FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

AngioVis Patient Persistency

Object Relation Mapping for Large Relational Datasets of Binary Data

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

by

Manuel Andre
0925715
in
Medical Informatics

Georg Ursits
1025726
in
Media Informatics and
Visual Computing

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Univ.–Doz. Dipl.–Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.–Ing. Gabriel Mistelbauer

Vienna, September 29, 2013

(Signature of Author) (Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

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.

(Ort, Datum)	(Unterschrift Verfasser)
(Ort, Datum)	(Unterschrift Verfasser)

Abstract

A usual workstation in medicine processes a significant amount of patients per day. It is critical in a clinical daily routine, to retrieve, store and access analysed data in a fast pace. An application in this context needs to be responsive and speed up current processes. Due to the necessity of storing all these data, current development in database technologies provides opportunities to improve their management. In this thesis, we will investigate, how such a technologies can be transferred to a specific application scenario. Furthermore there are several application parameters, like load time, response delay and integrity of the stored data that can be improved, to have a positive impact on a medical workflow. Apart those performance related parameters there are several other factors like extendibility, scalability and structure that are taken into consideration as well.

Contents

List of Figures

CHAPTER 1

Introduction

AngioVis (**A**ngiographic **Vis**ualization) is a diagnosis tool for visualisation of large peripheral *CTA* (**C**omputer **T**omographic **A**ngiography[1]) datasets. The main goal of this framework is to detect and categorise arterial diseases for clinical diagnosis and treatment planning. The framework uses the *DICOM* (**D**igital **I**maging and **Co**mmunications in **M**edicine) Standard which was developed by the DICOM Standards Committee. This Standard specifies the interaction between two or more devices, also across the network (with TCP/IP) in a specific file format as explained in Chapter 3.

1.1 Motivation

Arterial diseases becomes more frequent over the last years. With the AngioVis framework it is possible to load several *DICOM* files. These files will be converted into the f3d format for further processing. The f3d file format contains the 3D volume data for the volume rendering pipeline. Beside this pipeline there are many more others implemented. Through these different rendering pipelines the framework requires different persistency strategies. The framework consists of many plugins. Every plugin has a different job and is connected to the PluginManager of AngioVis. This manager handles a plugin's lifecycle, that basically means creates, deletes or registers a new plugin instance. Figure 1.1 shows a schematic representation of the AngioVis program layout in the context of PatientDB.

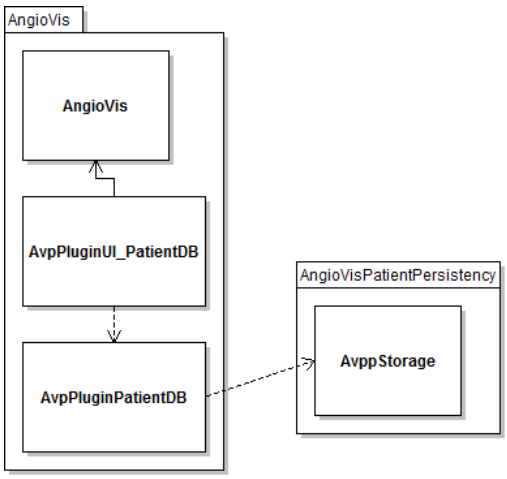

Figure 1.1: AngioVis PatientDB Draft

[1] visualisation of arteries, veins of interest (medical imaging technique)

The main goal of this thesis it to be as generic and as extendible as possible. Our meta requirements are an easy scalable architecture that stores many types of data especially huge amount of binary data. Another requirement is that available data from CT equipment can be imported, or, in other words, that available data can be inserted into the database.

To fit as many requirements, technically and functional as possible, we intend to build a layer to abstract the underlying persistency functionality, it is called AngioVisPatientPersistency (see Figure 1.1). The produced binary is loosely coupled to the main framework and is managed by the plugin mechanism.This plugin solves two inherent issues:

- offer the relational organization of patient meta data acquired from DICOM files
- manage certain specific files like the workspace of a patient, with the corresponding data and vessel tree

A similarity of all persistency layers is that they need to be interoperable, that means our target is to implement a layer which is built upon a *RDBMS* in the context of AngioVis. The aim of AngioVisPatientPersistency is, to be cross platform, allowing the above mentioned scale out and avoiding to be an architecture breaker for future extensions. The persistency layer might serve as a facade for a future client server architecture.

Persistence is one of the most important properties of a program that interacts with or stores data in a database. There are several important milestones during the development of such a system in order to establish or keep data integrity. How can huge amount of binary data be stored in a database? The communication between the program itself and the database shall be as fast as possible with a minimum of database requests (Chapter 3.3 - Caching). Because of that fact and the questions mentioned above there are several problems to solve to get an efficient and for the radiologists and physicians transparent workflow between the program and the database.

1.2 Problem Statement

Considering the in the motivation described environment the problem we solve, is the refactoring and optimization of a persistency solution heavily based on XML. The current solution stores numbers of XML-Files on the hard disk. One of the biggest limiting factors of this implementation is the slow read/ write speed of the hard disk, without proper optimizations. Beside the architecture being not scalable with respect to further extensions, every Workstation stores these XML-Files and forms a non managed decentralised architecture. The discussion of advantages or disadvantages of a decentralised or centralised architecture does never end. With a centralised architecture (client/ server) or with a data driven architecture (Chapter 3.1) the data is stored in a database, which can offer the best of both approaches. One the one hand it can guarantee strict integrity constraints, and on the other hand it can scale out like a decentralised architecture via clustering. Our program is based on a data driven architecture which replaces the XML-based infrastructure through three new Entities based on the DICOM Standard [5]:

Patient: The Patient entity contains Attributes which are associated with the Patient Information for example Patient's name (further details, see Chapter 3.1).

Study: The Study entity is connected with the Patient through the Patient's Primary Key with certain Study Information for example description, date and time (further details, see Chapter 3.1).

Series: The Series entity is connected with the Study through the Study's Primary Key with certain Series Information (further details, see Chapter 3.1).

The workflow of AngioVis shall be that several files (*.avw[2] , *.f3d[3] and vesseltree.xml[4]) will be loaded through the AngioVis framework. The plugin manager redirects these files to the AngioVis Patient Persistency Plugin. This plugin transforms their filenames with a obfuscation method (see Chapter 3.4) to a random string. After the obfuscation every file will be stored in the database and on the hard disk on a specific working directory. Storing this huge amount of binary data in the database is a challenge, because complete files will be stored. For example *.f3d-Files consuming a lot of disk space. Consider the scenario that these *.f3d-Files will be stored in the database. There are several advantages and disadvantages of storing binary data in a database or on the hard disk. On the one hand the binary data will be stored in the database so a unknown person cannot access information in an easy manner. But one of the disadvantages are that the database, or, in our, case the program cache (see Chapter 3.3) will be overloaded with a mass of binary data. On the other hand, the binary data will be stored on the hard disk, which implies the necessity to take care to obfuscate sensitive information. It is much easier to take a backup or to archive the working directory after a working day compared to a database setup. If the binary data will be stored in the database the whole database has to be backed up completely and brings further problems with it.

[2]Represents the AngioVis-Workspace and stores informations regarding the current working session.

[3]Is a specific internal data format of AngioVis for storing the 3D volume in an optimised way.

[4]Contains structured data to specially represent the vessels extracted from the 3D volume. [1]

Considering these issues, a possible solution to store the binary data needs to evaluated. One possible solution would be to store the data in a specific working directory on the hard disk. Only filenames will be stored in the database and because of the working directory it is easy to find and load the requested file. This solution and many more need to be considered and evaluated to optimise the current setup. Figure 1.2 shows the workflow of the reimplemented PatientDB. First of all files are passed through by the AngioVis framework to the new implemented plugin which takes care of them. These files will be stored in two locations, on the one hand in the database and on the other hand on the hard disk (see Chapter 3.4).

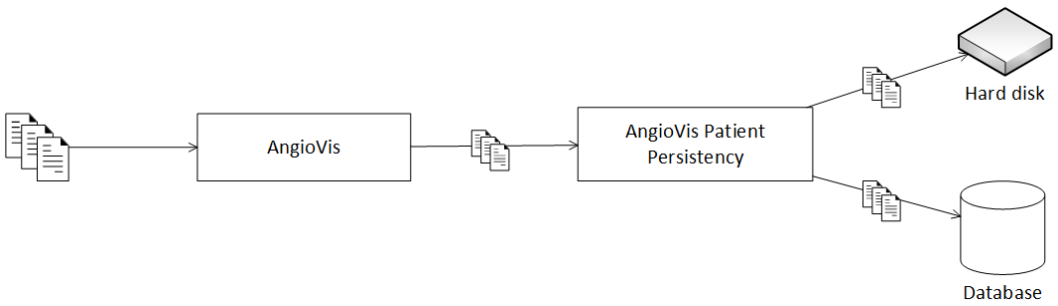

Figure 1.2: AngioVis PatientDB Data Flow

CHAPTER 2

State of the Art

This c PatientDB. To cover the most significant aspects, this section is divided into three topics, *Layered Architecture* and it's influence on our software design, persistency relevant design patterns and best practices, and finally the problems of mapping relational data into programming language entities.

