h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt
Fachbereich Informatik

A Dynamic Product Line for an Electronic Health Record Management System in Cancer Care

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

vorgelegt von
Patrick Spitzer (744230)

Referent:	Prof. Dr. Bernhard Humm
Korreferent:	Prof. Dr. Ralf Hahn
Ausgabedatum:	13.10.2016
Abgabedatum:	13.04.2017

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 13. April 2017

Patrick Spitzer

Abstract

Die Integration von Informationstechnik in die wachsende Komplexität der Biowissenschaft und des Gesundheitswesens ist ein wichtiger Faktor um Ärzte zu unterstützen, Patientensicherheit zu erhöhen und klinische Prozesse zu optimieren. Ein Verwaltungssystem für elektronische Gesundheitsakten hilft Kliniken, indem es Informationen über den Gesundheitsstatus eines Patienten digital abspeichert und zusätzlich Unterstützungs- und Verwaltungsdienste anbietet.

Diese Systeme müssen äußerst variabel sein, da viele medizinische Fachrichtungen, Informationsdienste, technische Besonderheiten und krankenhausspezifische Anspassungen zu berücksichtigen sind. Des Weiteren müssen sie an die umfangreiche, vielfältige und sich ständig verändernde medizinische Domäne flexibel anpassbar sein.

Um dieses Problem zu lösen, wird eine dynamische Software Produktlinie für ein Verwaltungssystem für elektronische Gesundheitsakten in dieser Thesis konzipiert und teilweise implementiert. Die Produktlinie ermöglicht die Entwicklung und Instanziierung mehrerer kundenspezifischer Produkte auf einer Platform. Dabei werden Kernfunktionalitäten und Softwareartefakte untereinander geteilt. Jede Anwendung läuft jedoch innerhalb ihres eigenen Kontexts. Das heißt, dass produktspezifische Daten, Funktionalitäten und Konfigurationen von anderen Anwendungen weder benutzt noch eingesehen werden können. Konkrete Anwendungen werden durch das Erstellen einer Konfiguration abgeleitet, welche die aktivierten Funktionalitäten und produktspezifischen Einstellungen definiert.

Neben der Architektur fokusiert sich diese Thesis auf die initiale Untersuchung der gemeinsamen und variablen Anforderungen, die Verwaltung der vielfältigen medizinischen Daten mit Hilfe des HL7 Reference Information Model's, die Vorstellung von grundlegenden Mechanismen für die Zugriffsverwaltung und das Erstellen einer flexiblen Benutzeroberfläche für das Verwalten von elektronischen Gesundheitsakten.

Abstract

Integrating information technology into the growing complexity of life science and healthcare is an important factor for supporting physicians, improving safety for patients and optimizing clinical processes. An electronic health record management system helps hospitals as it digitally stores information about the health status of a patient and additionally provides support or management services.

However, these systems must be highly flexible as they have to consider various medical specialties, information services, technical features and hospital specific customizations. Furthermore they have to be dynamically adaptable to the extensive, diverse and ever-changing medical domain.

To solve this problem, a dynamic software product line for an electronic health record management system is being designed and partially implemented in this thesis. The product line allows to develop and instantiate multiple customer-specific products on one platform and enables them to share core functionality and software artefacts. Each application runs in its own context, i.e. that the product-specific data, functionality and configuration of others can neither be seen nor used. Concrete applications are derived by creating a configuration that defines the enabled optional features and product-specific settings.

Besides the architecture, the thesis focuses on initially examining the common and variable requirements, managing the diverse medical data with the HL7 Reference Information Model, introducing fundamental access management mechanisms and creating a flexible user interface for managing electronic health records.

Contents

List of Figures

List of Tables

Acronyms

ANSI American National Standards Institute

BLOB Binary Large Objects

CA Core Assets

DAO Data Access Object

DSPL Dynamic Software Product Line

EF Entity Framework

EHR Electronic Health Record

EHRMS Electronic Health Record Management System

FEF Family Evaluation Framework

HL7 Health Level Seven

IIS Internet Information Services

ISO International Organization for Standardization

JSON JavaScript Object Notation

MDT Multidisciplinary Team Meeting

MIS Medical Information Service

MS Medical Specialty

MSSQL Microsoft SQL Server

OVM Orthogonal Variability Model

ORM Object-Relational Mapper

REST Representational State Transfer

RIM Reference Information Model

SaaS Software as a Service

SPL Software Product Line

SPLE Software Product Line Engineering

UIC User Interface Components

VAF Variable Attribute Fields

Chapter 1

Introduction

1.1 Project

This thesis was written within the scope of the project SAGE-CARE (SemAntically integrating Genomics with Electronic health records for Cancer CARE) which is funded by the European Commission and has several participants from the educational and commercial sector. The European Commission states that "the aim of this project is to bring together subject matter experts from the academic and non-academic sectors to create a holistic informatics platform for rapidly integrating genomic sequences, electronic health records (EHRs) and research repositories to enable personalised medicine strategies for malignant melanoma treatment." [Cor]

At the University of Applied Sciences Darmstadt, an EHR application for treating patients with melanoma cancer is currently developed for the SAGE-CARE project (see [HW15]; [BHW15]; [Bee15]; [Ide16]). Besides simply managing patient data, the system provides medical information services (MISs). Those services aim to support clinicians by providing person-specific and relevant information, such as suitable literature or evidence-based medical recommendations. Additionally, multidisciplinary team meetings (MDTs) are supported, where a team of consultants from multiple clinical disciplines come together and discuss the treatment of selected patients.

An important partner and consultant during the development process is NSilico Lifescience Ltd., who is also a participant in the SAGE-CARE Project. The company is located in Dublin and describes itself as "the provider of the world's most easy-to-use data management and analytics software for the lifesciences and health-

care industries. The company's offerings are based upon a unique and unrivalled blend of biological, computing, software-development and clinical experience and expertise which enables us to provide our customers with solutions which significantly increase the efficiency and accuracy of their work." [NSind] Their product Simplicity-MDT was the basis for the EHR application of the University of Applied Sciences Darmstadt.

1.2 Motivation

Integrating information technology into the growing complexity of life science and healthcare is an important factor for improving safety for patients. While a lot of effort is put in improving diagnosis and treatment methods, the integration of operational systems is not always seen as a priority. However, studies suggest that such systems can reduce error rates, increase performance of physicians and improve the clinical outcome. The main barriers for this situation are seen in financial limits, lack of standards and cultural barriers. [BG03]

While NSilico Lifescience Ltd. already had an existing EHR application for melanoma treatment, called Simplicity MDT, the enhancement for new medical specialties resulted in developing new applications because of the different requirements, concerning data and user interfaces. These systems are currently developed and maintained separately, which will cause a lot of maintenance effort if even more customized applications for medical conditions are added.

To solve this problem a multi-tenant aware dynamic software product line for an electronic health record management system (EHRMS) that covers multiple cancer diseases is being designed and partially implemented in this thesis. The product line allows to develop and instantiate multiple customer-specific products on one platform and enables them to share core functionality and software artefacts, instead of developing multiple single separated applications. Concrete applications are derived by creating a configuration that defines the activated features and product-specific settings. Each application will run in its own context, i.e. that the data, functionality and configuration of others can not be seen or used.

The thesis uses the software product line engineering approach for the design of the system. In doing so, it will focus on initially examining the common and variable requirements, designing the architecture and selected system functionalities, and im-

plementing first concepts. The prototypical implementation is based on the current SAGE-CARE melanoma application, which served as the main input for the design, together with Simplicity MDT from NSilico.

1.3 Structure

This thesis contains the hereafter described chapters. The fundamental problem of this thesis and the initial requirements that shall be met are defined in chapter 2. Relevant fundamentals and background knowledge is described in chapter 3. Thereupon, common and variable features for the product line are examined in chapter 4. Chapter 5 deals with the architecture design of the system and outlines multiple concrete concepts. Prototypical implementations for two selected concepts are shown in chapter 6. Subsequently, in chapter 7 the acquired designs are evaluated with the help of the Family Evaluation Framework and additionally by comparing it against the incipient defined requirements. Related work is shown and shortly summarized in chapter 8, while chapter 9 finally concludes the results of this thesis and determines aspects that were not addressed and are considered as future work.

Chapter 2

Problem Statement

The purpose of this thesis is to develop a dynamic software product line (DSPL) for customer-specific electronic health record management systems. These applications mainly differ according to their supported medical specialties and medical information services. Customers should be able to document their preconditions with the product line owner, who consequently must be able to configure the software product line (SPL) in such a way that a concrete product gets derived. More detailed requirements, including a commonality and variability analysis for the product line, are examined in chapter 4. All things considered, the following fundamental requirements were identified:

1. **Product line for EHR management:** The described concept shall outline the architecture of a product line for an EHRMS in cancer care. The product line is initially not designed for the mass market. Currently two hospitals use the commercial applications, which serve as main input for the SPL.

2. **Dynamic derivation of concrete products:** Concrete products shall be derivable from a software platform via configuration, covering all possible variants. Meaning that various medical specialties, medical information services and technical services have to be considered. Furthermore, configuration may concern every layer of the architecture.

3. **Configurability on different levels:** Products meet the requirements of a customer on different levels, e.g. a default configuration for a hospital and a differing specific configuration for a single physician. Consequently, where appropriate, different configuration levels must be considered.

4. **Multiple products on one instance:** The DSPL must be able to derive multiple products, each with its own configuration and data, on one running product line instance. It is mandatory, that concrete products must not know if they share their product line instance with other products.

5. **Fundamental security mechanisms:** Only valid users shall have access to their application. They should have their own account with its own corresponding roles and rights. For example a melanoma physician must not be able to view patients, which are not assigned to his area of responsibility.

6. **Change Tracking:** Changes to the persistent data in the SQL database shall be traceable. Additionally, data must not be deletable to ensure compliance standards.

7. **Testability:** Testing product lines is different from single system testing, as variability and multiple products have to be considered when writing tests. Additionally, like in single system testing, different testing levels have to be regarded, such as unit or integration tests. The thesis shall create an appropriate design for structuring and managing those multiple dimensions.

8. **Flexible data model:** To meet the needs of all products, the data model must be flexible and extensible. Negative effects of the flexibility on efficiency and convenience shall be counteracted.

9. **Flexible user interfaces:** A flexible user interface for EHR management with reusable elements shall be designed, enabling to build up patient data dynamically.

Chapter 3

Background

The subsequent chapter describes relevant fundamentals and background knowledge for this thesis. First, the principles and terminology of software product line engineering are explained, which is essential for this work. Next, EHRs, electronic health record management systems and additional related terms of the health care domain are defined. Finally the HL7 Reference Information Model is introduced, as it creates the structure for the relational data of the product line.

3.1 Software Product Line Engineering

3.1.1 Product Lines

Software product lines (SPLs) are defined as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are deployed from a common set of core assets in a prescribed way." [CN09, p. 5]

Others, like [PBL05] or [Ape+13], put emphasis on deriving products through mass customization of software platforms and reusable components. However in this thesis the definition from [CN09] will be used, since it is the most specific one.

Core Assets (CA) are mainly reusable software components, providing the platform for the SPL. Generally speaking, CA also include reusable "domain models, requirements statements, documentation and specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions." [CN09,

p. 14] Authors like [PBL05] and [LSR07] use the term artefact instead of asset. In this thesis the terms asset and artefact will be used synonymous, as suggested by [PBL05].

By taking suitable CA, configuring and exploiting defined variation mechanisms and possibly appending new application specific components, a concrete product is build. The products are all based upon the SPL architecture, which explains why this is the most crucial CA. Instead of implementing every application from scratch and just reusing independent libraries, building a product from a SPL is more a process of assembling and configuring CA. Consequently, for every product a build-plan must exist, which defines the used CA, the configurations and the additional product specific components. [CN09, pp. 5–6]

Developing products with the help of CA, configuration and possibly adding specific components in a predefined way is called Software Product Line Engineering (SPLE). This approach is defined as a "paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization"[PBL05, p. 14]

According to [PBL05] and [LSR07] the paradigm is made up of two development processes, respectively life-cycles: domain engineering and application engineering. These processes and their interrelations are visualized in figure 3.1.

In this context the domain is a "specialized body of knowledge, an area of expertise, or a collection of related functionality. For example, the telecommunications domain is a set of telecommunications functionality, which in turn consists of other domains such as switching, protocols, telephony, and network. A telecommunications software product line is a specific set of software systems that provides some of that functionality." [CN09, p. 14] The aim of domain engineering is to determine commonalities and variabilities in the SPL. During the process a platform is created, mainly consisting of requirements, architecture, components and tests. In the course of the application engineering process, the concrete products are derived from the previously created platform. Dividing SPLE into two processes helps to separate the different concerns, build a reusable platform, find commonalities and create specific applications by exploiting the variation points of the platform. [PBL05, pp. 20–21]

It is also helpful for the development process to classify all requirements (functional and non-functional) according to three types: [Hah16]; [LSR07]

1. **Common:** Valid for all products and hence a part of the platform.

2. **Optional:** Common for some products but not for all. Variability mechanisms have to be introduced to handle these mechanisms.

3. **Differing:** Customer-specific requirements, that are valid for one product only. These characteristics are not a part of the platform although they must be supported (e.g. provide a plug-in mechanism).

This classification simplifies the assignment of requirements to the two development processes, as common and optional characteristics are mainly handled in domain engineering, while differing requirements are a part of application engineering.

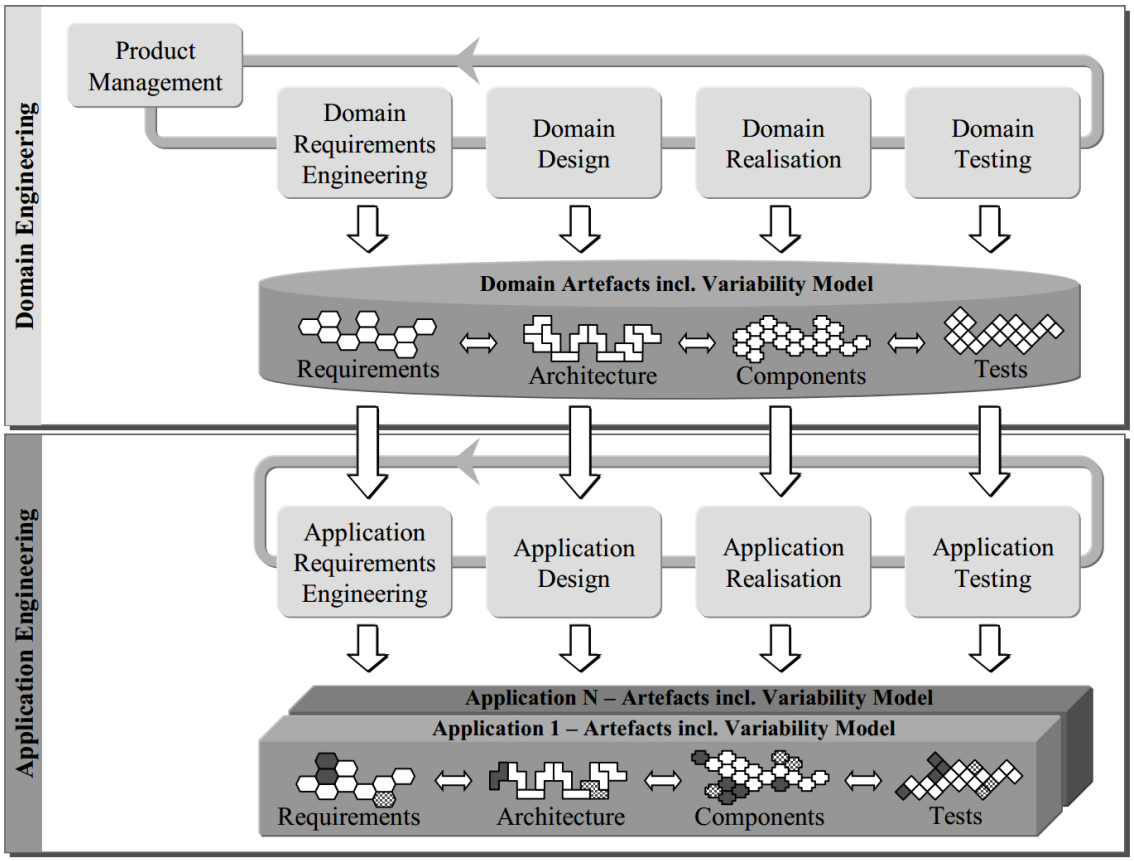


Figure 3.1: The two development processes of SPLE [PBL05, p. 22]

3.1.2 Domain Engineering

The domain engineering life-cycle is represented in the upper section of figure 3.1 and it consists out of five subprocesses. They are described in the following paragraphs. [PBL05, pp. 24–28] [LSR07, pp. 49–53]

Product Management: Deals with managing the scope of the SPL. The business goals are the input for this process, while the output should be a plan for future products and their assembly out of the existing products and artefacts.

Domain Requirements Engineering: In this subprocess the concrete variable and common requirements of the SPL with all its derived products get ascertained and documented. Consequently the two major steps are the commonality analysis and the variability analysis. Input for this process are multiple requirements sources, like stakeholders, the product roadmap, competitors products and fundamental requirements (e.g. non-functional requirements). Documented requirements, such as textual descriptions for common requirements or variability models for variable requirements, are the output.

Domain Design: This process is responsible for determining a reference architecture and designing the additional common and optional assets for the SPL, which are also valid for its derived products. The input is made up of the previously created domain requirements, defining common and variable demands. The output consists of the SPL-Architecture and a revised variability model, which includes internal variability.

When creating the architecture for a SPL, it is important to always consider all products, requirements, variation points and stakeholders. Conflicting needs must be solved with introducing variation mechanisms. Specialized architecture design methods for SPL do exists, e.g. *COPA*, *FAST*, *FORM*, *KobrA* or *QADA*, but are not further examined during this thesis (see [Acm] and [AK15] for additional reading).

Domain Realization: In this phase of the main process, the detailed design, reusable CA and variation mechanisms are realized. The assets can be either made, bought, mined from an existing component or commissioned to be created by other companies. The input of domain realization consists of the reference architecture and the software artefacts that have to be created.

Domain Testing: The last subprocess validates and verifies the previously created artefacts. The input consists of requirements, architecture, component and interface designs, and the implemented core assets. The test results are the output.

3.1.3 Application Engineering

The application engineering life-cycle is represented in the bottom lane of figure 3.1. Following four subprocesses can be described in the subsequent paragraphs. [PBL05, pp. 30–34] [LSR07, pp. 53–55]

Application Requirements Engineering: This subprocess is needed to initially specify the requirement for a concrete product. Therefore, the domain requirements and the product specific requirements are needed. The main task during this step is to find commonalities and differences between domain and application requirements, to assure optimal reuse of the platform. At the end of this process a complete requirements specification should exist where it is evident which artefacts are reused from the domain and which are application specific.

Application Design: This step deals with deriving a product architecture from the reference domain architecture. Variation points get configured and application specific architecture artefacts get prepared. Whereas the input encompasses the reference architecture and the application specification, the output consists of the application architecture.

Application Realization: Procedure of implementing the specific product. Application architecture and reusable domain artefacts are used as an input for this process. Domain components get configured if needed and application components get implemented. Composed together they form the application, which is the desired output.

Application Testing: This step is used to validate and verify the application and assures that quality requirements are met. The created application, with all its used application and domain artefacts, forms the input. The output is the result of all performed tests.

3.1.4 Variability

Variability in Software Product Line Engineering

Introducing variability, or sometimes synonymously called flexibility is mandatory for SPLE. A product platform has to support mass customization and variation mechanisms to realize the derivation of several products. Throughout the development process, described in chapter 3.1.2 and 3.1.3, the variation points have to be identified, documented and handled in the development assets. While domain engineering mostly handles identification and definition of variability, application engineering cares about exploitation by assigning the proper variants to a product. [PBL05, p. 8]

The term variability can be defined as "the ability or the tendency to change." [PBL05, p. 59] In SPLE variability can be broken down to variation points and variants. Variation points are varying items within domain assets, while variants are single options or instances of variation points. An example for a variation point in a SPL is a user identification mechanism, while its variants are fingerprint scanning, iris scanning or password authentication. [PBL05, pp. 61–62]

Types of Variability

For correct identification and documentation, one also has to distinguish between different types of variability. The first distinction should be made between variability in time and variability in space. While variability in time means that an asset will change over time and compensate its old version, variability in space implies that an asset exists in various versions which are valid at the same time. The differences between these two types are visualized in figure 3.2. It shows three mechanisms for identification. While the variants professional line and economy line coexist (variability in space), the professional line changes its identification mechanism over time from a keypad to a fingerprint mechanism (variability in time). [PBL05, pp. 65-67]

The second distinction that has to be made is between internal and external variability. External variability directly concerns the customer and results out of their requirements. The previously stated example of different identification mechanisms is also an illustration of external variability, because it is directly visible to the cus-

tomer. Internal variability is described as the flexibility that is hidden from the customer and it often results from the realization of external variability or from technical reasons. The ability to store a fingerprint as compressed or uncompressed image is an example for internal variability, because the customer isn't aware of it.

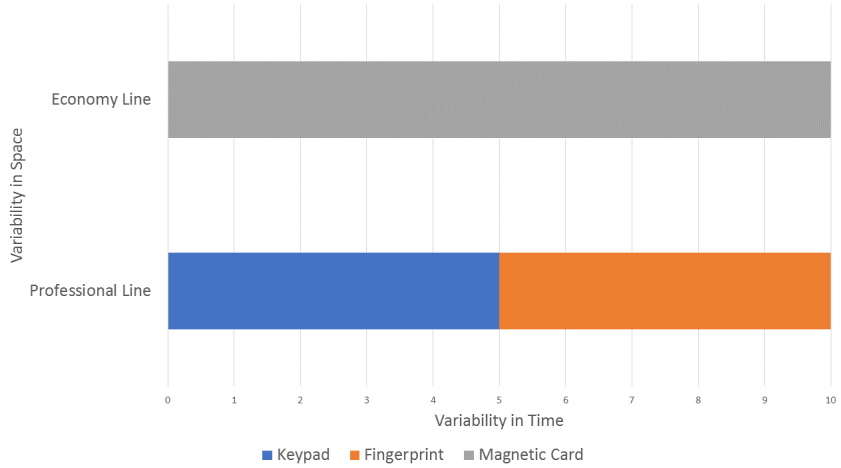


Figure 3.2: Variability in time and space according to [PBL05, p. 67]

Variability Documentation

A way to document variability is the orthogonal variability model (OVM), which was first described by [HP03] and further elaborated by [PBL05]. The OVM "defines the variability of a software product line. It relates the variability defined to other software development models such as feature models, use case models, design models, component models, and test models." [PBL05, p. 75] The graphical notation is realized with the elements illustrated in figure 3.3.

The sections *Variation Point*, *Variant*, *Variability Dependencies* and *Alternative Choice* show the elements that are needed to document the assembly of a variation point. If no alternative choice is modeled, then a default range [1..1] is used. These elements are used in the right section of figure 3.4, as they illustrate the variation point Door Lock. The *Artefact Dependencies* connect variation points and variants to other development artefacts, which aren't documented as a variability diagram such as use case diagrams. An example is also modeled in figure 3.4, where artefact dependencies connect the variability diagram with the use case diagram. *Constraint Dependencies* illustrate restrictions between two variability diagram elements. Whereas, for example, *requires_v_vp* means that a variation requires the

presence of the associated variation point.

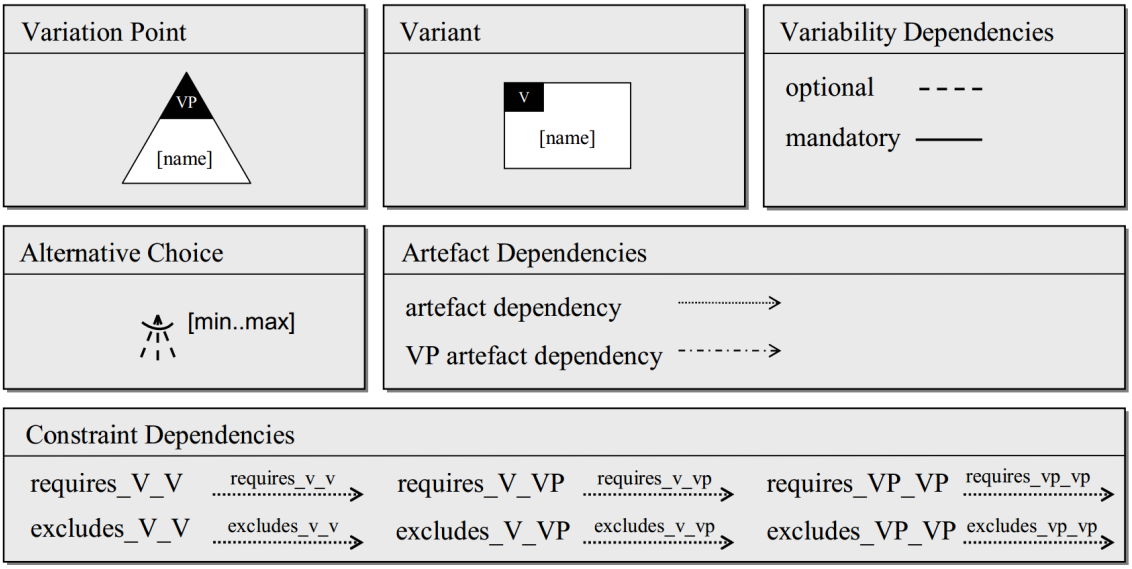


Figure 3.3: Graphical notation for variability models [PBL05, p. 85]

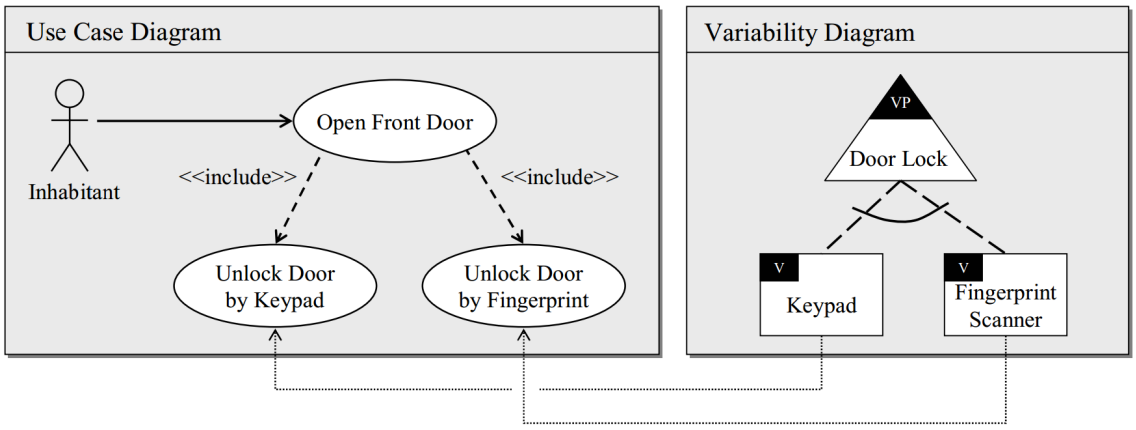


Figure 3.4: Example of orthogonal variability modelling [PBL05, p. 85]

3.1.5 Dynamic Software Product Lines for Cloud Computing

Dynamic software product lines (DSPLs) are specific types of SPLs, where variation points and their variants are bound only at run time, whereas traditional SPLs bind variability at different stages of development. The requirement for DSPLs emerged out of the problem that it is sometimes not possible to foresee all the features, that a product requires. Hence, features must be easily addable per configuration at run-

time instead of rebuilding. Accordingly this approach was developed to be able to react to changes in users needs. All things considered, a DSPL design is suited for systems, that ought to be adaptable at runtime by manual intervention, for example by the product owner or via a self-service portal. Additionally, this approach can be particularly beneficial for autonomic or self-adaptive systems. [CBK13]

DSPLs are often linked to reconfigurable Software as a Service (SaaS) applications. The SaaS model enables customers to access an application on demand, without buying or installing hardware or software. SaaS applications are usually hosted centrally on a cloud system and accessed via web browser. Their architecture should support distributable application layers, e.g. a three layer architecture, and loose coupling between the layers. [Feh14]

Both concepts meet customer requirements via variability management at run time, and reconfiguration over the product life cycle has to be considered. A concrete configuration of a configurable SaaS application can be seen as the equivalent to a product of a DSPL. Consequently, the configuration process can be seen as an equivalent to the product derivation, respectively application engineering process.

