Konzeption und Realisierung einer verteilten Adressdatenbank im Stil einer Publish/Subscribe-Architektur zum Austausch von Patientendaten zwischen autonomen medizinischen Informationssystemen

Diplomarbeit im Fach Informatik

vorgelegt von

Florian Dominik Rampp

geb. 30.07.1984 in Günzburg

angefertigt am

Department Informatik
Lehrstuhl für Informatik 6
Datenmanagement
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr. Richard Lenz
Dipl.-Inf. Christoph Neumann

Beginn der Arbeit: 15.10.2008
Abgabe der Arbeit: 31.03.2009

Erklärung zur Selbständigkeit

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Friedrich-Alexander-Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Diplomarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 31.03.2009

(Florian Dominik Rampp)

Zusammenfassung

Aufgrund des fehlenden Informationsaustausches zwischen Institutionen leidet die medizinische Versorgungskette in Deutschland unter der unzureichenden Verfügbarkeit von patientenbezogenen Daten. Im Rahmen dieser Diplomarbeit wird eine Lösung vorgestellt, die Patienteninformationen zwischen autonomen medizinischen Informationssystemen austauscht. Die drei Phasen des Informationsaustausches umfassen die Bündelung von Informationen die eingebracht werden sollen, die Rückführung dieser Informationseinheit in die Patientenakte und die darauf folgende Verteilung der Änderungen an weitere Teilnehmer. Das zu entwerfende System implementiert eine Publish/Subscribe-Architektur. Die Hauptziele sind der Verzicht auf eine zentrale Infrastruktur und die Orientierung an der traditionellen, dokumentenorientierten Arbeitsweise.

Eine Referenzimplementierung zeigt die praktische Umsetzbarkeit des Konzepts. Ein multidimensionaler Systementwurf führt zu Modulen mit einem genau umrissenen Funktionsumfang und klar definierten Schnittstellen. Aufgrund der Verteilung des Systems werden Kommunikationsprotokolle zur Datenübertragung zwischen Systemknoten benötigt. Dazu wurde eine Plugin-Architektur entworfen, die die Einbindung von beliebigen Protokollen ermöglicht. Des Weiteren werden verschiedene Möglichkeiten zur Adressierung von Systemkonten aufgezeigt. Das System ist unabhängig vom Typ der ausgetauschten Daten, was sich in der Einführung einer domänenunspezifischen Terminologie widerspiegelt. Als erster, ausgetauschter Informationstyp wurden Adressdaten ausgewählt, da diese allen Patientenakten gemein sind.

Design and Implementation of a Distributed Address Database Following a Publish/Subscribe Architecture to Share Patient Data Among Autonomous Healthcare Information Systems

Abstract

Due to the lack of inter-institutional information exchange, the medical supply chain in Germany suffers from the insufficient availability of patient-related data. In the course of this thesis, a solution is proposed that shares patient-centric information among autonomous healthcare information systems. The three phases of information exchange encompass the bundling of information to contribute, repatriating this information unit into the patient health record, and subsequently publishing the changes to subscribers. Therefore, the system to be designed follows a publish/subscribe architecture. The main objectives are the abdication of any central infrastructure and the adherence to a traditional document-oriented approach.

A reference implementation is provided that proves the feasibility of the overall concept. A multi-dimensional system design approach creates modules with a distinct scope of work and well-defined interfaces. Due to the distributed nature of the system, network communication protocols are required to support data transfer between nodes. Therefore, a plug-in architecture is designed that allows the integration of bindings to arbitrary protocols. Furthermore, different types of identifier schemes are used to address accounts. The system is agnostic of the exchanged information that is reflected by the introduction of a domain-unspecific terminology. As a first exemplary information type to be exchanged, address data, being common to all patient files, is chosen.

Contents

1 Introduction

The current state of inter-institutional supply chains in healthcare follows a traditional paper-based approach. It encompasses several participants including healthcare professionals and institutions in the primary and secondary care sectors. The increasing number of parties being involved in patient treatment requires seamless flow of patient-related information. This is even more critical with the advent of comprehensive, closely meshed teams of physicians and other accredited partners. Traditional paper-based information exchange reaches its limits and can't keep up with the changing requirements. Patient-related data is missing since it is not available to all treating institutions. Current clinical systems that provide information interchange are organized around a central infrastructure and thus are only deployed in hospitals connecting sections of expertise. Even existing region-wide healthcare systems bridging institutions require federated databases or central context managers. In the course of this thesis, a solution is presented that supports inter-institutional, cross-organizational flow of patient-related information. It provides distribution of health records without requiring a central infrastructure.