The choice of a layered architecture for a highly data driven application is standing to reason for a structured application design reasoned by the following deliberations. Every layer fulfils a specific purpose, handles a definite set of problems and communicates with its preceding or succeeding layer or set of layers. Layers are an easy mechanism to encapsulate functionality and differentiate each of them via clearly defined interfaces. There are numerous interpretations of this paradigms nowadays as discussed in Savolainen et al [11]. A meta view on the vast amount of layers, discussed in [11], allows a rough categorisation in three types of layers, starting from the bottom the Persistency layer, building on top of it Business Logic layer and the uppermost Presentation layer as depicted in Figure 2.1.

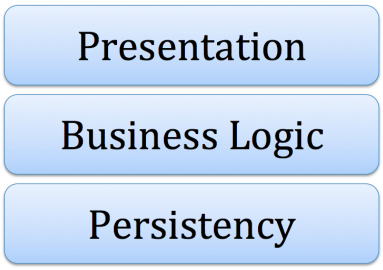

Figure 2.1: layered architecture

The AngioVis main framework covers the upper two layers, Presentation and Business Logic of the stack shown in Figure 2.1 The PatientDB Plugin implements the persistency layer for AngioVis, which is the foundation of any layered architecture. It's purpose is to ensure integrity and identity of stored data in the context of layered architecture. In terms of the above stated rough categorisation the PatientDB Plugin realises the persistency layer. As mentioned in Savolainen et al [11] there is a tendency to split every layer up into sublayers to define an application architecture of fine granularity.

Beside the layered architecture there are several other best practices and software development patterns that apply quite well to the requirement of an *ORM* (**O**bject **R**elation **M**apping). The automated generation of *ORM* code is quite complex and incorporates various mechanism such as correctness verification, type deduction based on available meta information of a *SQL*

(**S**tructured **Q**uery **L**anguage) schema, and relational algebra as discussed in Mehra et al [9]. Irrespective of the type hierarchie's origin, formalised in an *ERD* (**E**ntity **R**elation **D**iagram), there is one pattern, the *DAO* (**D**ata **A**ccess **O**bject) pattern, that fits the purpose of an extendible *ORM* outstandingly well. The main reason for using a *DAO* pattern is to differentiate the code and or mechanism to actually retrieve the data of various entity's. All implications and side effects of this process are completely separated from the objects that are used for the internal flow of the application. Cheng [7] discusses an adapted version of the traditional *DAO* pattern to overcome certain limitations and restrictions regarding the dynamic extendibility of an application architecture.

During implementation of an *ORM* there is one essential issue, namely the *ORM Impedance Mismatch* [13]. It is intrinsically hard to realise a data mapping from a table-based kind of data to a object oriented type of entity. While facing this problem, there is a indirect correlation between handling special characteristics of the underlying *RDBMS* and a dynamically typed highly abstracted application architectures. There are many possible solutions to cope with the intrinsic issues of implementing an *ORM*. Atwook [6] discusses a set of possible solutions and their implications from an implementation perspective.

CHAPTER 3

Method

This Chapter discusses the overall object oriented architecture of the PatientDB plugin. Furthermore, an object oriented design formalises real world entities and represents them in an intuitive way.

The PatientDB plugin is part of a layered architecture and is internally structured into layers as well, depicted in Figure 3.1. PatientDB represents the persistency layer in the context of AngioVis. The first sublayer contains all entities, Patient, Study, Series, File and FileType, that represent the relational part of data stored by the PatientDB Plugin. This circumstance is depicted in Appendix C.2. On top of this layer builds the second sublayer that contains all *DAO* classes. All *DAO*s are described as abstract classes. Every single entity's *DAO* therefore describes what functionality is offered and what the interface looks like. All abstract *DAO*s provide the recipe to implement further mechanisms of storing data, for example to use different databases. All *DAO*s have one default implementation for the underlying MySQL Database. Further ways of storing the data can be added with ease by providing additional implementations of the abstract *DAO*. The third and last sublayer encapsulates the management layer that implements all mapped business processes and advanced mechanisms like *Lazy Loading*, certain caching strategies and persisting binary and relational data in an underlying heterogeneous storage.

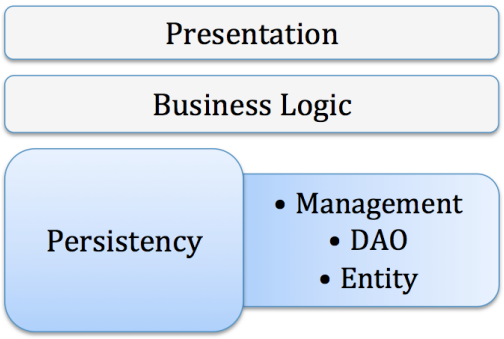

Figure 3.1: Layered Architecture - Persistency Layer in greater detail

The full stack of all these three sublayers provides the the mapping from objects to relations needed and implements the business process model required by AngioVis. The following chapters further explain the various fields covered by this persistency layer and provide implementation details.

3.1 Data Driven Design

The *NEMA* (**N**ational **E**lectrical **M**anufacturers **A**ssociation) started a cooperation with the *ACR* (**A**merican **C**ollege of **R**adiology) in 1983 to develop the *DICOM* standard. This industry leading standard for clinical applications defines basic approaches to

- *„... 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 systems of hospital information*
- *Allow the creation of diagnostic information data bases that can be interrogated by a wide variety of devices distributed geographically. ...“*

DICOM standard's introduction [3]

Hence the standard defines how data from different vendors is encoded, stored and transmitted. Furthermore a stack of meta data is standardised as well. Every entity defined by the *DICOM* standard possesses a certain set of attributes describing the modality of each individual entity. The standard even goes one step further in specifying each entity and provides verbose explanations instead of only defining each attribute. Each entity and attribute is described in a clinical context to give precise decisions of the actual data format. After this description that would suffice to deduce technical formal definitions like *ERD*s or other data modelling diagrams, small parts of these diagrams are drafted to communicate actual entities, relations and their cardinalities. [4] [5]

Analysing the reference implementation by *DICOM@OFFIS*, the *DCMTK* (**DI**C**OM**- **T**ool**k**it [2]), is an easy way to explore the *DICOM* standard in greater detail. The *DCMTK* is written in C++ which is beneficial for usage with the AngioVis environment, because it seamlessly integrates into the current framework. The use of this *API* significantly eases the process of parsing *DICOM* files, compared to writing a parser from scratch. The detailed standard and the implementation *DCMTK*, practically predefines the entities mapped into the database schema depicted in Appendix B. Our implementation does not fully store the vast amount of attributes provided by DICOM, but rather a certain subset. Attributes are stored if they contribute by identifying a data set for the actual end user, or contribute to visualization purposes.

Exactly three entities have been deduced from the *DICOM* standard, patient, study and series. All three entities serve valuable information for associating the *DICOM* file to a patient in a structured way. Based on the cardinalities[1] and the overall structure suggested by the standard [5], these three entities form a graph that strictly matches the characteristics of a tree (ever node has a set of children, only one parent and the whole graph does not have any circles). Each entity, described in the order below, takes up on level of the tree's depth. On each level multiple entities of the same type reside and are connected to exactly one parent entity and possibly multiple child entities. Exactly this hierarchy is described in the *DICOM* standard as well [5,

[1] In mathematics, the cardinality of a set is a measure of the „number of elements of the set“. [12]

46]. To match the requirements provided by the problem statement (see Chapter 1) two further entities are introduced. These two entities (File and FileType) are the leaves of the entity tree. These entities allow to connect binary data to the relational data from the standard.

Patient: This entity provides the patient's id and name, defined by the *DICOM* file. The id in this context is not the primary key but rather the id given by a clinical origin, „*Primary hospital identification number or code for the patient.*". The patient's name is simply the full name of the corresponding patient. Every Patient can have multiple associated Studies. [4]

Study: This entity provides the uid, description, date and time. The uid according to the standard, „*The unique identifier for the Study provided for this Requested Procedure.*", and therefore offers a viable search criterium on this entity, hence is used as unique key. The study's description depends on the clinical context of this study, „*Institution-generated description or classification of the Study (component) performed.*". The time and date information contains the start date and time of the study. Every Study might group one or more Series. [4]

Series: This entity provides the uid, description, date and time, just as the study does. All attributes are specified comparable to the study, apart from the description, which is specified as, „*Description of the Series* ". Every Series organises a number of Files. [4]

File: This entity is not entirely specified by the standard, it rather serves multiple purposes. On the one hand it represents a files node in the beforehand mentioned tree structure, and on the other hand it identifies the binary data of the actual file in the storage directory. The attributes stored are the name, size, checksum and its relation to a Series and it's FileType. The File entity's checksum is a MD5 hash of the whole binary representation of the file to guarantee integrity and identity. Based on the attribute's values the actual path of the file in the binary store directory is determined.