Linking DSPLs to configurable SaaS applications is not a new concept in literature, even though boundaries and characteristics of the two concepts are not always clearly defined, see [Sch13]; [SR12]; [RA11]; [Mie+09]; [BGP12]. However, an approach to classify and compare those concepts, with its possible characteristics like multi-tenancy or configurability, is provided by [Sch13]. The author compares them regarding the amount of customers and configurations per product instance, which is shown in table 3.1.

Category	Customer	Configuration	Product instance
Software product line	1	1	1
Configurable multi-instance SaaS application	1	1	1
Multi-tenant aware SaaS application	m	1	1 (shared by all customers)
Dynamic software product line	1	n	1
Reconfigurable multi-instance SaaS application	1	n	1
Reconfigurable multi-tenant aware SaaS application	m	n	1 (shared by all customers)

Table 3.1: Comparing DSPLs to configurable SaaS applications according to [Sch13]

In the final analysis, DSPLs and reconfigurable multi-instance SaaS applications differ mainly in their licensing and delivery model, while they both rely on resolving variation points with run-time mechanisms and configuration management. A reconfigurable multi-tenant aware SaaS application adds tenant-awareness to these concepts, but still relies on the DSPL principles. Therefore, in the course of this thesis the term DSPL will be mainly used, as the focus of this thesis is on software product line engineering and not on the licensing models of SaaS applications.

3.2 Electronic Health Records

Definitions of electronic health records (EHRs) are very much alike in their core principles, see [Int05], [Kir08] or [Hri10], since they all mention the saving of a patient's health data as digital information. However, in this thesis the following definition will be used: An EHR is a "repository of information regarding the health status of a subject of care, in computer processable form, stored and transmitted securely and accessible by multiple authorized users, having a standardized or commonly agreed logical information model that is independent of EHR systems and whose primary purpose is the support of continuing, efficient and quality integrated health care." [Int05, p. 2]

While this definition focuses on the storage of health care information, the other mentioned sources describe a broader scope of functionality for EHRs such as the integration of support or management services. A system with that sort of services will be called an electronic health record management system (EHRMS). This terminology makes it possible to distinguish between the actual data record and a fully functional system, enriched with support services. [Kir08, p. 326]

With an EHRMS the processes in health care can be automated, which results in increased efficiency and reduced costs for hospitals. Other advantages are the minimization of paper storage costs, eliminating the poor legibility of handwritten records, reduced medical errors through the use of terminologies and the integration of other clinical support services or systems. [Hri10, pp. 17–18]

The literature also mentions the the distinction between an EHR, an electronic medical record and an electronic patient record. An electronic medical record digitally documents only a single use of a health service. Furthermore an electronic patient record is considered as a system that documents the complete documentation of all

health-related information of a patient instead of the recording of just a single use of a health service or selected informations.

3.3 HL7 Reference Information Model

Health Level Seven (HL7) International describes itself as a "not-for-profit, ANSI-accredited standards developing organization dedicated to providing a comprehensive framework and related standards for the exchange, integration, sharing, and retrieval of electronic health information that supports clinical practice and the management, delivery and evaluation of health services." [Heab] The corporation is supported by health care providers, government stakeholders, payers, pharmaceutical companies, vendors and consulting firms.

An information model in general can be described as a "structured specification, expressed graphically and/or narratively, of the information requirements of a domain." [Int06] Furthermore it describes classes, properties, relationships and states.

The HL7 Reference Information Model (RIM) is an abstract model for representing and interchanging health information. It is an ISO-Standard and mainly encompasses class and state-machine diagrams. While it additionally provides several other models, this thesis only covers the main class diagram.

The RIM classes represent information, which must be documented and communicated within the health care environment. The abstract model consists of six back-bone classes: [Int06][Bee11]

- Entity: Physical things and beings of a health care environment, e.g. persons, organizations, materials, places, devices, etc.
- Act: Represents actions and happenings, e.g. issues, procedures, observations, medications, supply, registrations, etc.
- Role: The roles that are played by entities while participating in an act, e.g. patients, provider, consultants, employees etc.
- Participation: Relationship between a Role and an Act, e.g. a patient and his medication.
- ActRelationship: The binding of one Act to another, e.g. an issue and a consequential procedure.

- RoleLink: Relationship between two individual roles, e.g. a physician and his patient.

Act, Entity and Role, which will be called base classes within the scope of this thesis, are refined by sub-classes, for example a patient would inherit from the role class. The purpose of Participation, ActRelationship and RoleLink, from now on called association classes, is to enrich the relationship between two base classes with information and to add the possibility to dynamically build up new relationships. For example there would be no concrete link between a patient class and a medication class in an information system. Instead a participation object would be created to connect two instances of those classes. This generic approach of connecting the base classes, and therefore also the inheriting classes, via association classes is important since the HL7 RIM intends to be a standard for many information systems, which will likely have different data models in detail. The abstract HL7 RIM class model is visualized in figure 3.5. While the attributes of the back-bone classes and their subclasses are intentionally left out in this figure, since they are not relevant for this thesis, the entire suggested HL7 RIM is visualized in appendix A.

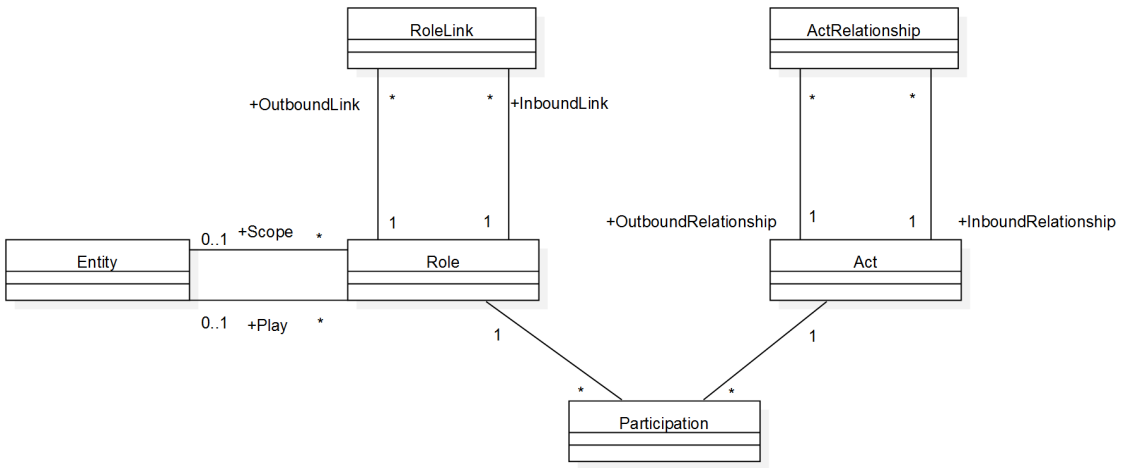


Figure 3.5: HL7 Reference Information Model

For further reading and a more detailed description, HL7 provides a comprehensive documentation of the RIM standard. [Heaa]

Chapter 4

Domain Requirements Engineering

As described in chapter 3.1.2, domain requirements engineering mainly performs a commonality and variability analysis with certain requirements sources as input. The purpose of this chapter is to find these common and variable requirements for the derived products and to document them with the help of the OVM, which was introduced in chapter 3.1.4. Input for this thesis were the products from NSilico Life-science Ltd., shortly introduced in chapter 1, the existing SAGE-CARE Melanoma EHR application and the initially defined fundamental requirements in chapter 2.

4.1 Commonality Analysis

While flexibility is of course one major aspect of product lines, it is especially important to ascertain as much commonalities as possible. They are the foundation for every SPL and its derived products. On the whole, the more commonalities exist, the less variability mechanisms have to be introduced in the architecture, which reduces the complexity. The following sections describe the commonalities in detail.

4.1.1 Managing Electronic Health Records of Patients

The management of patients with a cancer disease, e.g. breast cancer, thyroid cancer or melanoma, is the foundation for the product line and at least one medical specialty (MS) is implemented in every product. A MS is seen as certain type of a cancer disease within the scope of this thesis. Users must have the ability to search for,

add, edit and view the EHRs of patients. Although the SPL is focused on cancer diseases, the more universal term MS is used to indicate the possibility for extending the system with other similar disease types.

4.1.2 Technical Services

Within the scope of this thesis, technical services include every service that doesn't provide health care related information for the end-user and is mostly implemented to realize technical requirements. Technical services often don't concern the end-user, but provide a great potential for reuse. The common technical services are described throughout this section.

Multi-tenancy

A tenant is defined as "a group of users sharing the same view on an application they use", while the group is typically a legal entity. [KMK12] Multi-tenancy is therefore an approach to create multiple views on one application instance and assign one view to every group of users.

The separation of data between customers is mandatory in health care information systems, since they contain sensible data. As the product is designed as a DSPL, capable of serving many customers with one cloud instance, a multi-tenant architecture is needed to ensure that a tenant will not be able to access data and functionality from other tenants within the same instance.

In the course of this thesis, multi-tenancy will mostly concern data separation. However, the user interface and the access management have to be adapted to this requirement.

Access Management

A concept for authentication and authorization is needed in this SPL for similar reasons as multi-tenancy. Health care data must be managed very carefully and therefore the system must make sure that only valid user have access and that certain views, data and functionalities are only available for defined users.

Authentication is the process of validating the identity of a user. This thesis refers to human beings when using the term user, whereas the user concept can be extended

to e.g. machines, networks, or intelligent autonomous agents. Authorization, also known as access control, grants users access to certain resources, while preventing them from accessing resources they are not allowed to obtain. [Tod07]; [Ben06]; [Ame04]

The access management of this system has to check at least three questions, when a request is made:

1. **Authentication:** Has the request been made by a valid user?
2. **Authorization of tenant rights:** Has the tenant of the user the corresponding rights for the request?
3. **Authorization of user rights:** Has the user the corresponding rights for the request?

The design of the access management should also address the client and the server of the SPL. It is mandatory that the users only see their allowed parts of the user interface, mainly for usability reasons. However, this would not be sufficient on its own. Additionally, the server has to be secured against invalid requests, which is the crucial part. It must be ensured that only data is transfered to the client, which the querying user is allowed to see.

Data Historization

Integrating data historization into the SPL is especially important because it contains health care data. Saving and reading the history of patient data can, on the one hand, be helpful for physicians, e.g. showing the development of a tumor stage. On the other hand it can also be a legal requirement to track changes and ensure that no data is deleted completely. Due to that, historization is helpful to fulfill compliance standards. As a result, this requirement is a mandatory feature and will be integrated into every product.

Multi-Language-Support

Multi-Language Support, which belongs to the concept of software internationalization, is not a new concept, see [Hal97] or [Ess00]. But in times of globalization it

hasn't lost any relevance for software engineering. For this reason it is also seen as a common requirement for all products to support multiple languages.

Literature suggests that the term internationalization includes designing a software product for handling multiple languages and cultural conventions without redesigning the application, comprising its development artefacts, and therefore creating the flexibility to easily serve and create acceptance on multiple local markets. [Ess00, p. 25] In this thesis only the aspect of multi-language support is handled, while cultural conventions, such as different time or date formats, are left out.

Deployment:

The possibility of deploying the software to an application server is mandatory as the products will be distributed client-server applications. Additionally, it is a fundamental requirement to realize the possibility that a product can be deployed to a cloud system. As this is a mostly technology related topic, deployment is only covered by the requirements and not by the design of the system.

4.1.3 HL7 Reference Information Model as Data Model

The data model for the SPL must be based on the generic approach of the HL7 RIM, which was already described in section 3.3. This is due to its advantages in the light of creating a product line for an EHRMS:

- It is a common, maintained and well documented ISO-Standard for health care information systems.
- The model adds the flexibility to extend the data model without changing the relations between entity classes.
- The HL7 RIM appends the ability to create new links between concrete objects at run-time of the application through the usage of association classes.
- Compatibility to other systems is ensured, which implement the HL7 RIM and therefore between all products of the SPL.

4.1.4 Client Flexibility

When designing a SPL, not only flexibility on the application and data layer has to be considered, but also on the client. Concepts have to be made to seamlessly integrate optional and differing requirements.

Reusable User interface components

The approach of this section emerged out of the requirements for adapting the interfaces for adding and editing EHRs to the actual patient data and to the tenant configuration. The existing SAGE-CARE melanoma application used a static predefined form for adding and editing new patients. This has the disadvantage, that all the possible data of the patient is shown to the user. If a concrete product would support various medical specialties, the resulting form would show everything that a user could define for the patient and consequently be extremely large. To face this problem, dynamic forms with user interface components (UIC) are designed. Each UIC encapsulates a set of related data or functionality and can be reused, e.g. grouping the interface data for representing a breast cancer issue. The UIC are addable and rearrangeable by the user to build up a patient record. A navigationbar supports the user to keep an overview of the record and navigate through it. Additionally, tenants have the ability to add customized fields per component.

Variable attribute fields:

The user interfaces should also introduce variable attribute fields (VAF) per UIC. Tenants define per UIC which attribute fields will always be displayed and which attribute fields will be optional. These VAC can be then configured by the user.

4.1.5 Configuration Hierarchy

The product line can become comprehensive over time, needing many configurable items on various levels and serving multiple products and users. Already mentioned commonalities and following variabilities (see chapter 4.2) indicate, that a configuration framework has to be integrated in the architecture, that allows configuration on various levels. This configuration hierarchy is shown in figure 4.1 and is described as follows:

1. **Product line owner:** The product line owner is defined as the company developing the SPL. The owner configures and customizes the product line as a whole, introducing changes which possibly affect all products. This mainly includes implementing new functionalities, adding possible variants for the tenant, changing the data model and modifying common user interfaces.

2. **Tenant:** Products are obtained by hospital groups, hospitals or departments. For simplification and to easily realize that the SPL can serve cloud customers and customers with local servers, a tenant (multi-tenant SaaS context) will equal a product (SPL context). Hence, the tenant configuration process can be seen as the equivalent to the product derivation process of a SPL. This implies that if, for example, a hospital group wants a tenant for each hospital, then multiple configurations have to be created and consequently a product for each hospital is derived. The tenant configuration represents the requirements of a customer who orders and configures a concrete product. Accordingly it includes, but is not limited to, a selection of variants from the variability model. The tenant can only configure a subset of the product line owner configuration.

3. **User:** Users of a tenant refine their corresponding tenant configuration. This is, for example, meaningful for user dependent views, as end-users work everyday with the product and need possibilities to configure it according to their requirements.

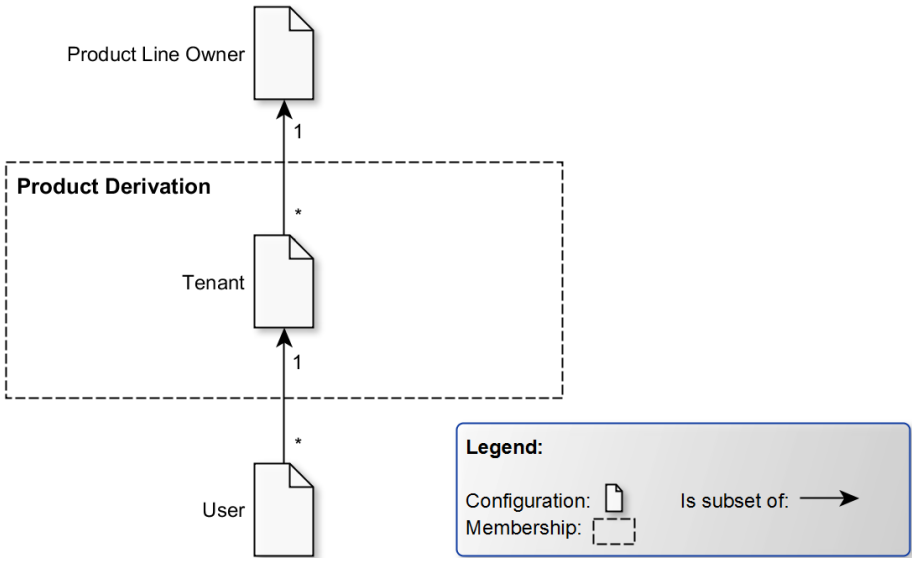


Figure 4.1: Configuration hierarchy

This thesis doesn't include a concrete implementable design for the level based configuration management. Instead, a reference to the corresponding level will be made, if something is configurable.

4.2 Variability Analysis

The variability analysis is responsible for defining and documenting the variation points and their variants in a systematic manner, as already introduced in chapter 3.1.4. As a result the following questions will be examined during this process for defining the variability: [PBL05, p. 63]

- What is the real world item that varies?
- Does it vary in the context of this SPL?
- Which variants can be identified for this SPL?

This is especially necessary to prepare the architecture with the appropriate flexibility mechanisms. The introduced variants in this chapter are analyzed for the SAGE-CARE SPL and partially already contain specific technologies, which is why variants are not described comprehensively within this chapter.

Variation points were categorized according to three main service categories, medical specialties, medical information services (MISs) and technical services, like shown in figure 4.2. While MSs are mandatory for managing EHRs of patients and technical services for common functionalities of information systems, MISs are considered to be optional as further explained in 4.2.2. The complete variability model for the SPL is visualized in appendix B. Presented variation points are generally configurable on the tenant level.

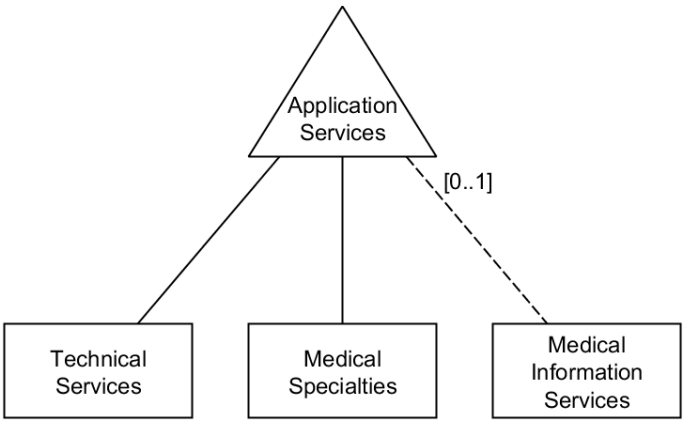


Figure 4.2: Main categories of the OVM

4.2.1 Medical Specialties

In the real world diseases vary widely. Supporting different diseases with different interfaces and services was also one of the main motivations for creating this SPL and consequently it is a variation point with the constraint that only cancer diseases are considered at first. This flexibility is important since not every hospital has consultants for every disease or wants to include new specialties over time. The currently planned variants for MSs are visualized in figure 4.3.

4.2.2 Medical Information Services

There are many approaches for personalized medicine and support services for physicians (see [Ide16]). The term MIS is used in this thesis as a more universal notion for clinical decision support service. Such services "provide clinicians, staff, patients, and other individuals with knowledge and person-specific information, intelligently filtered and presented at appropriate times, to enhance health and health care." [Ber09] The integration of MISs for personalized medicine is also an integral part of the SPL. Despite their importance, their functionality is seen as an addition to the fundamental EHR functionality and hence it is not mandatory to choose a specific amount. Certain MISs are also bound to additional fee-required systems and consequently raise the cost for the product. The variants for MISs and their relations to MS are depicted in figure 4.3.

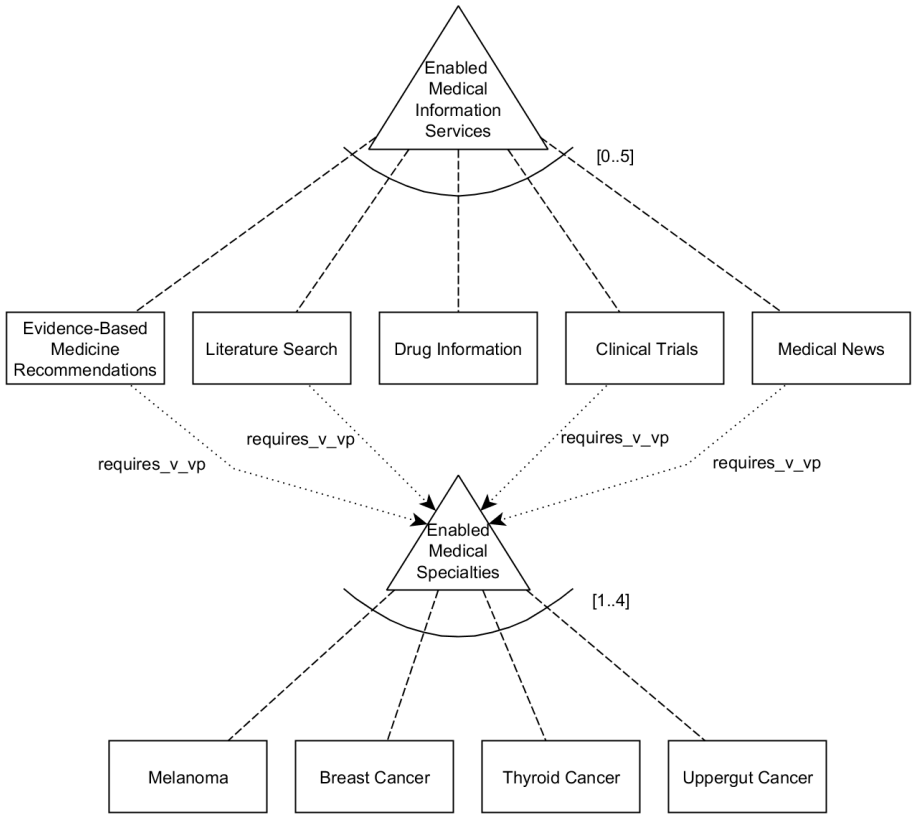


Figure 4.3: OVM for medical specialties and medical information services

4.2.3 Technical Services

User repositories

It is common that hospitals already have a central user repository or directory service for authentication, like Microsoft Active Directory. Others customers want to manage their own users within their product. This is why it is also important to introduce a variation point, which considers different user repositories and additionally includes the own database as a repository, if such services are not used. The variants for user repositories are visualized on the left side of figure 4.4.

Requirement for multiple tenants

Organizations can consist of multiple organizational units, such as hospital groups consist out of multiple hospitals, which need separate products. As already mentioned in the configuration hierarchy, products of this product line are obtained by

hospital groups, hospitals or departments. This forms a hierarchical structure, as many departments can belong to one hospital, which in turn can belong to a hospital group. As a result, the requirement of creating multiple tenants for one customer is also a variation point for the SAGE-CARE SPL. If a customer needs multiple tenants, than multiple tenant configurations may be created. This variation point is shown on the right side of figure 4.4.

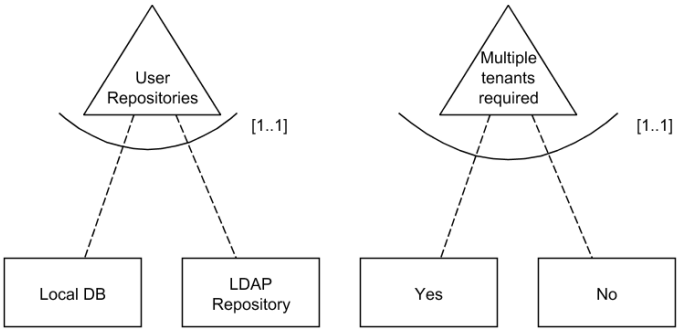


Figure 4.4: OVM for user repositories and multi-tenancy

Supported languages

Language is an obvious variability in the real world. Even though multi-language support is identified as a commonality, the supported languages should be configurable per product to meet the needs of multiple markets. Current languages for the SAGE-CARE SPL are shown in figure 4.5.

Deployment endpoint

A huge amount of application servers and cloud providers exist on the market. Different endpoints for the deployment of the SAGE-CARE SPL products are also considered as important, as not every customer wants to have their data on a cloud system while others rate the convenience of these system as important. Since deploying the application to multiple needed computing platforms is already possible with the existing SAGE-CARE melanoma application, which is the basis for the product line, this is not further examined within this thesis. The proposed deployment endpoints can be viewed in figure 4.5.

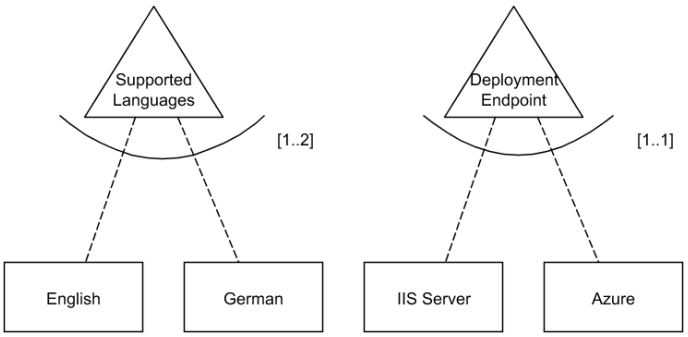


Figure 4.5: OVM for supported languages and deployment endpoint

Additional Services

Additionally the requirement for other technical services was detected, e.g. reporting, data-backup and data-integration services. Reporting provides consultants with relevant statistical health care data. The data-backup service reduces the risk for data loss by backing up the application data onto different distributed nodes, like shown by [LYW13]. Finally, the data-integration realizes access to various independent data sources. [LÖ09]

These are common functionalities for an information system and especially a backup service would meet the requirements of the health care domain. However, these services are not further elaborated within this thesis as those are considered out of scope due to their priority.

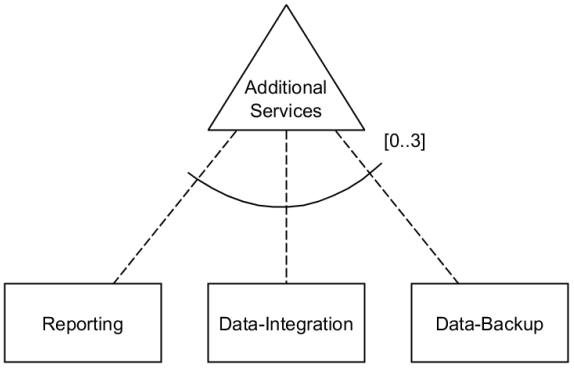


Figure 4.6: OVM for additional technical services

Chapter 5

Domain Design

As stated in chapter 3.1.2, domain design conceptualizes the architecture and its comprised components for the product line with the help of the previously created common and variable requirements. The following chapter doesn't describe the whole system in detail, but rather focuses on designing selected parts, especially the mandatory variants. This includes the architecture of the system, technical services, domain testing and client variability mechanisms for showing EHRs.

5.1 System Overview

The system is designed according to the principles of a DSPL, respectively a reconfigurable multi-tenant aware SaaS application (see chapter 3.1.5). This is an appropriate approach to face the relatively small amount of variation points and to solve the dependencies between the MSs and MISs. Designing the platform as a traditional SPL, according to the classification of table 3.1, would have the significant drawback that adding new MSs or MISs to an existing product would result in redeployment. Additionally, a cloud instance would have to be created for every product, which is not scalable for a rising number of products. The overview of this system and the interaction with the configuration hierarchy is visualized in figure 5.1.