1.1 The Current State

The medical supply chain in Germany involves physicians of the primary and the secondary healthcare sector. While office-based physicians constitute the primary care, the secondary care encompasses hospitals, pharmacies, and laboratories. Additional participants in the healthcare sector include health insurance funds and associations of statutory health insurance physicians. These healthcare sectors instrument *Healthcare Information Systems* in order to gather and process patient-related information.

If the current institution cannot provide the needed spectrum of diagnosis and therapeutic measures, patients are referred to another institution using *letters of referral*. As a consequence, responsibility for medical, organizational, financial, and legal liability is delegated. *Discharge letters* provide the referring institution with information about diagnosis, significant investigations, medication, etc. concerning the treatment of the patient in the institution he/she has been referred to. These two examples of information flows traditionally use paper-based documents. The patient is not necessarily involved in data interchange directly, but may only serve as surrogate for postal delivery. The described system of information exchange is based on the assumption of patients trusting in their physicians and physicians trusting in each other.

Another scenario is the advent of treating patients in comprehensive, closely meshed teams of physicians and accredited partners [RL03]. Especially, chronic diseases, like cancer, diabetes, and cardiac insufficiencies result in closer collaboration of physicians [LSLG00]. This requires a comprehensive, long-term exchange of patient-related information bridging institutions and sections of authority and expertise.

Existing healthcare frameworks for data exchange focus on institutions of the secondary care. Complex hospitals are provided with protocol standards that support information interchange between organizational sections. These protocol standards presume a central infrastructure potentially involving federated databases, transaction monitors, or context managers. An example is the XDS[1] standard from IHE[2] that allows for distributed document repositories [ACC07]. Nonetheless, a central system node is required for registering documents and providing an index for retrieval. Even existing systems following RHIN[3] architectures [WVM01] require a central infrastructure for providing wide-area information exchange.

1.2 The Emerging Problem

While an increasing number of parties are involved in patient treatment, no comprehensive system for inter-institutional data exchange is available. The consequence is missing patient-related information, like discharge letters, not being available to all involved institutions [Sam04]. This problem is even more critical with the advent of physicians being organized in inter-institutional, closely meshed teams. A system for providing nation-wide information exchange without the need for a central infrastructure is currently not available. Lacking such a platform, comprehensive functional and data integration [LBK07] bridging institutions is out of reach.

Any solution has to cope with the obstacle of coupling heterogeneous and isolated legacy systems. They were neither designed for cooperation nor do they support current data integration standards. Furthermore, introducing a new information exchange platform is always a trade-off between the retention of local autonomy and enforcing data integration by common standards.

1 Cross Enterprise Document Sharing
2 Integrating the Healthcare Enterprise
3 Regional Healthcare Information Networks

1.3 Supporting Information Exchange Between Institutions

To solve the described problem, a solution is introduced in the course of this thesis that supports the flow of patient-related information between institutions, be they in primary or secondary care. This system does not require central infrastructure and, thus, allows for the maintenance of an institution's local autonomy. It also supports closely meshed teams distributed over several institutions and sections of expertise, as elaborated above. The system to be designed permits the controlled distribution of a patient file to trusted physicians and other institutions involved in the medical supply chain. Originating in institutional EHRs[1] [PB05], information will be contributed to a patient-centered health record, potentially comprising the entire medical history. The patient will be the lone authority over his/her data and, thus, governs the information exchange by allowing healthcare parties to subscribe to subsets of the patient file.

Furthermore, the system to be designed will foster data integration between institutions by offering a facility for transporting HL7[2] CDA[3] documents. CDA provides a standard for XML[4]-structured clinical documents with well-defined encoding, structure, and semantics. They are persistent in nature and contain human-readable as well as optional structured parts to be processed by software. The structured parts instrument existing standards for medical ontologies, like SNOMED[5] or LOINC[6], to represent medical concepts. CDA does not require a specific way of transporting the documents. Besides the solution presented in this thesis, other transportation facilities are HL7 v2 or v3 messages, as well as instrumenting email, or FTP[7].