FileType: The FileType is an entity that makes the File entity more modular. The FileType is outsourced from the file, because both a Series and a File are associated to a FileType. Basically this entity only adds a bit of normalization to not store file types duplicated, or if certain file types change their descriptions to only change them once.

When integrating a persistency layer for example in web environments one quite popular mindset is *CRUD* (**C**reate **R**ead **U**pdate **D**elete) [10]. Basically all actions needed to manage an entity when considering it's modality in an object oriented paradigm are provided. Apart from basic interactions, more complex use cases that need more sophisticated queries can be tailored specifically or are built on top of these actions.

All provided *CRUD* functions are implemented in the `Manager` (see Figure C.1 on page 28) and delegated to specific implementations.

- `insert{File,Patient,Series,Study}` corresponds to **C**reate
- `get{File,Patient,Series,Study}ById` corresponds to **R**ead

- `update{File,Patient,Series,Study}` corresponds to **U**pdate
- `delete{File,Patient,Series,Study}` corresponds to **D**elete

3.2 Business Processes

The wide spread cognition of *Business Processes* is their usage for corporate development, establishing processes for administration or the usage for production line optimization. But the term of *Business Process* gains increasingly importance in IT projects. Requirements are the start of every project or small program. *Business Processes* are deduced from requirements and finally define the frame for the underlying data infrastructure and the *Business Logic* implemented.

Looking at one out of the many definitions for *Business Processes*, the relation to the programming context is getting more evident.

> *„a set of linked activities that take an input and transform it to create an output. Ideally, the transformation that occurs in the process should add value to the input and create an output that is more useful and effective to the recipient either upstream or downstream.“*
>
> by Johansson et al [8]

This definition fits what the management layer does. As mentioned in Chapter 3 on page 7, the top and third sublayer of the PatientDB plugin encapsulates all business requirements and interaction mechanisms. On the one hand simple *CRUD* related operations are provided, and on the other more complex *Business Processes* are implemented. Each of these processes is implemented as one function in the manager. Every process requires a set of certain parameters, which are described in [**?**]. Basically two specific types of functions were needed to fulfil all initial requirements. All entities of the current schema are structured as a tree, like mentioned in 3.1. When holding a reference to a specific entity there are obviously four scenarios that are required:

1. get all entities of a certain type
2. get a entity by a certain attribute
3. get the ancestor of the currently referenced entity
4. get all descendants of the currently referenced entity

These operations are implemented in the `Manager` class to extend the basic *CRUD* operations. This set of operations is the initial one to satisfy these requirements:

- `getAll{Files,Patients}` - allows to get a full list of all currently stored files/ patients. These two operations are the most significant ones to get all entities of the corresponding type. Either all files are selected and their ancestors retrieved or all patients are selected and their descendants retrieved. Each patient represents a main organisation

unit for its studies, series and files. Every file is indirectly associated to exactly one patient, via its series and study.

- `get{PatientByPid,StudyByUid,SeriesByUid,FileByName, FileType ByName}` - all these functions help retrieving their entities by unique keys, constraints that are communicated by requirements or specifications but are not used for the primary key identifying their entity.

  This example demonstrates the ability to retrieve a patient by an unique constraint defined by the standard, which is not the primary key used (would be a numeric value):

  ```
  manager->getPatientByPid("pt9871234");
  ```

- `get{PatientByStudy,StudyBySeries,SeriesByFile}` - returns the corresponding ancestor of a study, series or file.

  ```
  Series *series = manager->getSeriesByUid("uid9871234");
  manager->getFilesBySeries(series);
  ```

- `getAll{StudiesByPatient,SeriesByStudy,FilesBySeries}` - provides a list of all descendants related to the passed entity.

There are generally two different approaches to deduce models for structured data from *Business Processes*. Either *Business Processes* induce all requirements and the whole persistency layer is designed based on technical formalisation of these, or the data already suggests certain *Business Processes* to be added and or refined in advance.

3.3 Entity Management

In PatientDB's architecture entities are the first class citizens. They are the main tools for organising dataflows and bridging between *Business Processes* respectively the underlying persistency. Basically, by reading the managers function definitions all currently implemented *Business Processes* can be obtained. To handle entities, their provided operations and current state, every entity is designed as a simple state machine. An entity's state is defined by `EntityStatusType`, which equals one of the following values, `STUB`, `LOADED` or `DIRTY`.

The entity manager or just Manager internally caches entities and their current state. When the Manager hands out entities, they are never shallow copies. Every entity handed out references exactly the same memory used by the Manager to administrate the entity's data. In case a not cached entity is accessed, it will be loaded lazily, meaning that not all entities are fully loaded from the database right away and this is the reason for the existence of an `EntityStatusType` of `STUB`.

There are two `EntityLoadingModes` available `EAGER` and `LAZY`. These two modes are used to control the behaviour when loading an entity. The default behaviour is loading lazily, whenever possible. For example when a patient entity is loaded all it's corresponding studies are loaded as well, but only their stubs. This offers the possibility of having a slight preview off the upcoming structure, in terms of count of descendants and their IDs.

Implementing this semi-automatic type of lazy loading was quite easy considering the simplicity of the entities and their structure. The caching is quite beneficial, because repeated reads are cheap and if needed the reloading of data can be forced. Recent changes are reflected in the cache as well, and are consistently stored before being flushed and written to the database. The caching strategy definitely takes load off the database connection. The combinations between the entity state machine and the loading mode are quite interesting. An entity cannot be loaded in any circumstance. For example if the `EntityStatusType` is `DIRTY` and the `EntityLoadingMode` is `EAGER` the entity cannot be updated with the loaded data because of the possibility of loosing unsaved changes. The caching mechanism implemented in the manager handles this issue and others to maintain integrity and a coherent state of the already loaded data.

3.4 Binary Data Management

The `FileManager` class manages the obfuscation, insertion, deletion and checksum generation of incoming files or filenames. Binary data management is something special, as mentioned in Chapter 1. Especially the `FileManager` and the `File` entity has a deep relationship. The `File` entity relies on some function from the `FileManager`, for example to erase or load the binary data from hard disk. The clue is that the `File` entity uses functions that only manipulate existing data. Data that is about to be modified will be loaded into the cache and marked with one of the `EntityStatus` flags. If the correct flag is set, the changes will be done and written to file. The `File` entity is an abstraction layer on top of the `FileManager` class. Another interesting feature of the File and `FileManager` collaboration is inspired by smart pointers and the copy-on-write mechanism.

The `FileManager` uses a flavour of the *COW* (**C**opy-**o**n-**w**rite) optimization strategy. Referenced data is not copied into the current context as long as it is not changed or written. This optimization strategy provides several benefits. The most evident one is a small memory footprint of objects or processes. Every object or process gets a pointer to the resource. The other idea the `FileManager` uses are smart pointers. A smart pointer acts like a pointer with more features. One of the feature is the automatic memory management. Automatic memory management deals with the topic of abstracting memory allocation and deallocation and usually is hidden behind a facade type of abstraction mechanism.

The `File` makes use of some ideas from these mechanisms. It is a small memory footprint facade, that offers the actual entity's data. The binary data of the file is only read and passed through via the `FileManager` if explicitly requested. Basically the `File` only offers an *API* (**A**pplication **P**rogrammable **I**nterface) to access the binary data in the working directory, instead of storing its data in the main memory and bloating the persistency layer.

The obfuscation method is a special step which has to be done before the files can be stored in the actual working directory. Figure 3.2 describes the workflow of the obfuscation method before the file can be stored. First of all the file is passed from AngioVis to our PatientDB. The filename is read from the *DICOM* Header. The next step is converting the filename into a MD5 hashed binary representation which is Base64 encoded. When this process is finished the file is moved to the working directory with the obfuscated filename.

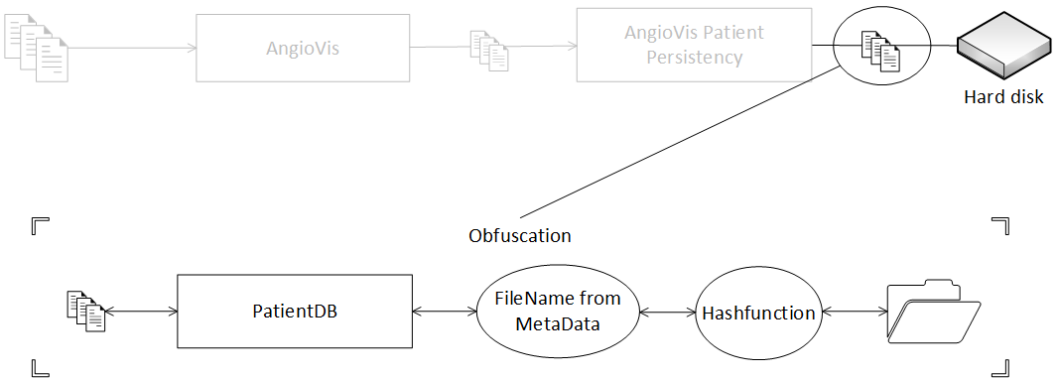

Figure 3.2: AngioVis Patient Persistency Obfuscation

Only the plain filename will be stored in the database and the checksum. There are several steps to generate the checksum and store the filename in the database. The first step is the same like the workflow in Figure 3.2. Files are passed from the AngioVis framework to our PatientDB. After that step the file will be loaded chunk by chunk and the hash function generates a MD5 representation of the binary data. The result will be stored in the database and just in case a file is loaded by one of our `File` entity instances the whole process starts back to front. The specific filename will be loaded from the database for the Base64 decoding. If the file can be found in the working directory the two filenames will be compared and the checksum generation process starts again. Last but not least, if the checksum matches with the calculated checksum the file integrity is guaranteed. Otherwise it will be rejected.