The bottommost layer is represented by the technical infrastructure, i.e. the hardware and server platform for any needed software. Albeit, this is out of scope for this thesis and not further examined.

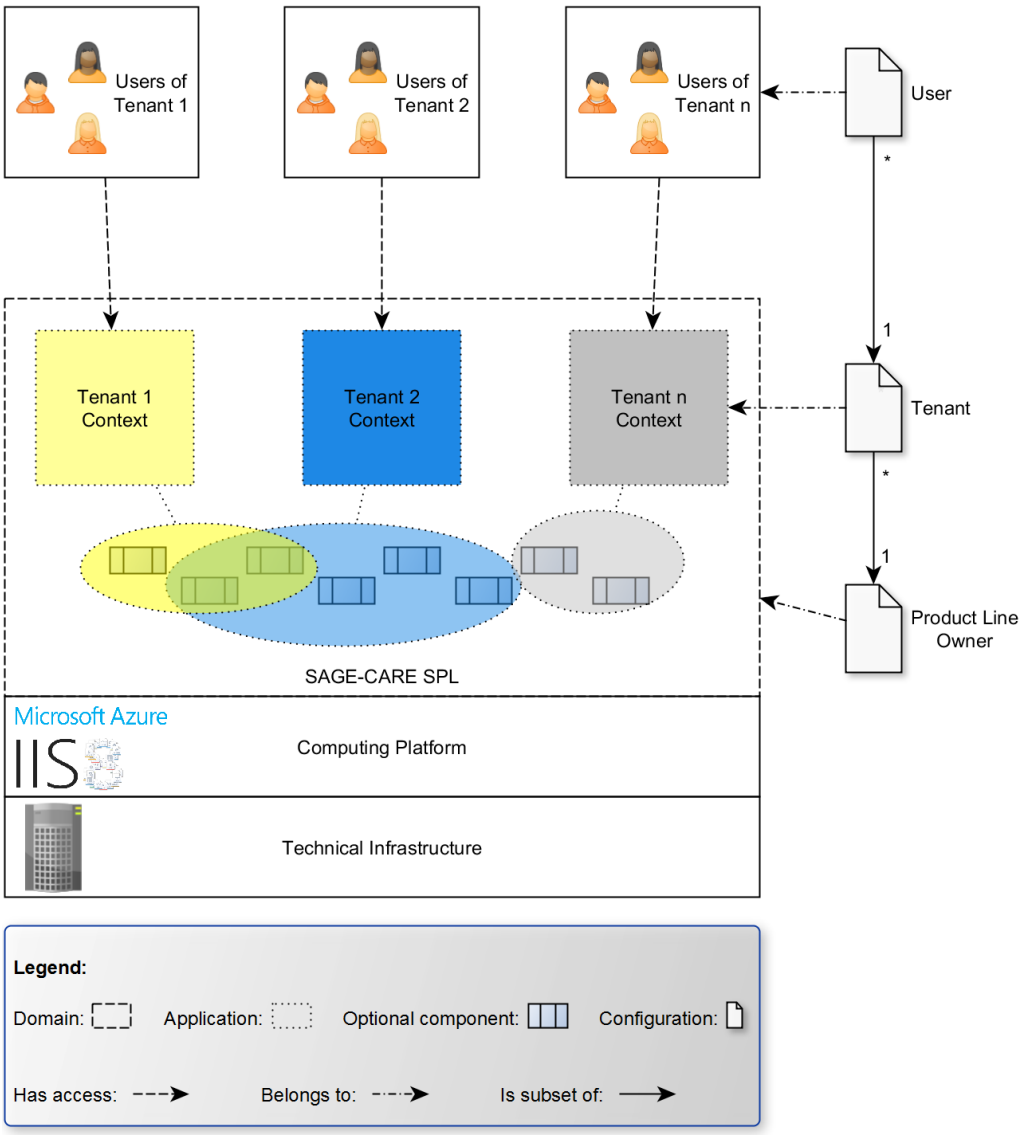


Figure 5.1: SPL system overview

The next layer is the computing platform. This includes the complete software basis, enabling the product line and its applications to run. In the SAGE-CARE context, Microsoft Azure or locally installed Microsoft Internet Information Services (IIS) are the relevant platforms considering the deployment. Hence, they are shown as an example in the depiction.

Subsequently the SAGE-CARE SPL is shown, which is mainly described in this thesis. In the product line context, this is the domain with its contained core assets. These artefacts are the reusable software components, used by the concrete applications. The platform is configured with the help of the product line owner

configuration.

Since this is a multi-tenant aware DSPL, the applications are able to run in one domain instance, while being bound to their own context. This means they are not aware about any other applications on the domain instance. As already defined, a concrete product is mapped to a tenant. Hence, in the course of this thesis the terms tenant context, application and product are used synonymously. Although the term tenant is used for describing the technical concepts, such as a multi-tenant data architecture, while the other terms are more convenient for the SPLE and business point of view. The applications are configured by its corresponding tenant configuration, defining the tenant context. This includes the selected variants, represented by the optional software components in the figure, and additional tenant specific data and user interface elements.

The users, at the topmost section, belong and have access to exactly one tenant. A user configuration is assigned for every user, which represents a subset of the tenant configuration.

5.2 Architecture

The architecture is closely related to the existing EHR application of the SAGE-CARE project and already considers its functionality. It is structured in three layers, which are loosely coupled. An overview of the system is visualized in figure 5.2, with the help of a UML package diagram. In the SAGE-CARE SPL, outer packages are realized as separate projects, while inner packages are implemented as folders. Except for the data layer, which consists of multiple independent existing systems. Since this is technology related, the more independent terms layers and sections are used.

Packages with a grey outline are not further discussed in this thesis, as they have no importance for the product line context. However, they are added for the sake of completeness.

The first layer, called client layer, consists of the *EHR client*, which is the configurable client for all products of the product line. It is designed as a web client, that is loosely coupled to the *REST WebAPI* and contains two sub packages. The view packages contains the view components, being made up of the user interface and corresponding view logic. Whereas the service package contains reusable client

logic, shared by multiple view components. A basic concept of realizing variants in the client is shown in chapter 5.7. Concrete concepts, like dynamic views with user interface components (UIC), for the user interfaces of the client are outlined in chapter 5.10

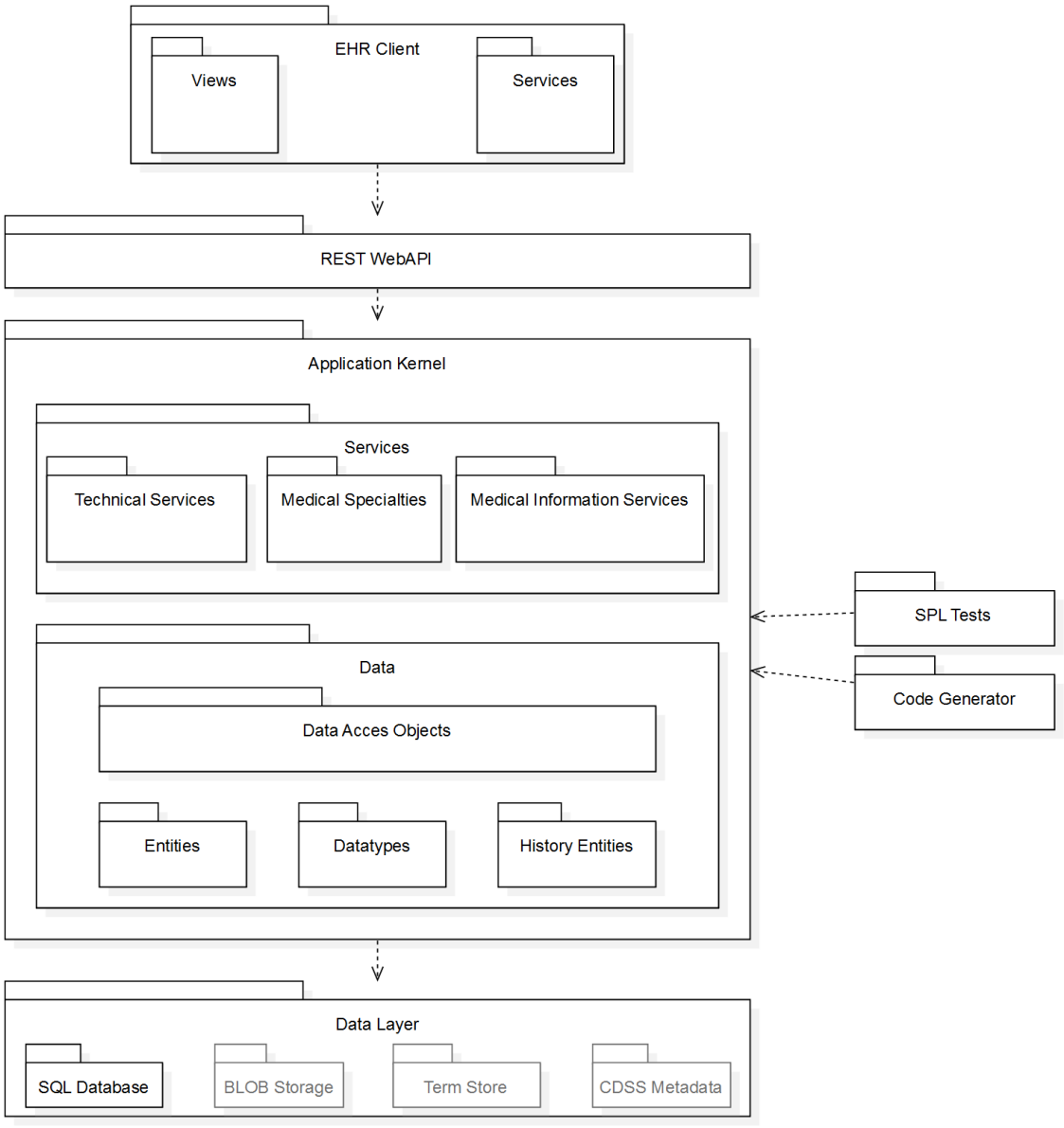


Figure 5.2: SPL architecture

The next layer, named application layer, consists out of the *REST WebAPI* and the *Application Kernel*, which are tightly coupled. The *REST WebAPI* is mainly a facade for the application kernel, which provides an interface for the client and is designed according to the Representational State Transfer (REST) principle. This

paradigm makes use of web technologies for communication and designs the application interface according to provided resources (see [Fie00] for further reading).

The *Application Kernel* contains the business logic of the application. It is mainly made up of the different types of services, that were already identified in chapter 4.2, and a data access layer. Currently the functionality concerning the MSs is covered by the data access layer, as it includes adding and editing patient data. However, it is assumed that business logic for the specialties will follow and therefore it is considered in the architecture. Some MISs also already exist in the current SAGE-CARE melanoma application. But for the SPL they have to be made aware of various specialties by extending their interface with a parameter, determining the specialty, and integrating variability mechanisms. The data access layer uses data access objects (DAOs) and an object-relational mapper (ORM), for accessing a relational database. The packages *Entities*, *DataTypes* and *HistoryEntities* contain the data items of the system and are further explained in chapter 5.4 and 5.5.

The third layer contains the persistent data of the SPL and will be addressed as data layer in this thesis. It is made up of different storage types, although only the relational *SQL Database* is relevant in the course of this thesis. The *BLOB Storage* is used for storing files, e.g. images of melanomas, while the *Term Store* and *CDSS Metadata* are used for MISs (see [Ide16]).

Two additional projects are mentioned in this thesis, that are tools for the product line owner. The *SPL Tests* project contains all software tests for the product line and is further described in chapter 5.9. The *Code Generator* is a framework for creating a flexible data model and is designed in chapter 5.4.

5.3 Generic Data Model

As already stated in chapter 4, the HL7 RIM data model is required, because of its flexibility and propagation in the health care domain. However, the data model for this thesis is slightly modified to handle some of the disadvantages without losing the compatibility to the standard.

Looking at the official data model in figure 3.5, the connection between an `Entity` and a `Role` is not realized via an association object, in contrast to the other relations. The associations also have a specific meaning and forces the connections to either be scope or player-related, instead of letting an association object define the type of

relationship. Because of this inconsistency, the relationship is realized via a newly introduced association class called EntityRoleRelationship. This class takes care of connecting an Entity to a Role, as illustrated in figure 5.4.

Another disadvantage is the overuse of attributes that configure the association classes and accordingly specify the relationship between two base classes. For instance, 19 attributes are settable for ActRelationship according to the reference model shown in appendix A. Those were replaced by the much simpler and convenient mechanism, of using two relationship type attributes. The value of these attributes represent the variable names for each site of the relationship, which would normally be used when designing a data model with concrete relations.

The principle of relationship types and its connection to a concrete relationship are visualized in figure 5.3 with the help of a UML Object Model. The depiction shows a concrete relationship between a `Patient` and a `Medication` object at the top. The bottom section shows how this is mapped to the relationship type principle. The relationship type attributes of the `Participation` object are labeled as `ActType` and `RoleType`, to make an assignment to the corresponding object possible.

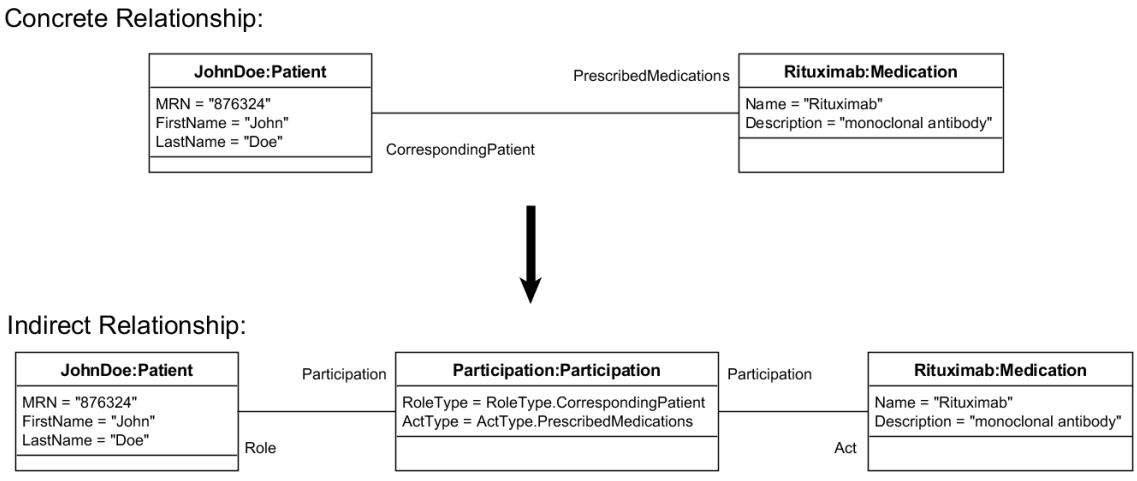


Figure 5.3: Relationship type principle

This principle is used for all association objects, creating the abstract data model shown in figure 5.4, where all other classes in the SPL will inherit from. The data model ensures enough flexibility for the given requirements, while not losing the compatibility to the HL7 standard. The designed classes of this chapter are integrated in the *Entities* section of the architecture, see figure 5.2.

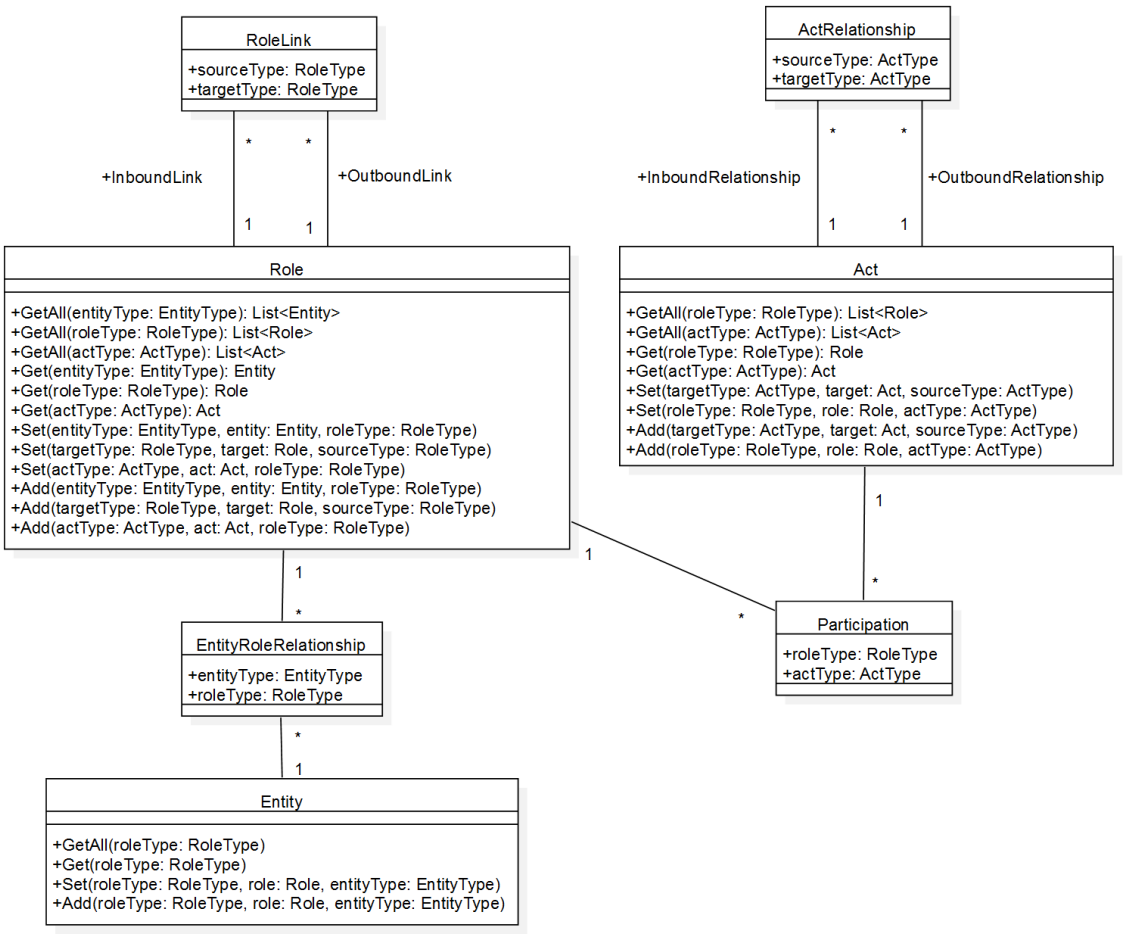


Figure 5.4: Class model of the SPL

Besides its advantages, the usage of association objects has the disadvantage of making it difficult to navigate between two objects. For example if a `Patient` object (derived from `Role`) has a relationship to a `Medication` object (derived from `Act`) and multiple other `Acts`, the developer would have to iterate over the collection of `Participations` and find the `Medications` via the `actType` variable. This issue can be be addressed by implementing an efficient data structure combined with getter and setter methods, like further elaborated in chapter 6.1.2. A getter method, respectively accessor method, is used for accessing an object without modifying it, whereas a setter method, also known as mutator method, modifies an object.[Hor10, p. 46] These operations are integrated into the base classes, which makes them available for all classes. Following methods are designed to make the navigation more convenient, as already shown in figure 5.4:

- `GetAll`: Gets all related objects with the passed relationship type.

- `Get`: Returns a single related object with the relationship type from the parameter. Assumes, that only one related object with the corresponding type exists.
- `Set`: Creates a new association object, which connects the caller to the passed object. Sets the relationship types on the association object. Ensures that the caller has only one relationship with the corresponding types. This method is used to set 1:1 relationships.
- `Add`: Same as the Set method, but simply adds the the association without ensuring unique occurrence. Used for realizing 1:n or n:m relations.

Additionally to the HL7 RIM back-bone classes, other classes were identified as particularly important, respectively helpful, in the currently existing products and during the design of the SPL. Hence, they are described ancillary:

- `AbstractEntity`: This class is the parent class for all HL7 RIM back-bone classes and hence for all classes in the data model. This simplifies the process of managing properties, that concern the complete data model like a table identifier or a tenantID (see chapter 5.6). However, the `AbstractEntity` is designed as an abstract class and can not be instantiated.
- `Patient`: The patient is the central class for the basic patient information. Every additional patient information, e.g. corresponding medications or issues, are linked directly or indirectly to this class via the association objects.
- `Issue`: Parent class for all MSs. Has a similar fundamental meaning for the system, as the patient object. A patient is only added to the system if he has an issue and an issue always belongs to a patient. This reflects the situation of the real world. A hospital only treats a person if an issue exists, which makes the person a patient of the hospital.

5.4 Code-Generation of the Data Model

The generic data model is an important step towards a flexible data management in the SPL. However, the flexibility and especially extensibility is even more enhanced when combined with a configurable code-generation tool. This framework was firstly

introduced and designed by [HW15] and is integrated in the *Code Generator* project as shown in figure 5.2. It is outlined in this thesis to show the advantages of this framework for a product line and the integration into the configuration hierarchy.

Foundation for the framework is the data model configuration with the help of a spreadsheet, containing three worksheets. The first worksheets defines the entity classes and its superclasses. An example for the HL7 back-bone classes and the patient class is shown in figure 5.5.

1	Class	Superclass	Comment
2	Entity	AbstractEntity	
3	Role	AbstractEntity	
4	Act	AbstractEntity	
5	ActRelationship	AbstractEntity	
6	RoleLink	AbstractEntity	
7	Participation	AbstractEntity	
8	EntityRoleRelationship	AbstractEntity	
9	Patient	Role	

Figure 5.5: Entity classes worksheet

The second worksheet defines the entity attributes for the entities of the first worksheet. This sheet is visualized in figure 5.6 and shows again the patient class as example. Beside the attributes, it also defines the cardinality, with the possible values *0*, *1* or ***. If *CardinalityMax* is set to ***, a list gets created.

1	Classes	AttributeName	Datatype	CardinalityMin	CardinalityMax	Comment
63	Patient	Mobility	Mobility	0	1	
64	Patient	MRN	String	1	1	
65	Patient	HomeSituation	HomeSituation	0	1	
66	Patient	ResidentialCare	ResidentialCare	0	1	
67	Patient	ClinicallyReviewed	ExtendedBoolean	0	1	
68	Patient	Comments	String	0	*	
69	Patient	Warfarin	ExtendedBoolean	0	1	
70	Patient	NOACs	ExtendedBoolean	0	1	
71	Patient	Antiplatelet	ExtendedBoolean	0	1	
72	Patient	DrugRegimen	ExtendedBoolean	0	1	

Figure 5.6: Entity attributes worksheet

Lastly, the third spreadsheet is designed for defining enumerations, which is shown in figure 5.7. The column *Type* defines the enumerated types, while the *Label* column lists their members.

1	Type	Label	Comment
38	Mobility	Full Mobility	
39	Mobility	Not mobile	
40	Mobility	With help	
41	Mobility	Unknown	

Figure 5.7: Enumerations worksheet

The spreadsheet is loaded by the code generator, which subsequently generates

the configured classes. Generated entities are saved in the *Entities* folder of the *Application Kernel*, while Enumerations belong to *Datatypes* (see figure 5.2). An example for the shown patient class is shown in listing 5.1, implemented with C#.

```
public partial class Patient : Role
{
   public CNSIssue? CNSIssue {get; set;}
   public Mobility? Mobility {get; set;}
   public String MRN {get; set;}
   public HomeSituation? HomeSituation {get; set;}
   public ResidentialCare? ResidentialCare {get; set;}
   public ExtendedBoolean? ClinicallyReviewed {get; set;}
   public ICollection<String> Comment {get; set;}
   public ExtendedBoolean? Warfarin {get; set;}
   public ExtendedBoolean? NOACs {get; set;}
   public ExtendedBoolean? Antiplatelet {get; set;}
   public ExtendedBoolean? DrugRegimen {get; set;}
}
```

Listing 5.1: Generated patient class

This is an important addition to the SPL, as it introduces a shift from programming to configuring the data model. Hence, it increases the flexibility and makes the increasing model better manageable. In the context of a product line it also has to be considered, that this sheet concerns the product line owner and the tenant configuration. The product line owner manages data items, which are valid for the whole platform. Tenant specific data items belong to the tenant configuration, but are nevertheless managed in the same spreadsheet, as only a small amount of tenant specific data is expected. It is supposed, that the flexible data model will already cover most of the tenant specific needs. However, the *Comment* column, which is added to every worksheet, is envisaged for documenting the connection to the tenant configuration.

5.5 Data Historization

As already described in section 4.1.2, implementing a data historization is required because this system is used in the health care domain. For this reason, changes have to be comprehensible and data shouldn't be deletable. Although the historization can be used to show the disease process to a physician, the historization is integrated in the *Technical Services* section of the architecture, see figure 5.2. This is due to

the fact, that it is designed in a universal way, covers the complete data model and helps to fulfill compliance standards.

In the context of this thesis, data historization means saving every change that was made to a table entry. This means more precisely that every create, update and delete operation on a table gets tracked in a corresponding historization table. To fully automate and integrate the data historization, the component for code-generation, introduced in chapter 5.4, gets extended with the functionality of creating additional history entities. These history entities are saved in the *History Entities* section of the architecture, as shown in figure 5.2.

For every created entity of the code-generator, additionally a corresponding history entity gets automatically created. These history entities have the same attributes as their corresponding entity to ensure compatibility and are mapped automatically to the *SQL Datase* of the *Data Layer*, since an ORM is used. Following additional attributes are added to the history entities:

- `OriginalID`: Historizes the database identifier of the corresponding entity.
- `Action`: Documents if an add, modify or delete operation was executed.
- `User`: Documents the user who was responsible for the data modification

The result of this creation process is visualized in figure 5.8.

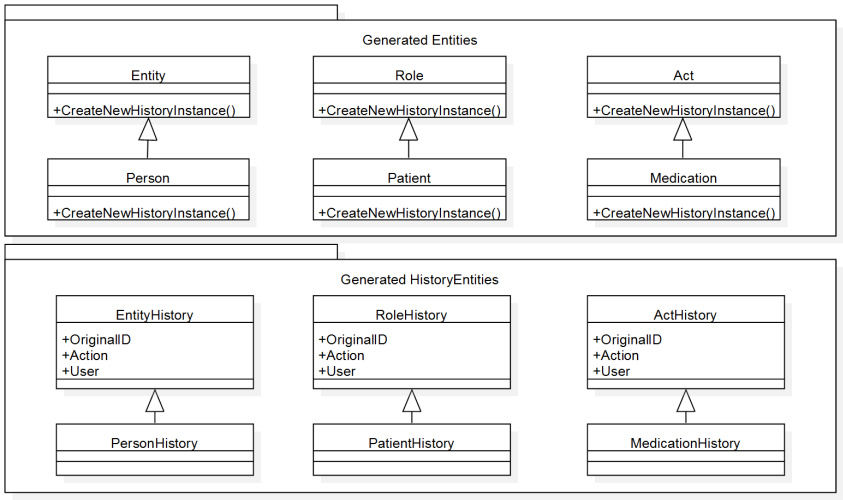


Figure 5.8: History entities

The factory method (see [Gam+11, p. 107]) `CreateNewHistoryInstance()` is also automatically created and overridden in every class. It returns a new instance of the

corresponding history entity object. For example the `CreateNewHistoryInstance()` method of a `Patient` object would return a new instance of `PatientHistory`. As a result, the two hierarchies are connected on a programmatic level and can be used for the automated historization process.

