other different options, like XML or JSON for human readable formats. Binary formats might be machine dependent (like endianess; defining how many bits represent certain entities like numbers). Furthermore, annotations might be used in later improvements for various tasks, such as remote procedure calls and for providing information about the code at runtime.

During the implementation of the compiler, performance was not a main concern as the language was designed to be the glue for various natively implemented building blocks (points-of-interest, sub-views, menus, embedded media, etc.). In early stages of the implementation of the web-based DICOM viewer, ADICT was used to even implement the viewer part. Due to performance reasons, this was rewritten in native JavaScript in later software versions.

4.7 Language Reference

In this section the language is described in a formal way. The syntax of Active Dicom Script can be described as a context free grammar, referred to as CFG. Such a system consists of a number of production rules of the following form:

$$A \rightarrow \alpha \tag{4.2}$$

The left-hand side of the above rule is called a non-terminal symbol. The right-hand side may consist of a series of terminal and non-terminal symbols. The right-hand side might also be empty. Context free means that a substitution rule might be applied regardless of neighboring symbols. Non-terminal symbols are written in *italic* fonts. To separate left-hand side and right-hand side, an arrow, →, is used. Tokens are written in **bold**. Special characters, such as carriage return, are written in capital letters, e.g., CR. Ellipsis (…) are used to write lists in a concise way, for instance ‘a’…‘z’ represents all characters from ‘a’ to ‘z’. Simplifications in the notation are used for better readability such as verbal descriptions of character sets. Epsilon, ϵ, is used as the empty rule appearing only as right-hand side expression.

4.7.1 Token Rules

As shown in Figure 4.2, the compiler is divided into three parts each one providing input for the following one. The first part is the scanner, which produces tokens acting as input for the parser. Tokens are simple text elements, such as numbers, identifiers, and keywords. This section describes the rules for these basic elements.

$$
\begin{aligned}
\textit{EndOfLine} &\rightarrow \text{CR} \mid \text{LF} \mid \text{CR LF} && (4.3)\\
\textit{BlankChar} &\rightarrow \textit{EndOfLine} \mid \text{SPACE} \mid \text{TAB} && (4.4)\\
\textit{Digit} &\rightarrow \mathbf{0} \ldots \mathbf{9} && (4.5)\\
\textit{Alpha} &\rightarrow \mathbf{a} \ldots \mathbf{z} && (4.6)\\
\textit{IdentifierStart} &\rightarrow \textit{Alpha} \mid \_ && (4.7)\\
\textit{IdentifierChars} &\rightarrow \textit{Alpha} \mid \_ \mid \textit{Digit} \mid \mathbf{?} && (4.8)\\
\textit{Identifier} &\rightarrow \textit{IdentifierStart IdentifierChars} && (4.9)\\
\textit{String} &\rightarrow \text{“ “} \mid \text{“ } \textit{StringChars} \text{ “} && (4.10)\\
\textit{StringChars} &\rightarrow \textit{any character except for “ and CR/LF}\\
&\mid \textit{StringEscCharSequence} && (4.11)\\
\textit{StringEscCharSequence} &\rightarrow \backslash\ \textit{StringEscChar} && (4.12)\\
\textit{StringEscChar} &\rightarrow t \mid r \mid n \mid \text{“} \mid \backslash && (4.13)\\
\textit{IntNumber} &\rightarrow \textit{Digit IntNumber} \mid \textit{Digit} && (4.14)\\
\textit{Number} &\rightarrow \textit{IntNumber} \,\mathbf{.}\, \textit{IntNumber} \mid \textit{IntNumber} && (4.15)\\
\textit{BinaryToken} &\rightarrow \textit{base64 string} && (4.16)
\end{aligned}
$$

4.7.2 Boolean and Mathematical Expressions

Boolean and mathematical expressions are those that use function calls, various boolean operators and mathematical operations. If no distinction is necessary, we use the term expression to refer to either a boolean or mathematical expression. The following production rules describe the syntax of boolean expressions.

$$
\begin{aligned}
\textit{Expression} &\rightarrow \textit{ExpressionL1}\\
&\mid \textit{ExpressionL1} \,\mathbf{and}\, \textit{ExpressionL1}\\
&\mid \textit{ExpressionL1} \,\mathbf{or}\, \textit{ExpressionL1} && (4.17)\\
\textit{ExpressionL1} &\rightarrow \textit{ExpressionL2}\\
&\mid \textit{ExpressionL2} \,\mathbf{eq}\, \textit{ExpressionL2}\\
&\mid \textit{ExpressionL2} \,\mathbf{ne}\, \textit{ExpressionL2}\\
&\mid \textit{ExpressionL2} \,\mathbf{lt}\, \textit{ExpressionL2}\\
&\mid \textit{ExpressionL2} \,\mathbf{le}\, \textit{ExpressionL2}\\
&\mid \textit{ExpressionL2} \,\mathbf{gt}\, \textit{ExpressionL2}\\
&\mid \textit{ExpressionL2} \,\mathbf{ge}\, \textit{ExpressionL2} && (4.18)
\end{aligned}
$$

Mathematical expressions are built on top of boolean expressions. Operator precedence is ensured due to the following production rules.

$$
\begin{aligned}
\textit{ExpressionL2} &\rightarrow \textit{ExpressionL3} \\
&\mid \textit{ExpressionL3} \texttt{ + } \textit{ExpressionL3} \\
&\mid \textit{ExpressionL3} \texttt{ - } \textit{ExpressionL3} && (4.19)\\
\textit{ExpressionL3} &\rightarrow \textit{ExpressionL4} \\
&\mid \textit{ExpressionL4} \texttt{ * } \textit{ExpressionL4} \\
&\mid \textit{ExpressionL4} \texttt{ / } \textit{ExpressionL4} \\
&\mid \textit{ExpressionL4} \texttt{ \% } \textit{ExpressionL4} && (4.20)\\
\textit{ExpressionL4} &\rightarrow \textit{ExpressionL5} \\
&\mid \textit{PathExpression} && (4.21)\\
\textit{PathExpression} &\rightarrow \textit{ExpressionL4} \texttt{ : } \textit{ExpressionL5} && (4.22)\\
\textit{ExpressionL5} &\rightarrow \textit{ExpressionTopLevel} \\
&\mid \textit{FCall} \\
&\mid \textit{ArrayAccess} && (4.23)
\end{aligned}
$$

The following production rules are more complex and are therefore dealt with individually. In many of the upcoming production rules we need a list of expressions, e.g., parameter list:

$$
\begin{aligned}
\textit{ExprList} &\rightarrow \textit{Expression} \\
&\mid \textit{Expression} \texttt{ , } \textit{ExprList} && (4.24)\\
\textit{ParamList} &\rightarrow \texttt{( )} \\
&\mid \texttt{(} \textit{ExprList} \texttt{)} && (4.25)\\
\textit{ArrayParamList} &\rightarrow \texttt{[} \textit{ExprList} \texttt{]} && (4.26)\\
\textit{IdentifierList} &\rightarrow \textit{Identifier} \\
&\mid \textit{Identifier} \texttt{ , } \textit{IdentifierList} && (4.27)
\end{aligned}
$$

With the help of the above rules we can now define function calls and array accesses.

$$
\begin{aligned}
\textit{FCall} &\rightarrow \textit{ExpressionTopLevel ParamList} && (4.28)\\
\textit{ArrayAccess} &\rightarrow \textit{ExpressionTopLevel ArrayParamList} && (4.29)
\end{aligned}
$$

The final expression production rule, i.e., the top level expression, is more complicated since more cases have to be considered. Lambda definitions are specified in Section 4.7.5.

$$
\begin{aligned}
\textit{ExpressionTopLevel} \rightarrow\ & \textit{Identifier} \mid \textit{Number} \\
\mid\ & \textbf{this} \mid \textbf{true} \mid \textbf{false} \\
\mid\ & \textit{String} \\
\mid\ & \textbf{null} \mid \textbf{nil} \\
\mid\ & \textbf{+}\ \textit{ExpressionTopLevel} \\
\mid\ & \textbf{-}\ \textit{ExpressionTopLevel} \\
\mid\ & \textbf{not}\ \textit{ExpressionTopLevel} \\
\mid\ & \textbf{(}\ \textit{Expression}\ \textbf{)} \\
\mid\ & \textbf{new}\ \textit{NewExpression} \\
\mid\ & \textit{LambdaDef}
\end{aligned}
\tag{4.30}
$$

4.7.3 New Operator

Since *new* is a very important operator it, is described in an own section. It is used to create new objects, arrays, and binary data.

$$
\begin{aligned}
\textit{NewExpression} \rightarrow\ & \textbf{array}\ \textit{ArrayParamList} \\
\mid\ & \textbf{binary}\ \textbf{[}\ \textbf{new object}\ \textit{ObjectExpression}\ \textbf{;}\ \textit{BinaryToken}\ \textbf{]} \\
\mid\ & \textbf{object}\ \textit{ObjectExpression}
\end{aligned}
\tag{4.31}
$$