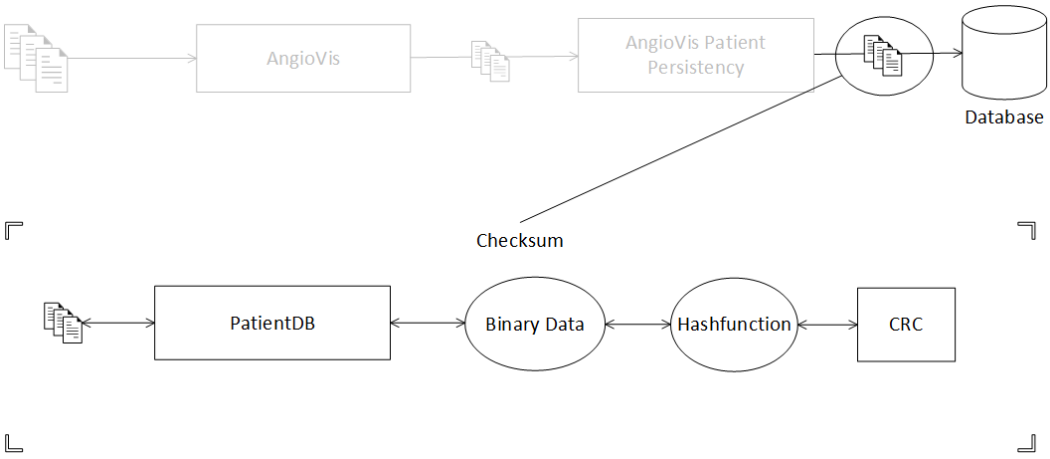

Figure 3.3: AngioVis Patient Persistency Checksum Generation

3.5 Results

To demonstrate the improvements PatientDB gains several test cases prove its advantages. Basically 5 test cases of various length and load on the overall system are provided. We focused on the following types of tests:

1. Prove functionality of simple *CRUD* operations.
2. Test the stubbing and caching strategies.
3. Test more complex business cases apart from simple *CRUD* operations.
4. Measure the pure binary data throughput with stubbed relational data, to prove the increased import speed.
5. The last test case is a full stack simulation, 360 *DICOM* files were loaded and the whole relational tree was constructed based on the *DICOM* header information.

The default dataset the previous PatientDB implementation was tested with 10 files that in total make up a binary amount of 1.2GB. The loading process took a long time in any case and was cancelled before finishing. The PatientDB reimplementation handles this dataset, the fourth described test case, within one minute and thirty seconds. To further stress test the new PatientDB, the last test case is used, with a dataset of 360 *DICOM* files. This test run that totally reconstructs the database terminated after a six minutes thirty of execution.

CHAPTER 4

Conclusion

The implementation of the PatientDB, provides further integrity constraints and organises relational data and binary data in an easy to administrate fashion.

This challenge was attacked by providing a *Layered Architecture* that introduces separation of concerns into the PatientDB plugin. Furthermore, optimizations like caching and lazy loading are applied to speed up the entity management. Entities are managed coherently in one management layer on top and offer an *API* to manipulate their data in a *CRUD* way as well as providing further implementations of more complex *Business Processes*.

The extendibility of the applied application structure was the focus of this implementation. The whole persistency layer is designed with principles of object orientation and further best practices in mind. Future work of this implementation is further increase of extendibility and an even more generic approach like configure entities on a configuration file basis.

APPENDIX A

Integration and Lifecycle Documentation

A.1 Preface

This Appendix describes the setup procedure for AngioVis in three phases.

Phase one will cover all steps we took to add external dependencies and set external tools up. This includes the a full Database Management System Stack including the integration into the Qt framework.

Phase two covers the basic process of forward engineering the mysql model represented by the *avpp.mwb*[1], model changes and their implications and overall remarks regarding the full model lifecycle.

[1] **M**ySQL **W**ork**b**ench

A.2 Phase 1 - External requirements

For Phase 1 are basically three external tools/ dependencies needed:

1. *MySQL Server*, the initial SQL instance PatientDB was developed with
2. *MySQL Workbench*, not necessarily needed but useful for forward- engineering[2]
3. *QtMySQL Plugin*, mandatory qt plugin that needs to be configured and compiled against the current qt source distribution and the running MySQL server instance.

All three of these setup steps are described in the following three sections.

MySQL Server Setup

Our preferred way of installing a MySQL Server instance was the zip archive distributed by oracle on the MySQLl website[3], mainly because the zip version does not create any side effects apart from the needed windows service entry.

1. Download a version of the MySQL community edition, we used the most recent version of MySQL 5.6, in our case `http://dev.mysql.com/get/Downloads/MySQL-5.6/mysql-5.6.11-winx64.zip/from/http://cdn.mysql.com/`
2. Unpack the downloaded archive to a preferred location on the Hard disk, we used `C:/MySQL/`
3. Open a command line prompt `cmd.exe` with administrative privileges and navigate to `C:/MySQL/mysql-5.6.11-winx64/bin`
4. Execute the following command A.1 to install the MySQL daemon as windows service:

```
1 C:/MySQL/mysql-5.6.11-winx64/bin>mysqld.exe --install
```

Listing A.1: Installation of mysql daemon as windows service

5. Depending on the current environment the start type of the MySQL windows service needs to be adjusted via the service panel in the windows settings. We changed the start type in our case to manual as illustrated in the figure A.1

[2] Process of maintaining the PatentDB's schema on a graphical basis and on the fly conversion to a MySQL Script that is executed automatically to reflect changes right away on the running MySQL instance

[3] `http://dev.mysql.com/downloads/mysql/`

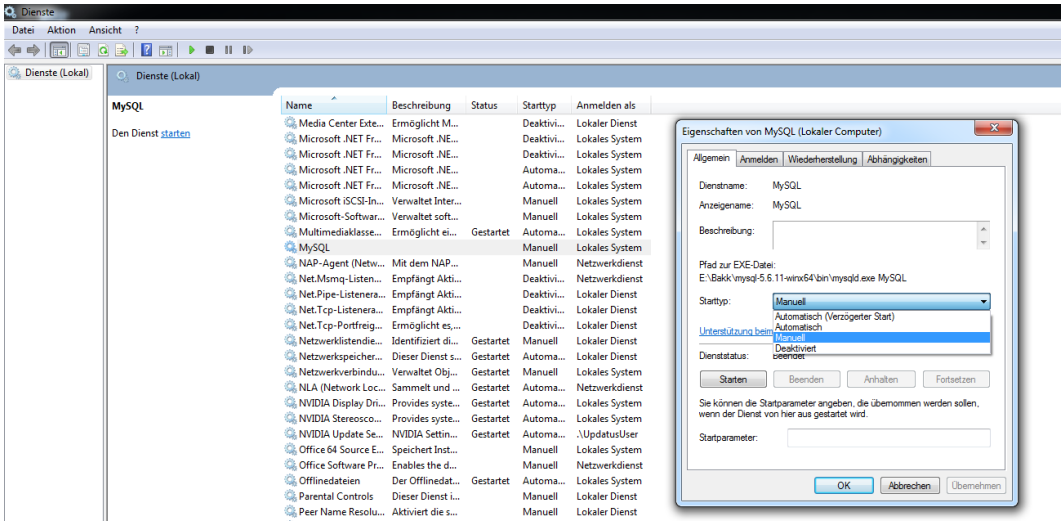

Figure A.1: MySQL daemon as windows service

MySQL Workbench Setup

The MySQL Workbench was the administration utility of our choice due to the following reasons:

- easily maintain and administrate user privileges and access rights
- forward- engineering capability to react to schema changes in a fast paced manner
- easily deploy and manage testing datasets

The Workbench's setup is a standalone setup as well, for the only reason to be able to cleanly deinstall MySQL without any traces.