While being critical to data protection and privacy, the legal implications accompanying the solution are out of scope of this thesis.

1 Electronical Health Records
2 Health Level 7
3 Clinical Document Architecture
4 Extended Markup Language
5 Systematized Nomenclature of Medicine
6 Logical Observation Identifiers Names and Codes
7 File Transfer Protocol

2 Methods

This chapter provides an overview of the design process and applied methods used in shaping and implementing the functionality of the system to design. The current state and the emerging problem of inter-institutional healthcare supply chains have been explained in the introduction. A solution is proposed that fulfills well-defined objectives described in chapter 3. An initial overview of the functionality of the system to be designed is obtained by following a basic interaction scenario. Several phases and artifacts are distinguished, while an additional scenario is introduced which presumes a closer collaboration of physicians in expert teams.

These basic assumptions about the extent and functionality of the system to be designed are elaborated on in depth in chapter 4. The proposed solution to the problems and requirements outlined in the previous chapter is a system called DEUS. It is agnostic of the type of data exchanged which is implemented by introducing a level of abstraction. Subsequently, different actors and roles are established that reflect this abstraction. The basic unit of information exchange is the Digital Card which will be introduced together with other artifacts of DEUS. The basic interaction scenario is elaborated by closely examining the sequence of phases and steps that are needed for information interchange. Trust relationships and semantics associated with contributing and publishing information are explained. Further requirements of DEUS include an identifier scheme for accounts, communication protocols for inter-node information exchange, and authentication. A use case analysis helped in gathering the requirements and the basic outline of DEUS. By taking the point of view of a user in the different roles, such as a physician or a patient, necessary use cases were found.

Since DEUS enables information exchange between medical healthcare information systems running on different servers, the system is distributed over many nodes. Therefore, the basic principles of a distributed system, including communication, addressing, system identification, and service discovery need to be taken into account. An overview of some fundamentals of distributed system design is given in chapter 5. Two alternatives for identifiers together with a method for service and metadata discovery is described. After adopting the principles of protocol design, it was decided to keep the system independent of a specific communication protocol. Rather, a general protocol is devised adhering to a top-down approach of deriving functionality from use cases. Various bindings of this protocol to concrete communication protocols are introduced and evaluated.

UML[1] and a multidimensional system modeling approach are used to decompose DEUS into subsystems, tiers, and sublayers and thus obtain modules with a well-defined interface and scope of operation. These design considerations resulted in the system architecture described in chapter 6. As an example of data to be exchanged by instrumenting DEUS, address and contact data is taken, since this is a common subset each health record contains. A data model for address and contact data was created using ERDs[2] that is influenced by the address data standard vCard. This modelling resulted in a data model, encompassing vCard and other contact data standards.

The scope of work and the functionality of each of the system modules is explained in chapter 7. Here, the details of their responsibilities and the interfaces they offer to other modules are described. A component model called OSGi[3], which offers life cycle support for each module, is used to implement the components. The use cases found in the requirements analysis are realized by interaction of certain modules. This collaboration and further design issues like inter-module dependencies, design patterns, and module versioning are explained.

To ease development, several frameworks and technologies are used including tools supporting the build cycle and the deployment phase. The runtime environment has to provide a container for installing and executing the OSGi components. These implementation topics are covered in chapter 8.

On designing and developing DEUS, several points of extensibility were discovered. Chapter 9 explains how the scope of the system can be enlarged by implementing omitted functionality and elaborating advanced concepts. Future work can build upon the efforts described in this thesis and the existing DEUS reference implementation.

1 Unified Modelling Language
2 Entity-Relationship Diagrams
3 Open Services Gateway Initiative

3 Requirements Analysis

In the course of this thesis, a system will be designed that provides inter-institutional, interdisciplinary distribution of patient-related data. This system should fulfill some requirements, with the most important being the retention of local autonomy by the abdication of any central infrastructure and integration of existing, heterogeneous systems. It furthermore should adhere to the document-oriented approach currently found in healthcare supply chains. Another objective is the empowerment of the patient to be the lone sovereign of his/her data and thus control the distribution of it. A basic interaction scenario is outlined that involves contribution of patient-related data to a health record and subsequent publication of this information to interested parties. A second scenario introduces a collaborative approach of treating patients in closely meshed, comprehensive teams of physicians. In collaborating teams, even more than in classical cooperation, a seamless flow of information among the involved institutions is essential.