For this purpose, an integration into the transaction context, which is responsible for handling all the transactions that are made to a database, has to be conducted. Whenever the transaction context saves an entity to the database, it can at first create a corresponding history entity via the factory method. The next step to detect all the changes that were made to the entity. As stated previously, changes are introduced through add, delete or modify operations. Normally an EHR system wouldn't support deleting patient information anyway, but it is implemented nevertheless to counteract data loss. If the used technology doesn't already offer change tracking functionality, the change detection has to be implemented manually, in order to compare the changed entity to the existing database entry. However, this mechanism is not further elaborated in this thesis since such features are usually implemented in modern object-relational mapping frameworks such as Entity Framework or Hibernate. [Mica]; [Red] Finally the detected changes have to be saved. Depending on the performed operation, the following strategy is recommended:

- **Added:** Every property with its value gets historized.
- **Deleted:** Every property with its original value gets historized.
- **Modified:** Only properties where the new value differs from the original value from the database get saved. The new value is used for the historization.

This approach is resulting in the example historization table in figure 5.9. It shows the history table for a `BreastCancer` entity. The example object with the `OriginalID` 77 got initially added, then modified twice and in the end deleted.

BreastCancerHistory Table

Id	Action	OriginalID	ScoreB	Diagnosis	...
1	Added	77	3	Ductal carcinoma in situ	...
2	Modified	77	5A	null	...
3	Modified	77	null	Invasive carcinoma	...
4	Deleted	77	5A	Invasive carcinoma	...
...	...	...	...	...	...

Figure 5.9: Example of a BreastCancerHistory database table

5.6 Multi-Tenant Data Architecture

Multi tenancy is declared as a commonality for the product line, since without this concept, it wouldn't be possible to serve multiple customers with one cloud instance or separate finer grained parts of a superior business unit. Only the view on the relational application data is covered in this chapter, although multi-tenancy has influences on many other parts of the system, like the access management or the user interface.

The terminology is not always consistent, but literature basically distinguishes between three approaches for a multi-tenant data architecture: [CCW06]; [KMK12]; [Sch14]; [Micb]

- **Separate Databases**: Assigns each tenant its own database. This is the simplest approach and has the advantages of high security and easy backup mechanism since data is logically separated. However this approach may have a negative influence on flexibility since increasing the number of tenants can be expensive at a certain amount because of the high number of databases.
- **Shared Database, Separate Schemas**: This method shares a single database between all tenants but separates them by creating a schema for each tenant. This approach is very similar to the separate databases principle and therefor still has a negative impact on flexibility since creating new tenants can become expensive .
- **Shared Database, Shared Schema**: When sharing the database and the schema, the tables must include an additional column with the tenant ID. This ID assigns every row to the related tenant. The approach has the lowest overhead when it comes to adding or removing tenants, since they are just represented by an ID in the table, and therefore makes them easily configurable and scalable. Another advantage is the simplicity of querying and joining data from different tenants, e.g. to collect data for a clinical trial. The drawbacks are poor performance when querying the database since it creates large tables and the results must be filtered by the tenantID, although this can be eliminated by creating table views per tenant. Another disadvantage is the sparsity of tenant dependent columns. However, sparsity can be handled by state-of-the-art databases like MSSQL Server, as described by [HW15].

Because of its advantages for a flexible solution and the manageable disadvantages the shared database, shared schema approach is used. It is also the most appropriate for a multi-tenant SaaS application, as suggested by [CCW06], because of its advantages when handling large amounts of tenants with a small number of database servers. The impact of this principle on all database tables is visualized in figure 5.10 where the column *TenantID* assigns the corresponding row to a tenant. Considering the already described data-model, this addition is appropriately integrated as property of the `AbstractEntity` class.

Id	TenantID	FirstName	...

Id	TenantID	MRN	...

Id	TenantID	ScoreB	...
1	5	5A	...
2	1	1	...
3	1	3	...
4	7	4	...

Figure 5.10: Tables for the Shared Database, Shared Schema principle

To ensure security and performance, the Tenant View Filter pattern is proposed by [CCW06]. It states that for every tenantID there should be a tenant filter, containing a tenant dependent view for each table. Clients are only allowed to work with the filtered data according to their tenant. This process is visualized in figure 5.11. For the sake of simplicity, the client in the diagram directly communicates with the transaction manager which is responsible for handling the database connections and queries. Considering the initially described architecture, the client would have to communicate with the *REST WebAPI*.

The process steps in the figure are described as follows:

1. Users perform a login at a certain tenant. The session of a user is always bound to his corresponding tenant.

2. When a user sends a data query to the system, with the help of the client interface, he hands over his tenantID to the application layer. User requests without the tenantID are not possible anymore.

3. The application layer queries the tenant filters of the database, handing over the tenantID.

4. The database returns the corresponding table views, if they exists. If not, the table views must be created. A view enables the transaction manager to filter tables efficiently according to the tenantID.

5. In the next step the transaction manager sends the user query to the tenant filtered data, which is a subset of the entire relational data.

6. The queried data gets returned from the SQL database to the application layer.

7. The queried data gets returned from the application layer to the client.

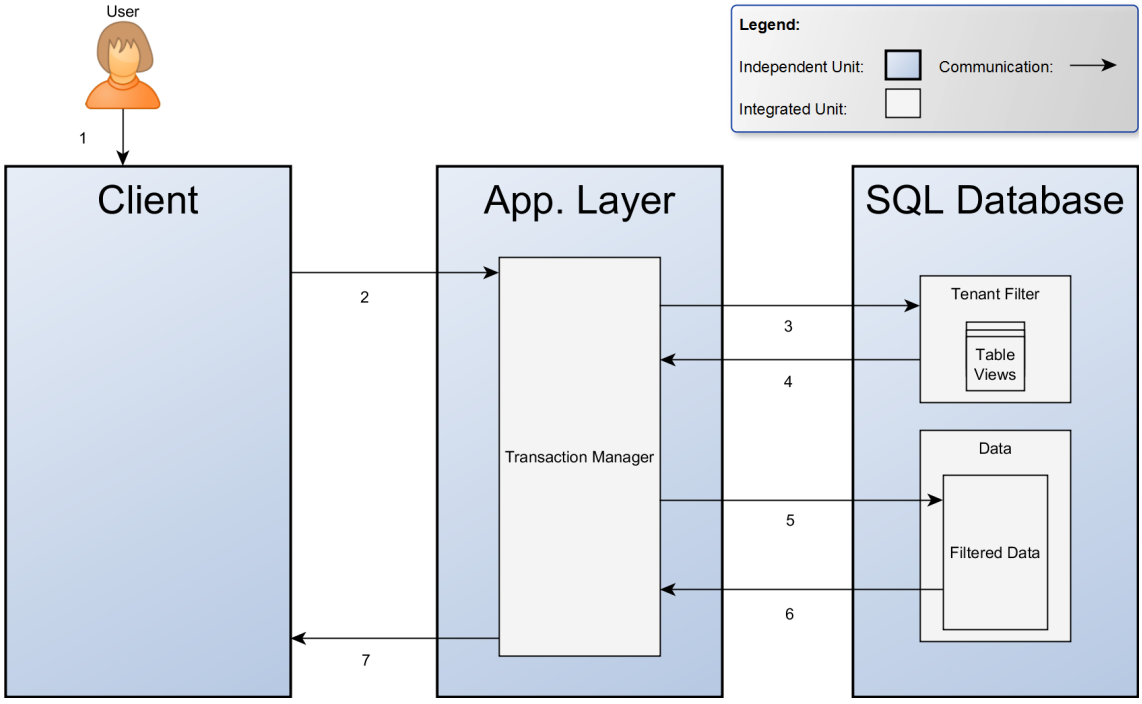


Figure 5.11: Multi-tenancy integration with the Tenant View Filter pattern

Finally through this way of multi-tenant integration adding and removing tenants are a matter of configuration for the product line owner, if the transaction context can automate the view creation. However, since this is a common implementation problem, it is supported in ORM frameworks like Microsoft EF (see [Bér16] for further reading).

5.7 Access Management

As already described in section 4.1.2 the access management is mandatory for all products of the product line to prevent not allowed access to the system. This section describes the initial design for the SPL access management. But since IT-security is a very comprehensive topic, it is not covered completely in this thesis.

The requirements already identified two access restriction points, one concerning the application layer, the other concerning the client layer. Hence, this subject is also divided in two subsections.

5.7.1 Application Layer

The access restriction mechanisms for the application layer represent the crucial part for the access management. They prevent not permitted access to the system, even if a user would see not allowed functionality or would manually send a request to the server.

The attributes of a user are bundled in an object called the user account, which contains a userID as the unique identifier. For the authentication process, a user has to provide his userID and a credential to the system. The credential provides evidence for the users claimed identity, e.g. a password which is only known to the user and the system. If userID and credential match together, the user is authenticated. A security context or token can be bound to the users connection as evidence that the user has already been authenticated. Otherwise a user would have to provide his userID and credential for every request. [Tod07, pp. 4–5] [Ben06, pp. 3–4]

After the authentication, the requests of the user can be authorized. Authorization can be defined per operation or resource that an interface offers and is often realized via a role based access control model, as described by [Ame04]. This model is based on on users, roles and permissions. A role is defined as "a job function within the context of an organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role." [Ame04, p. 2] A permission allows a role to perform an operation or access a resource. As shown in figure 5.12, users and roles have a many-to-many relationship, as well as roles and permissions.

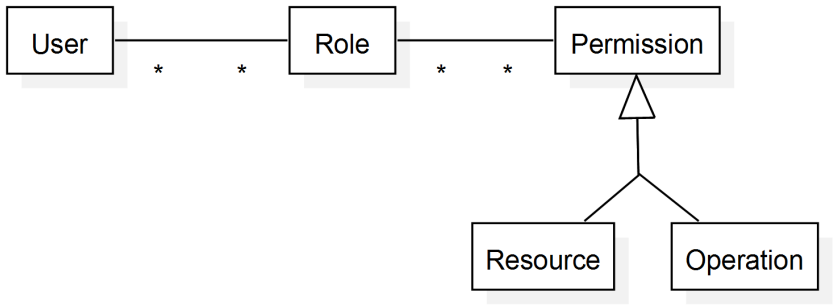


Figure 5.12: Relationship between users, roles and permissions according to [Ame04]

Another level of authorization is the verification of the tenant rights. The tenant rights are derived from the tenant configurations and mainly represent the chosen variants of the tenant. For example, if a tenant chose only the *Melanoma* variant of the variation point *Enabled Medical Specialties*, then all users of this tenant would only be able to manage patients with a melanoma.

With the help of this model, the authentication and authorization process, visualized in figure 5.13, for the SPL can be designed. The process is additionally based on the security, authentication, and authorization documentation for ASP.NET Web API, since this is a state-of-the-art framework. [Mice]

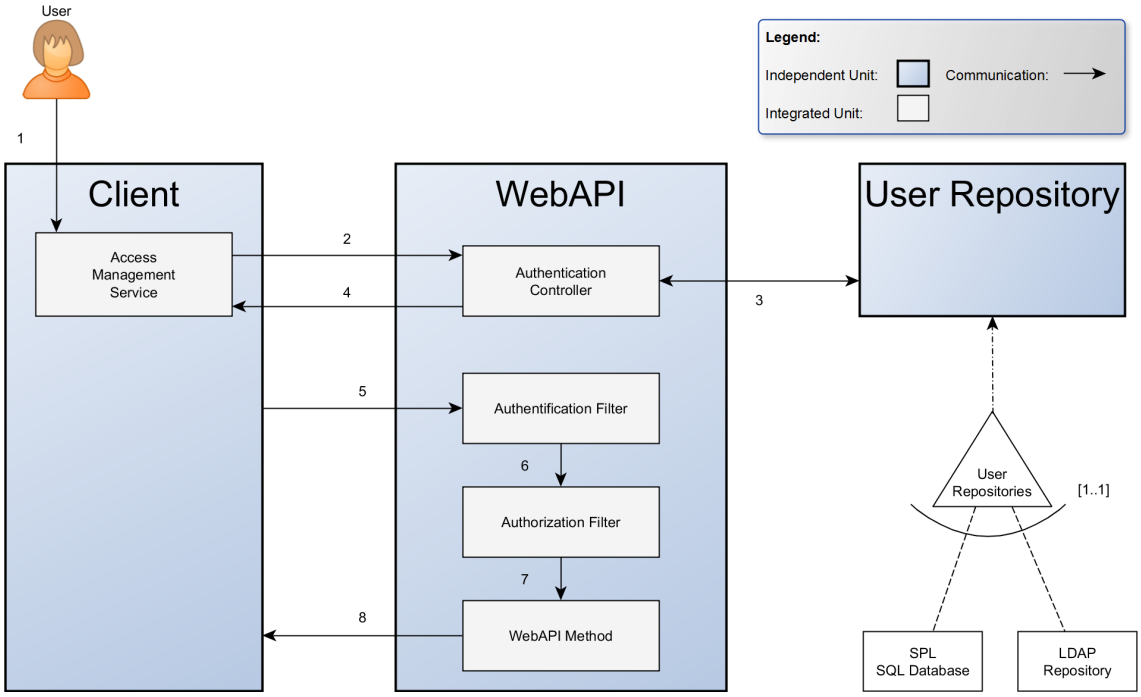


Figure 5.13: Authentication and authorization in the SPL

The numbers in the figure represent the following steps:

1. At first the user hands over the userID, tenantID and credential, in this case a password, to the *Access Management Service* of the *Client*. The *Access Management Service* is responsible for handling all access management related functionalities and data at the client. It is implemented in the *Services* section of the *EHR Client*, since it is used by multiple user interfaces.

2. User data gets handed over to the *WebAPI* of the server.

3. The *Authentication Controller* of the *WebAPI* validates the data against a *User Repository*. As stated in section 4.4, the user repository is a variation point and hence different repositories must be supported. A strategy pattern as variation mechanism is considered as appropriate, since this variation point represents a family of related algorithms. [Gam+11, p. 315]

4. If the user is valid, the already described token or security context is returned. Otherwise an appropriate exception must be sent back to the client.

5. Every subsequent request from the *Client*, has to contain the token, the tenant, the WebAPI method, method parameters and, if the WebAPI method is dependent on medical specialties, the concrete specialty is needed. With this information, the questions from chapter 4.1.2 can be checked. At first the token gets validated by a *Authentication Filter* to see if the request has been made by a valid user.

6. The next filter that applies is the *Authorization Filter*. First, it checks whether the user's corresponding tenant has the rights to perform the method. This is validated with the help of the tenant configuration. After that, the user's corresponding rights are checked with the help of the assigned roles. Both filters are applied before every call of a WebAPI method and are therefore suitable for an aspect-oriented design.

7. The method gets executed if authentication and authorization succeeds, otherwise the user gets an appropriate exception gets created.

8. The result of the WebAPI method is returned to the client.

Already proposing concrete roles or even designing a user management system is not part of this thesis anymore and is considered as future work.

5.7.2 Client Layer

A second access restriction point was already mentioned in the requirements for the access management (see chapter 4.1.2). As stated, this is necessary to hide interfaces and functionality, which the user is not allowed to see. Additionally this is important for the product derivation process, as the presented mechanisms result in showing the users of a tenant the right interfaces for the corresponding product.

The initial authentication is realized with a login page on the client side, where the user can type in his username and credential. Without a valid security context, the user can't get past this interface. As this is a common mechanism for authentication and the communication between client and application layer was already described in chapter 4.1.2, it is not further elaborated in this thesis. However, even if this procedure is obvious, it is still mandatory and for a better understanding, the user interface for the authentication is visualized in figure 5.14.

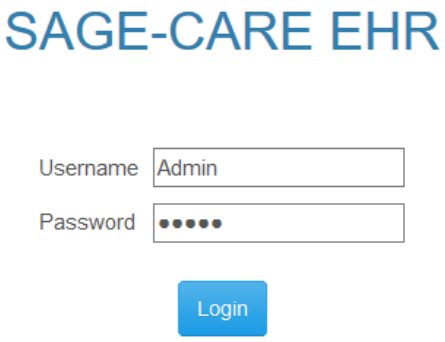


Figure 5.14: Login page for user authentication on the client

Subsequent user interfaces are filtered with an analogical principle like on the server. This means elements of the user interface can be assigned with an authorization filter, like methods of the WebAPI, which authorizes according to user rights, tenant rights and tenant membership. The authorization for tenant membership is additionally needed to realize tenant specific, respectively differing, UI elements. This principle is visualized by means of two scenarios in figure 5.15. The scenarios both show the same interface, where the user has to select a medical specialty for a new patient. However, the interface has to adapt dynamically to tenant and user rights since it is bound to the VP *Enabled Medical Specialties*. To be more precisely, the buttons on the interface have to annotated with the client authorization filter, which includes them dynamically.

- **Scenario 1**: The tenant *Cork Hospital* has the right to manage patients of all MS variants and so does the currently logged in user *Joe Bloggs*.
- **Scenario 2**: The tenant *Dublin Hospital* selected the MS variants *Breast Cancer*, *Thyroid Cancer* and *Uppergut Cancer*. However, the currently logged in user *Jon Doe* is only authorized to manage patients with breast and thyroid cancer.

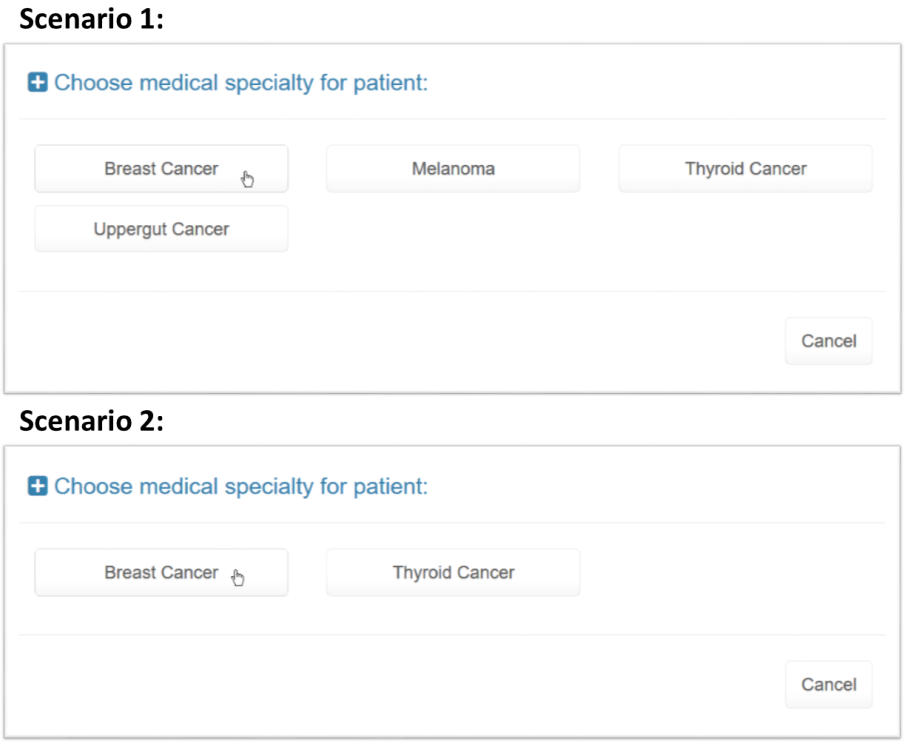


Figure 5.15: Adapting the user interface according to tenant and user configuration

For an efficient authorization process, the tenant and user rights must be saved in the already introduced *Access Management Service* on the Client (see chapter 5.7). When a user successfully logs in to the system and gets a response from the *Authentication Controller*, as shown in step 4 in figure 5.13, the controller has to additionally send back the corresponding tenant and user rights. These rights are saved in the *Access Management Service*, which exports an interface with the following method for the UI elements authorization filter:

- `isUserAllowed(neededTenantRights, neededUserRights, neededTenantMembership)`: The first parameter hands over the tenant rights, which are needed for viewing the UI element, while the second one concerns the needed

user rights and the third the needed tenant membership. The needed rights are compared to the provided rights from the *Authentication Controller*. As already stated they are saved in the *Access Management Service* anyway and don't need to be passed as parameter. The method returns true if the user provides the needed rights and membership, otherwise false is returned.

If the method returns false the client authorization filter has to take care of excluding the UI elements. Producing a query to the application layer per authorized element of the user interface would result in a great quantity of requests and possible poor performance.

This design fundamentally enables authentication, and authorization via user specific interfaces on the client. However, a more detailed design of the *Access Management Service* and the authorization filter for UI-Elements is considered as future work and is not part of the thesis anymore.

5.8 Multi-Language Support

The multi-language support is a common requirement, but the design has to consider, that a tenant can configure the supported languages for his product. Therefore, it is a matter of product line owner configuration to add, modify and remove languages for the platform and a matter of tenant configuration to select the languages for a concrete product. The multi-language component is integrated in the *Technical Services* section of the architecture, see figure 5.2.

A common way to structure and store translations are resource files. These files include simple key-value pairs and a locale identifier in their filename. The key-value pairs consist of a unique identifier for the translatable text across all files as key and the corresponding translatable text as value. The locale identifier is an easily identifiable language code, e.g. *en* for English or *de* for German, which can vary dependent on the used technology. [Ess00, pp. 31–32]

Table 5.1 shows an example for two resource files for English and German and the Microsoft .NET naming conventions and file types. Programming languages like C# or Java are able to organize their files in a similar way and offer mechanisms to read them at application run-time by simply stating the basic filename and a given locale identifier. [Ora]; [Micd]

Language:	English	German
File name:	translations.en.resx	translations.de.resx
Content:	`pOverview = "Patient overview", login = "Login", logout = "Logout"`	`pOverview = "Patientenübersicht", login = "Anmelden", logout = "Abmelden`

Table 5.1: Resource files example

For the SPL exactly this principle is used for internationalization. For the resource files, the naming convention is equivalent to the example: `translations.«language identifier».«file extension»`. Three design units have to be integrated into the system for this concept:

- **Multi-Language Service**: Service for the client which saves the currently selected translations, enabling views to query the key-value pairs efficiently. Additionally it takes care on the client side, that a tenant can only select his supported languages.
- **Multi-Language Controller**: Defines the interface for the Multi-Language component on the *REST WebAPI*.
- **Multi-Language Component**: Is integrated in the *Application Kernel* and contains the application logic and the resource files to ensure performance and availability.

The interaction between these units is visualized in figure 5.16, where the numbers represent the following process steps:

1. When a user opens the client, the *Multi-Language Service* sends a request to the *Multi-Language Controller* handing over the tenantID as parameter.
2. The controller sends back the supported languages and the default language for the current tenant, which is declared in the tenant configuration. The *Multi-Language Component* is not queried, as no business logic is needed.
3. The *Multi-Language Service* sends a query to the *Multi-Language Controller*, handing over a language identifier as parameter. The language is either se-

lected by the user or the default language is used when the client is opened for the first time.

4. The *Multi-Language Controller* passes the query to the *Multi-Language Component* on the server.

5. The *Multi-Language Component* loads the appropriate resource file and returns it in a suitable data format, such as a dictionary.

6. The controller returns all translations for the selected languages to the Multi-Language Service, where user interfaces can query single translations. Querying the server for every single key-value translation pair is considered as bad design, as it would result in a large amount of requests.

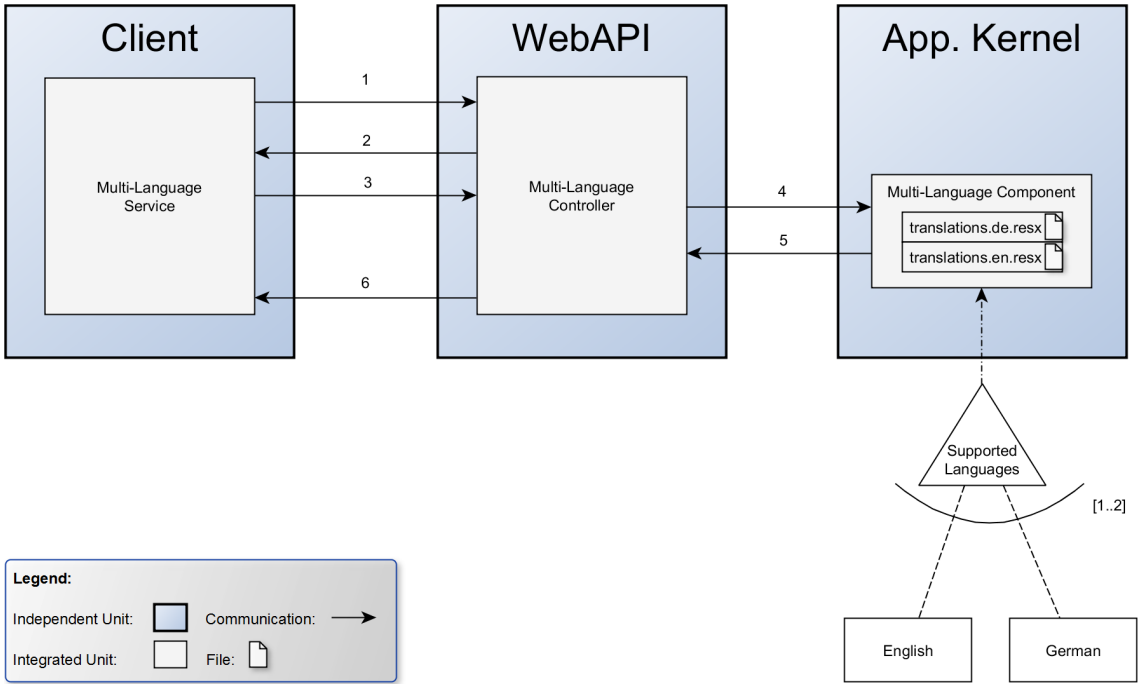


Figure 5.16: Multi-language support in the SPL

5.9 Product Line Testing

As described in chapter 3.1.2 and 3.1.3, testing is an integral part of SPLE. This chapter should demonstrate the difficulties of testing in a SPL and especially show a solution how to organize tests in a way that is sufficient for the requirements of a DSPL and integrate them into the proposed architecture.

The testing process of product lines differs from single application testing, as there are multiple running applications which can be tested and variability mechanisms have to be considered when designing the tests. The testing process of a DSPL slightly simplifies the process, as it relies on derivation via configuration and run-time mechanisms, instead of derivation via building independent products. Hence, concrete products doesn't have to be build before testing. However, it can become more complex to structure the tests as everything is realized with one large platform. As a result a categorization of tests according to multiple dimensions is proposed in this chapter to easily structure and run tests. The following dimensions, each with multiple categories, were identified:

- **Variation Points**: Since variability is ubiquitous in the SPL, it has to be considered in the testing process that tests are responsible for certain variants. Hence a variation point becomes a dimension for structuring tests, with its corresponding variants as categories.
- **Products**: Tests have to assignable to concrete products, respectively tenants, which form the categories of this dimension. Tenant specific parts should be avoided in a DSPL but they are not prohibited. This is why they have to be considered and testable.
- **Testing Levels**: Different testing levels, like system, integration or unit tests, are also available in single application testing but are not less important for DSPL testing.

A separation of tests on a per file basis would not be sufficient, since this would mean that every test could be assigned to exactly one category. But a unit test could belong to a specific variant, while also only being valid for selected tenants. Simply agreeing to structure the tests according to one dimension would be a possible solution but this would restrict the test functionality extensively. For example agreeing on structuring tests only according to their related tenant would mean that tests for commonalities would have to be rewritten for every tenant.

To solve this problem an aspect-oriented approach is suitable, since it is a cross-cutting concern. This means meta data must be added to every test, assigning them to their respective test categories. The used test framework exploits this meta data and makes it possible to filter and run tests according to their assigned categories.