1. Download a version of the MySQL workbench, we used the most recent version, that fits out MySQL community server version, in our case: `http://dev.mysql.com/get/Downloads/MySQLGUITools/mysql-workbench-gpl-5.2.47-win32-noinstall.zip` from `http://cdn.mysql.com/`
2. Unpack the downloaded archive to the same location as the MySQL server `C:/MySQL/`
3. Create and softlinks as preferred and desired
4. Bring up the MySQL Workbench application and click on *New Server Instance*
5. The following wizard guides through the basic configuration of MySQL Workbench in conjunction with the actual MySQL community server instance. We connect the PatientDB Plugin as well as the MySQL Workbench via the TCP/IP Stack to our MySQL server as shown in figure A.2

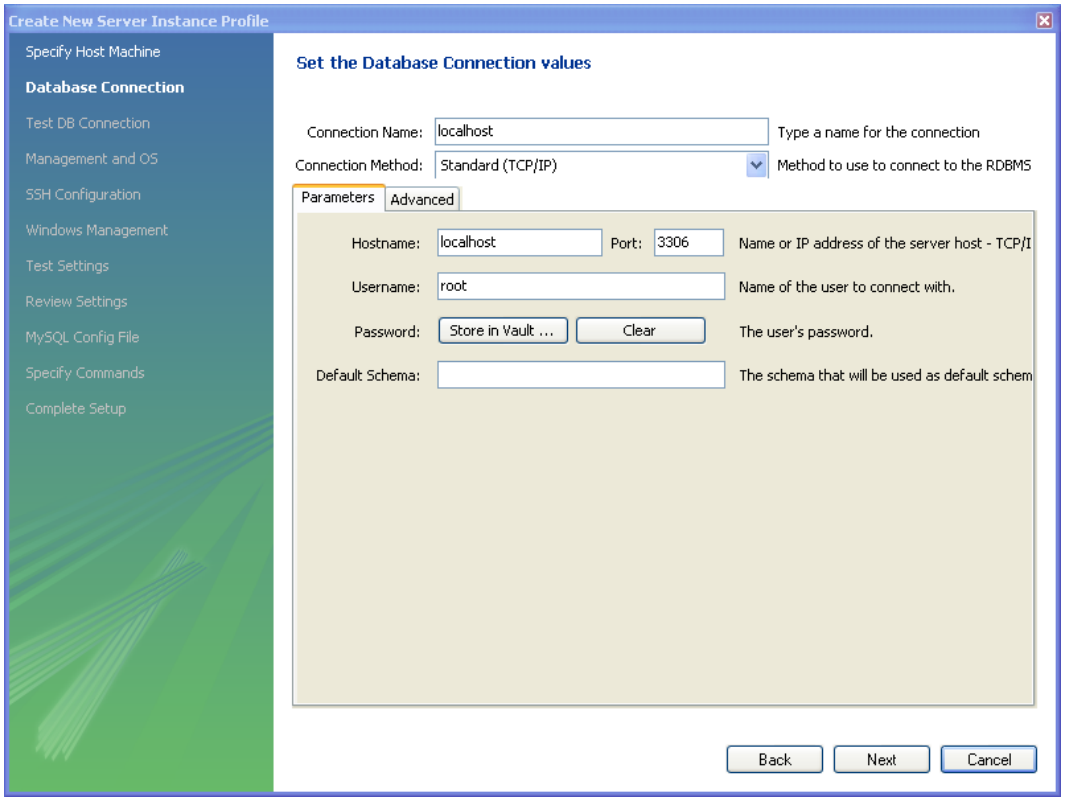

Figure A.2: MySQL Workbench connection settings

6. The following steps in the wizard test the connection to the database server and offer the possibility to configure and test the default configuration file `my-default.ini` usually located in the root directory of the MySQL server installation.

7. After successfully configuring the MySQL Workbench two new items should appear in the main-application, a new connection (as side effect of the connection settings) and the newly created server instance as shown in figure A.3

8. After these configuration steps it's easily possible to check and analyse the currently persisted patient data directly and to apply any needed schema changes via forward engineering. To update the schema just open the corresponding `avpp.mwb` apply all needed changes and execute the changed schema

QtMySQL Plugin setup

The last step of Phase 1 is to enable the internal MySQL support of Qt via it's QtMySQL plugin, that's directly available in the everywhere distribution we installed previously.

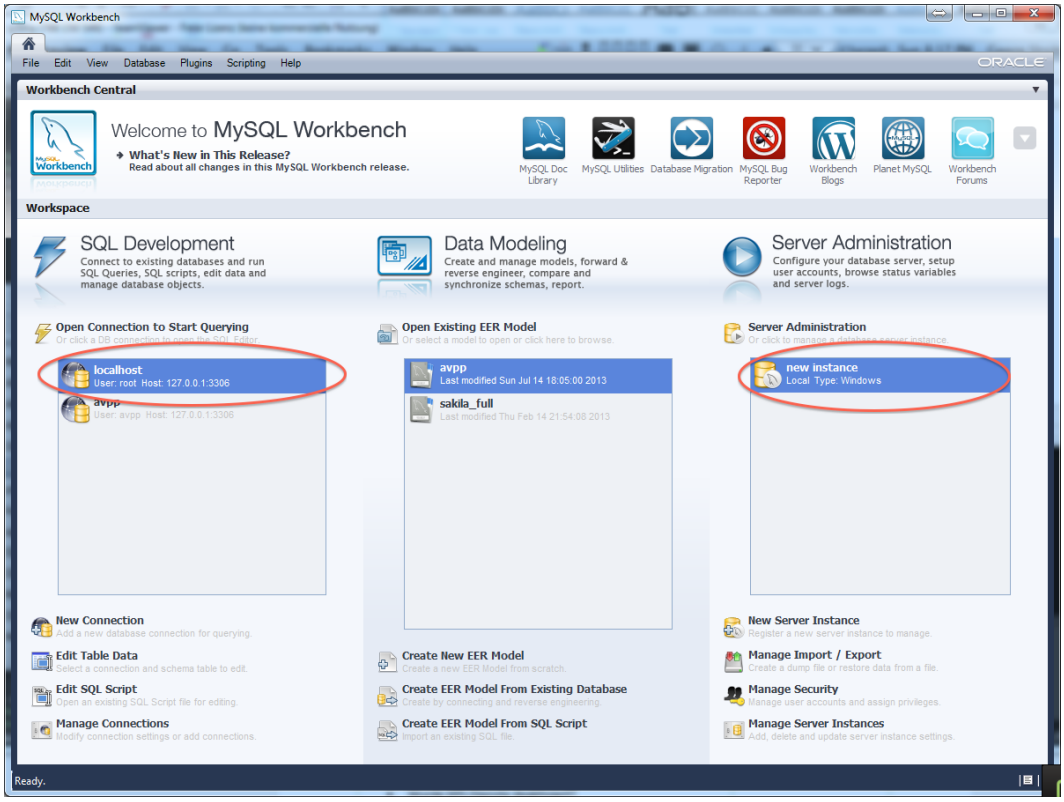

Figure A.3: MySQL Workbench successful configuration

1. Detailed and probably most up to date compilation instructions are available at `http://qt-project.org/doc/qt-4.8/sql-driver.html#how-to-build-the-qmysql-plugin-on-windows`, we proceeded following the steps below.

2. Navigate to the QtMySQL plugin directory in the Qt distribution, execute the qmake command with adapted parameters and run nmake to build the needed plugin.

```
1 cd %QTDIR%\src\plugins\sqldrivers\mysql
2 C:\Qt\4.8.4\bin\qmake "INCLUDEPATH+=C:/MySQL/mysql-5.6.11-
    winx64/include" "LIBS+=C:/MySQL/mysql-5.6.11-winx64/lib/
    libmysql.lib" mysql.pro
3 nmake
```

Listing A.2: Compilation of QtMySQL plugin

3. To make use of these libraries, the generated, lib and dll files in `qt/plugins/sqldrivers/` (`qsqlmysqld4.dll`, `QtSql4.dll`) and the MySQL dll `C:/MySQL/mysql-5.6.11-winx64/lib/libmysql.dll` needs to be copied next beside the actual generated binary

A.3 Phase 3 - Model Lifecyclemanagement

This Phase covers the basic usage of the MySQL Workbench to manage the lifecycle of the used SQL Schema. Two basic operations will be explained:

1. Editing the basic model represented by the *avpp.mwb*, changing default insert values and managing foreign key constraints
2. Forward engineering the changed model to achieve two things:
   - reset the current database status to factory status
   - reflect all changes to the model in the new database status

The MySQL Model and MySQL Workbench

To operate on the MySQL Model *avpp.mwb* provided, the second column Data Modelling in figure A.3 offers all operations needed.

1. To load the MySQL workbench file *avpp.mwb* click on the menu entry *Open Existing EER Model* as depicted in figure A.3.
2. An entry *avpp* will appear, click on this menu item.
3. The opened view provides the following capabilities:
   - The EER Editing Screen as shown in figure A.4 is a graphical editing tool to manipulate the current Schema.
   - The operations on the left side allow to change foreign key constraints and therefore relationships between our Entities. Take care while changing anything in this scheme, changes here are not automatically reflected in the AngioVis Patient Persistency Plugin.
   - The most important section are the tabs on the bottom pane. These tabs allow to change the current default inserts and intrinsics of the tables like datatypes used in the schema.
4. One of the probably most important operations of the Data Modelling Suite of MySQL workbench is the forward engineering functionality, that directly applies changes made on the EER screen to the current Schema instance represented by the MySQL server.