3.1 Objectives

The proposed solution should adhere to the main objectives elaborated in this section. *No central mediator* should require the institutions to give up their local autonomy in favor of a central hegemony. This includes the abdication of any kind of central infrastructure like joint databases, transaction monitors, or central context managers.

Since physicians are accustomed to a *document-oriented* working practice, the system should also reflect this approach in its operation and basic data model. Self-contained and viable information units constitute documents adhering to this principle. Their viability results from contained context information [LBK07], such as the author, subject, labels, and descriptions that uniquely identify the document. This furthermore involves the incorporation of medical concepts inside documents rather than in the interface of a system. Since frequent changes of system interfaces entail compatibility, operation, and deployment problems, interfaces should be kept simple and lightweight. This approach follows the *deferred design principle* [Pat03] of evolutionary systems by not freezing design decisions about domain semantics into a fixed interface. Due to the continuous adaption of medical models and terminologies to current knowledge, a document-oriented approach using lightweight interfaces is preferred over a service-oriented solution with semantically rich interfaces. In comparison to the traditional approach of rather comprehensive and large documents, the proposed solution favors the exchange of smaller, viable units of information. Smaller information units improve the structure of health records by providing a higher selectivity in retrieval and display.

Furthermore, a *patient-centric approach* should reflect the medical supply chain in Germany with its focus on the patient. The patient should maintain control of the exchanged data and, thus, is empowered with the responsibility of deciding which information is shared with whom. While this change of notion appears quite radical, 'Patient Empowerment' is an upcoming issue in eHealth [eHe03]. If a patient is unable to govern information exchange, the sovereignty over his/her data can be delegated to a legitimate other party, be it an institution or a person.

Existing healthcare information systems are heterogeneous considering the used software and platform. In order to be as integrative and compatible as possible, the solution should not be coupled to a specific platform. Vendor and technology lock-in is avoided by keeping the architecture and the application core independent of any used remote invocation technology, storage mechanism, or middleware frameworks.

3.2 The Basic Interaction Scenario

The basic interaction scenario is depicted in figure 3.1 involving three main actors. A person named 'Alice', assuming the role of *Patient* visits a physician, named 'Dr. Higgins'[1], taking on the role of *Treatment Provider* here. The information gathered during her visit at Dr. Higgins has to be shared among other people including the healthcare professional 'Prof. Bob', assuming the role of *Audience Party.* The three introduced top level user roles embody the main actors of the interaction scenario. The required trust connections between the different actors are being thought of as established earlier.

3.2.1 Contribution and Repatriation Phase

The information unit, containing data about the treatment of Alice, is bundled by the HCIS[2] of Dr. Higgins. It contains, for example, a diagnostic finding, clinical evidence, a diagnosis, a therapeutic measure, an order, or a prescription. The process of subsequently importing the unit into the HCIS' local system extension is called *Contribution.* Afterwards, the contributed unit is signed, using cryptographic methods, and transferred to the patient's system account. Alice, as being the concerned person and the sovereign of information interchange, then decides

1 The name is a homage to the 'Higgins Project', that introduced the notion of an Information Card as self-contained, viable document representing a part of a party's identity.

2 Healthcare Information System

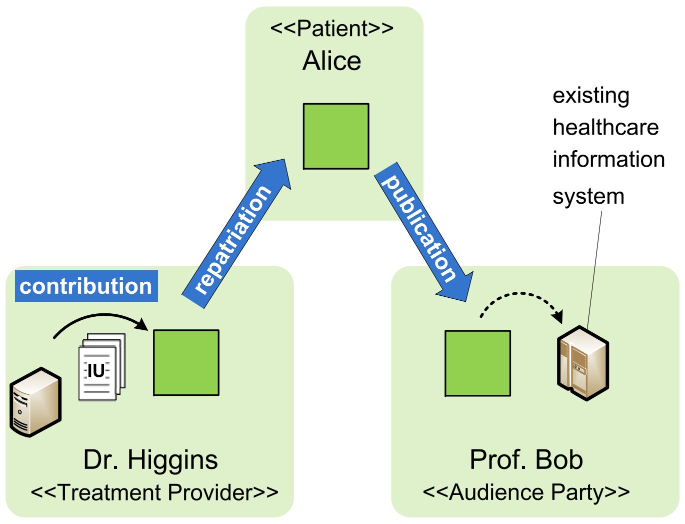


Figure 3.1: The basic interaction scenario around contribution, repatriation and publication

whether the contributed information unit is assimilated into her health record. The process of transferring a unit of information to the patient system and the successive decision about the acceptance of it is called *Repatriation*.