Objects are one of most important building blocks of applications and are extensively used in the examples throughout the thesis. Object expressions are either inline object definitions or constructor calls.

$$
\begin{aligned}
\textit{ObjectExpression} \rightarrow\ & \textbf{:}\ \textit{IdentifierList ParamList} \\
\mid\ & \{\,\} \\
\mid\ & \{\ \textit{ObjectDefinitionLines}\ \} && (4.32) \\
\textit{ObjectDefinitionLines} \rightarrow\ & \textit{ObjectDefinitionLine} \\
\mid\ & \textit{ObjectDefinitionLine}\ \textbf{;}\ \textit{ObjectDefinitionLines} && (4.33) \\
\textit{ObjectDefinitionLine} \rightarrow\ & \textit{Identifier}\ \textbf{:}\ \textit{Expression} && (4.34)
\end{aligned}
$$

4.7.4 Statements

Building upon all the above described production rules, we can now define the final set of production rules, namely those for statements. The statement set of production rules contains all programming language constructs of Section 4.4 onwards. The following production defines

all the available statements of ADICT:

$$
\begin{aligned}
\textit{AnnotatedStatement} &\rightarrow \textit{Annotation AnnotatedStatement} \\
&\mid \textit{Statement} && (4.35)\\
\textit{StatementNoBlock} &\rightarrow \textit{DefStatement} \\
&\mid \textit{IfStatement} \\
&\mid \textit{LoopStatement} \\
&\mid \textbf{break} \\
&\mid \textbf{continue} \\
&\mid \textit{BlockStatement} \\
&\mid \textit{ReturnStatement} \\
&\mid \textit{CondStatement} \\
&\mid \textit{Expression} \\
&\mid \textit{AssignmentStatement} && (4.36)\\
\textit{Statement} &\rightarrow \textit{StatementNoBlock} \\
&\mid \textit{BlockStatement} && (4.37)
\end{aligned}
$$

4.7.4.1 Variable Definition

$$
\begin{aligned}
\textit{DefStatement} &\rightarrow \textbf{def}\ \textit{Identifier} \\
&\mid \textbf{local}\ \textit{Identifier} \\
&\mid \textbf{def}\ \textit{Identifier} = \textit{Expression} \\
&\mid \textbf{local}\ \textit{Identifier} = \textit{Expression} && (4.38)
\end{aligned}
$$

The keywords **def** and **local** can be used interchangeably. **def** is mostly used to express that a function is being defined.

4.7.4.2 If Statement

$$
\begin{aligned}
\textit{IfStatement} &\rightarrow \textbf{if}\ (\ \textit{Expression}\ )\ \textit{BlockStatement} \\
&\mid \textbf{if}\ (\ \textit{Expression}\ )\ \textit{BlockStatement IfElse} && (4.39)\\
\textit{IfElse} &\rightarrow \textbf{else}\ \textit{BlockStatement} \\
&\mid \textbf{elseif}\ (\ \textit{Expression}\ )\ \textit{BlockStatement} \\
&\mid \textbf{elseif}\ (\ \textit{Expression}\ )\ \textit{BlockStatement IfElse} && (4.40)
\end{aligned}
$$

4.7.4.3 Loops

$$
\begin{aligned}
\textit{LoopStatement} &\rightarrow \textit{BasicLoop} \\
&\mid \textit{WhileLoop} \\
&\mid \textit{UntilLoop} \\
&\mid \textit{ForLoop} \qquad (4.41) \\
\textit{LoopBody} &\rightarrow \textit{StatementNoBlock}\ \textbf{;} \\
&\mid \textit{BlockStatement} \\
&\mid \epsilon \qquad (4.42)
\end{aligned}
$$

The basic loop statement does not check any condition and the programmer has to use the **break** statement in order to exit the loop:

$$
\textit{BasicLoop} \rightarrow \textbf{do}\ \textit{LoopBody}\ \textbf{loop} \qquad (4.43)
$$

The next production rule defines the well known while loop found in many programming languages:

$$
\begin{aligned}
\textit{WhileLoop} &\rightarrow \textbf{do while (}\ \textit{Expression}\ \textbf{)}\ \textit{LoopBody}\ \textbf{loop} \\
&\mid \textbf{do}\ \textit{LoopBody}\ \textbf{loop while (}\ \textit{Expression}\ \textbf{)} \qquad (4.44)
\end{aligned}
$$

A slight variation to the classic while-loop statement is the until loop, which negates the loop condition:

$$
\begin{aligned}
\textit{UntilLoop} &\rightarrow \textbf{do until (}\ \textit{Expression}\ \textbf{)}\ \textit{LoopBody}\ \textbf{loop} \\
&\mid \textbf{do}\ \textit{LoopBody}\ \textbf{loop until (}\ \textit{Expression}\ \textbf{)} \qquad (4.45)
\end{aligned}
$$

One of the most often used loop statements is the for-loop. It is used to iterate over containers such as lists, arrays, or maps. It is used for counting purposes as well:

$$
\begin{aligned}
\textit{ForLoop} \rightarrow\ & \textbf{do for}\ \textit{Identifier}\ \textbf{from}\ \textit{Expression}\ \textbf{to}\ \textit{Expression} \\
& \textbf{step}\ \textit{Expression}\ \textit{LoopBody}\ \textbf{loop} \qquad (4.46)
\end{aligned}
$$

4.7.4.4 Cond/Switch Statement

Most programming languages provide a language construct for checking many conditions at once. Often such a statement is only applicable to ordinal types, i.e., those that can be mapped to integer types.

$$
\begin{aligned}
\textit{CondStatement} &\rightarrow \textit{StdCondStatement} \\
&\mid \textit{BreakingCondStatement} \qquad (4.47)
\end{aligned}
$$

In ADICT the cond-statement is used to test conditions and execute statements based on the evaluated result of the given expression. There are two possibilities. Defined by *BreakingCondStatement* is a switch statement, which automatically adds a **break** statement after each line. The rule, *StdCondStatement*, is the second possible way to write a cond-statement in ADICT.

$$
\begin{aligned}
\textit{BreakingCondStatement} &\rightarrow \textbf{cond break end} \\
&\mid \textbf{cond break } \textit{CondLines} \textbf{ end} && (4.48)\\
\textit{StdCondStatement} &\rightarrow \textbf{cond end} \\
&\mid \textbf{cond } \textit{CondLines} \textbf{ end} && (4.49)\\
\textit{CondLines} &\rightarrow \textit{CondLine}\,\textbf{;} \\
&\mid \textit{CondLine CondLines} && (4.50)\\
\textit{CondLine} &\rightarrow \textit{Expression}\,\textbf{:}\,\textit{Statement} && (4.51)
\end{aligned}
$$

4.7.4.5 Assignments

An assignment in ADICT is described with the following production rule:

$$
\textit{AssignmentStatement} \rightarrow \textit{Expression} = \textit{Expression} \qquad (4.52)
$$

4.7.4.6 Return Statement

The return–statement is used to exit functions and return a value to the caller if provided:

$$
\begin{aligned}
\textit{ReturnStatement} &\rightarrow \textbf{return } \textit{Expression} \\
&\mid \textbf{return} && (4.53)
\end{aligned}
$$

4.7.4.7 Blocks of Statements

Statements can be grouped. A group of statements is treated as a statement, which allows the programmer to insert multiple statements where a single one is allowed:

$$
\begin{aligned}
\textit{BlockStatement} &\rightarrow \{\,\} \\
&\mid \{\,\textit{StatementList}\,\} && (4.54)\\
\textit{StatementList} &\rightarrow \textit{Statement}\,\textbf{;} \\
&\mid \textit{Statement}\,\textbf{;}\,\textit{StatementList} && (4.55)
\end{aligned}
$$

4.7.5 Lambda Definitions

Functions are one of the most important constructs of programming languages. They enable concepts such as code reuse, recursions, and local variables with the use of a stack. Many programming languages distinguish between functions, which do not return a value, calling

them procedures, and ones returning a value. ADICT does not make such a distinction, there are just functions.

$$\begin{aligned} \textit{LambdaDef} &\rightarrow \textbf{lambda ( )}\ \textit{BlockStatement} \\ &\mid \textbf{lambda (}\ \textit{IdentifierList}\ \textbf{)}\ \textit{BlockStatement} \end{aligned} \tag{4.56}$$

4.7.6 Annotations

Annotations can be added to statements. It is used for data sections.

$$\begin{aligned} \textit{Annotation} &\rightarrow \textbf{@}\ \textit{Identifier} \\ &\mid \textbf{@}\ \textit{Identifier}\ \{\ \textit{ObjectDefinitionLines}\ \} \end{aligned} \tag{4.57}$$

4.7.7 Program

The following production rule defines the starting point for programs. A valid program can either be empty or a list of statements.

$$\begin{aligned} \textit{Program} &\rightarrow \epsilon \\ &\mid \textit{StatementList} \end{aligned} \tag{4.58}$$

4.8 Limitations

The implemented language is not without limitations or shortcomings. The created compiler is kept simple to reduce development efforts. Furthermore, we implemented the compiler tailored to features needed and supported by the target language, i.e., JavaScript. Since JavaScript does not have types, structures, complex types, such as unions (records with variants), complex classes and much more, we omitted such advanced language constructs in our implementation.