This approach is based on the concept of test categories from the Microsoft .NET framework and due to that this framework is used to explain this principle further. However, this concept is basically not programming language related and it is also included in other frameworks like JUnit for Java. [Jun]; [Micc]

In C# a `[TestCategory()]` attribute is used to add this kind of meta-information to methods or classes. The parameter of this attribute is the category, to which the test belongs. This principle is shown in listing 5.2, where a unit test is additionally assigned to the melanoma medical specialty and the concrete products for two hospitals.

```
1 [TestMethod]
2 [TestCategory("UnitTest"), TestCategory("Melanoma"),
     TestCategory("CorkHospital"), TestCategory("DublinHospital")]
3 public void GetMelanomaDataTest()
4 {
5 //Testimplementation
6 }
```

Listing 5.2: Implementation example of test categories

The integrated test framework can extract this meta data and let the product line owner or build- and deployment-scripts filter and combine these categories, when running the tests. As this is also already supported by the two mentioned technologies, this is not further designed in this thesis.

Since the number of tests can become large, the tests should be managed in a separate project, which references the application layer. Tests for the SAGE-CARE product line are therefore integrated in the *SPL Tests* project, as shown in figure 5.2.

5.10 Client Variability

When designing the product line, not only flexibility on the application and data layer has to be considered, but also on the client. Concepts have to be made to seamlessly integrate optional and differing requirements. The fundamental concept of excluding UI elements per authorization filter was already described in chapter 5.7.2. However, in this chapter two concrete concepts are presented which show the design of flexible and customized views for the EHRMS.

5.10.1 Patient Records with User Interface Components

The requirements for this concept were already fundamentally described in chapter 4.1.4. As stated, the existing user interfaces for adding and editing patients should be redesigned with the help of reusable user interface components (UIC) that are aligned to the actual patient data and group related data or functionality. Every MS and MIS in the product line is represented by at least one UIC, addressing the variation points from chapter 4.2.1 and 4.2.2. Figure 5.17 shows the example of a component, arranging the classification data for breast cancer.

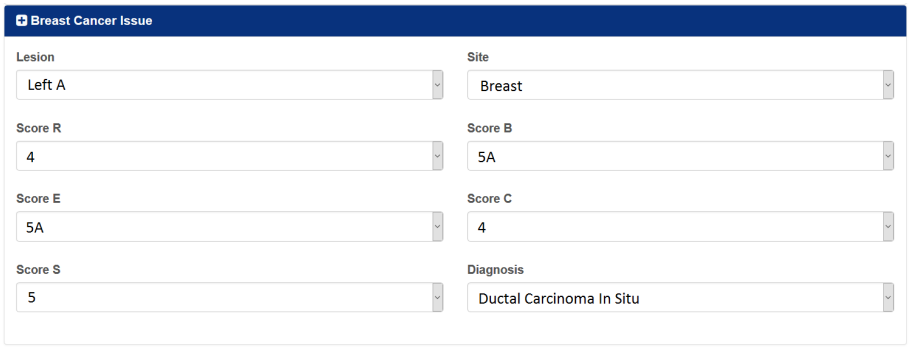


Figure 5.17: Example UIC for breast cancer

These components are aligned vertically, to visualize the paper form of a patient record. The UIC concept could be used for other use cases, like dynamically building up patient overviews for group meetings of physicians, but it is initially designed for adding and editing patients. The design provides the following functionalities for meeting the requirements:

1. UIC can be classified according to three dimensions: common/optional, static/dynamic and leaf/node. Common components are available for every product, while optional components are linked to a variant. Static types have a pre-defined place and are arranged at the top or the bottom of the page. While dynamic components adapt themselves to the actual patient data and can be added, moved and deleted by the user in a dynamic center section. The order of the dynamic UIC is part of the tenant configuration, e.g. always show MS first. Node components can be extended with other components while leafs are not extendable.

2. The concept for navigation on the EHR is integrated into the existing sidebar of the SAGE-CARE melanoma application EHR interface. Additionally, it

provides functionality for managing the UIC.

3. Tenant specific fields are possible with the help of the access management mechanisms from chapter 5.7.2.

An example of this concept is visualized in figure 5.18, where the EHR of two different patients and products are shown. *Product A* shows a patient with breast cancer. The *Breast Cancer* UIC contains other components, as this element is designed as a node. These elements can be dynamically added and removed from the node. Apart from that, only static UIC are shown, as dynamic components only show the actual patient data. *Product B* shows another patient with a melanoma and additionally with an enabled *Drug Informations* variant in the tenant configuration, which adds the component for showing drug interactions.

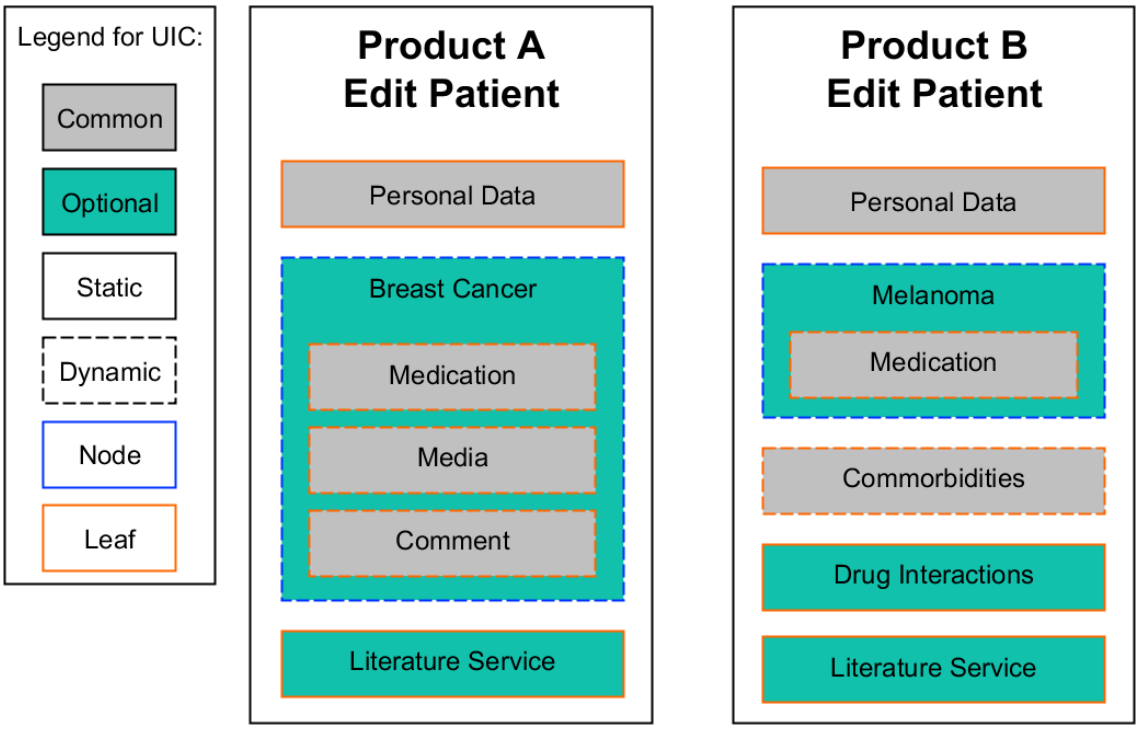


Figure 5.18: User interface components concept overview

The functionality of adding, removing and moving dynamic components and the navigation concept is grouped in a sidebar, instead of inserting them also in the EHR form, which can become big as more components are added. This separates the functionality for navigating and structuring the components (navigationbar) from viewing and editing their data (EHR). The navigationbar represents the UIC from the patient record as a label and consequently creates a summarized overview.

For adding new components, an add button is displayed in the navigation bar. When clicked, a dialog like illustrated in figure 6.5 is opened, showing the available dynamic UIC for the current patient. Moving components is realized via drag'n'drop, which is an easy and user friendly concept. Removing components is also possible via drag'n'drop. The user is able to drag a component label from the sidebar into a paste bin, which is a common concept and creates consistency with the drag'n'drop mechanism for moving components. Normally patient data is not deleted in an EHR. But as this concept gives the user the ability to build up patients freely, it has to be considered that users make mistakes, which have to be reversible. This case has not been further elaborated during this thesis as the main goal was to initially create a free and flexible concept. The same applies to constraints for allowing only specific leaf elements for certain nodes. However, two considerations were made:

- Give the user the freedom to remove every component he wants, since the data historization ensures that data can't be deleted completely anyway (see chapter 5.5). Certainly, the UI must then provide a recovery functionality.
- Only newly added UIC are deletable. Already saved components can just be modified or rearranged.

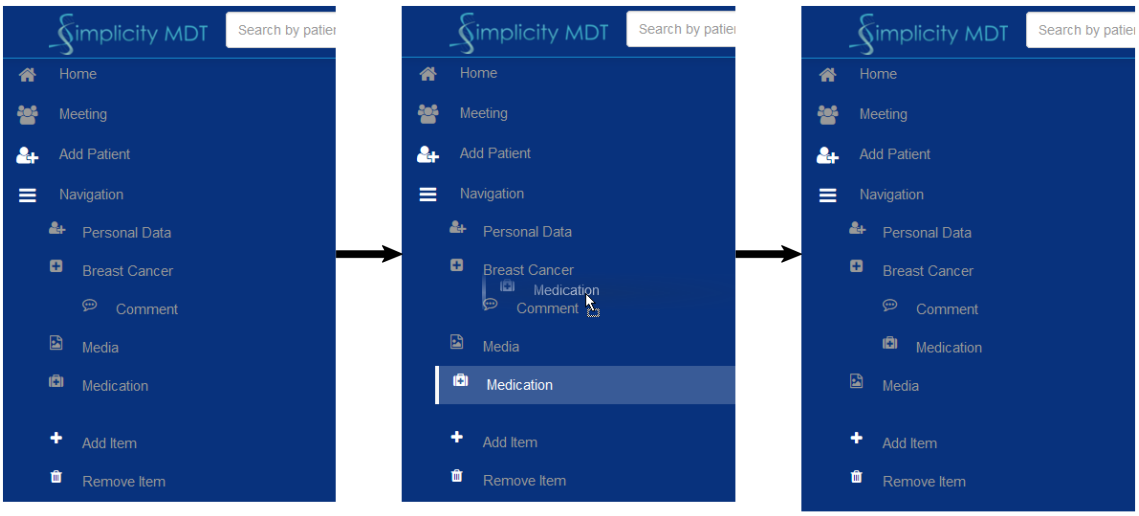


Figure 5.19: Prototype for the UIC navigationbar

A mapping of these changes to the server data model is already possible, since the HL7 RIM is used, where new relations can be build up dynamically with association objects. The user interaction model of the navigation bar and a drag'n'drop example is visualized in figure 5.19. Simply clicking on the component label of the sidebar

would focus the UIC on the EHR interface. A screenshot of the current prototype, showing the sidebar and the EHR interface with the help of an example patient is shown in appendix C.

The UIC framework is integrated completely in the client layer with the help of two main view components and one service, belonging to the corresponding sections shown in figure 5.2. The following units have to be considered

- `PatientRecordView`: Contains all independent UIC and the main patient record view. The main patient record view is concerned with building up the EHR form, with the help of UIC, according to the patient data and the tenant and user rights, as shown in figure 5.18. The UIC are independent view components, each with a view definition, e.g. an HTML file, and a data definition, e.g. a JSON file. The data-definition is necessary, as adding an UIC to the patient record view also means adding data to the patient data object.
- `PatientRecordNavigationBar`: The view for the described navigationbar for the `PatientRecordView`, as shown in figure 5.19. It implements the drag and drop and navigation functionality.
- `PatientRecordService`: Contains the patient data, communicates with the server and exports an interface, that allows access to the data for the views. This structure is equal to the observer pattern, like described by [Gam+11, p. 293], and ensures that both views work with the same data. Additionally it provides the data in a usable format for the views, since the data from the application layer is not always appropriate for such a view, as shown in chapter 6.2.2.

For tenant specific fields, no new mechanisms are needed, as this was already covered by the fundamental mechanisms of the client authorization with the validation for tenant membership (see section 5.7.2). If tenants want a product specific field, it must be added by the product line owner in the spreadsheet for the code-generator, which enables the persistent storage of the data, and on the UIC component with the authorization filter. This is sufficient as it is not expected that tenants want a large variety of product dependent fields, since the UI itself already provides flexibility.

Querying the patient data on the application layer, is in the current system already possible with the `PatientController` class at the *REST WebAPI*. This controller exposes the patient resource for the client and is callable per HTTP. Momentarily

this controller returns the complete patient data to the client, including all issues. For the product line the URL is changed to `"/patients/{id}/{issue}"`. The placeholder `{id}` stands for the database id of the patient object, while the `{issue}` defines which specialty of the patient should be returned. As already stated in chapter 5.7.1, specialty dependent interfaces of the WebAPI must have a parameter which determines the concrete medical specialty. Just returning all diseases of a patient would not be sufficient anymore, since the access management must be able to check if the currently logged in user is allowed to manage patients with the corresponding disease.

5.10.2 Variable Attribute Fields

Variable attribute fields (VAF) extend the functionality of the previous designed UIC. Per UIC an optional amount of fields are declared via the tenant configuration. The user has the possibility to select the fields which should be shown on his interface. Selected fields are saved per user in the corresponding user configuration.

This concept is useful to meet the needs of specific users, as medical data has a comprehensive scope and multiple users from multiple departments have to work on the same system. For example, the clerical staff of a hospital is more concerned with administration related patient data and needs to add a health insurance policy number to the patient's personal data. Whereas a physician has no use of this attribute field, but rather wants to see the height and weight as personal data of the patient.

A prototype implementation of this concept is shown in figure 5.20. The top part of the picture shows a simplified example of a personal patient data UIC, where all optional fields are deselected. In the bottom right corner of the UIC a button gets inserted which indicates that the tenant enabled VAF for this UIC. The bottom section of the figure shows the dialog which is displayed when the button is clicked. In the dialog the user can then select the VAF, that are displayed on the UIC. In this particular example, the insurance policy number was selected and added to the UIC. Additionally, this setting is saved in the user configuration and loaded whenever this user opens a new EHR.

Three components have to be designed for the VAF:

- `VAFButton`: Encapsulates the button and its corresponding dialog, like shown

in figure 5.20. This creates the possibility to reuse the button on every UIC. This component is integrated in the *View* section of the *Client*.

- `VAFDirective`: The `VAFDirective` is responsible for annotating fields on the UIC. The annotation links the field to a tenant. Since this annotation is only providing functionality, it is integrated into the *Service* section of the *Client*.
- `VAFService`: Is responsible for providing the needed data for the `VAFButton`, enabling and disabling fields and querying the user configuration. When a UIC is instantiated, the `VAFDirective` sends its annotated optional fields with their corresponding tenants to the `VAFService`. Having the optional fields, the service queries the user configuration to declare fields as selected or deselected. The optional fields for the current tenant and user can then be queried by the `VAFButton` and shown at the selection dialog. If a field is selected at the `VAFButton` dialog, the `VAFService` is queried to enable the field. It is integrated into the *Service* section of the *Client*.

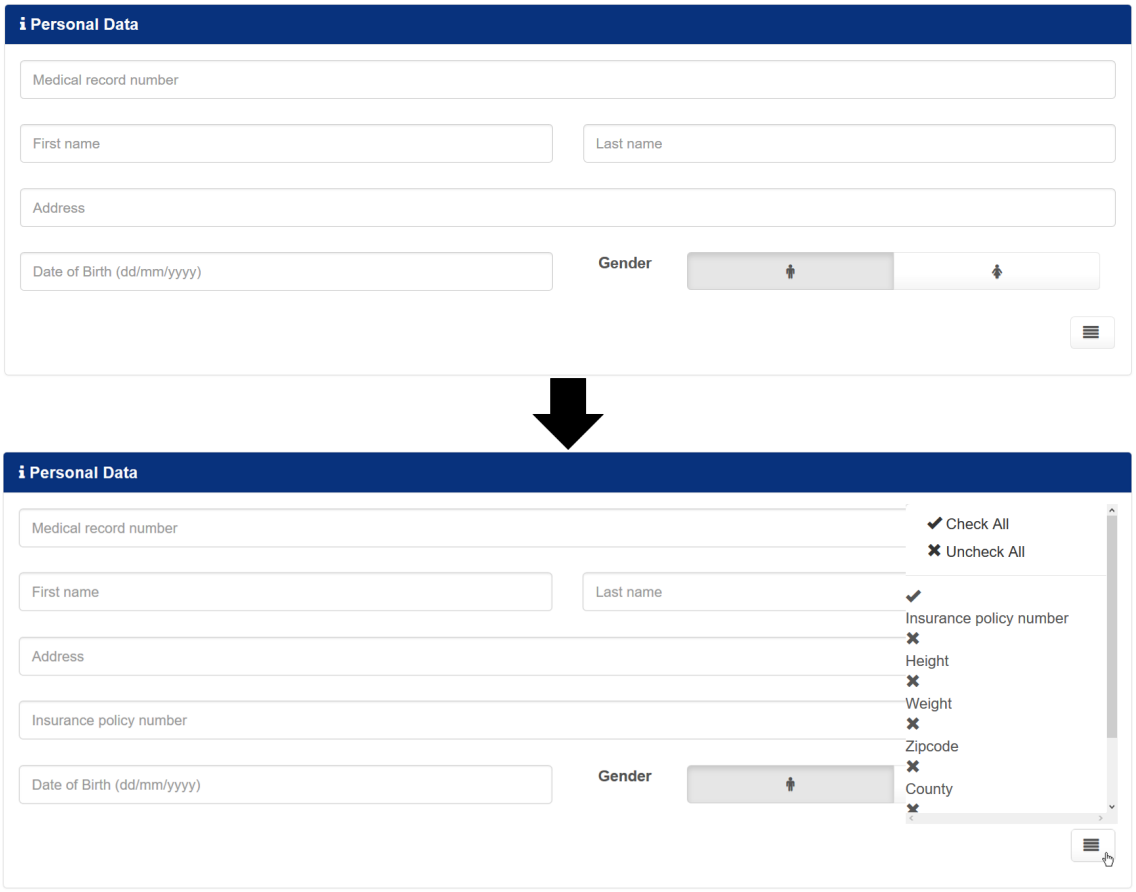


Figure 5.20: Prototype implementation for the VAF

Chapter 6

Domain Realization

Only selected parts of the concept from chapter 5 were implemented during this thesis. The implementation is using a fork of the existing SAGE-CARE melanoma application and enhances it towards a SPL. The client is implemented as a web application with the technologies HTML, CSS and JavaScript. Additionally the frameworks AngularJS and Bootstrap are widely used. At the application layer Microsoft .NET technologies and the programming language C# are used. Communication between *EHR Client* and *REST WebAPI* is realized via the HTTP protocol and the Microsoft ASP.NET WebAPI, which results in a RESTful API design and JSON as data exchange format. The used object-relational mapper (ORM) is Entity Framework (EF), which allows the easy utilization of a Microsoft SQL Server (MSSQL) as data layer. The shown implementations in this chapter were done without tenancy awareness and the configuration hierarchy as this is currently not implemented in the SAGE-CARE application.

6.1 Generic Data Model

6.1.1 Application Layer and Database Implementation

As described in section 5.3 and shown in figure 5.4, a slightly modified but still compatible version of the HL7 RIM is used as the data model. The implementation is based on the approach from [HW15] for managing EHR data in a flexible yet efficient way and it is realized with the Code First approach of EF, which allows to describe a data model by using C# classes and predefined conventions. Created entity classes

are automatically mapped to the object-relational MSSQL database. The `Act` class is shown in listing 6.1 as an implementation example of the HL7 RIM with EF. It also already features the `ActRelationshipList` and `ParticipationList` classes, which are further described in chapter 6.1.2. Nevertheless, they basically contain a list of `ActRelationship` and `Participation` objects.

```
public partial class Act : AbstractEntity
{
   [InverseProperty("Source")]
   public ActRelationshipList OutboundRelationship {get; set;}

   [InverseProperty("Target")]
   public ActRelationshipList InboundRelationship {get; set;}

   public ParticipationList Participation {get; set;}
}
```

Listing 6.1: Act Class

The corresponding association class for connecting two `Act` classes is shown in listing 6.2. The source code of the other HL7 RIM base and association classes are not further shown in this thesis, since there are no differences regarding the implementation.

```
public partial class ActRelationship : AbstractEntity
{
   public ActType SourceType { get; set; }
   public ActType TargetType { get; set; }
   public Act Source { get; set; }
   public Act Target { get; set; }
}
```

Listing 6.2: ActRelationship Class

The relationship type, introduced in chapter 5.3 and called `ActType` in the listing, is realized as an enumeration to ensure consistent and readable variable names for the whole application. An enumeration was created for every base class and was consequently called `ActType`, `RoleType` and `EntityType`.

Relationships to other entity classes must be realized through public properties, otherwise EF will not map them to the database. On the relational database, these relationships are automatically realized with foreign keys. A 1:n or n:m relationship must be realized through public properties which implement the `ICollection<T>` interface, which is used to define generic collections.

Another important information for EF in this particular example is the pointer to the inverse property, realized with the `[InverseProperty("")]` attribute. This is mandatory because of the multiple relationships between Act and ActRelationship, since this is an association class with one incoming and one outgoing relationship. Otherwise EF would not be able to e.g. assign the `OutboundRelationship` property to the `Source` property.

There are three strategies for mapping classes to relational databases: Table-Per-Hierarchy, Table-Per-Type and Table-Per-Concrete Type. Table-Per-Hierarchy uses one database table for a whole inheritance chain, for example a Role table. Inheriting classes are also saved within this table, instead of an own table for e.g. a Patient. An additional column is created per table, called the discriminator, which saves the concrete type of a table entry. Since all other entities in the information system will inherit from the HL7 RIM base classes, the Table-Per-Hierarchy strategy will be used for efficient access to the inheriting classes without creating expensive joins. [Kan15]; [HW15]

The result of this strategy and the automated mapping of relations via foreign keys (*Source_Id* and *Target_Id*) are visualized as an example table for Act and ActRelationship in figure 6.1. The example shows a mapping of a `BreastCancerIssue` and a `Medication` object with an object of the appropriate association class.

Act Table

Id	Discriminator	ScoreB	Name	...
1	BreastCancerIssue	5A	null	...
2	Medication	null	Rituximab	...
...	...	...	...	...

ActRelationship Table

Id	SourceType	TargetType	Source_Id	Target_Id
1	BREASTCANCERISSUE	MEDICATIONS	1	2
...	...	...	...	...

Figure 6.1: Relational Database Tables for Act and ActRelationship

6.1.2 Efficient Getter and Setter Methods for the Server

When it comes to the implementation, the generic and flexible approach of this data model shows some drawbacks, which are illustrated in listing 6.3. This example shows the difference between the access of related objects in a data model with direct relationships and the SPL data model, with the help of the 1:n relationship between a patient and its corresponding medications. The third way shows the usage of the already introduced getter and setter methods, designed in chapter 5.3.

```
1  ...
2  //Accessing the medications via a property
3  var medicationsObj = patient.Medications;
4
5  //Accessing the medications via association objects
6  var medications = patient.Participation.Where(p => p.ActType ==
       ActType.MEDICATIONS).Select(p => p.Act).OfType<Medication>();
7
8  //Accessing the medications via the new getter and setter methods
9  var medications = patient.GetAll<Medication>(ActType.Medications);
10 ...
```

Listing 6.3: Resulting Navigation Method of HL7 RIM

The example indicates the following disadvantages of the implementation with association objects:

- Inconvenient: Developers are used to navigate to related objects via properties or via getter and setter methods. With association objects they always have to filter the list for the right relationship type, then select the the right navigation property and at last ensure type safety.
- Inefficient: To get access to a corresponding object, one always has to iterate over the list of association objects.
- Error-prone: The statement is clearly oversized for a simple navigation, involves several method invocations and checks for null pointers should normally be included for a safe access.

Hence, the proposed efficient getter and setter methods are implemented, hiding the complex association objects, creating a convenient and accustomed mechanism for the developer and taking care of exception handling.

As already mentioned, simply iterating over all existing association objects is inefficient, as the number of association objects increases. Other classes than a simple list implementation must be used by the methods. Instead of mapping the 1:n relationships in the HL7 RIM data model with a default list implementation, a `BindingList`[1] is used. It provides a generic list implementation, which can be used by EF because it implements the `ICollection<T>` interface, with the addition of supporting data-binding mechanisms. These mechanisms are used for binding the data to a `MultiValueDictionary`[2], which allows efficient access to the values in the list. In contrast to a normal dictionary implementation, the `MultiValueDictionary` assigns multiple values to one unique key, which is therefore also capable of mapping the 1:n relationship. The relationship type, which was introduced in chapter 5.3 and represents the variable name in a concrete relationship, is used as a key, while the values are references to the corresponding objects in the list. This creates the possibility to navigate efficiently to related objects by there variable name, without taking care of the association objects.

The interaction of these two classes is shown in listing 6.4, using the example of a `BindingList` implementation for the `Participation` class. As the implementations for the other association objects are analogical, they are not further described in this thesis. For realizing the data-binding between the list and the dictionary, the `ListChangedEvent` method is called whenever changes occur, which is one of the provided mechanisms. The `ListChangedEventArgs e` parameter delivers informations about the change type, e.g. if an item was added or modified, and provides a reference to the changed object. Nevertheless the implementation of this method must take care of updating the dictionaries with the corresponding Roles and Acts.

```
1 public class ParticipationList : BindingList<Participation>
2 {
3    public MultiValueDictionary<RoleType, Role> RolesDict { get;} = new
         MultiValueDictionary<RoleType, Role>();
4    public MultiValueDictionary<ActType, Act> ActsDict { get;} = new
         MultiValueDictionary<ActType, Act>();
5
6    private void ListChangedEvent(object sender, ListChangedEventArgs e)
7    {
8       switch (e.ListChangedType)
9       {
```

[1] https://msdn.microsoft.com/en-gb/library/ms132679(v=vs.110).aspx
[2] https://www.nuget.org/packages/Microsoft.Experimental.Collections

```
10          case ListChangedType.ItemAdded:
11            //Add item to dictionary
12          break;
13          case ListChangedType.ItemDeleted:
14            //Delete item from dictionary
15          break;
16       }
17    }
18 }
```

Listing 6.4: Implementation of the ParticipationList class

As already stated in the design chapter, the getter and setter methods are implemented in the HL7 base classes, since all other classes inherit from them and are therefore capable of reusing the methods. How the methods interact with the `BindingList` and the `MultiValueDictionary` is shown in listing 6.5. This listing shows the example of the `GetAll` method in the `Act` class, which returns related Roles for an Act by its associated `RoleType`. The usage of this method is illustrated in the last statement of listing 6.3. The implementation of the other proposed methods for the `Act` class are shown in appendix D. Regarding the implementation for the `Role` and `Entitiy` class, there is no distinctive feature. Consequently they are not further shown in this thesis.

```
1  public partial class Act : AbstractEntity
2  {
3     public ParticipationList Participation {get; set;}
4
5     public IList<T> GetAll<T>(RoleType relationshipType) where T : Role
6     {
7        List<T> result = null;
8
9        if (Participation != null &&
             Participation.RolesDict.ContainsKey(relationshipType))
10       {
11          var roles = Participation.RolesDict[relationshipType];
12          result = roles.OfType<T>().ToList();
13       }
14
15       return (result!=null && result.Count>0 ? result : null);
16    }
17 }
```

Listing 6.5: Implementation of Getter and Setter Methods

The newly introduced methods indicate the following advantages and solve the problems described at the beginning of this chapter:

- Convenient: Even though C# developers are used to properties instead of methods for object access, the usage of getter and setter methods is still a common way of accessing and modifying objects.
- Efficient: The underlying dictionary structure allows efficient access via the relationshiptype
- Centralized Exception Handling: The getter and setter methods take care of exception handling instead of handing the responsibility to the developer at every iteration over the association objects.
- Reduced complexity: While the advantages of the data model with decoupled associations can still be exploited, the developer does not have to work directly with it.