3.2.2 Publication Phase

Following the assimilation of the new information, the changes to the health record of the patient are published to the subscribers of Alice's patient file, among them Prof. Bob. This phase is called *Publication*. While only a single unit of information, being viable and self-contained as previously described, is contributed and subsequently assimilated into the patient's health record, this may trigger a list of changes to the patient file. It depends on the semantics of the assimilation how contributed information changes the existing data. Thus, the act of Publication does not only transfer the newly contributed information unit, but a more sophisticated data structure, called a patch, reflecting the change in the state of the health record.

Dr. Bob collects the received information about Alice in a local patient file. This local copy represents a replicated version of only a subset of the patient file, because not every contributed information unit necessarily has to be published to all subscribers. So, the process of publication constitutes an unidirectional synchronization of a subset of the patient's health record with the replicated version of the subscribers. Existing HCISs can subsequently access this local copy and retrieve information from it to be used inside the HCIS.

3.3 Artifacts

The health record of Alice constitutes the subject of interest and is named *Personal Patient File (PPF)*. It consists of a list of information units representing the medical history of the patient as far as it is known to the system. The subscriber side replication of the health record contains a subset of the information units of the Personal Patient File and is called *Foreign Patient File (FPF)*. A healthcare professional in the role of Audience Party will require access to patient related information of more than one patient. Thus, Foreign Patient Files of multiple patients are collected in a depository called *Distributed Patient Folder (DPF)*. The described artifacts and their relations are visualized in figure 3.2.

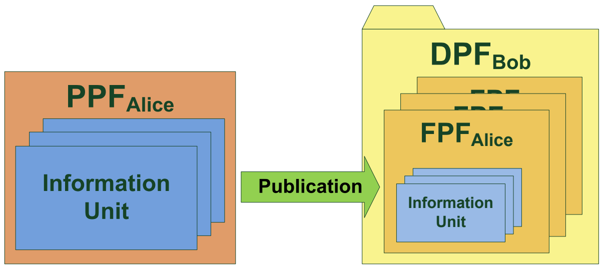


Figure 3.2: The relationship of the artifacts Personal Patient File, Foreign Patient File and Distributed Patient Folder

3.4 A Collaborative Scenario

The introduced basic interaction scenario constitutes a cooperative approach rather than a collaborative one. In a cooperative environment, participants are organizationally independent, the degree of interdependence is low. While in a collaborative scenario organizational independence is kept, the degree of work-product interdependence is high. The collective results obtained by a team of physicians and accredited partners would be impossible when working alone.

An example for a collaborative scenario is a breast cancer treatment center with closely meshed teams of oncologists, radiologists, post-operative care, and other healthcare professionals. Here, patient interaction is counterproductive and information should be exchanged unfiltered among the team without a patient to govern information interchange. The system to be designed should also support this collaborative scenario by providing means of bundling actors into teams and bypassing decision processes involving the patient.

3.5 Conclusion

As explained in the previous chapter, there is a necessity for a comprehensive solution to the problem of inter-institutional information exchange of patient-related data. Therefore, a system should be designed, that enables heterogeneous Healthcare Information Systems to communicate and exchange information, without the need for a central mediating instance. Adhering to a patient-centric design guarantees the sovereignty of the patient over his/her data and its distribution. The exchanged artifacts should constitute self-contained, viable documents following the document-oriented approach to ease the adoption by mirroring familiar working practices. A basic interaction scenario encompassing three different types of actors was introduced, as well as several artifacts of information exchange. Another scenario constitutes a collaborative approach of treating patients in closely meshed teams of healthcare professionals. Both scenarios should be supported by the designed system.