As mentioned, the target language, lacked types and therefore such a feature was omitted as it would certainly increase complexity of the compiler (e.g. semantical analysis, type checking and conversions). In some cases, operator precedence might not be intuitive or as one might expect known from other languages requiring the placement of additional parenthesis.

Speed was not the primary goal. Therefore, the whole code is compiled every time an ActiveDICOM is opened or viewed. Caching of compiled code might speed up page-load times and sharing of code, e.g., libraries, might save memory. JavaScript engines are quite sophisticated virtual machines, often using various advanced optimizations techniques such as just-in-time compilation. Our implementation heavily relies on the virtual machines making our generated code to perform well enough.

4.9 Summary

This chapter provided an extensive overview of our developed domain specific scripting language **A**ctive **DIC**OM Scrip**t** (ADICT). We discussed our design decisions and implementation aspects. This was then followed by a short introduction to JavaScript.

Section 4.4 gave a comprehensive and thorough introduction to our scripting language. We provided many examples with increasing difficulty showcasing our intended language. Ranging from introductory examples, like variable definitions, we incorporated various advanced examples later into the section such as recursive sum functions. The discussion of objects, arrays, and annotated data then conclude the description of ADICT.

We, provided an insight into the implementation of our small compiler package. It provides a syntax highlighter and a pretty printer as well, both used in the web-application for a better readability of code. Our designed compiler follows a tiered architecture where its implementation consists of a scanner, parser, and emitter.

Afterwards a more formal description of our language was given. Using a CFG rule system, where we described the language based on token rules to boolean/mathematical rules and finally statement rules.

Our language has limitations and which we describe in this chapter as well, which surfaced during the realization of our idea/contribution. Being a simple compiler, many advanced features, like types, optimizations and code modularization were neglected or only touched briefly.

CHAPTER

5 Results

After demonstrating several interactive elements, we present results of ActiveDICOM with two medical cases in this chapter. The first case is a vessel stenosis (see Figure 5.1), whereas the second case deals with a pulmonary embolism (see Figure 5.2). Since both clinically relevant cases are rather difficult to spot, interactive elements support their recognition and examination.

In order to provide preliminary feedback on ActiveDICOM, we consulted a radiologist. With an informal interview we identified interdisciplinary medical discussion rounds between radiologists and surgeons as one possible application scenario. Using an ActiveDICOM pre-set, i.e., predefined building blocks, the entire workflow of a surgery can be planned by radiologists and surgeons during their meeting. Afterwards, the surgeon receives the ActiveDICOM as discussed and prepared, in order to proceed with the surgery. Nowadays, surgeons are still printing images in order to view them on light boxes, as mentioned by the radiologist. However, our approach allows the surgeon to view several visualization techniques, embedded into the DICOM image, on a digital light box even during the surgery.

5.1 Vessel Stenosis

A clinically relevant case is presented in Figure 5.1 through a blood vessel stenosis. Such a vascular pathology prevent blood from flowing through the vessels and leads to a lack of oxygen in the affected organs. In order to analyze a blood vessel, MIP is usually not sufficient, since concentric calcifications usually appear to block the entire vessel. This can be clarified by creating a cut along the centerline of the affected blood vessel in order to assess its lumen, i.e., the interior of the vessel. The cut is realized by a curved planar reformation (CPR), described in the work of Kanitsar et al. [41]. However, many CPR images have to be inspected by the radiologist in order to judge the severity of the stenosis. The number of images can be reduced, as described by Mistelbauer et al. [52], to one single static image. This technique features vessel visualizations using Curvicircular Feature Aggregation (CFA). Nevertheless, for inspecting the

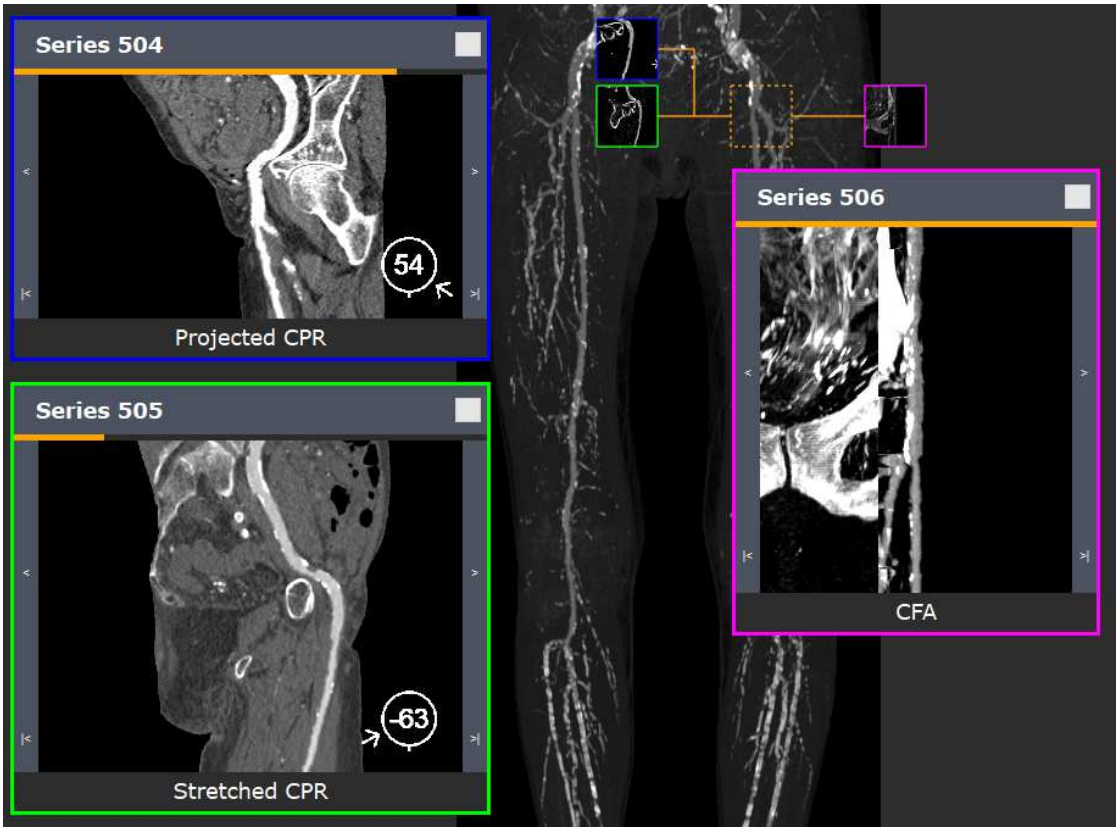


Figure 5.1: An ActiveDICOM showing a vessel stenosis highlighted in the underlying DICOM image by the dashed rectangle of the interactive menu. If the menu is expanded by the user, small preview images are shown. By selecting such an image, the corresponding integrated view is displayed. In the case of the projected and stretched CPRs, the whole series consists of multiple images that can be scrolled through. The CFA image consists of only one picture. The views are visually linked by the color of their borders.

region of the stenosis, a local CPR together with an axial slice image would be beneficial. As stated in the study by Schertler et al. [70], this would correspond to the suggested combination of methods to provide an accurate report.

In Figure 5.1 we demonstrate such a case. The underlying DICOM image is a MIP visualization of the whole vasculature of the patient, whereas the POI highlights the blood-vessel stenosis. If expanded, the user can inspect the specific region with a series of projected and stretched CPR images as well as with a single CFA image. The CFA serves as an additional orientation image, with a MIP and **Min**imum **I**ntensity **P**rojection (MinIP) aggregation on the left and right side respectively. Without the possibilities of ActiveDICOM, the radiologist would have to, either inspect the images side-by-side to compare the ROI with the stenosis, or memorize the region and perform a mental comparison. This might be sustainable for a small set of images, but with an increasing number of images and more than one ROI it will quickly