   This operation is easily achieved by pressing `Ctrl + g` while viewing the EER Screen. This operation brings up a popup that allows specifying further details. The most important step is the configuration screen, shown in figure A.5, of the forward engineering process that allows configuring what and how will be applied to the current database instance.

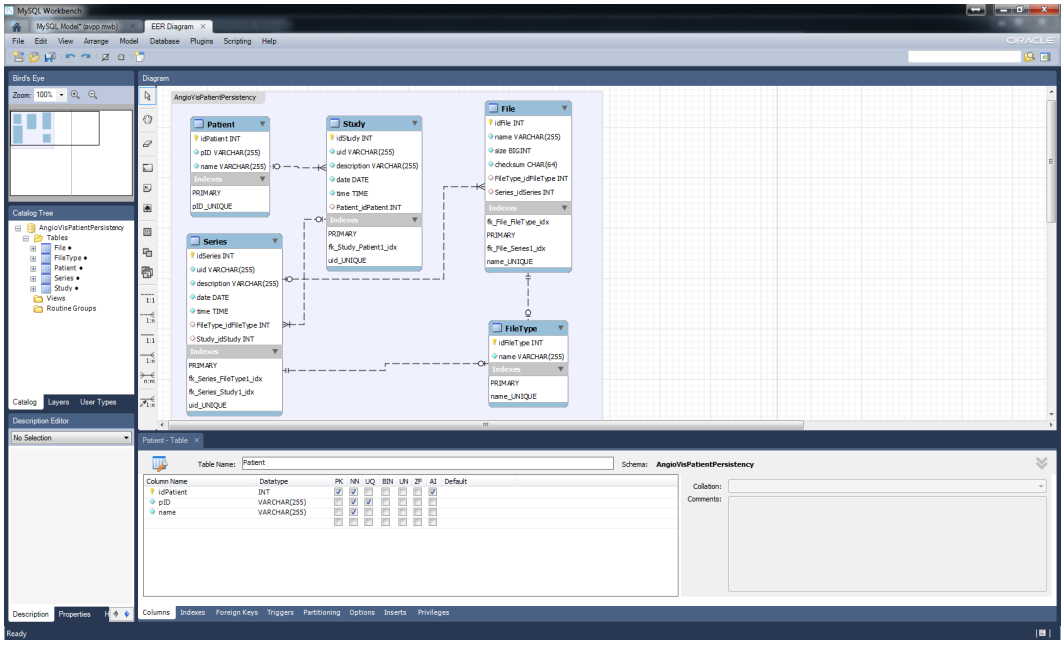

Figure A.4: MySQL Workbench EER Diagram administration screen

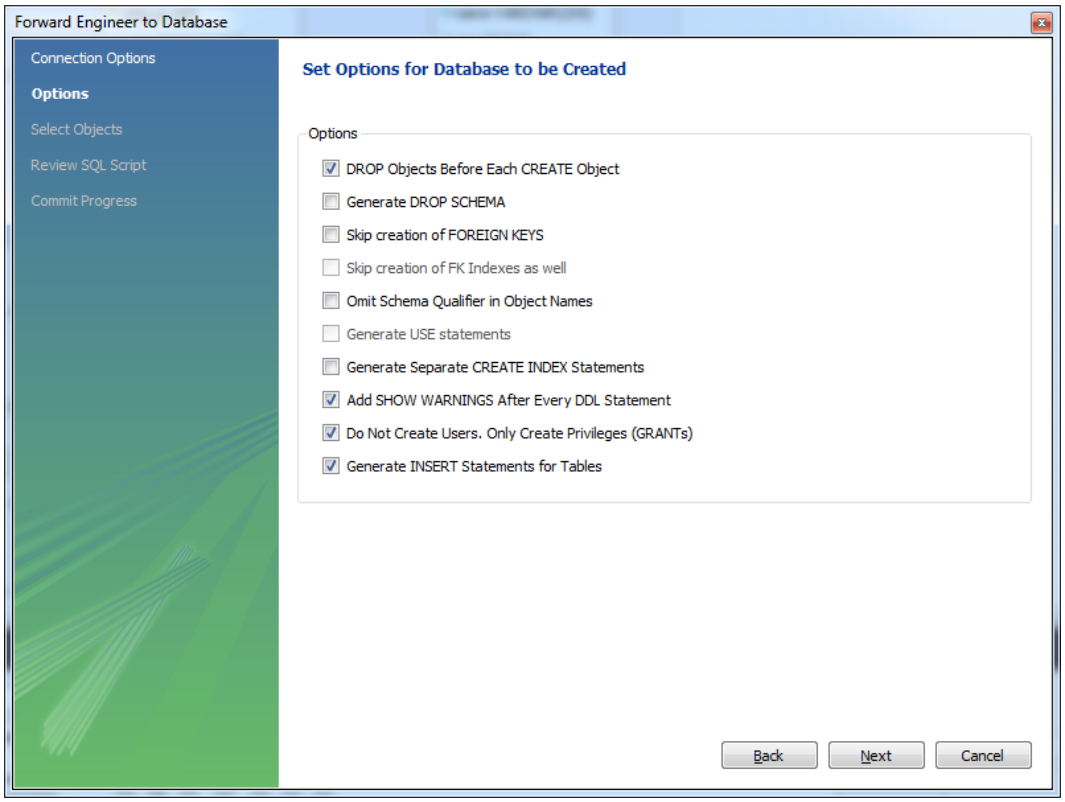

Figure A.5: MySQL Workbench forward engineering dialogue

APPENDIX B

Entity Relationship Diagram

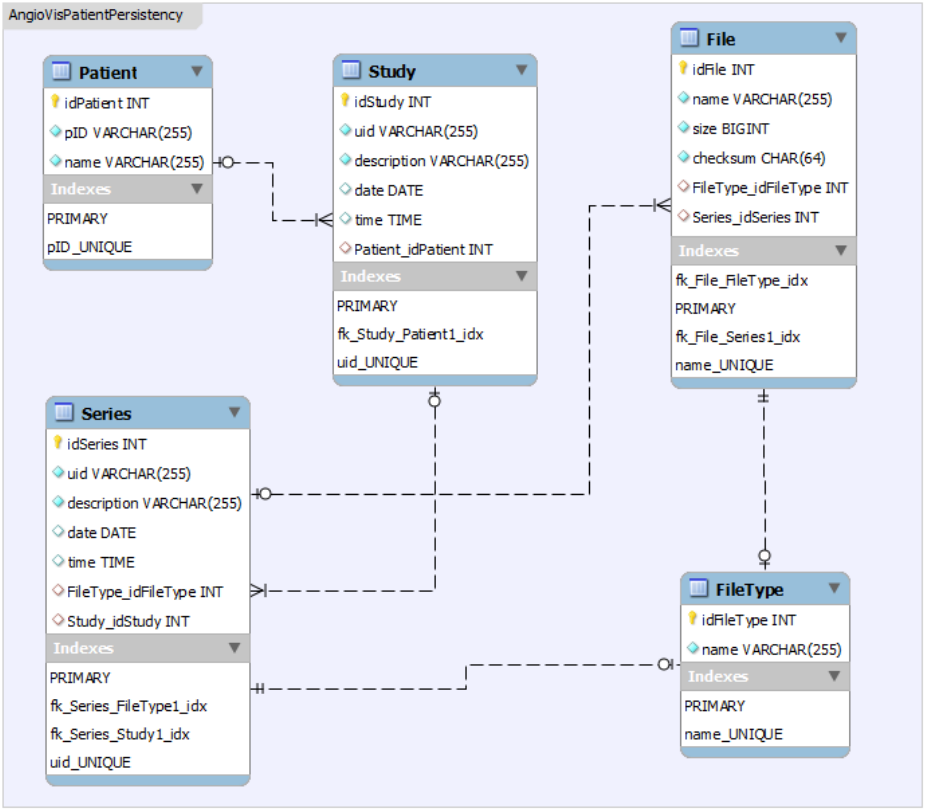

Figure B.1: Entity Relationship Diagram

The entity relationship diagram depicted in Figure B.1 shows all five entities deduced from the standard, patient, study, series, file and file type. Every patient is related to zero or more studies and stores the patient name and patient id. One study belongs to zero or one patients and is related to zero or more series'. One series is related to zero or one study and organises zero or more files. Study and series share a common set of attributes, uid, description, date and time.

One file knows its name, size, checksum and type. Both a series and a file are restricted to one file type. In case of the series this means one series stores one or many files of the same type.