4 Outline of DEUS

By introducing DEUS, a solution to the emerging problems explained in chapter 1 is proposed. DEUS is an acronym for *Distributed Electronic Patient File Update System.* The system fulfills the described requirements by adhering to patient-centric and document-oriented principles. The classic approach of publish-subscribe systems is extended by the notion of a mediator, incorporated by the patient, governing the distribution of medical information related to him/her. DEUS employs a distributed architecture, instrumenting peer-to-peer communication to avoid the usage of a central infrastructure. Furthermore, the system is domain-unspecific and agnostic of the content of documents leading to the ability to share arbitrary data using the developed information distribution architecture. To reflect this generic approach, a layer of abstraction is introduced that provides a generic, domain-unspecific terminology. This approach fosters system evolution by adhering to the 'deferred design principle' [Pat03]. It furthermore reflects a layering style that follows the layered approach of system evolution in [LBK07].

In this chapter, a basic distinction is made between DEUS accounts and the nodes they reside on. These accounts are associated with actors that assume domain-unspecific distribution roles. The introduced actors and their assumed roles no more reflect a medical domain. Artifacts being involved in information exchange are elaborated by introducing a domain-independent notation as well.

The basic interaction scenario introduced in section 3.2 is detailed further by describing the setup and teardown of trust relationships between actors. Semantics concerning the assimilation of contributed information units into the Personal Patient File are elaborated. The subsequent publication of the changed state of the health record involves principles of synchronization. A group-based approach for choosing the subscribers and filtering the data to be published is presented below. Different possibilities are identified to initially publish historic data after a subscription is established. Furthermore, DEUS is also able to support the collaboration scenario of section 3.4. The chapter closes with an overview of some technical requirements concerning identification of accounts, used communication protocols, and authentication.

4.1 DEUS Nodes vs. DEUS Accounts

Existing HCIS with an installed DEUS extension or independent servers offering DEUS functionality are called DEUS *nodes*. Since DEUS is multitenant, each node may host an arbitrary number of DEUS *accounts*. Each actor participating in information exchange has its own account, which is uniquely identified by a user ID. While functionality can be restricted on a per-account basis to reflect the scope of work of the different actors, each node nonetheless supports the full functionality of DEUS.

For setting up an account, a patient may freely choose a DEUS node. This node may either be self-administrated or offered by a trusted third party hosting DEUS accounts on behalf of the user. While the latter is probably the more prevalent scenario, DEUS does not enforce this decision. Examples of third parties that offer patients to host patient accounts might be hospitals, the patient's general practitioner, or health insurance companies.

Interaction between DEUS accounts may result in communication over network links depending on whether the accounts reside on the same node. Instrumented protocols for this communication will be elaborated later.

4.2 DEUS Actors and Their Assumed Roles

The requirement analysis revealed three healthcare actors participating in DEUS. The first actor is the *Treatment Provider*, that contributes information to the health record of a patient. The second actor involved in DEUS is the *Patient*, being the person concerned of contributed information and governing the distribution of his/her data. If another physician requires read access to the health record of a patients, the physician will subscribe to the patient file and thus become the *Audience Party* actor.

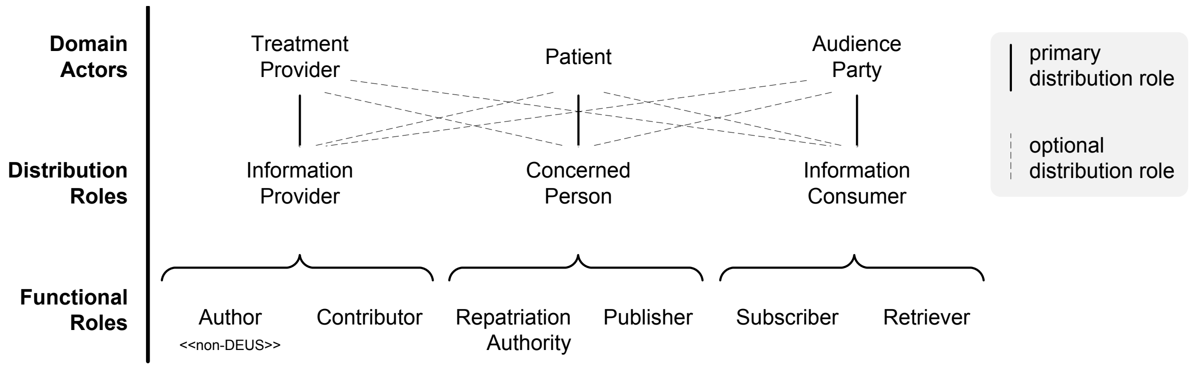


Figure 4.1: The three DEUS actors and their roles in a distribution scenario.