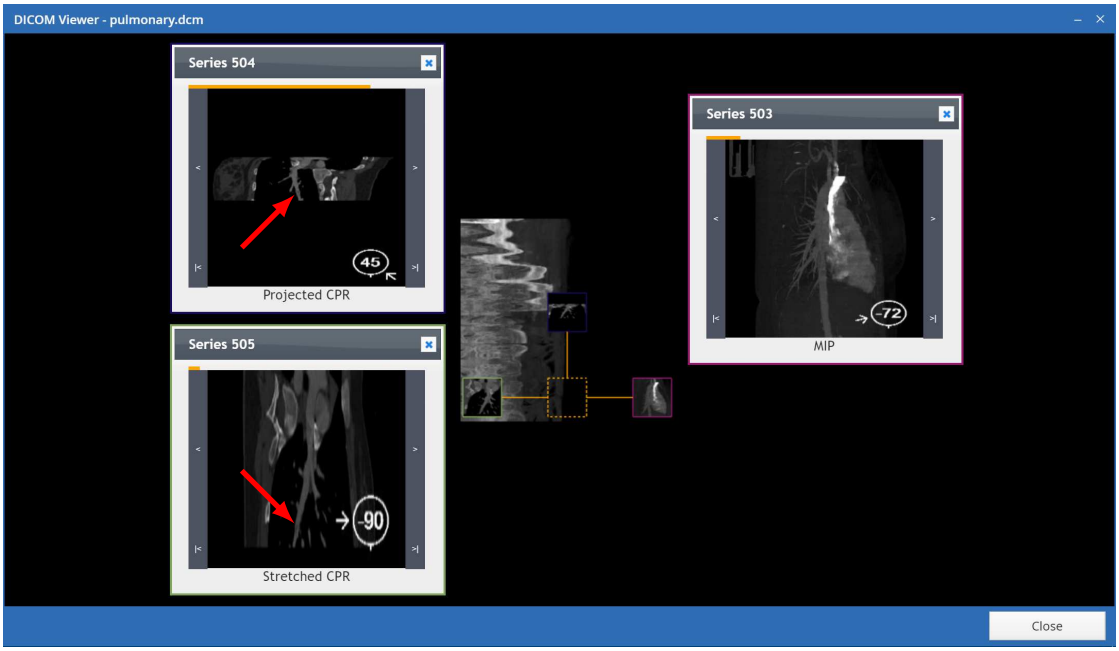


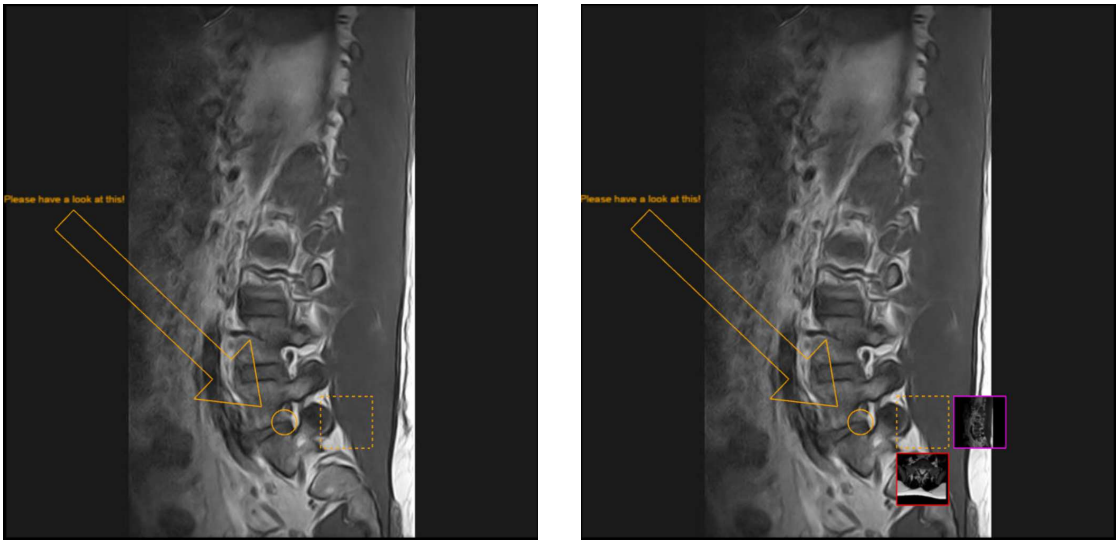
Figure 5.2: A pulmonary embolism (red arrows not generated by ADICT) analyzed within an ActiveDICOM. The underlying DICOM image is a CFA, showing the embolism at the bottom, since the MinIP to the right of CFA comes close to the centerline of the corresponding blood vessel. The interactive menu consists of two series of CPR images as well as a MIP image. Although the embolism is quite small, it can be spotted in the images.

become intractable.

5.2 Pulmonary Embolism

A different case is demonstrated in Figure 5.2 through a pulmonary embolism. An embolism is a detached clot of plaque, traveling within the blood stream. As long as it does not block a blood vessel and prevents blood from flowing, it is not dangerous. When blocking a vessel, all successive organs will lack oxygen, leading to starvation. Such situations are very dangerous, especially if they happen in the brain or lung and, therefore, should be diagnosed and treated as fast as possible.

Pulmonary embolisms might be difficult to spot and, again, various techniques at hand might aid the performance of the radiologist during medical reporting. The ActiveDICOM shown in Figure 5.2 depicts a CFA as underlying DICOM image, where the embolism is highlighted with an interactive menu. When clicking on the menu, it expands to CPR views of the specific blood vessel and a MIP view showing the thorax. In the CPR views, the user is capable of rotating around the vessel centerline, inspecting the vessel' lumen. The whole setup allows a thorough investigation of the embolism.



(a) Point of interest and menu.

(b) Expanded menu node.

Figure 5.3: Demonstration of the implemented points-of-interest (POIs) and menus. The arrow is animated and starts "blinking" as soon as the image is loaded, thus pointing the user to the desired location.

5.3 Multimedia and Interaction

Figure 3.22 demonstrates how to create an ActiveDICOM. With the developed tool chain we created a complex, interactive, and multi-media enriched ActiveDICOM (see Figure 5.3 and Figure 5.4). The ActiveDICOM features the following elements described in the thesis:

Interactive menu. The menu combines points-of-interests, guiding the user to a specific region of interest in the DICOM and an unfolding grid based interactive button gallery (referred to as grid or grid-menu). Interacting with the menu nodes reveals neighboring nodes providing successively more options. The buttons of the grid are automatically aligned and can be textured. Furthermore, the color of the border of the grid buttons, can be customized, i.e., Figure 5.3b. In our presented examples the color of the border of the grid-menu buttons matches the border color of the interactive dialogs, depicted, for instance, in Figure 5.1.

Point-of-Interest. Animated, interactive geometric primitives. The featured ActiveDICOM directs the user's attention to a certain region by drawing circles and arrows (see Figures 5.3a, 5.3b, 5.4a, and 5.4b). By adding animations to those graphical primitives the user's focus should be grasped and guided where needed.

Interactive dialogs. After selecting the rectangular nodes, displayed when interacting with the menu, dialogs are displayed. The content of these dialogs depends on the media library used for embedding data. The dialogs feature navigation buttons, a caption,

and an information bar at the bottom. Figures, 5.5a and 5.5b show all the various implemented dialogs.

Embedded videos. In order to demonstrate that even videos can be embedded into an ActiveDICOM one of the discussed dialogs involves an embedded video. The video data is embedded into the DICOM file, using our developed annotated data feature. Multiple videos can be added as well.

Embedded interactive DICOMs. It is also possible to display full featured DICOMs within an ActiveDICOM. The dialog, displaying the embedded images, is able to cycle through a collection of embedded DICOMs. All the displayed DICOMs are embedded into the file also using the annotated data feature.

Embedded documents. In order to demonstrate that whole documents, such as PDF files, can be embedded and displayed when the user interacts with the ActiveDICOM, we added a sample PDF document (see Figure 5.5b). The PDF file is embedded into the ActiveDICOM as well.

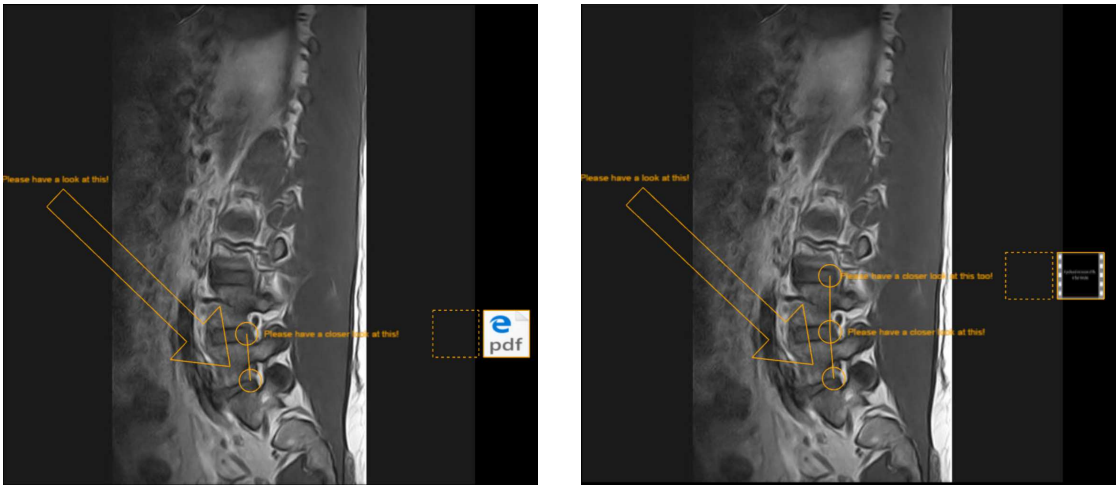
Embedded images. All the images, used as button textures for the grid-based menu are embedded in the DICOM too.