6.1.3 Client Implementation

As already stated, the client mainly uses the web technologies HTML, CSS and Javascript and is loosely coupled to the server. Due to that, the getter and setter methods from the server are not reusable on the client. When requesting the server, the data is sent in the HL7 RIM structure. It is transformed by the Json.NET[3] framework to the JavaScript Object Notation (JSON) for data transfer to the client. This causes the same disadvantages to apply as in chapter 6.1.2. An example of a server response is visualized on the left side of figure 6.3. Just using the data in the HL7 RIM structure is again not a suitable approach. Because additionally it has to be considered, that the client heavily relies on two way data binding between the views and their corresponding models because of AngularJS, which is why an efficient access to related objects is mandatory. As a result an approach was developed, which is specifically customized for the requirements of a dynamically typed script language.

This approach introduces a new client services, called `HL7RelationshipService` and realized as angular factory[4], which maps the server data model to the client

[3] `http://www.newtonsoft.com/json`
[4] `https://docs.angularjs.org/guide/providers`

data model, such as e.g. Microsoft Entity Framework takes care of mapping data between the business logic and the database. The service processes incoming data into a usable format for the client and translates it back into the server data model when it is sent back. Main functionality is the deletion of association objects and the adding of related objects as direct reference. This is simply possible because JavaScript is dynamically typed and therefore the requirement of building up relationships dynamically between objects is already fulfilled by technology and doesn't need a generic approach like on statically typed languages. Hence, three views on the data exist in the system: a relational view for the database, an object-oriented view for the statically typed application kernel and an object-oriented view without association objects for the dynamically typed client. An example for all three views is shown in figure 6.2.

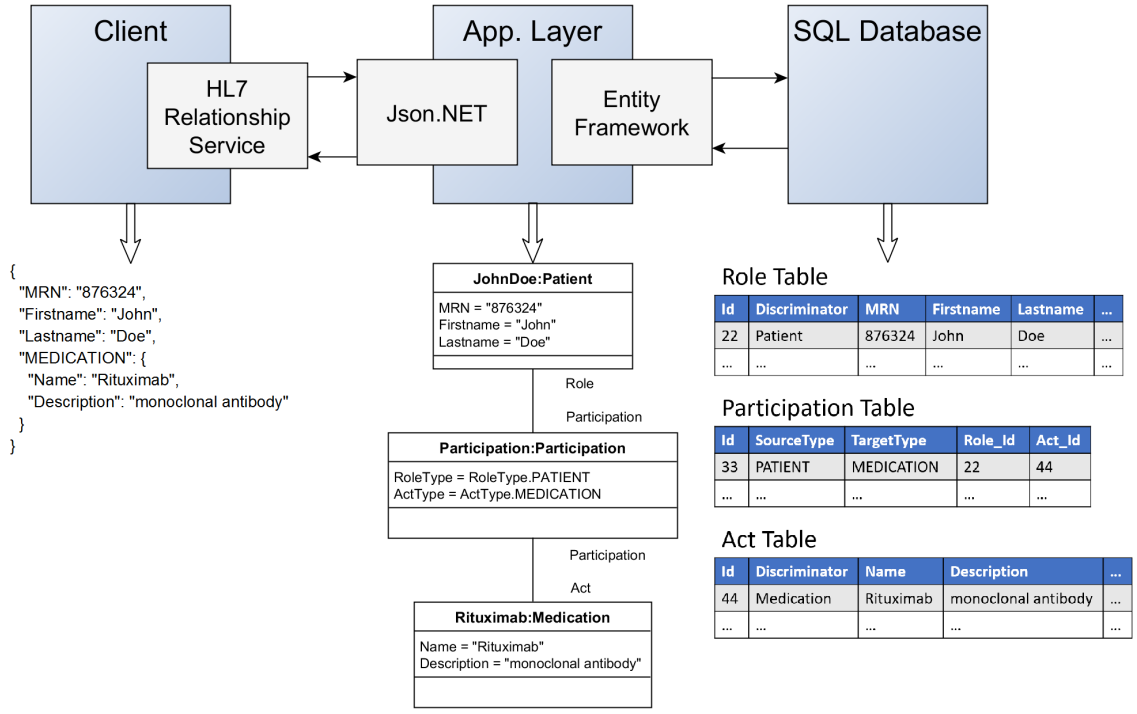


Figure 6.2: Three Different Views on the Data

This approach makes the introduction for getter and setter methods on the client dispensable and creates the possibility for convenient, efficient and yet flexible client development.

The `HL7RelationshipService` contains two important public methods for the data mapping, `removeHl7Relations(data)` and `insertHl7Relations(data)`. The first method is called to modify the incoming HL7 RIM data. It iterates recursively

over the variables of the `data` parameter and removes HL7 association objects. The detection of HL7 association objects is done via their `$type` variable, which is created automatically from ASP .NET WebAPI and contains the server data type as value. The relationshiptypes of the association object get mapped to variable names, as described in chapter 5.3. Additionally a new variable is added, called `$sourceAssociation`. This is necessary to flag this object as a server object which was initially connected via an association object. A transformation example is shown in figure 6.3.

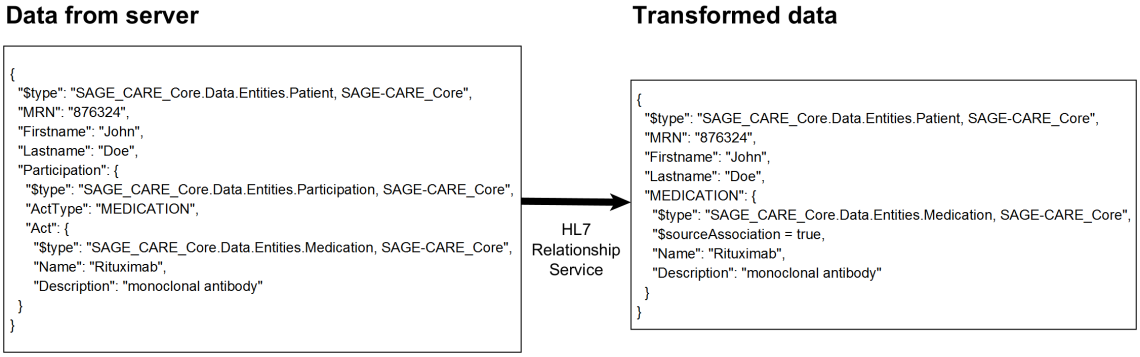


Figure 6.3: Hl7RelationshipService data mapping

When sending data back to the server, the `insertHl7Relations(data)` is called once which reverses the changes from `removeHl7Relations(data)`. It iterates again recursively over the `data` parameter and whenever an object contains the `$sourceAssociation` variable, it gets connected again via an association object to its parent object. Consequently, after this method got invoked the data is in the HL7 RIM structure again, which is processable for the server.

6.2 Designing Patient Records with User Interface Components

The concept of using independent UIC for designing patient records was already introduced in chapter 5.10.1. It was also already stated, that the user interacts with two interfaces: a navigationbar for navigating and structuring the UIC and the patient record for viewing and editing the patient data. The design chapter identified three components: the patient record service, the patient record view and the patient record navigationbar. These units were prototypically implemented

and are described in the following chapters. During the implementation phase, the tenant aware access management was not yet designed and hence is not considered in these chapters. A consequence is that a user currently sees every possible variant, every defined field and the complete patient data is retrieved from the server in the prototype.

6.2.1 Patient Record Service

The patient record view service is responsible for being the subject in the observer pattern, containing data processing methods that are needed for the realization of this concept and configuring the UIC. It is implemented in the service section of the client layer, see figure 5.2, as AngularJS factory in the file `patientRecordService.js`.

In the current implementation the service queries the the patient object with all its corresponding information from the application layer and provides it for the observers. As described in chapter 5.3, the patient is one of the main objects of the system, beside the issue classes. The data processing is needed to meet the requirements of dynamic UIC that get arranged according to the patient data. Since dynamic UIC can be leafs and nodes, the patient data, which is just a data object with all corresponding additional objects like diseases or medications, needs to be processed in a structure that is suitable for such a tree structure.

The basis for the service and the processing is the UIC configuration file, which is implemented in a JSON file since this is natively supported format in JavaScript. This file defines which UIC are implemented and configures all needed values for the views. An extract of the currently implemented `UICConfiguration.json`, showing the configuration for two components is shown in listing 6.6.

```
1 [
2 {
3    "serverDataType": "SAGE_CARE_Core.Data.Entities.BreastCancerIssue,
        SAGE-CARE_Core",
4    "labelValue": "Breast Cancer",
5    "treeType": "NODE",
6    "htmlPath": "breastcancerissue/breastcancerissue.html",
7    "jsonPath": "breastcancerissue/breastcancerissue.json",
8    "propertyName": "BREASTCANCERISSUE",
9    "icon": "fa-plus-square"
```

```
10 },
11 {
12    "serverDataType": "SAGE_CARE_Core.Data.Entities.Medication,
          SAGE-CARE_Core",
13    "labelValue": "Medication",
14    "treeType": "LEAF",
15    "htmlPath": "medication/medication.html",
16    "jsonPath": "medication/medication.json",
17    "propertyName": "MEDICATION",
18    "icon": "fa-medkit"
19 }
20 ]
```

Listing 6.6: Configuration of the UIC

Every UIC is represented as JSON object in an array, with the following properties:

- `serverDataType`: The corresponding server data type, which contains the data for this UIC.
- `labelValue`: Value for the navigationbar label and the heading for the UIC on the patient record.
- `treeType`: Determines if the UIC is a leaf or a node element.
- `htmlPath`: Path to the html file which defines the UIC view.
- `jsonPath`: Path to the html file which defines the UIC data model.
- `propertyName`: Name of the variable in which the corresponding data was saved.
- `icon`: Unique identifier for the icon that is presented for the UIC on the user interface. Currently the implementation uses the Font-Awesome icon collection[5].

The dynamic UIC must be sortable for the tenant, mappable to a tree structure to represent nodes and leaf components and rearrangeable by the user. Consequently a normal object received by the server, such as the patient object shown in figure 6.3, can not be used as object for data-binding. On the one hand the variables of JavaScript objects are not sortable. On the other hand for realizing the dynamic view with AngularJS, the ngRepeat[6] directive and the angular-drag-and-drop-lists[7]

[5] `https://github.com/FortAwesome/Font-Awesome`
[6] `https://docs.angularjs.org/api/ng/directive/ngRepeat`
[7] `https://marceljuenemann.github.io/angular-drag-and-drop-lists/demo/#/nested`

framework are needed, which use an array as input parameter for instantiating an HTML template once per item.

The method `createPatientTreeModel(patient)` of this service is responsible for the data processing, producing a suitable data model for the view components. This data model is from now on called tree data model and is basically a representation of the relevant variables from the patient data object as an array, enriched with information of the UIC configuration. The method takes the complete patient data object as parameter and recursively iterates over all its variables, searching for supported UIC datatypes in the `$type` variable of related objects. This means that UIC are bound to concrete server data types in this implementation, as already shown in the service configuration (listing 6.6). If an object with a supported data type is found, it is added to the array of the tree data model including its corresponding settings of the UIC configuration. These settings are added to realize a generic and efficient way to include the UIC on the interface.

The resulting data model fulfills the requirements for the used technologies and represents the arrangement of the UIC on the user interface. The following sections 6.2.2 and 6.2.3 give further explanations on why this data processing is needed and how the data model is used in the views.

A simplified example for the data processing is shown in figure 6.4 for a patient with breast cancer and one medication. The method creates an object with an `items` variable, which contains all variables of the main patient object that are supported for an UIC including its settings from the configuration. In the example it is the `BreastCancerIssue` object from the `Patient`. The method also recursively searches in objects with a supported datatype for more supported objects. This means that the method iterates over the variables of the `BreastCancerIssue` object and adds the `Medication` and `Comment` object to a new `items` array.

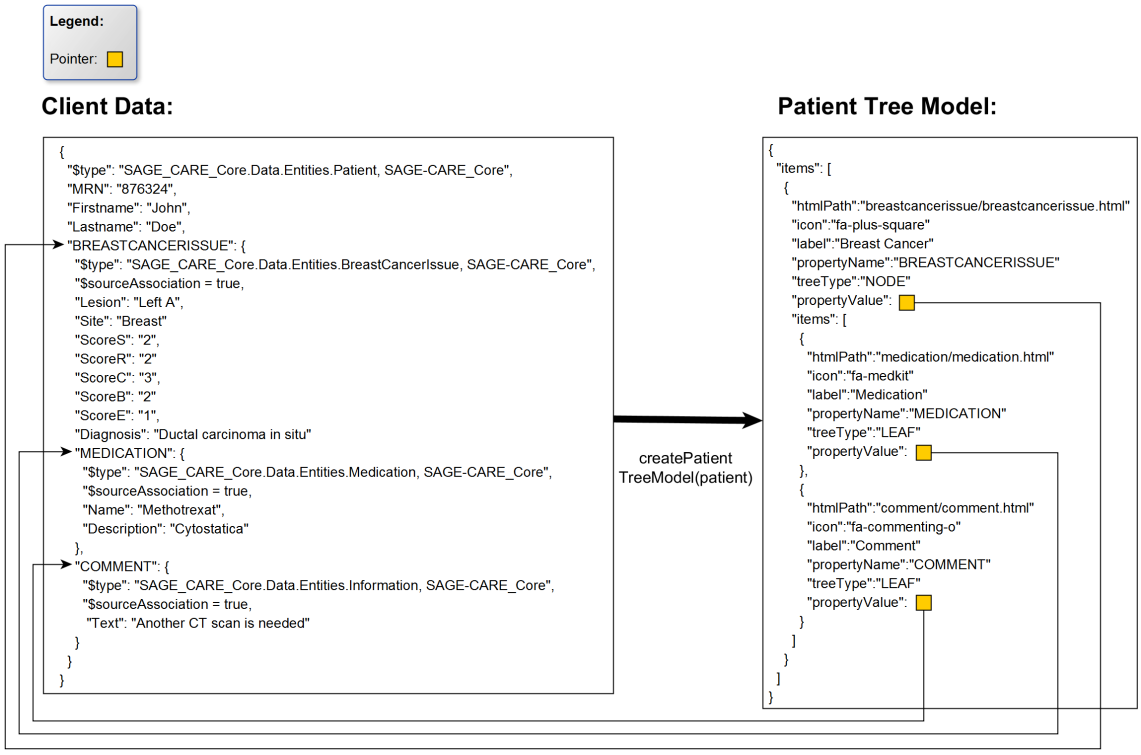


Figure 6.4: Mapping between a patient object and its tree data model

6.2.2 Patient Record View Component

As stated previously the patient record component shows the patient data with the help of UIC and is integrated into the view section of the client, see figure 5.2. The patient record component consists out of three main files and multiple sub folders:

- `patientRecordDirective.js`: AngularJS directive[8] for reuse purposes. Enables to reuse the patient record component in every HTML file, just by typing `<patient-record-directive> </patient-record-directive>`. This is important for reusing the component on the interfaces for adding and editing a patient, which slightly differ. As this is a standard AngularJS approach, it is not further explained in this thesis.
- `patientRecordView.html`: HTML file for recursively loading the UIC.
- `patientRecordController.js`: Client logic for the patient record user interface. Since most of the view logic is covered by the patient record service and the `patientRecordView.html`, this controller is not further described.

[8] `https://docs.angularjs.org/guide/directive`

- UIC subfolders: Every UIC gets its own folder, containing an HTML file for its view structure and a JSON file for its data structure, as already shown in the configuration example in listing 6.6.

The `patientRecordView.html` file makes use of the ngRepeat directive and the previously introduced tree data model from the `patientRecordViewService`, as shown in listing 6.7. An iteration over the patient data object with ngRepeat would technically not be possible, as it needs an array. At first the static UIC get included, represented by a personal data UIC in the example. This is done with the help of the path to its HTML file. Afterwards the ngRepeat iterates over the root `items` array in the tree data model and includes the script with the id `patientRecordViewRecursion`. This script loads the HTML with the path that is saved in the `htmlPath` variable. Finally, if the object has child objects as well in his own `items` variable, the script gets included recursively. With the data from figure 6.4 the HTML code would generate a patient record view interface like shown in appendix C.

```
1  <div class="container">
2     <ul>
3        <uib-accordion close-others="false">
4        <!-- Static UIC -->
5        <li>
6           <div id="personaldata"
                ng-include="'app/components/patientRecordView/UIC/personaldata
                /personaldata.html'"> </div>
7        </li>
8        <!-- Dynamic UIC -->
9        <li ng-repeat="childItem in vm.treeDataModel.items"
             ng-include="'patientRecordViewRecursion'"></li>
10       </uib-accordion>
11    </ul>
12 </div>
13
14 <script type="text/ng-template" id="patientRecordViewRecursion">
15    <uib-accordion-group is-open="true"
          ng-attr-id="{{childItem.propertyName +
          childItem.propertyValue.Id}}">
16       <div ng-include="'app/components/patientRecordView/UIC/' +
             childItem.htmlPath"></div>
17       <ul ng-if="childItem.items.length > 0">
18          <uib-accordion close-others="false">
19             <li ng-repeat="childItem in childItem.items"
                   ng-include="'categoryTree'"></li>
```

```
20          </uib-accordion>
21        </ul>
22    </uib-accordion-group>
23 </script>
```

Listing 6.7: Code of patientRecordView.html

6.2.3 Patient Record Navigationbar

The patient navigation and drag and drop functionality is currently integrated in the existing sidebar of the SAGE-CARE melanoma application. The assembly of UIC labels is done in a similar way as in the `patientRecordView.html` with the help of ngRepeat and the tree data model from the `patientRecordViewService`. Except that it doesn't include the UIC HTML saved in the `htmlPath`, but only shows the contents of the `labelValue` and `icon` property. Prototypes of the navigationbar were already shown in figure 5.19 and appendix C.

The drag and drop functionality is implemented with the help of the angular-drag-and-drop-lists framework, providing an angular directive for modifying ngRepeat lists with the HTML5 drag and drop API[9]. It instantiates all items from the tree data model as draggable elements. If UIC are declared as nodes with their `treeType` setting in the UIC configuration, the drag and drop framework declares them as container. This means that they are able to include further drag and drop elements.

As the navigationbar and the patient record view both use the same tree model data object from the patient record service, with the help of the observer pattern and angular data-binding, they are always synchronized. As a result, if the user moves a UIC label on the navigationbar, the patient record view gets updated automatically.

The current implementation only offers the ability to add new UIC to the patient via a button on the navigationbar, whereas the delete functionality is currently not designed (see chapter 5.10.1). The add button opens a new dialog with the help of the angular-dialog-service framework[10]. The dialog reads the `UICConfiguration.json` and offers the user a button per supported UIC, as shown in figure 6.5. If the user adds an element, an object gets created with the corresponding data model, deposited in the JSON file of the `jsonPath` setting. This object gets appended to the currently used patient data object, and as a result the tree data model and the

[9] `https://www.w3.org/TR/html/editing.html#drag-and-drop`
[10] `https://github.com/m-e-conroy/angular-dialog-service`

dependent views are updated with the new UIC.

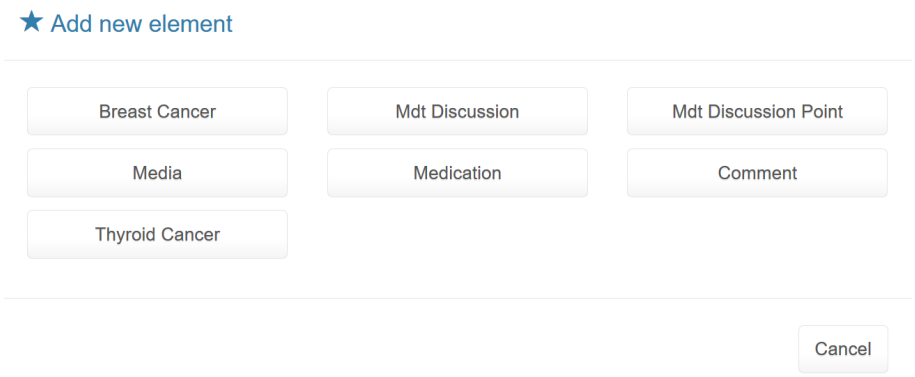


Figure 6.5: Adding new UICs to a patient record

Chapter 7

Evaluation

As this thesis creates the initial concept for the SAGE-CARE SPL, it is important to consider the defined requirements, but also evaluate with more universal requirements from the literature. Consequently two evaluations were made. Chapter 7.1 uses the Family Evaluation Framework (FEF) to determine if universal needs for product lines were addressed. Whereas chapter 7.2 evaluates the specific requirements for this SPL, at first defined in chapter 2.

7.1 Family Evaluation Framework

7.1.1 Overview

The FEF is a result of the European ESAPS and CAFÉ projects and aims to evaluate organizations according to their performance in SPLE. Academic institutions, as well as companies from different European countries, like Siemens, Bosch or Philips, worked together in these projects on multiple research topics concerning SPLs (see [van02] for further reading).

Evaluation within the framework is done according to four dimensions: business, architecture, process and organization. Each dimension is then divided into five levels with three to four evaluation aspects. The levels can be reached incrementally. That means if a company wants to reach a certain level, it has to satisfy all of the requirements from the lower levels. The result of the FEF is an evaluation profile, which declares a level for each evaluation aspect.

The dimensions are rated independently. As a result, a company can for instance reach level five in the architecture dimension, but level one in the organization dimension. Similar to other level based evaluation systems, like Capability Maturity Model Integration (CMMI), the FEF doesn't require specific practices to reach a level, but focuses on certain objectives that need to be achieved. [LSR07, pp. 79–82]

As this thesis is only concerned with the architecture of a SPL, the dimensions business, process and organization are left out, since the aspects couldn't be evaluated. However, the architecture dimension is evaluated with the help of the designed concept and prototype implementation, according to the asset reuse level, the reference architecture and the variability management. [LSR07, p. 86]

7.1.2 Execution

The execution of the evaluation is based on the performed example from [LSR07, pp. 100–104]. For each evaluation aspect the level definitions are quoted, followed by a level classification for the SAGE-CARE SPL.

Asset Reuse Level

This aspect intends to consider the usage extent of core assets in products. The levels are defined as follows:

- Level 1: There is no or only unsystematic reuse.
- Level 2: There is a common third-party infrastructure defined and in use. There is only ad hoc reuse, mainly based on the repository of the third-party products.
- Level 3: There is a common platform defined as a collection of common assets in a domain repository. Reuse is restricted to this platform, and it is restricted by architectural constraints.
- Level 4: There is systematic and managed reuse based on an asset repository, with explicit variability in the assets.
- Level 5: There is systematic reuse based on an asset repository, with explicit variability in the assets plus automated product derivation mechanisms.

As this product line is designed as DSPL, all products are based on the same configurable platform, reusing the same assets. The architecture and product derivation of the SAGE-CARE SPL heavily relies on reuse, since differing features are rarely needed and are currently all solved by configuration, e.g. variable attribute fields (see chapter 5.10.2). Variability is explicitly defined in the assets, e.g. in access management (see chapter 5.7.1) or multi language support (see chapter 5.8). Product derivation is done via configuration, as outlined in section 4.1.5. The FEF does not define the term "automated product derivation mechanisms" of level 5. Albeit, configuration of the SPL has to be done manually and must be aligned to the variability model, which is the reason that level 5 is not reached. Subsequently, the SPL reaches level 4 for the asset reuse level.

Reference Architecture

The purpose of this aspect is to ascertain how much the application architecture is determined by the reference architecture. Ensuing levels are stated:

- Level 1: There is no software product line architecture.
- Level 2: The software product line architecture is derived from the third-party infrastructure. It only enforces the use of this infrastructure.
- Level 3: It is in use for the applications. It contains rules and determines the use of the platform. This incorporates the common use of certain quality solutions as offered by the reference architecture.
- Level 4: There is an explicit reference architecture that determines where application architectures may vary. Many quality solutions are incorporated in the software product line architecture.
- Level 5: It determines the application architectures completely. There is automated configuration support to derive specific applications. Quality is supported through the managed use of specific variation points.

The reference architecture from chapter 5.2 defines the architecture for every concrete product of the SPL. The derivation via configuration is a fundamental principal of a DSPL and thus also of the SAGE-CARE SPL (see chapter 4.1.5). Yet again, there is neither an automated configuration support nor a self-service for customers designed, as the SPL is not yet planned for a large amount of customers and the

configuration effort per product is manageable. Therefore only the criteria for level 4 is reached.

Variability Management

With this point of view, the explicit use of variation points and variability mechanisms should be examined. The following levels are defined:

- Level 1: Variability is not managed.
- Level 2: Only variability offered by the third-party infrastructure is somewhat limited. The remainder of the variation is open to be determined by the application architecture.
- Level 3: The reference architecture determines which configurations of domain assets are allowed within applications. It determines explicit variation points, where application-specific variants may be bound.
- Level 4: The software product line architecture determines which configurations are allowed for application architectures. The reference architecture determines variation points and restricts the allowed variants for most of these variation points. It determines rules that application-specific variants have to obey.
- Level 5: It is fully integrated in the architecture. Variability is described in models or languages that are semantically and syntactically standardized within the organization. Variants are derived automatically.

The variability in the designed SPL is fully integrated into the domain architecture, as specific application architectures do not exist, see chapter 5.2. The domain architecture constraints the variants of concrete applications and determines the rules for the products. However, the OVM and textual descriptions were used in chapter 4. Consequently, an automatic derivation of variants is not possible and level 4 is reached for variability management.

7.1.3 Summary

As the SAGE-CARE SPL reaches level 4 for every evaluation aspect, the architecture dimension is also assessed with level 4. This level is rated as follows: “At this level,

the domain commonality and variability is captured and a reference architecture is specified for the complete software product line. Domain assets include support for deriving products. Variability management is explicitly addressed in the software product line architecture". [LSR07, p. 88]

According to the FEF, primarily the automation for deriving products is in need for improvement for this design. Although, as already stated, the framework doesn't give an explicit example or definition of the automation. It is assumed that a product derivation via a self-service for the customer can be seen as a fully automated derivation process, as the product line owner wouldn't have to interact in any way.

7.2 Requirements Evaluation

This chapter evaluates the outlined DSPL by comparing concept and prototype implementation with the requirements defined in Chapter 2.

Requirement 1: This requirement states that an architecture of a software product line for an EHRMS in cancer care shall be outlined. The proposed architecture in chapter 5.2 is structured in three layers, which are loosely coupled and can be distributed. The design considers the currently examined common and variable requirements. Concepts and prototypes could be allocated to the various layers and sections of the architecture. Currently it is also already possible to deploy a configured application to a local Microsoft IIS server and to the Microsoft Azure cloud. As the SPL is not designed for the mass market, a concept for extensive load balancing was not designed initially. However, requirement 1 is met with the architecture.

Requirements 2 and 3: The derivation of products via configuration and the multiple configuration levels were further described in chapter 4.1.5 and always considered in the multiple designs of chapter 5. The main variation points, medical specialties, medical information services and technical services, were also further described and considered. However, no concrete concept for the configuration was made during this thesis, even though the requirements were further examined and a basic outline was made. As a result, the requirements 2 and 3 are not met completely.

Requirement 4: The multi-tenancy data architecture from chapter 5.6 creates the foundation in the data layer. Regarding the application and the client layer, multi-tenant awareness is integrated into the access management as described in chapter 5.7. All things considered, the concepts describe how the interface is customized according to the tenant of the user, how not permitted access to the application layer is prevented and how data is seperated between tenants. Users are always assigned to a specific tenant and cannot see or access data and features of other tenants. Consequently requirement 4 is seen as fulfilled.