APPENDIX C

UML class diagram

This Appendix contains all diagrams of relevant classes to the *PatientDB* plugin of the AngioVis Framework. All classes are categorised into the three layers depicted in Figure 3.1:

Management layer	**DAO layer**	**Entity layer**
`class Manager`	`class PatientDao`	`class Entity`
`class FileManager`	`class PatientDaoSql`	`class Patient`
`class Connection`	`class StudyDao`	`class Study`
`class PsqlConnection`	`class StudyDaoSql`	`class Series`
`class SqlHelper`	`class SeriesDao`	`class File`
`class Settings`	`class SeriesDaoSql`	`class FileType`
	`class FileDao`	
	`class FileDaoSql`	
	`class FileTypeDao`	
	`class FileTypeDaoSql`	

C.1 Management and Configuration classes

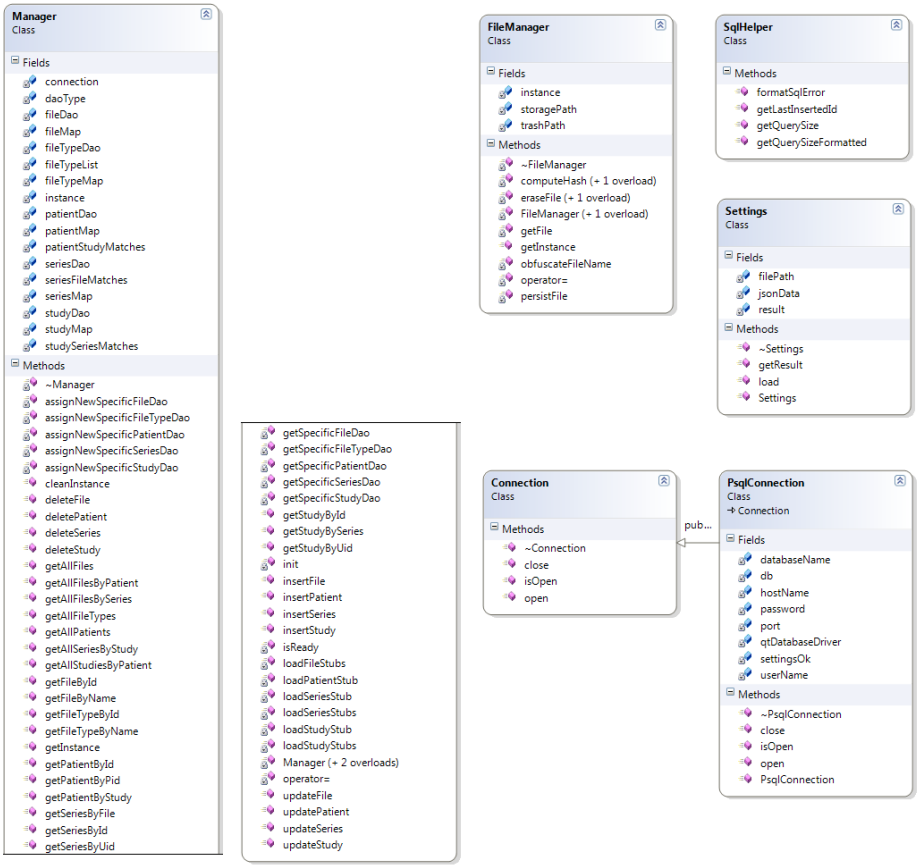

Figure C.1: Management and Configuration classes

- `class Manager` is the aggregation of the whole API, all business cases are provided via this class. Internally the caching strategy and the lazy loading are implemented in the `Manager` as well.
- `class FileManager` encapsulates the file access and is tightly coupled to the `File` entity.
- `class Connection` is an abstract class that defines the base interface implemented by specific connection classes. The assumption is that a connection to the underlying persistency mechanism is needed, bearing the base requirement of a RDBMS in mind.
- `class PsqlConnection` is the default implementation to persist via SQL. This class implements the beforehand mentioned abstract class `Connection` and specifies behaviour related to SQL.

- `class SqlHelper` is a convenience class that offers heavily reused and therefore encapsulated functionality for interaction with the underlying sql API.
- `class Settings` helps loading a JSON settings file and accessing the configurations provided.

C.2 Entity Classes

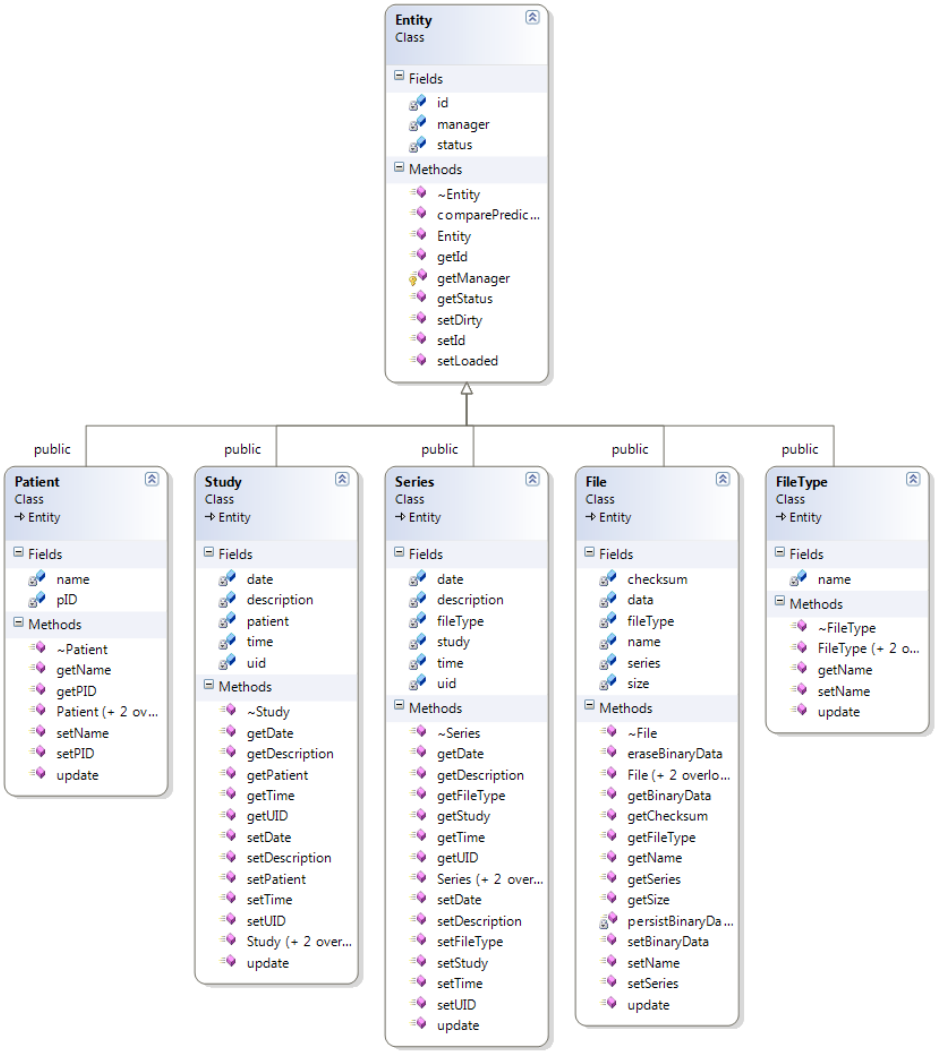

Figure C.2: abstract Entity, Patient, Study, Series, File and FileType class

- `class Entity` is an abstract class that defines basic details regarding every entity, like having an id (artificial integer keys only), referencing the manager and being in a certain status.
- `class Patient` encapsulates all attributes (pID, name), access functions and a unified way to validate this entity.
- `class Study` encapsulates all attributes (date, description, patient, time, uid), access functions and a unified way to validate this entity.

- `class Series` encapsulates all attributes (date, description, fileType, study, time, uid), access functions and a unified way to validate this entity.
- `class File` encapsulates all attributes (checksum, data, fileType, name, series, size), access functions and a unified way to validate this entity.
- `class FileType` encapsulates all attributes (name), access functions and a unified way to validate this entity.