The ADICT listing for figures 5.3, 5.4, and 5.5 is based on Listing 1 from page 52. Interacting with the menu and the provided options, dialogs are shown. The information displayed in the dialogs may vary greatly. In the resulting ActiveDICOM the first two dialogs (see Figure 5.5a) display a series of further DICOMs to the user. Those embedded DICOMs are interactive. The user can change the windowing function. Figure 5.5b displays all dialogs embedded into the DICOM. It further demonstrates the possibility of embedding videos and arbitrary documents within an ActiveDICOM.

5.4 Support for Visual Analytics

In order to further showcase the capabilities of ActiveDICOM we implemented a template-engine (see Figure 5.6) for applying visual analytics. The template engine is using our Active DICOM Script and combines it with special markup instructions, similar to JSP [10] or Apache Velocity [17]. Using a new source identifier at the start of a file, “#@ADICT”, the rendering of an ActiveDICOM is different. Scripting elements have to be enclosed within @{...}. The goal of visual analytics is to present complex, big data in such a way, that it is possible to extract important/relevant information. Therefore different visualization techniques are combined. In our example we are combining HTML, diagrams rendered using D3.js [5], and interactive DICOMs. All those different technologies are glued together using our template engine. An HTML document is embedded into the DICOM featuring ActiveDICOM commands.

The implemented template engine is able to generate HTML documents and embed various resources such as:

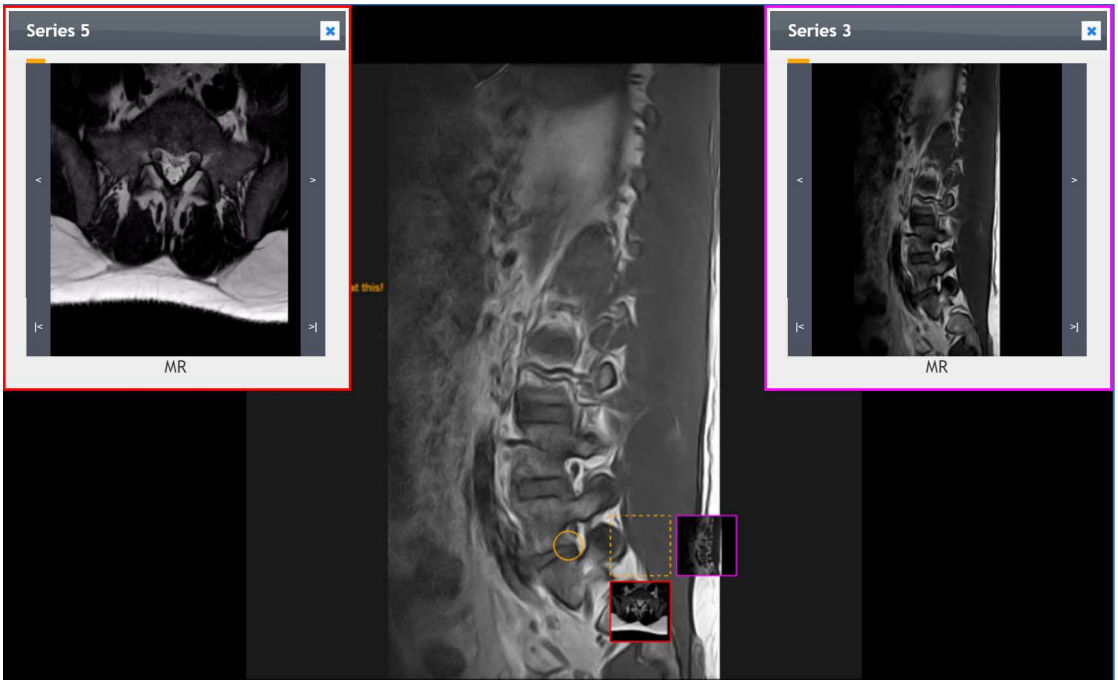


(a) (b)

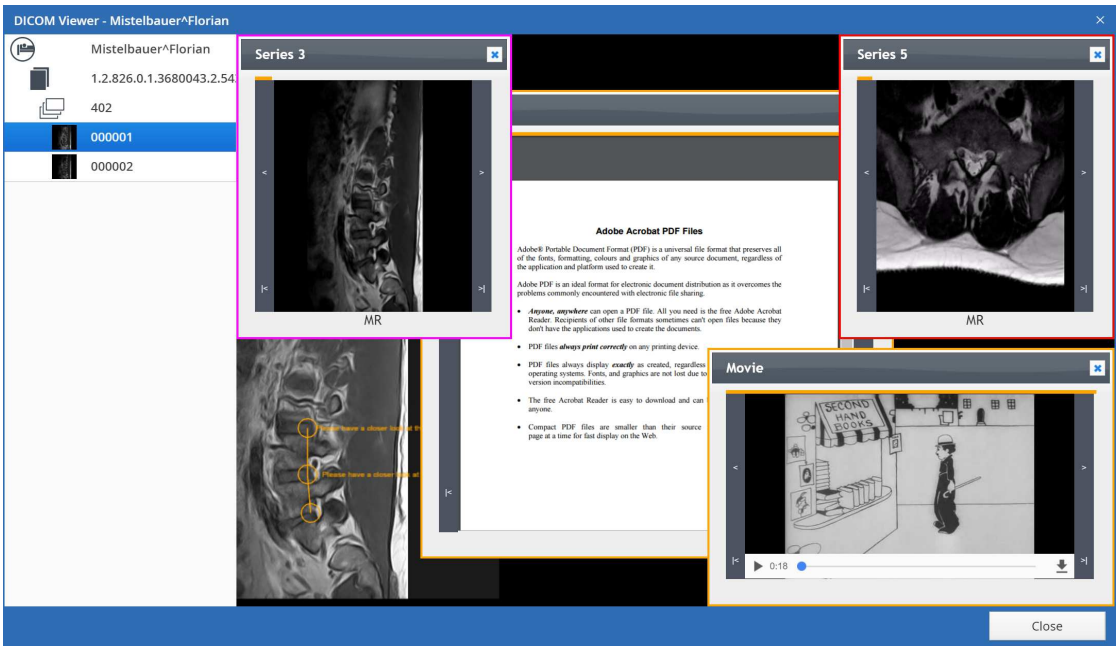
Figure 5.4: Demonstration of a fully implemented and expanded menu. It shows an embedded PDF document and a movie clip. (a) The second menu node with its expanded options. (b) The third menu node showing an embedded video option.

- **HTML + CSS.** It allows the programmer to create visually pleasing report like documents. Such reports may contain text and image information extracted from the DICOM.
- **DICOM**. Arbitrary medical images capable of being displayed by the viewer might be integrated.
- **Statistical diagrams.** Statistical diagrams can be embedded using data stored as CSV in the ActiveDICOM.
- **Images.** Even images can be embedded, using our developed annotated data elements.

Although many of our examples might be qualified to be labeled with the term visual analytics we wanted to demonstrate that is also possible to explicitly generate reports. These reports can contain various visualizations bringing information from a large data set in form, such that well informed decisions can be made in complex situations.

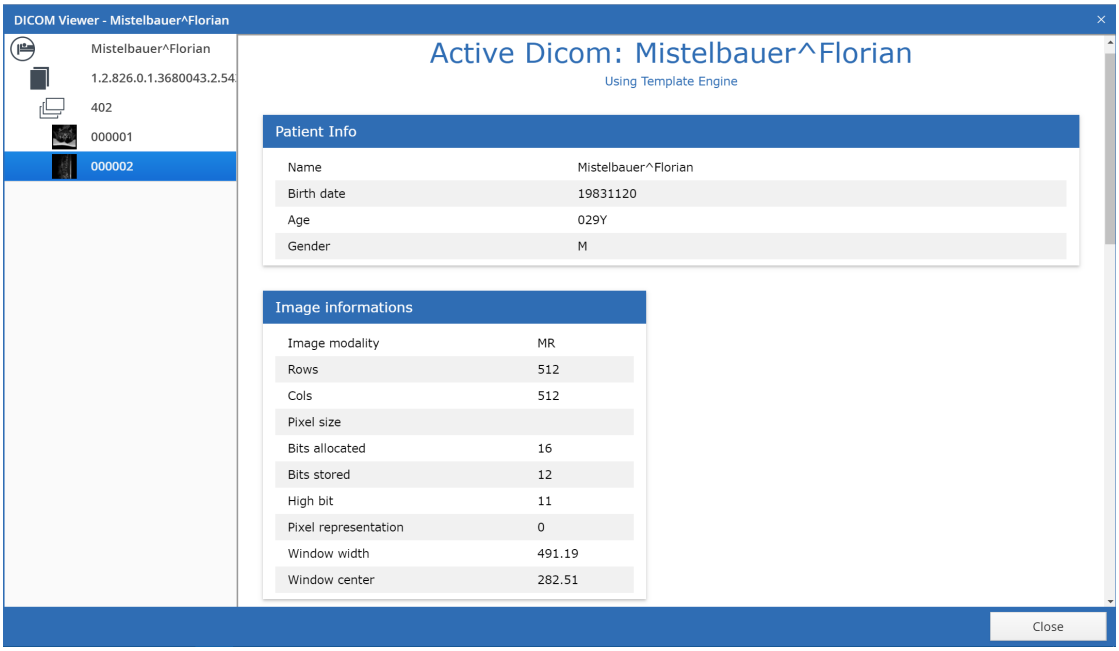


(a)