Requirement 5: With the presented access management mechanisms in chapter 5.7, fundamental designs for authentication and authorization were made. Users have their own account, linked to a tenant, multiple roles and multiple rights. The access management provides role and tenant based mechanisms for the authorization on the WebAPI and the client. For this reason, fundamental security mechanisms, described in requirement 5 are met.

Requirement 6: With the introduced historization component it is possible to track all changes, i.e. create, update and delete operations, that were made by users to the SQL database. Every change can be traced back to a user and its corresponding tenant. The ability to track delete operations and historize the values of the objects, ensures that data can't be deleted completely and is recoverable if needed. Due to this, the requirement is fulfilled with the presented concept in chapter 5.5.

Requirement 7: As stated in the requirements, the main problem for product line testing is the handling of multiple variants, products and additionally linking them to testing levels. The introduced concept of test categories allows the developer to classify tests simultaneously to multiple categories of the three main dimensions. There are also no restrictions, concerning the categories or dimensions, which makes the concept extensible. The used Microsoft .NET testing framework from the existing implementation exploits the categories and permits the execution of all tests from specific categories. However, as stated in the design chapter, this functionality is also integrated in other testing frameworks. As the DSPL integrates all products into the domain architecture, the tests are implemented in an isolated project. Thus, separating the concerns and ensuring a better manageability of the tests. All things

considered, the presented concept conforms the requirements.

Requirement 8: As already evaluated by [HW15], the code generation framework enables that new entities and attributes may be added easily by editing the introduced spreadsheet. Additionally the generation ensures up-to-date classes and the ORM assures up-to-date database tables. The newly implemented association classes also enable that relationships between objects can be build up and changed at run-time. This creates possibilities like in dynamically typed programming languages while still ensuring type-safety. Consequently, the data model can bee seen as flexible and extensible.

Getter and setter methods are an established concept and create an accustomed way for developers to work with the data model without using the association classes. Furthermore, they are combined with a dictionary structure to ensure efficiency. Tests of all methods revealed, that the response time doesn't rise above 1 ms no matter how large the association lists are. This is due to the fact, that accessing dictionaries can be done with $\mathcal{O}(1)$ complexity. The only method depending on the size of the list is the second call of the set method with the same relationship type, as it first has to delete the old entry in the list to ensure that the variable is only added once to the association list. However, the first response times that were greater than 1 ms occurred at 10000 and more items in the corresponding list, which is already an unlikely use case scenario. As a result, also convenience and efficiency is reached with the concept and the requirement is met.

Requirement 9: The introduced patient records with UIC extend the flexibility of the old static interface, as it allows an adaption of the interface to the actual patient data. Additionally, new UIC can be added and rearranged by the user to give them the possibility to create individual patient records and meet the demands of the tenant. The components can be extended with tenant dependent fields and even meet the needs of users with the introduced VAF. They are reusable for multiple interfaces by including them, as they are encapsulated in their own files. Currently they are already reused for the interfaces for adding and editing patients. Due to this, the initially defined requirement is fulfilled, as they are flexible, reusable and arrangeable by the user.

Chapter 8

Related Work

As already mentioned in chapter 3.1.5 and 3.2 neither the combination of DSPL with reconfigurable multi-tenant aware SaaS applications, nor EHR applications are new research fields. However, to the best of my knowledge, no publication describes the combination of an EHRMS with a DSPL. Most of the examined publications were either concerned with designing a single EHR application or about security of EHRs in the Cloud.

The paper from [BM13] outlines a cloud-based approach for the design of an interoperable EHRMS. Similar to this thesis they describe the architecture for an EHRMS, but with the explicit focus on semantic interoperability, data integration, and security. Albeit, the paper focuses on single-system engineering instead of creating a product line.

In the publications of [LSW10], [CA13] and [KS11], security structures are examined for patient records on cloud systems. Especially the first publication also gives insight on client platform security, by virtually dividing an application in so called trusted virtual domains. This means that a client platform provides different functionalities, which are virtually separated to prevent not permitted access.

Another publication from [Kuo11] summarizes the general opportunities and challenges of cloud computing to improve health care services, to benefit health care research and to change the face of health information technology. All in all, the author discusses the aspects shown in table 8.1.

Aspects:	Opportunities:	Challenges:
Management:	Lower cost of new IT infrastructure Computing resources available on demand Payment of use on a short-term basis as needed	Lack of trust by health care professionals Organizational inertia Loss of governance Uncertain provider's compliance
Technology:	Reduction of IT maintenance burdens Scalability and flexibility of infrastructure Advantage for green computing	Resource exhaustion issues Unpredictable performance Data lock-in Data transfer bottlenecks Bugs in large-scale distributed cloud systems
Security:	More resources available for data protection Replication of data in multiple locations increasing data security Dynamically scaled defensive resources strengthening resilience	Separation failure Public management interface issues Poor encryption key management Privilege abuse
Legal:	Provider's commitments to protect customer's data and privacy Development of guidelines and technologies to enable the construction of trusted platforms by not-for-profit organizations Fostering of regulations by government for data and privacy protection	Data jurisdiction issues Privacy issues

Table 8.1: Opportunities and challenges of cloud computing to improve health care services according to [Kuo11]

The SPLiCE (Software Product Line for healthCarE) project aims to propose a model-driven engineering method for healthcare information systems. A SPL shall thereby be created, which integrates clinical data models, described according to the "openEHR" specifications, and architecture models, specified in the "Acme" architecture description language. [Gom+12]

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this thesis, a a multi-tenant aware dynamic software product line for an electronic health record management system in cancer care was proposed, aiming to fulfill the needs for the product line owner, the customers and the users. Focus of this thesis was on identifying the common and variable requirements, designing the fundamental architecture and examining selected parts of the system. While other parts that were already described in publications, were left out. The structure of this thesis was aligned on the domain engineering process for SPLs, described in chapter 3.1.2. Considering the basic input for the product line, three existing applications were regarded. Two of them are currently in use at hospitals, developed and marketed by NSilico. The third, which created the architectural foundation for the SPL, was developed during the SAGE-CARE project.

With the help of this input, the common and variable requirements were examined during the domain requirements engineering process, described in chapter 4. It was ascertained, that the variation points currently can be divided into three main categories: technical services, medical specialties and medical information services. Another important part was the common requirement for a configuration hierarchy, that considers the product line owner, the tenant and the user. Chapter 5 dealt with the design of the domain. The architecture itself was designed to be capable of running on a cloud system, deriving instances per configuration and meeting the common and variable requirements. Focus of the design chapter was also on integrating the mandatory variants into the architecture. Although the access man-

agement and the configuration management both need more research, since these are comprehensive topics and hence they were not examined in detail. Designs for MISs were left out since they were already described by [Ide16]. However, concepts for showing the MSs and MISs to the user were addressed in chapter 5.10. The prototype implementation for two of the designs were shown in chapter 6. Implementing the HL7 RIM data model and combining it with efficient getter and setter methods helped to improve the flexibility of the data layer and created a foundation for persisting comprehensive medical data. Realizing a first prototype of the UIC provided variability for the client, creating a clear user interface while still beeing able to handle the various constellations of variants. Evaluating the work of this thesis in chapter 7 revealed that the approach followed universal requirements for a SPL, shown with the help of the FEF. Yet the requirements evaluation showed that there is still a need for a more comprehensive configuration framework, like demonstrated in the corresponding chapter, whereas the other requirements were met.

Supporting physicians with information technology offers a great potential for the optimization of clinical processes. An EHRMS that includes the functionality of MISs may help to reduce error rates, to increase performance of consultants and to improve the clinical outcome. Additionally, creating the system as a DSPL and emphasizing flexibility helps to overcome possible barriers when introducing a EHRMS in a hospital. The cloud-ready architecture creates the opportunity for an easy access to the system without buying and setting up a technical infrastructure in a hospital. Nevertheless, if it is needed because of data security, it is still possible to deploy the application to a private application server. The DSPL also allows flexible pricing models. Hospitals can start with a single specialty for a department to either test the system, or to realize a phased adoption. They also don't need to pay and operate an extensive and monolithic system, but rather get a customized application for their needs.

All in all, a SPL can be an extensive system, especially combined with the complex health care domain. Consequently, only the fundamental work was done in this thesis and there is still future work left.

9.2 Future Work

It is currently planned to integrate more MSs, MISs and technical services into the product line and hand it over to NSilico Lifescience Ltd. who will promote it as a commercial product. For this purpose at least the additional work described in this chapter is required.

9.2.1 Configuration

The configuration of the product line was initially described and outlined. Nevertheless, no concrete design was made for the multiple configuration levels. It is assumed that the product line owner and the tenant configuration will be spread over multiple files on the system, instead of having one file per tenant and product line owner. The same applies to the user configuration, which currently only covers the VAF. However, while the tenant configuration can be made manually together with the product line owner, a user configuration should be created automatically and be changed via the user interface as the amount of users can become large. Obviously also the tenant configuration could be realized with a self-service portal. But as the SPL is initially not designed for the mass market, a configuration with the product line owner is already seen as sufficient.

Presently there is also no design for hierarchical or linked tenants. For example if a hospital group is made up of five hospitals and each hospital shall have a separate tenant but the same tenant configuration. This would result in generating five independent tenant configurations instead of creating one configuration and linking the various tenants to it. It has to be further examined if this is a feasible approach or if the more simpler approach from this thesis is preferable.

Besides the technical design for the configurations itself, a concept for corporate identity should be made. This leaves tenants the possibility to at least integrate their own logos and colors and gives users the look and feel of a truly customized application.

9.2.2 Security

Currently only the fundamentals for the access management are designed. This is helpful, since the access management is mandatory for an EHRMS and has influence on many other designs. However, since the topic is very comprehensive, future work has to be done.

Currently no concrete concept for configurable roles and users is presented. To meet the requirements of every customer, a configurable access management should be introduced. This means that tenant and user rights are modeled in a sophisticated access management system, where users, roles and permissions are defined. Those definitions belong to the tenant configuration and can be used by the SPL.

Another important topic is the focus on secure communication and encryption. On the one hand, an introduction of the Transport Layer Security (TLS) protocol for the communication between server and client would secure the data transfer. On the other hand, if performance is not influenced to badly, an encryption of sensible patient data in the cloud could be contemplated.

In addition to the historization, the current SAGE-CARE Melanoma application provides a basic auditing functionality. The auditing is currently just logging the execution of a *REST WebAPI* method. A preparation for the product line should be considered as this is a useful feature, when being well integrated with the access management. In other words, this feature should log user and tenant data of method calls and provide an alert system if not permitted access was detected.

Legal standards are an additional issue that was not addressed in this thesis and should be examined when the product line is sold commercially.

9.2.3 Client Flexibility

The current UIC implementation for showing EHRs is only a first prototype, which shows the general user interaction model. Access management needs to be integrated to make tenant aware user interfaces. There is also a need for a better WebAPI interface implementation for querying the EHR data. Currently the whole patient is transferred to the client. Regarding the access management, this is not a recommended behavior, since a consultant can be assigned to one specific MS, which gives him only the rights to manage patients with this specific disease. Accordingly

the interface needs to return the patient with just one specific issue and only if the role of the user provides the corresponding permissions. If a finer grained access management is needed, it is additionally recommended to split up the patient data query into multiple requests. This means that only the fundamental patient data is loaded at first and the remaining data is lazily loaded, while the access management on the server decides if the user provides the appropriate permissions.

As stated in chapter 5.10.1, there is currently no concrete concept for deleting currently added UIC again. The design presented two possible ways but did not elaborate it further. It is recommended that only unsaved changes should be deletable to emphasize for the user that patient data should not be deletable at all. Although in this product line data can not be deleted but only historized.

The VAF, represented in chapter 5.10.2, still need to be more thoroughly designed and implemented. This is especially dependent on the concrete design of the user configuration, which is is currently only available as a concept.

9.2.4 Additional Technical Services

Like described in chapter 4.2.3 additional technical services were initially identified but not further elaborated during this thesis due to their priority. Especially a data-integration and a backup service should be introduced when preparing the product line for a commercial distribution. The data-integration should consider integrating HL7 RIM or openEHR (see [Bea+08] for further reading) compliant data. Backup services should be able to automatically save the current application data regularly on a remote server to counteract data loss.

9.2.5 Miscellaneous

Internationalization was only covered in terms of multi-language support. An extension to cultural conventions, like differing measuring units or symbols, should be considered. Additionally, only languages that are based on the Latin alphabet are implemented. Other alphabets, e.g. the Chinese alphabet, introduce greater changes and most likely need a reworked and more detailed concept.

The currently designed and implemented MISs in the SAGE-CARE Melanoma application aren't yet revised for the product line. For example the literature service,

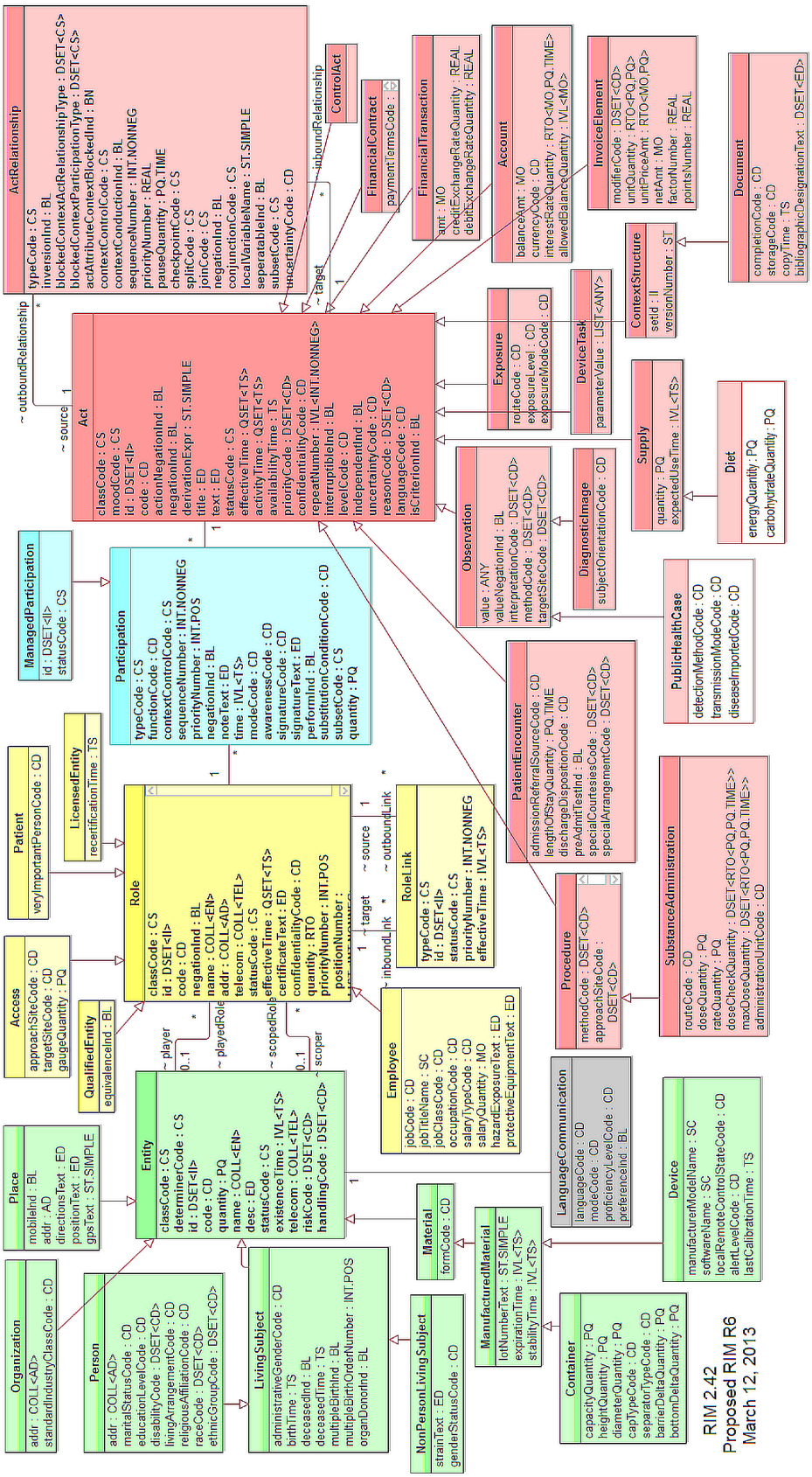
which searches relevant literature according to the patient's data, needs a mapping strategy for each specialty as they all have diverse classification data. Consequently they need to add a MS parameter in their interface, which determines the selection of the correct algorithm.

As this thesis was concerned with designing the system from a product line point of view, profound SaaS topics like load balancing or scalability were left out. Moreover, the requirements for a technical infrastructure were not further examined.

Appendices

Appendix A

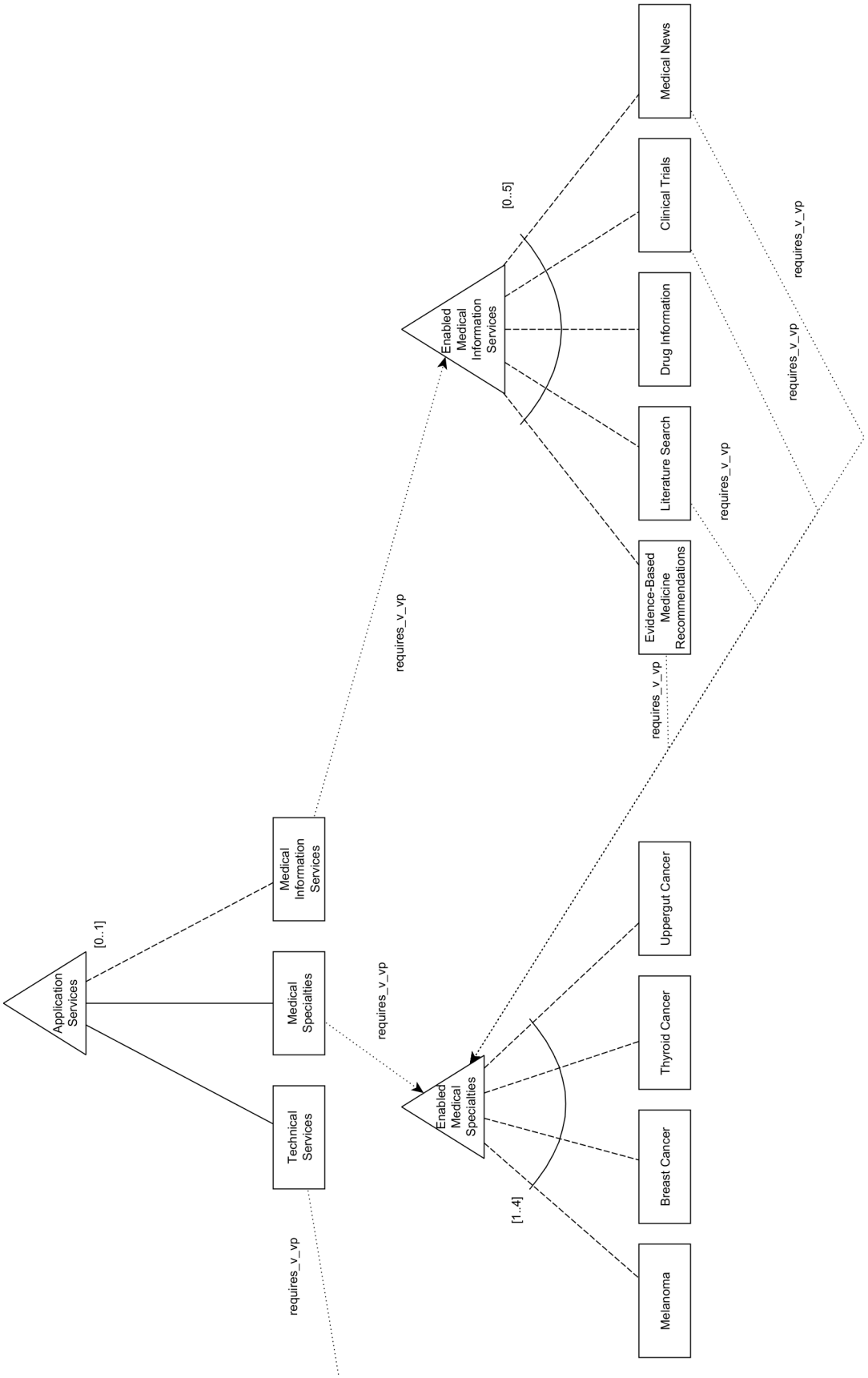
HL7 Reference Information Model

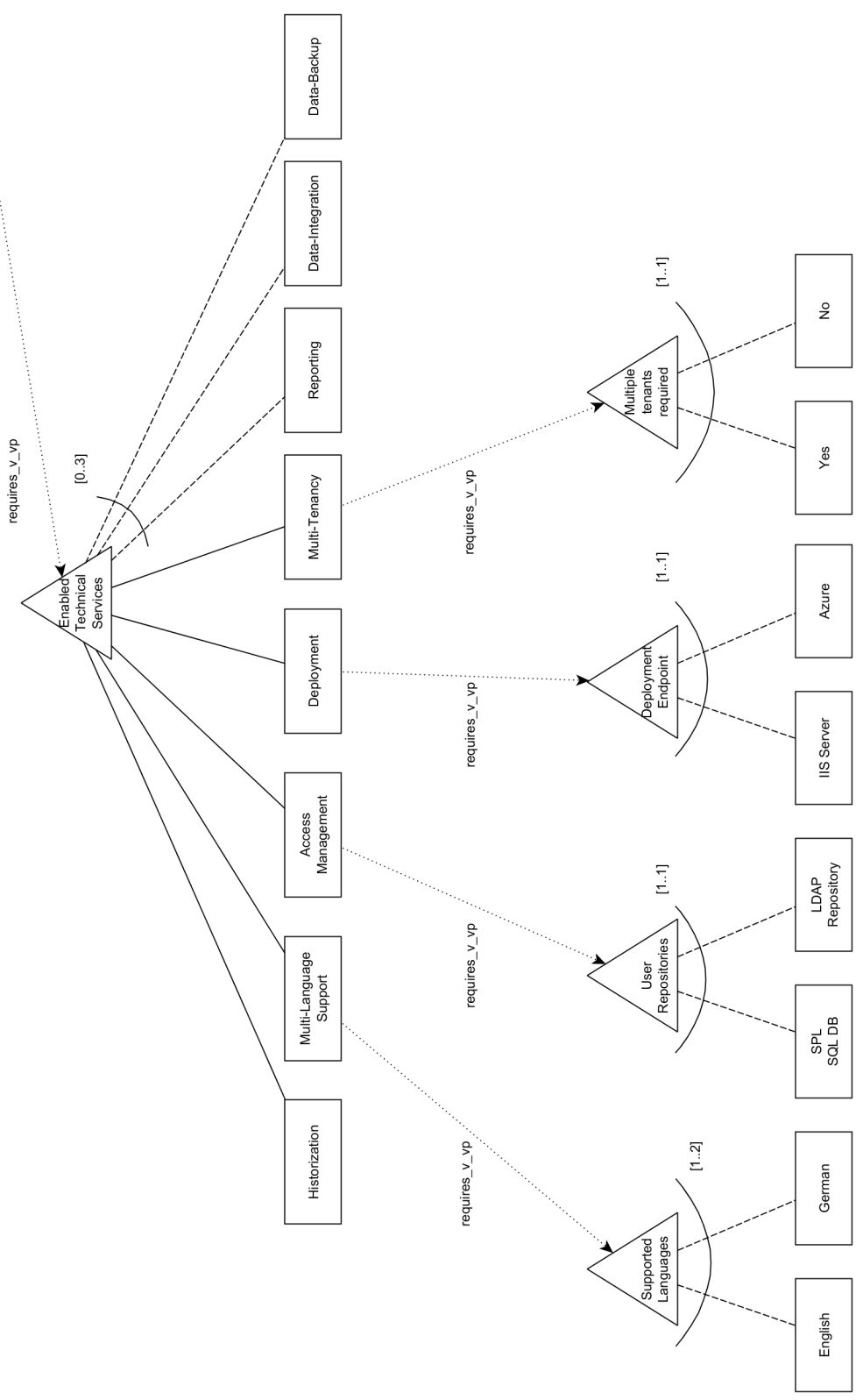

ActRelationship
typeCode : CS
inversionInd : BL
blockedContextActRelationshipType : DSET<CS>
blockedContextParticipationType : DSET<CS>
actAttributeContextBlockedInd : BN
contextControlCode : CS
contextConductionInd : BL
sequenceNumber : INT.NONNEG
priorityNumber : REAL
pauseQuantity : PQ.TIME
checkpointCode : CS
splitCode : CS
joinCode : CS
negationInd : BL
conjunctionCode : CS
localVariableName : ST.SIMPLE
seperatableInd : BL
subsetCode : CS
uncertaintyCode : CD
~ outboundRelationship
~ inboundRelationship
~ source
~ target
ControlAct
FinancialContract
paymentTermsCode : CS
FinancialTransaction
amt : MO
creditExchangeRateQuantity : REAL
debitExchangeRateQuantity : REAL
Account
balanceAmt : MO
currencyCode : CD
interestRateQuantity : RTO<MO,PQ.TIME>
allowedBalanceQuantity : IVL<MO>
InvoiceElement
modifierCode : DSET<CD>
unitQuantity : RTO<PQ,PQ>
unitPriceAmt : RTO<MO,PQ>
netAmt : MO
factorNumber : REAL
pointsNumber : REAL
Document
completionCode : CD
storageCode : CD
copyTime : TS
bibliographicDesignationText : DSET<ED>
ContextStructure
setId : II
versionNumber : ST
Act
classCode : CS
moodCode : CS
id : DSET<II>
code : CD
actionNegationInd : BL
negationInd : BL
derivationExpr : ST.SIMPLE
title : ED
text : ED
statusCode : CS
effectiveTime : QSET<TS>
activityTime : QSET<TS>
availabilityTime : TS
priorityCode : DSET<CD>
confidentialityCode : CD
repeatNumber : IVL<INT.NONNEG>
interruptibleInd : BL
levelCode : CD
independentInd : BL
uncertaintyCode : CD
reasonCode : DSET<CD>
languageCode : CD
isCriterionInd : BL
Exposure
routeCode : CD
exposureLevel : CD
exposureModeCode : CD
DeviceTask
parameterValue : LIST<ANY>
Supply
quantity : PQ
expectedUseTime : IVL<TS>
Diet
energyQuantity : PQ
carbohydrateQuantity : PQ
ManagedParticipation
id : DSET<II>
statusCode : CS
Participation
typeCode : CS
functionCode : CD
contextControlCode : CS
sequenceNumber : INT.NONNEG
priorityNumber : INT.POS
negationInd : BL
noteText : ED
time : IVL<TS>
modeCode : CD
awarenessCode : CD
signatureCode : CD
signatureText : ED
performInd : BL
substitutionConditionCode : CD
subsetCode : CS
quantity : PQ
Observation
value : ANY
valueNegationInd : BL
interpretationCode : DSET<CD>
methodCode : DSET<CD>
targetSiteCode : DSET<CD>
DiagnosticImage
subjectOrientationCode : CD
PublicHealthCase
detectionMethodCode : CD
transmissionModeCode : CD
diseaseImportedCode : CD
Patient
veryImportantPersonCode : CD
LicensedEntity
recertificationTime : TS
Role
classCode : CS
id : DSET<II>
code : CD
negationInd : BL
name : COLL<EN>
addr : COLL<AD>
telecom : COLL<TEL>
statusCode : CS
effectiveTime : QSET<TS>
certificateText : ED
confidentialityCode : CD
quantity : RTO
priorityNumber : INT.POS
positionNumber :
RoleLink
typeCode : CS
id : DSET<II>
statusCode : CS
priorityNumber : INT.NONNEG
effectiveTime : IVL<TS>
PatientEncounter
admissionReferralSourceCode : CD
lengthOfStayQuantity : PQ.TIME
dischargeDispositionCode : CD
preAdmitTestInd : BL
specialCourtesiesCode : DSET<CD>
specialArrangementCode : DSET<CD>
Access
approachSiteCode : CD
targetSiteCode : CD
gaugeQuantity : PQ
QualifiedEntity
equivalenceInd : BL
~ player
~ playedRole
~ scopedRole
~ scoper
~ inboundLink
~ outboundLink
Employee
jobCode : CD
jobTitleName : SC
jobClassCode : CD
occupationCode : CD
salaryTypeCode : CD
salaryQuantity : MO
hazardExposureText : ED
protectiveEquipmentText : ED
Procedure
methodCode : DSET<CD>
approachSiteCode : DSET<CD>
SubstanceAdministration
routeCode : CD
doseQuantity : PQ
rateQuantity : PQ
doseCheckQuantity : DSET<RTO<PQ,PQ.TIME>>
maxDoseQuantity : DSET<RTO<PQ,PQ.TIME>>
administrationUnitCode : CD
Place
mobileInd : BL
addr : AD
directionsText : ED
positionText : ED
gpsText : ST.SIMPLE
Entity
classCode : CS
determinerCode : CS
id : DSET<II>
code : CD
quantity : PQ
name : COLL<EN>
desc : ED
statusCode : CS
existenceTime : IVL<TS>
telecom : COLL<TEL>
riskCode : DSET<CD>
handlingCode : DSET<CD>
LanguageCommunication
languageCode : CD
modeCode : CD
proficiencyLevelCode : CD
preferenceInd : BL
Device
manufacturerModelName : SC
softwareName : SC
localRemoteControlStateCode : CD
alertLevelCode : CD
lastCalibrationTime : TS
Organization
addr : COLL<AD>
standardIndustryClassCode : CD
Person
addr : COLL<AD>
maritalStatusCode : CD
educationLevelCode : CD
disabilityCode : DSET<CD>
livingArrangementCode : CD
religiousAffiliationCode : CD
raceCode : DSET<CD>
ethnicGroupCode : DSET<CD>
LivingSubject
administrativeGenderCode : CD
birthTime : TS
deceasedInd : BL
deceasedTime : TS
multipleBirthInd : BL
multipleBirthOrderNumber : INT.POS
organDonorInd : BL
Material
formCode : CD
ManufacturedMaterial
lotNumberText : ST.SIMPLE
expirationTime : IVL<TS>
stabilityTime : IVL<TS>
NonPersonLivingSubject
strainText : ED
genderStatusCode : CD
Container
capacityQuantity : PQ
heightQuantity : PQ
diameterQuantity : PQ
capTypeCode : CD
separatorTypeCode : CD
barrierDeltaQuantity : PQ
bottomDeltaQuantity : PQ
RIM 2.42
Proposed RIM R6
March 12, 2013