C.3 Data Abstraction Object Classes

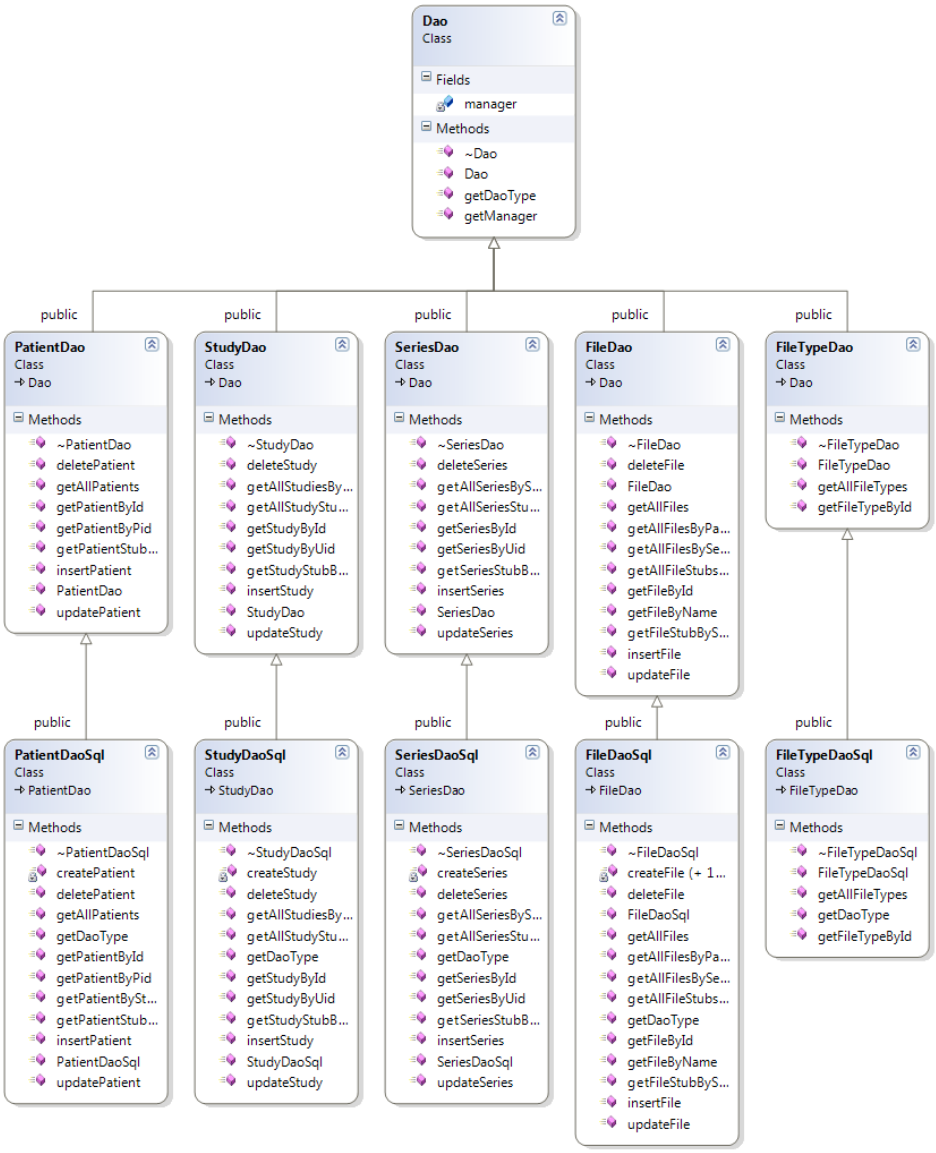

Figure C.3: Data Abstraction Object classes

- `class Dao` the heart of this DAO layer, that defines the basic characteristics of every DAO. Every DAO holds reference to the `Manager` and offers getters for its type and its stored reference to the `Manager`.
- Patient: `class PatientDao`, `class PatientDaoSql` encapsulates abstract interface and all CRUD functions for patients

- Study: `class StudyDao`, `class StudyDaoSql` encapsulates abstract interface and all CRUD functions for studies
- Series: `class SeriesDao`, `class SeriesDaoSql` encapsulates abstract interface and all CRUD functions for series
- File: `class FileDao`, `class FileDaoSql` encapsulates abstract interface and all CRUD functions for file
- FileType: `class FileTypeDao`, `class FileTypeDaoSql` encapsulates abstract interface and all CRUD functions for filetypes

C.4 Custom Types and Enumerations

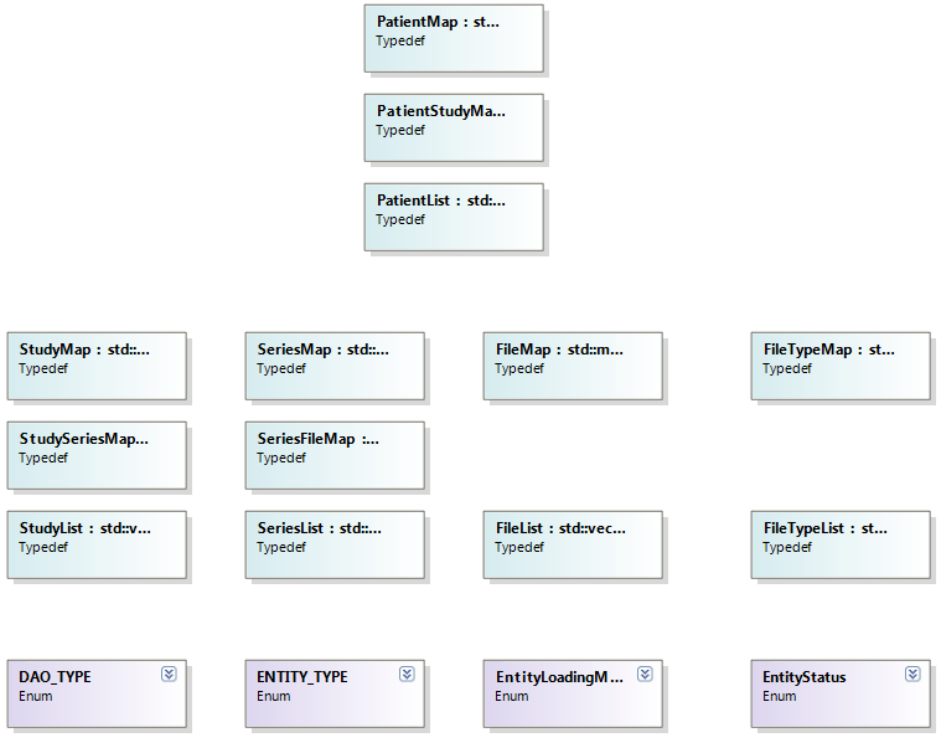

Figure C.4: Custom types and enumerations to introduce abstract types for Patients, Studies, Series, Files and FileTypes and the collections to store them

All in Figure C.4 shown datatypes introduce custom types for collections used to store Entities or data needed for the caching strategy. Introducing these types serves two purposes. First of all it increases the code's readability while making heavy use of the standard library's template classes. Last but not least but not least it makes replacing the standard collections fairly easy, because all usages throughout the codebase are masked by those alias types. For example if the PatientMap, which associates a patient's id to it's entity needs to be stored in a `MyCustomMap` instead of an `std::map` and both classes offer the same interface it's simply changing this typedef.

- Patient: `PatientMap`, `PatientStudyMap`, `PatientList`
- Study: `StudyMap`, `StudyStudyMap`, `StudyList`
- Series: `SeriesMap`, `SeriesStudyMap`, `SeriesList`
- File: `FileMap`, `FileList`

- FileType: `FileTypeMap`, `FileTypeList`
- Enumerations:
  - `DAO_TYPE`
    0. SQL
  - `ENTITY_TYPE`
    0. PATIENT
    1. STUDY
    2. SERIES
    3. FILE
    4. FILETYPE
  - `EntityStatus`
    0. STUB
    1. LOADED
    2. DIRTY
  - `EntityLoadingMode`
    0. LAZY
    1. EAGER

Bibliography

[1] AngioVis. http://angiovis.org/. Accessed: 2013-10-16.

[2] National Electrical Manufacturers Association. http://medical.nema.org/dicom/2011/11_03pu.pdf. Accessed: 2013-07-28.

[3] National Electrical Manufacturers Association. *Digital Imaging and Communications in Medicine (DICOM) - Part 1: Introduction and Overview*. National Electrical Manufacturers Association, 2011.

[4] National Electrical Manufacturers Association. *Digital Imaging and Communications in Medicine (DICOM) - Part 3: Information Object Definitions*. National Electrical Manufacturers Association, 2011.

[5] National Electrical Manufacturers Association. *Digital Imaging and Communications in Medicine (DICOM) - Part 4: Service Class Specifications*. National Electrical Manufacturers Association, 2011.

[6] Jeff Atwook. http://www.codinghorror.com/blog/2006/06/object-relational-mapping-is-the-vietnam-of-computer-science.html. Accessed: 2013-07-28.

[7] Fang Cheng. A new dao pattern with dynamic extensibility. In *Information and Computing Science, 2009. ICIC '09. Second International Conference on*, volume 1, pages 23–26, 2009.

[8] H.J. Johansson, P. McHugh, A.J. Pendlebury, and W.A. Wheeler. *Business Process Reengineering: Breakpoint Strategies for Market Dominance*. Wiley & Sons, 1993.

[9] K.K. Mehra, S.K. Rajamani, S.K. Jha, and A.P. Sistla. Verification of object relational maps. In *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pages 283–292, 2007.

[10] O.M. Pereira, Rui L. Aguiar, and M.Y. Santos. CRUD-DOM: A model for bridging the gap between the object-oriented and the relational paradigms. In *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, pages 114–122, 2010.

[11] J. Savolainen and V. Myllarniemi. Layered architecture revisited - comparison of research and practice. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 317–320, 2009.

[12] Wikipedia. http://en.wikipedia.org/wiki/cardinality. Accessed: 2013-09-02.

[13] Wikipedia. http://en.wikipedia.org/wiki/object-relational_impedance_mismatch. Accessed: 2013-07-28.