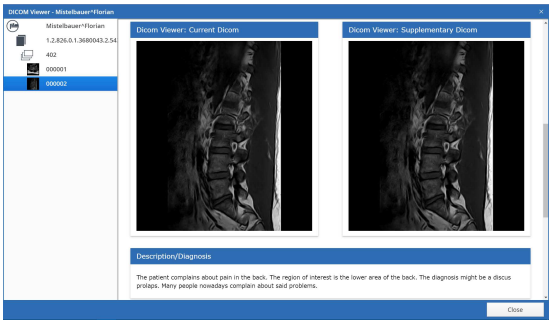


(b)

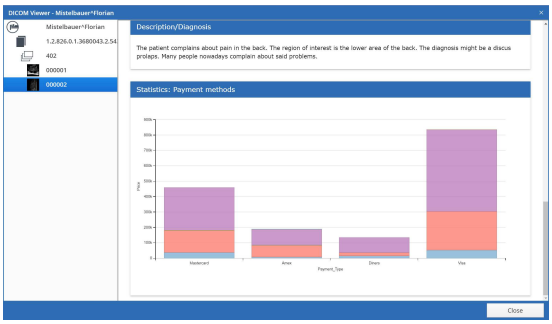
Figure 5.5: (a) Interactive DICOMs implemented in the ADICT script language. Both dialogs display a series of images, capable of changing the windowing function. With the help of the navigation buttons one can cycle through the images. (b) Demonstration of the implemented interactive concepts and multi-media elements. ActiveDICOM features a vast range of implemented embeddable elements, such as documents, movies, DICOM series. The embedded video is taken from [75].



(a)



(b)



(c)

Figure 5.6: Example, showcasing the support for visual analytics. (a) Depictions of patient information and image information displayed by rendering HTML using our template engine. (b) Rendering an HTML document using the build-in template engine. It allows for the creation of complex visualizations using embedded DICOM viewers and statistics. The embedded viewers allow the viewer the same level of control as our standard viewers shown throughout the thesis. (c) The statistic diagrams are derived from embedded CSV documents rendered using **d3.js** (data shown is a mock-up and just for demonstration purposes).

Listing 32: Interactive, multi media enriched ActiveDICOM.

```
#!ADICT
# setting the window center of the current dicom
sys:setCurrentDicomWC(200);
# setting the window width of the current dicom
sys:setCurrentDicomWW(500);
# print an arrow
local a1=poi:createArrow(-10, -130, -200, 50, 0.2);
# print text to the image
poi:textOut(-200, 70, "Please have a look at this!");
# show a circle
local c1=poi:createCircle(12, -145, 12, 100);
local c2=poi:createCircle(8, -95, 12, 100);
c2:visible = false;
local c3=poi:createCircle(8, -35, 12, 100);
c3:visible = false;
# drawing a line from c1 to c2
local l1 = poi:line(8, -95, 12, -145);
l1:visible = false;
# drawing a line from c2 to c3
local l2 = poi:line(8, -35, 8, -95);
l2:visible = false;
# text near node c2
local t2 = poi:textOut(100, -95, "Please have a closer
    look at this!");
t2:visible = false;
# text near node c3
local t3 = poi:textOut(100, -35, "Please have a closer
    look at this too!");
t3:visible = false;
#animate the arrow (blinking)
setInterval(lambda() {
    a1:visible = not (a1:visible);
}, 500);
# click handler for the circle 1
c1:click = lambda() {
    c2:visible = not (c2:visible);
    c3:visible = false;

    l1:visible = c2:visible;
    t2:visible = c2:visible;
    l2:visible = false;
    t3:visible = false;
```

Listing 33: Interactive, multi media enriched ActiveDICOM (Cont.).

```
    m1:visible = not(c2:visible) and not(c3:visible);
    m2:visible = not(c3:visible) and c2:visible;
    m3:visible = c3:visible;
};

c2:click = lambda() {
    c3:visible = not (c3:visible);
    l2:visible = c3:visible;
    t3:visible = c3:visible;

    m2:visible = not(c3:visible) and c2:visible;
    m3:visible = c3:visible;
};
# first menu root node
local m1=menu:create(72, -145);
# right node
local right_1=m1:root:leaf(1,0);
right_1:frameColor=sys:rgb(255,0,0);
# south node
local down_1=m1:root:leaf(0,-1);
down_1:frameColor=sys:rgb(255,255,0);
right_1:url=sys:getBinaryDataUrl("right_1");
down_1:url=sys:getBinaryDataUrl("down_1");
# now build the menu
m1:build();
# registering callbacks
down_1:callback=lambda(event) {
    down_1_media:toggle();
    down_1_media:first();
};

right_1:callback=lambda(event) {
    right_1_media:toggle();
    right_1_media:first();
};
# right subview
local right_1_media=media:create();
local down_1_media=media:create();

right_1_media:width(256):height(300);
right_1_media:init();
right_1_media:title("Series 3");
```

Listing 34: Interactive, multi media enriched ActiveDICOM (Cont.).

```
right_1_media:label("MR");
right_1_media:position(72, -145);
right_1_media:hide();
right_1_media:frameColor(right_1:frameColor);

down_1_media:width(256):height(300);
down_1_media:init();
down_1_media:title("Series 5");
down_1_media:label("MR");
down_1_media:position(72, -145);
down_1_media:hide();
down_1_media:frameColor(down_1:frameColor);
# embedding elements...
right_1_media:lib():embeddedDicom("dcm_s3_1");
right_1_media:lib():embeddedDicom("dcm_s3_2");
right_1_media:lib():embeddedDicom("dcm_s3_3");
right_1_media:lib():embeddedDicom("dcm_s3_4");
right_1_media:lib():embeddedDicom("dcm_s3_5");
right_1_media:lib():embeddedDicom("dcm_s3_6");
right_1_media:lib():embeddedDicom("dcm_s3_7");
right_1_media:lib():embeddedDicom("dcm_s3_8");
right_1_media:lib():embeddedDicom("dcm_s3_9");
right_1_media:lib():embeddedDicom("dcm_s3_10");
right_1_media:lib():embeddedDicom("dcm_s3_11");
right_1_media:lib():embeddedDicom("dcm_s3_12");
right_1_media:lib():embeddedDicom("dcm_s3_13");
right_1_media:lib():embeddedDicom("dcm_s3_14");
# showing the first element
right_1_media:first();

# embedding elements...
down_1_media:lib():embeddedDicom("dcm_s5_1");
down_1_media:lib():embeddedDicom("dcm_s5_2");
down_1_media:lib():embeddedDicom("dcm_s5_3");
down_1_media:lib():embeddedDicom("dcm_s5_4");
down_1_media:lib():embeddedDicom("dcm_s5_5");
down_1_media:lib():embeddedDicom("dcm_s5_6");
down_1_media:lib():embeddedDicom("dcm_s5_7");
down_1_media:lib():embeddedDicom("dcm_s5_8");
down_1_media:lib():embeddedDicom("dcm_s5_9");
down_1_media:lib():embeddedDicom("dcm_s5_10");
down_1_media:lib():embeddedDicom("dcm_s5_11");
```

Listing 35: Interactive, multi media enriched ActiveDICOM (Cont.).

```
down_1_media:lib():embeddedDicom("dcm_s5_12");
down_1_media:lib():embeddedDicom("dcm_s5_13");
down_1_media:lib():embeddedDicom("dcm_s5_14");
down_1_media:lib():embeddedDicom("dcm_s5_15");
down_1_media:lib():embeddedDicom("dcm_s5_16");
# showing the first element
down_1_media:first();
# second menu root node
local m2=menu:create(235, -95);
# right node
local right_2=m2:root:leaf(1,0);
right_2:url=sys:getBinaryDataUrl("pdf_icon");
# now build the menu
m2:build();
m2:visible = false;
# registering callbacks
right_2:callback=lambda(event) {
  right_2_media:toggle();
  right_2_media:first();
};
# right subview
local right_2_media=media:create();

right_2_media:width(512):height(512);
right_2_media:init();
right_2_media:title("Document");
right_2_media:label("");
right_2_media:position(180, -95);
right_2_media:hide();
right_2_media:frameColor(right_2:frameColor);

# embedding elements...
right_2_media:lib():embeddedDocument("embedded_pdf_1");
# third menu root node
local m3=menu:create(220, -35);
# right node
local right_3=m3:root:leaf(1,0);
right_3:url=sys:getBinaryDataUrl("video_icon");
# now build the menu
m3:build();
m3:visible = false;
```