Appendix B

Orthogonal Variability Model

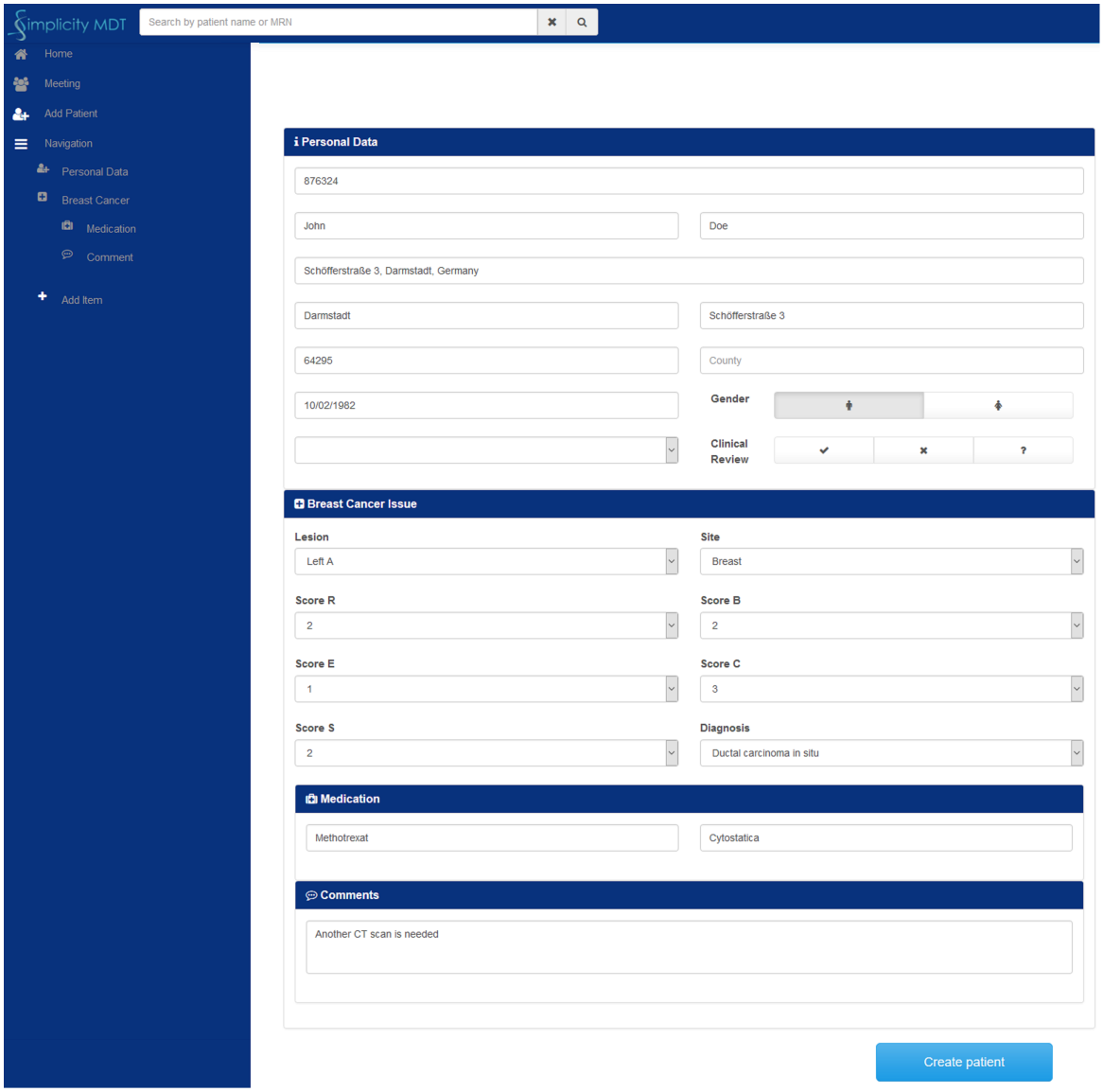
The OVM is split up on two pages, because of its size. The second figure is the left side of the total picture and shows the required variation point *Enabled Technical Services* for the variant *Technical Services*.


Application Services
[0..1]
Medical Information Services
Medical Specialties
Technical Services
requires_v_vp
requires_v_vp
requires_v_vp
Enabled Medical Information Services
[0..5]
Medical News
Clinical Trials
Drug Information
Literature Search
Evidence-Based Medicine Recommendations
requires_v_vp
requires_v_vp
requires_v_vp
Enabled Medical Specialties
[1..4]
Uppergut Cancer
Thyroid Cancer
Breast Cancer
Melanoma


requires_v_vp
[0..3]
Enabled Technical Services
Data-Backup
Data-Integration
Reporting
Multi-Tenancy
Deployment
Access Management
Multi-Language Support
Historization
requires_v_vp
Multiple tenants required
[1..1]
No
Yes
requires_v_vp
Deployment Endpoint
[1..1]
Azure
IIS Server
requires_v_vp
User Repositories
[1..1]
LDAP Repository
SPL SQL DB
requires_v_vp
Supported Languages
[1..2]
German
English

Appendix C

Add Patient User Interface



Appendix D

Act Class Implementation

```
1  public partial class Act : AbstractEntity
2  {
3
4     private ActRelationshipList _OutboundRelationship;
5     private ActRelationshipList _InboundRelationship;
6     private ParticipationList _Participation;
7
8     [InverseProperty("Source")]
9     public virtual ActRelationshipList OutboundRelationship { get; set; }
10
11    [InverseProperty("Target")]
12    public virtual ActRelationshipList InboundRelationship { get; set; }
13
14    public virtual ParticipationList Participation { get; set; }
15
16
17    public IList<T> GetAll<T>(RoleType relationshipType) where T : Role
18    {
19       List<T> result = null;
20
21       if (Participation != null &&
             Participation.RolesDict.ContainsKey(relationshipType))
22       {
23          var roles = Participation.RolesDict[relationshipType];
24          result = roles.OfType<T>().ToList();
25       }
26
27       return (result!=null && result.Count>0 ? result : null);
28    }
29
30    public IList<T> GetAll<T>(ActType relationshipType) where T : Act
```

```
31      {
32          var result = new List<T>();
33          if (OutboundRelationship!=null &&
                OutboundRelationship.TargetDict.ContainsKey(relationshipType))
34          {
35              var dictionaryTargets =
                    OutboundRelationship.TargetDict[relationshipType];
36              result.AddRange(dictionaryTargets.OfType<T>());
37          }
38          if (InboundRelationship != null &&
                InboundRelationship.SourceDict.ContainsKey(relationshipType))
39          {
40              var dictionarySources =
                    InboundRelationship.SourceDict[relationshipType];
41              result.AddRange(dictionarySources.OfType<T>());
42          }
43
44          return (result.Count > 0 ? result : null);
45      }
46
47
48      public T Get<T>(RoleType relationshipType) where T : Role
49      {
50          T result;
51          var getAllResult = this.GetAll<T>(relationshipType);
52          result = getAllResult != null ? getAllResult[0] : null;
53          return result;
54      }
55
56      public T Get<T>(ActType relationshipType) where T : Act
57      {
58          T result;
59          var getAllResult = this.GetAll<T>(relationshipType);
60          result = getAllResult != null ? getAllResult[0] : null;
61          return result;
62      }
63
64      public void Set(ActType targetTypeCode, Act target, ActType?
            sourceTypeCode = null)
65      {
66          if (OutboundRelationship != null &&
                OutboundRelationship.TargetDict.ContainsKey(targetTypeCode))
67          {
68              for (int i = 0; i < this.OutboundRelationship.Count; i++)
69              {
70                  var currentActRelationship =
                        this.OutboundRelationship.ElementAt(i);
```

```
71              if (currentActRelationship.TargetType == targetTypeCode)
72              {
73                 this.OutboundRelationship.RemoveAt(i);
74              }
75           }
76        }
77        this.Add(targetTypeCode, target, sourceTypeCode);
78     }
79
80     public void Set(RoleType roleTypeCode, Role role, ActType? actTypeCode
           = null)
81     {
82        if (Participation != null &&
              Participation.RolesDict.ContainsKey(roleTypeCode))
83        {
84           for (int i = 0; i < this.Participation.Count; i++)
85           {
86              var currentParticipation = this.Participation.ElementAt(i);
87              if (currentParticipation.RoleType == roleTypeCode)
88              {
89                 this.Participation.RemoveAt(i);
90              }
91           }
92        }
93        this.Add(roleTypeCode, role, actTypeCode);
94     }
95
96
97     public void Add(ActType targetTypeCode, Act target, ActType?
           sourceTypeCode = null)
98     {
99        var actRelationship = new ActRelationship(sourceTypeCode, this,
              targetTypeCode, target);
100       if (this.OutboundRelationship == null) this.OutboundRelationship =
              new ActRelationshipList();
101       this.OutboundRelationship.Add(actRelationship);
102       if (target.InboundRelationship == null) target.InboundRelationship
              = new ActRelationshipList();
103       target.InboundRelationship.Add(actRelationship);
104    }
105
106    public void Add(RoleType roleTypeCode, Role role, ActType? actTypeCode
           = null)
107    {
108       var participation = new Participation(roleTypeCode, role,
              actTypeCode, this);
```

```
109         if (this.Participation == null) this.Participation = new
                ParticipationList();
110         this.Participation.Add(participation);
111         if (role.Participation == null) role.Participation = new
                ParticipationList();
112         role.Participation.Add(participation);
113     }
114 }
```

Bibliography

[Acm] *Proceedings of the 26th International Conference on Software Engineering. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA*. eng. In collab. with Mari Matinlassi. ACM Special Interest Group on Software Engineering. Washington, DC: IEEE Computer Society, 2004. ISBN: 0-7695-2163-0. URL: http://dl.acm.org/citation.cfm?id=998675.

[AK15] Hwi Ahn and Sungwon Kang. *A Comparison of Software Product Line Architecture Design Methods from the Practicality Viewpoint*. 2015. URL: https://www.researchgate.net/publication/266171804_A_Comparison_of_Software_Product_Line_Architecture_Design_Methods_from_the_Practicality_Viewpoint (visited on 2017-01-28).

[Ape+13] Sven Apel et al. *Feature-oriented software product lines. Concepts and implementation*. eng. Berlin: Springer, 2013. 315 pp. ISBN: 978-3-642-37520-0.

[Bea+08] T. Beale et al. *EHR Information Model. The openEHR Reference Model*. Ed. by openEHR Foundation. 2008. URL: http://www.openehr.org/releases/1.0.2/architecture/rm/ehr_im.pdf (visited on 2017-04-06).

[Bee11] George W. Beeler. *Introduction to: HL7 Reference Information Model (RIM)*. Health Level Seven International. 2011. URL: https://www.hl7.org/documentcenter/public_temp_F88F5C4B-1C23-BA17-0CCBB0C201C99D4B/calendarofevents/himss/2011/HL7%20Reference%20Information%20Model.pdf (visited on 2017-02-07).

[Bee15] Ulrich Beez. "Terminology-Based Retrieval of Medical Publications". Master thesis. Darmstadt: University of Applied Science Darmstadt, 2015.

[Ben06] Messaoud Benantar. *Access control systems. Security, identity management and trust models*. eng. Boston, MA: Springer Science+Business Media Inc, 2006. 261 pp. ISBN: 9780387277165. DOI: 10.1007/0-387-27716-1. URL: http://dx.doi.org/10.1007/0-387-27716-1.

[Ber09] Eta S. Berner. *Clinical Decision Support Systems. State of the Art*. Agency for Healthcare Research and Quality. 2009. URL: https://healthit.ahrq.gov/sites/default/files/docs/page/09-0069-EF_1.pdf (visited on 2017-04-07).

[BG03] David W. Bates and Atul A. Gawande. "Improving safety with information technology". eng. In: *The New England journal of medicine* 348 (25 2003). Journal Article Research Support, U.S. Gov't, P.H.S., pp. 2526–2534. ISSN: 0028-4793. DOI: 10.1056/NEJMsa020847. eprint: 12815139.

[BGP12] Luciano Baresi, Sam Guinea, and Liliana Pasquale. "Service-Oriented Dynamic Software Product Lines". In: *Computer* 45 (10 2012), pp. 42–48. ISSN: 0018-9162. DOI: 10.1109/MC.2012.289.

[BHW15] Ulrich Beez, Bernhard G. Humm, and Paul Walsh. "Semantic AutoSuggest for Electronic Health Records". In: *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*. (Las Vegas, NV, USA). 2015, pp. 760–765. DOI: 10.1109/CSCI.2015.85.

[BM13] Arshdeep Bahga and Vijay K. Madisetti. "A cloud-based approach for interoperable electronic health records (EHRs)". eng. In: *IEEE journal of biomedical and health informatics* 17 (5 2013). Journal Article, pp. 894–906. ISSN: 2168-2194. DOI: 10.1109/JBHI.2013.2257818. eprint: 25055368.

[Bér16] David Bérubé. *Row level security in EntityFramework 6 (EF6)*. Ed. by Microsoft. 2016. URL: https://blogs.msdn.microsoft.com/mvpawardprogram/2016/02/09/row-level-security-in-entityframework-6-ef6/ (visited on 2017-03-23).

[CA13] Brian Coats and Subrata Acharya. "The forecast for electronic health record access". In: *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining - ASONAM '13*. the 2013 IEEE/ACM International Conference. (Niagara, Ontario, Canada). Ed. by Jon Rokne and Christos Faloutsos. New York, New York, USA: ACM Press, 2013, pp. 937–942. ISBN: 9781450322409. DOI: 10.1145/2492517.2500329.

[CBK13] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang, eds. *Systems and software variability management. Concepts, tools and experiences*. eng. Capilla, Rafael (Hrsg.) Bosch, Jan (Hrsg.) Kang, Kyo-Chul (Hrsg.) Heidelberg: Springer, 2013. ISBN: 978-3-642-36582-9.

[CCW06] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. *Multi-Tenant Data Architecture*. Microsoft. 2006. URL: https://msdn.microsoft.com/en-us/library/aa479086.aspx (visited on 2017-02-24).

[CN09] Paul Clements and Linda Northrop. *Software product lines. Practices and patterns*. 7. print. SEI series in software engineering. Boston et al.: Addison-Wesley, 2009. ISBN: 0201703327.

[Cor] *SemAntically integrating Genomics with Electronic health records for Cancer CARE*. CORDIS. 2017. URL: http://cordis.europa.eu/project/rcn/194165_de.html (visited on 2017-02-14).

[Ess00] Bert Esselink. *A Practical guide to localization*. Vol. v. 4. Language international world directory. 2000. ISBN: 9789027298188. URL: http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=e000xat&AN=334739.

[Feh14] Christoph Fehling. *Cloud computing patterns. Fundamentals to design, build, and manage cloud applications*. eng. Wien: Springer, 2014. ISBN: 978-3-7091-1567-1. URL: http://lib.myilibrary.com/detail.asp?id=635690.

[Fie00] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". University of California, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (visited on 2017-03-22).

[Gam+11] Erich Gamma et al. *Design patterns. Elements of reusable object-oriented software*. eng. 39. printing. Addison-Wesley professional computing series. Boston: Addison-Wesley, 2011. 395 pp. ISBN: 0201633612.

[Gom+12] Antônio Tadeu Azevedo Gomes et al. "SPLiCE. A Software Product Line for Healthcare". In: *Proceedings of the 2nd ACM SIGHIT symposium on International health informatics - IHI '12*. the 2nd ACM SIGHIT symposium. (Miami, Florida, USA). Ed. by Gang Luo et al. New York, New York, USA: ACM Press, 2012, p. 721. ISBN: 9781450307819. DOI: 10.1145/2110363.2110447.

[Hah16] Ralf Hahn. *Software Product Line Engineering. How to develop similiar products efficiently*. Darmstadt, 2016. URL: https://www.fbi.h-da.de/fileadmin/personal/r.hahn/2015WS/SWPLE/RH_SWPLE.pdf (visited on 2017-03-14).

[Hal97] Patrick A.V Hall. *Software without frontiers. A multi-platform, multi-cultural, multi-nation approach*. Wiley series in software engineering practice. Chichester [etc.]: J. Wiley, 1997. X, 338. ISBN: 9780471969747.

[Heaa] *About HL7*. Health Level Seven International. 2017. URL: http://www.hl7.org/about/index.cfm?ref=nav (visited on 2017-02-06).

[Heab] *HL7 Reference Information Model*. Health Level Seven International. 2017. URL: http://www.hl7.org/implement/standards/rim.cfm (visited on 2017-02-07).

[Hor10] Cay S. Horstmann. *Java concepts*. 6th ed. Hoboken: John Wiley & Sons, 2010. 666 S. ISBN: 9780470509470.

[HP03] Günter Halmans and Klaus Pohl. "Communicating the variability of a software-product family to customers". In: *Software and Systems Modeling* 2 (1 2003), pp. 15–36. ISSN: 1619-1366. DOI: 10.1007/s10270-003-0019-9.

[Hri10] Vagelis Hristidis. *Information discovery on electronic health records*. Chapman & Hall/CRC data mining and knowledge discovery series. Boca Raton: Taylor & Francis, 2010. ISBN: 9781420090413.

[HW15] Bernhard G. Humm and Paul Walsh. "Flexible yet Efficient Management of Electronic Health Records". In: *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*. (Las Vegas, NV, USA). 2015, pp. 771–775. DOI: 10.1109/CSCI.2015.84.

[Ide16] Johannes Idelhauser. "A Clinical Decision Support System for Personalised Medicine". Master thesis. Darmstadt: University of Applied Science Darmstadt, 2016.

[Jun] *JUnit Test Categories*. JUnit. URL: http://junit.org/junit4/javadoc/4.12/org/junit/experimental/categories/Categories.html (visited on 2017-03-01).

[Kan15] Joydip Kanjilal. *Entity Framework Tutorial*. 2nd ed. Olton Birmingham: Packt Publishing Ltd, 2015. ISBN: 9781783550029. URL: http://gbv.eblib.com/patron/FullRecord.aspx?p=3564802.

[Kir08] Wilhelm Kirch, ed. *Encyclopedia of Public Health*. eng. Dordrecht: Springer-Verlag Berlin Heidelberg, 2008. ISBN: 978-1-4020-5614-7. URL: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10284664.

[KMK12] Rouven Krebs, Christof Momm, and Samuel Kounev. *Architectural Concerns in Multi-Tenant SaaS Applications*. 2012. URL: https://se2.informatik.uni-wuerzburg.de/pa/uploads/papers/paper-371.pdf (visited on 2017-04-12).

[KS11] Alexander Kaletsch and Ali Sunyaev. *Privacy Engineering: Personal Health Records in Cloud Computing Environments*. 2011. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.667.7673&rep=rep1&type=pdf (visited on 2017-04-02).

[Kuo11] Alex Mu-Hsing Kuo. "Opportunities and challenges of cloud computing to improve health care services". eng. In: *Journal of medical Internet research* 13 (3 2011). Journal Article, e67. ISSN: 1438-8871. DOI: 10.2196/jmir.1867. eprint: 21937354.

[LSR07] Frank Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action. The Best Industrial Practice in Product Line Engineering*. 1. Aufl. s.l.: Springer-Verlag, 2007. ISBN: 9783540714361. URL: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10187468.

[LSW10] Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. "Securing the e-health cloud". In: *Proceedings of the ACM international conference on Health informatics - IHI '10*. the ACM international conference. (Arlington, Virginia, USA). Ed. by Ümit V. Çatalyürek et al. New York, New York, USA: ACM Press, 2010, p. 220. ISBN: 9781450300308. DOI: 10.1145/1882992.1883024.

[LYW13] Mingtao Lei, Wenbin Yao, and Cong Wang. "A Trust-Based Data Backup Method on the Cloud". In: *Trustworthy Computing and Services*. Ed. by Yuyu Yuan, Xu Wu, and Yueming Lu. Vol. 320. Communications in Computer and Information Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 178–185. ISBN: 978-3-642-35794-7. DOI: 10.1007/978-3-642-35795-4_23.

[LÖ09] Ling Liu and M. Tamer Özsu, eds. *Encyclopedia of database systems*. eng. Springer reference. New York, NY: Springer, 2009. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9. URL: http://dx.doi.org/10.1007/978-0-387-39940-9.

[Mica] *BindingList Class Documentation*. Microsoft. URL: https://msdn.microsoft.com/en-gb/library/ms132679(v=vs.110).aspx (visited on 2017-02-22).

[Micb] *Design patterns for multitenant SaaS applications and Azure SQL Database*. Microsoft. 2017. URL: https://docs.microsoft.com/en-us/azure/sql-database/sql-database-design-patterns-multi-tenancy-saas-applications (visited on 2017-04-12).

[Micc] *Group and Run Automated Tests Using Test Categories*. Microsoft. URL: https://msdn.microsoft.com/en-us/library/dd286683.aspx (visited on 2017-03-01).

[Micd] Microsoft, ed. *How to: Create a Localized Version of a Resource File*. URL: https://msdn.microsoft.com/en-us/library/aa992030(v=vs.100).aspx (visited on 2017-02-28).

[Mice] *Security, Authentication, and Authorization in ASP.NET Web API*. Microsoft. 2012. URL: https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/ (visited on 2017-02-27).

[Mie+09] Ralph Mietzner et al. "Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications". In: *2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*. 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, PESOS. (Vancouver, BC, Canada). IEEE, 2009, pp. 18–25. ISBN: 978-1-4244-3716-0. DOI: 10.1109/PESOS.2009.5068815.

[NSind] NSilico, ed. *NSilico - About*. n.d. URL: http://www.nsilico.com/About (visited on 2017-02-14).

[Ora] *Internationalizing the Sample Program*. Oracle. URL: https://docs.oracle.com/javase/tutorial/i18n/intro/steps.html (visited on 2017-02-28).

[PBL05] Klaus Pohl, Günter Böckle, and Frank Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*. eng. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2005. ISBN: 3540243720. DOI: 10.1007/3-540-28901-1. URL: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10229377.

[RA11] Stefan T. Ruehl and Urs Andelfinger. "Applying software product lines to create customizable software-as-a-service applications". In: *Proceedings of the 15th International Software Product Line Conference on - SPLC '11*. the 15th International Software Product Line Conference. (Munich, Germany). Ed. by Ina Schaefer, Isabel Klaus, and John Schmid. New York, New York, USA: ACM Press, 2011. ISBN: 9781450307895. DOI: 10.1145/2019136.2019154.

[Red] *Hibernate Envers - Easy Entity Auditing*. Red Hat Inc. 2012. URL: https://docs.jboss.org/hibernate/envers/3.6/reference/en-US/html_single/ (visited on 2017-03-07).

[Sch13] Julia Schroeter. *Feature-Based Configuration Management of Reconfigurable Cloud Applications*. Dresden, 2013. URL: http://www.qucosa.de/fileadmin/data/qucosa/documents/14141/diss-schroeter-04-26-revised-final-pdfA1b.pdf (visited on 2017-04-12).

[Sch14] Jan Schaffner. *Multi Tenancy for Cloud-Based In-Memory Column Databases. Workload Management and Data Placement*. @Potsdam, Univ., Diss., 2013. eng. In-Memory Data Management Research.

Schaffner, Jan (author.) Dordrecht: Springer, 2014. 140 pp. ISBN: 9783319004976. DOI: 10.1007/978-3-319-00497-6. URL: http://gbv.eblib.com/patron/FullRecord.aspx?p=1317762.

[SR12] Klaus Schmid and Andreas Rummler. "Cloud-based software product lines". In: *Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1*. the 16th International Software Product Line Conference. (Salvador, Brazil). Ed. by ACM. New York, New York, USA: ACM Press, 2012, p. 164. ISBN: 9781450310956. DOI: 10.1145/2364412.2364440.

[Tod07] Dobromir Todorov. *Mechanics of user identification and authentication. Fundamentals of identity management*. Boca Raton: Auerbach Publications, 2007. ISBN: 9781420052206.

[Ame04] American National Standards Institute, ed. *Role Based Access Control*. ANSI INCITS 359-2004. New York, 2004.

[Int05] International Organization for Standardization, ed. *Health informatics - Electronic health record - Definition, scope and context*. ISO/TR 20514:2005. Switzerland, 2005. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39525.

[Int06] International Organization for Standardization, ed. *Health informatics - HL7 version 3 - Reference information model - Release 1*. ISO/HL7 21731:2006. Switzerland, 2006. URL: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=40399.

[van02] F. van der Linden. "Software product families in Europe. The Esaps & Cafe projects". In: *IEEE Software* 19 (4 2002), pp. 41–49. ISSN: 0740-7459. DOI: 10.1109/MS.2002.1020286.