Listing 36: Interactive, multi media enriched ActiveDICOM (Cont.).

```
# registering callbacks
right_3:callback=lambda(event) {
  right_3_media:toggle();
  right_3_media:first();
};

# right subview
local right_3_media=media:create();

right_3_media:width(426):height(240);
right_3_media:init();
right_3_media:title("Movie");
right_3_media:label("");
right_3_media:position(180, -95);
right_3_media:hide();
right_3_media:frameColor(right_3:frameColor);
# embedding elements...
right_3_media:lib():embeddedVideo("movie_1");
```

CHAPTER

6 Conclusion and Outlook

In this thesis, we proposed a novel approach for enhancing DICOM images with interactive elements. For this reason, we designed a language, called ADICT, which acts as a glue between data, interaction, and visualization. By extracting and translating ADICT to specific target languages such as JavaScript, we presented a web-based viewer for ActiveDICOM. We proposed several interactive elements, encoded into the image itself and displayed by our viewer. Hence, we render static images active, enhancing the users' possibilities to explore the underlying data, as demonstrated in our results. Adhering to the definition of visual analytics we even support combining various visualizations, such as statistics, images, HTML, and DICOMs into one report like ActiveDICOM. Furthermore our examples demonstrated how one can create content and feature rich interactive medical images guiding the user's attention to certain regions of interest. With our annotated embedded data mechanism it is possible to create rich data heavy interactive medical images for various medical examination purposes. Using our implemented template engine it is possible to create reports integrating various interactive media objects in a visually pleasing way. This way it is possible to create appealing overviews which might incorporate data from a vast array of different sources, e.g., linking/embedding different DICOMs not necessarily embedded in the main image.

Future improvements involve a more powerful language, enriched by types and more complex annotations, such as encryption and quality-measuring ones. Annotations could act as preconditions or postconditions, partial encryption or decryption, and as addition of arbitrary embedded media. Our ActiveDICOM viewer should be extended to support multi-layered images and provide features for collaborative working, such as user-specified text annotations. Additionally, we want to improve the searching procedure of the viewer to navigate through a large-scale collection of DICOM images. Future features of ActiveDICOM should include thumbnail previews, embedded videos of various formats, or even PDF export. Exporting a whole viewer in a PDF document was one of the reasons a scripting language was developed in order to act as an abstraction layer for different output targets.

All ActiveDICOMs presented in this thesis have been manually composed using the developed software. Nevertheless, our intended future goal is to provide templates for various

application scenarios, in order to render this process transparent to the users, i.e., physicians or assistants. Another future aspect is a graphical assistant to further supplement this process, following the "**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et" principle.

List of Figures

List of Tables

Bibliography

[1] Apache Ivy™. The agile dependency manager. http://ant.apache.org/ivy/. Accessed: 2016-12-02.

[2] Apache Tomcat®. https://tomcat.apache.org/index.html. Accessed: 2016-12-02.

[3] AuntMinnie. http://www.auntminnie.com/. Accessed: 2017-01-10.

[4] clariPACS: Fast web PACS. http://www.claripacs.com/. Accessed: 2017-01-10.

[5] Data-Driven Documents. https://d3js.org/. Accessed: 2017-01-02.

[6] Eclipse. https://eclipse.org/. Accessed: 2016-12-02.

[7] ECMAScript. https://en.wikipedia.org/wiki/ECMAScript. Accessed: 2016-09-04.

[8] Hibernate. Everything data. http://hibernate.org/. Accessed: 2016-12-02.

[9] Java SE Development Kit 8. http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. Accessed: 2016-12-02.

[10] Java Server Pages. http://www.oracle.com/technetwork/java/javaee/jsp/index.html. Accessed: 2017-01-02.

[11] JavaScript 3D library. https://threejs.org/. Accessed: 2016-12-02.

[12] Jersey. RESTful Web Services in Java. https://jersey.java.net/. Accessed: 2016-12-02.

[13] jQuery. https://jquery.com/. Accessed: 2016-12-02.

[14] jQuery UI. https://jqueryui.com/. Accessed: 2016-12-02.

[15] MySQL. http://www.mysql.com/. Accessed: 2016-12-02.

[16] Open Source Clinical Image and Object Management. http://www.dcm4che.org/. Accessed: 2016-12-02.

[17] The Apache Velocity Project. http://velocity.apache.org/. Accessed: 2017-01-02.

[18] The four essential freedoms. https://www.gnu.org/philosophy/free-sw.en.html. Accessed: 2017-02-015.

[19] The Open Source Definition. http://opensource.org/osd. Accessed: 2017-02-015.

[20] The Open Source Definition. https://opensource.org/osd. Accessed: 2016-12-02.

[21] TypeScript. http://www.typescriptlang.org/index.html. Accessed: 2016-11-30.

[22] Vaadin - thinking of U and I. https://vaadin.com/home. Accessed: 2016-09-04.

[23] Gennady Andrienko, Jean-Daniel Fekete, Carsten Görg, Daniel Keim, Jörn Kohlhammer, and Guy Melançon. *Information Visualization Human-Centered Issues and Perspectives*, volume 4950 of *LNCS State-of-the-Art Survey*, chapter Visual Analytics: Definition, Process, and Challenges, pages 154–175. Springer, 2008.

[24] E.J.C. Arguiñarena, J.E. Macchi, P.P. Escobar, M. del Fresno, J.M. Massa, and M.A. Santiago. Dcm-Ar: A fast flash-based web-PACS viewer for displaying large DICOM images. In *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 3463–3466, Aug 2010.

[25] J.-P. Balabanian, I. Viola, and M. E. Gröller. Interactive illustrative visualization of hierarchical volume data. In *Proceedings of Graphics Interface*, pages 137–144, 2010.

[26] T. Becker, D. Onnasch, and R. Simon. Interoperability for image and non-image data in the DICOM standard investigated from different vendor implementations. In *Proceedings of Computers in Cardiology*, pages 675–678, 2001.

[27] M.A. Bochicchio, A. Longo, and L. Vaira. Extending web applications with 3d features. In *Proceedings of the 13th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 93–96, Sept 2011.

[28] Gong Chao. Human-computer interaction: Process and principles of human-computer interface design. In *Proceedings of the International Conference on Computer and Automation Engineering (ICCAE)*, pages 230–233, March 2009.

[29] crazypaco. GE LightSpeed CT scanner at Open House, Monroeville, Pennsylvania. Wikipedia, the free encyclopedia, 2012. Accessed: 2017-01-05.

[30] Drgnu23. Conversion of a DICOM-format X-ray from a patient of User Drgnu23, a ten year old male. This is the patient's left hand, posterior-anterior projection. Wikipedia, the free encyclopedia, 2005. Accessed: 2017-01-05.

[31] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[32] Edward J. Escott and David Rubinstein. Free DICOM Image Viewing and Processing Software for Your Desktop Computer: What's Available and What It Can Do for You. *RadioGraphics*, 23(5):1341–1357, 2003. PMID: 12975521.

[33] Josep Fernàndez-Bayó, Octavio Barbero, Carles Rubies, Melcior Sentís, and Lluis Donoso. Distributing medical images with internet technologies: A DICOM web server and a DICOM java viewer. *RadioGraphics*, 20(2):581–590, 2000. PMID: 10715352.

[34] A. Gentile, A. Santangelo, S. Sorce, and S. Vitabile. Novel human-to-human interactions from the evolution of HCI. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 600–605, 2011.

[35] D. Golubovic, G. Miljkovic, S. Miucin, Z. Kaprocki, and V. Velisavljev. WebGL implementation in WebKit based web browser on Android platform. In *Proceedings of Telecommunications Forum (TELFOR)*, pages 1139–1142, 2011.

[36] Daniel Haak, Charles-E. Page, and Thomas M. Deserno. A survey of dicom viewer software to integrate clinical research and medical imaging. *Journal of Digital Imaging*, 29(2):206–215, 2016.

[37] Gabor T. Herman. *Fundamentals of Computerized Tomography*. Springer-Verlag London, 2. edition, 1980.

[38] Hg6996. PET/CT-System with 16-slice CT. Wikipedia, the free encyclopedia, 2009. PET/CT-System with 16-slice CT; the ceiling mounted device is an injection pump for CT contrast agent.

[39] Jan Ainali. Philips MRI in Sahlgrenska Universitetsjukhuset, Gothenburg, Sweden. Wikipedia, the free encyclopedia, 2008. Accessed: 2017-01-05.

[40] Jens Maus. This is a transaxial slice of the brain of a 56 year old patient (male) taken with positron emission tomography (PET). Wikipedia, the free encyclopedia, 2010. Accessed: 2017-01-05.

[41] A. Kanitsar, D. Fleischmann, R. Wegenkittl, P. Felkel, and Meister E. Gröller. CPR - curved planar reformation. In *Proceedings IEEE Visualization*, pages 37–44, 2002.

[42] Michael Kaserer. DICOM Web Viewer. July 2013.

[43] Daniel Keim, Jörn Kohlhammer, Geoffrey Ellis, and Florian Mansmann. *Mastering the Information Age - Solving Problems with Visual Analytics*. Wiley, 1 edition, 2010.

[44] Kieran Maher. Examples of T1 weighted, T2 weighted and PD weighted MRI scans. Wikipedia, the free encyclopedia, 2006. Accessed: 2017-01-05.

[45] R. I. Kitney, S. Claesen, and J. Halls. A comprehensive web-based patient information environment. In *Proceedings of the 23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, volume 4, pages 3584–3585, 2001.

[46] P. Kohlmann, S. Bruckner, A. Kanitsar, and M. E. Gröller. LiveSync: deformed viewing spheres for knowledge-based navigation. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1544–1551, 2007.

[47] P. Kohlmann, S. Bruckner, A. Kanitsar, and M. E. Gröller. LiveSync++: enhancements of an interaction metaphor. In *Proceedings of Graphics Interface*, pages 81–88, 2008.

[48] P. Kohlmann, S. Bruckner, A. Kanitsar, and M. E. Gröller. Contextual Picking of Volumetric Structures. In *Proceedings of IEEE Pacific Visualization*, pages 185–192, 2009.

[49] E.J. Melicio Monteiro, C. Costa, and J. Lues Oliveira. A DICOM viewer based on web technology. In *Proceedings of the IEEE 15th International Conference on e-Health Networking, Applications Services (Healthcom)*, pages 167–171, Oct 2013.

[50] Mikael Häggström. Computed tomography of human brain. Wikipedia, the free encyclopedia, 2008. Computed tomography of human brain, from base of the skull to top. Taken with intravenous contrast medium.

[51] Gabriel Mistelbauer, Hamed Bouzari, Rüdiger Schernthaner, Ivan Baclija, Arnold Köchl, Stefan Bruckner, Milos Srámek, and Meister Eduard Gröller. Smart Super Views - A Knowledge-Assisted Interface for Medical Visualization. In *Proceedings of the IEEE Conference on Visual Analytics Science and Technology (IEEE VAST)*, pages 163–172, 10 2012.

[52] Gabriel Mistelbauer, Anca Morar, Andrej Varchola, Rüdiger Schernthaner, Ivan Baclija, Arnold Köchl, Armin Kanitsar, Stefan Bruckner, and Meister Eduard Gröller. Vessel Visualization using Curvicircular Feature Aggregation. *Computer Graphics Forum*, 32(3):231–240, 2013.

[53] M.M. Mobeen and Lin Feng. High-Performance Volume Rendering on the Ubiquitous WebGL Platform. In *Proceedings of IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 381–388, June 2012.

[54] Paul Nagy. Open source in imaging informatics. *Journal of digital imaging*, 20(1):1–10, 2007.

[55] National Electrical Manufacturers Association. The DICOM Standard. http://medical.nema.org/standard.html, 2011. Accessed: 2017-01-05.

[56] Meredith Northam, James Koonce, and James G. Ravenel. Pulmonary nodules detected at cardiac CT: comparison of images in limited and full fields of view. *American Journal of Roentgenology*, 191(3):878–881, Sep 2008.

[57] Alex Olwal, Oscar Frykholm, Kristina Groth, and Jonas Moll. Design and Evaluation of Interaction Technology for Medical Team Meetings. In *Proceedings of the 13th IFIP TC 13 International Conference on Human-computer Interaction - Volume Part I*, INTERACT'11, pages 505–522, 2011.

[58] Horst R. Portugaller, Helmut Schoellnast, Klaus A. Hausegger, Kurt Tiesenhausen, Wilfried Amann, and Andrea Berghold. Multislice spiral CT angiography in peripheral arterial occlusive disease: A valuable tool in detecting significant arterial lumen narrowing? *European Radiology*, 14(9):1681–1687, 2004.

[59] Bernhard Preim and Charl Botha. *Visual Computing for Medicine*. Morgan Kaufmann, 2014.

[60] PhilippeA. Puech, Loïc Boussel, Samir Belfkih, Laurent Lemaitre, Philippe Douek, and Régis Beuscart. DicomWorks: Software for reviewing dicom studies and promoting low-cost teleradiology. *Journal of Digital Imaging*, 20(2):122–130, 2007.

[61] Honea R., McCluggage Charles W., Parker B., O'Neall D., and Shook Keith A. Evaluation of commercial PC-based DICOM image viewer. *Journal of Digital Imaging*, 11:151–155, 1998.

[62] D. Radosevic and B. Klicek. Development of a Higher-Level Multimedia Scripting Language. In *Proceedings of the 23rd International Conference on Information Technology Interfaces*, pages 201–208, 2001.

[63] P. Rautek, S. Bruckner, and M. E. Gröller. Semantic Layers for Illustrative Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1336–1343, 2007.

[64] P. Rautek, S. Bruckner, and M. E. Gröller. Interaction-Dependent Semantics for Illustrative Volume Rendering. *Computer Graphics Forum*, 27(3):847–854, 2008.

[65] Jonathan C. Roberts. State of the Art: Coordinated & Multiple Views in Exploratory Visualization. In *Proceedings of the International Conference on Coordinated & Multiple Views in Exploratory Visualization*, pages 61–71, 2007.

[66] Timo Ropinski, Ivan Viola, Martin Biermann, Helwig Hauser, and Klaus Hinrichs. Multimodal Visualization with Interactive Closeups. In *Proceedings of the Conference on Theory and Practice of Computer Graphics*, pages 17–24, 2009.

[67] D. J. Rosenkrantz and R. E. Stearns. Properties of Deterministic Top Down Grammars. In *Proceedings of the First Annual ACM Symposium on Theory of Computing*, STOC '69, pages 165–180, New York, NY, USA, 1969. ACM.

[68] Antoine Rosset, Luca Spadola, and Osman Ratib. OsiriX: An open-source software for navigating in multidimensional DICOM images. *Journal of Digital Imaging*, 17(3):205–216, 2004.

[69] J. J. Saleem, A. L. Russ, P. Sanderson, T. R. Johnson, J. Zhang, and D. F. Sittig. Current challenges and opportunities for better integration of human factors research with development of clinical information systems. *Yearbook of Medical Informatics*, 1(4):48–58, 2009.

[70] Thomas Schertler, Thomas Frauenfelder, Paul Stolzmann, Hans Scheffel, Lotus Desbiolles, Borut Marincek, Vladimir Kaplan, Nils Kucher, and Hatem Alkadhi. Triple rule-out CT in patients with suspicion of acute pulmonary embolism: Findings and accuracy. *Academic Radiology*, 16(6):708–717, Jun 2009.

[71] Johanna Schmidt, Eduard Gröller, and Stefan Bruckner. VAICo: Visual analysis for image comparison. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2090–2099, 2013.

[72] Markus Steinberger, Manuela Waldner, Marc Streit, Alexander Lex, and Dieter Schmalstieg. Context-preserving visual links. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2249–2258, 2011.

[73] D. Stiller, R. Shupe, J. deHaan, and D. Richardson. *The ActionScript 3.0 Quick Reference Guide: For Developers and Designers Using Flash*. O'Reilly Media / Adobe Dev Library, 2008.

[74] Roberto Stramare, Giuliano Scattolin, Valeria Beltrame, Marco Gerardi, Marco Sommavilla, Cristina Gatto, Paolo Mosca, Leopoldo Rubaltelli, Carlo Riccardo Rossi, and Claudio Saccavini. Structured reporting using a shared indexed multilingual radiology lexicon. *International Journal of Computer Assisted Radiology and Surgery*, 7(4):621–633, 2012.

[75] Sullivan, Pat (director) / Universal , Keen Cartoon . Charlie in Turkey. http://www.openimages.eu/media/685703/Charlie_in_Turkey, 1919. Cartoon movie (animated film) starring the tramp Charlie. Running time: 9:39. Accessed: 2017-01-14.

[76] Chia-Chi Teng. Managing DICOM image metadata with desktop operating systems native user interface. In *Proceedings of the 22nd IEEE International Symposium on Computer-Based Medical Systems (CBMS)*, pages 1–5, Aug 2009.

[77] I. Viola, M. Feixas, M. Sbert, and M. E. Gröller. Importance-Driven Focus of Attention. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):933–940, 2006.

[78] C.D. Von Land, V. Lashin, A. Oriol, and J.J. Villanueva. Object-oriented design of the DICOM standard and its application to cardiovascular imaging. In *Proceedings of Computers in Cardiology*, pages 645–648, 1997.

[79] J. Walkenbach. *Excel VBA Programming For Dummies*. Wiley, 3. edition, 2013.

[80] X.Compagnion. Embryo at 14 weeks (profile). Wikipedia, the free encyclopedia, 2006. Accessed: 2017-01-05.

[81] Xianyi Yang and Guo Chen. Human-Computer Interaction Design in Product Design. In *Proceedings of the First International Workshop on Education Technology and Computer Science (ETCS)*, volume 2, pages 437–439, March 2009.

[82] R K Zeman, H Lyshkow, B S Garra, and T Gillespy. Viewing DICOM-compliant CT images on a desktop personal computer: use of an inexpensive DICOM receive agent and freeware image display applications. *American Journal of Roentgenology*, 172(2):305–308, 1999.