Since the DEUS architecture is domain-unspecific, the terminology used should also be domain-unspecific. Thus, each of the introduced actors assumes by its DEUS account a *distribution role* reflecting a primary interest in the overall distribution scenario. This is depicted by the vertical lines in figure 4.1. The domain-unspecific role of *Information Provider* contributes information about a *Concerned Person*, who decides about sharing this contribution with *Information Consumers*.

Besides a primary role, each actor can potentially assume, by its DEUS account, each distribution role as it is visualized by the dashed lines. For example, the patient can act as Information Provider about himself/herself, contributing information like allergies or legacy paper document scans. Similarly, a treatment provider can use its DEUS account in the role of a Concerned Person to provide and publish business card information or consultation hours information. Employing DEUS in another domain involves identifying the main actors using domain-specific names. It is possible to restrict which distribution roles can be assumed by a healthcare actor.

Certain responsibilities can be deduced from distribution roles forming *functional roles*. The Information Provider acts as *author* of the information, which takes place inside the HCIS and is not part of DEUS. The Information Provider acts as *contributor* by bundling an information unit, signing it and handing it over to the actual DEUS account. The Concerned Person acts as *repatriation authority* by deciding about the validity of a repatriated information unit. The Concerned Person acts as *publisher* by applying the selection of subscribers and performing the publication transfer. The Information Consumer acts as *subscriber* by establishing subscriptions to the account of the Concerned Person and by accepting published information units. Finally, the Information Consumer acts as *retriever* by accessing the information pool. Each functional role will be further detailed in chapter 7 when the subsystems related to each role will be introduced.

4.3 DEUS Artifacts

An information unit, as mentioned in the previous chapter, is formalized in the DEUS context by introducing the concept of a Digital Card incorporating a viable information unit. The second type of artifact involved in information exchange are dossier files including the Personal Patient File and the Foreign Patient File. They will be renamed to reflect the domain-unspecific orientation of DEUS.

4.3.1 The Digital Card

Following the document-oriented principle, the artifacts of data interchange need to be self-contained and viable. DEUS introduces the term *Digital Card* incorporating the notion of the information unit as used in the previous chapter[1]. Digital Cards become the subject of interest of information exchange and constitute the smallest piece of patient-related data.

Their viability is reflected by a composite primary key, uniquely identifying each Digital Card: Since each Digital Card is created by a single Information Provider, the ID of the Information Provider is part of the primary key of a Digital Card. Each Digital Card contains information related to a single Concerned Person, thus, his/her ID is included in the primary key. Due to the possibility of an Information Provider contributing multiple Digital Cards about a Concerned Person, the third part of the primary key of a Digital Card is a discriminator chosen by the Information Provider. This context information semantically associates the Digital Card with actors and thus adheres to the notion of a context as described in [LBK07]. The model of the introduced Digital Card is depicted in the ERD in figure 4.2.

Concluding, a Digital Card represents a document as required by the document-oriented approach, although the intended granularity is a more fine-grained one than the one experienced in a paper-based environment. By carrying its own context information, a Digital Card is viable and can exist independently of the system it stems from.

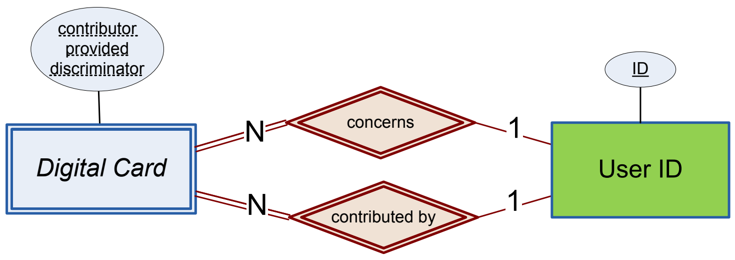


Figure 4.2: The data model of the introduced Digital Card

4.3.2 Dossier Files and Depository Folders

DEUS is agnostic of the problem domain by not being aware of the content of the transferred Digital Cards, which is also reflected by introducing generic distribution roles in section 4.2. This principle also drives a renaming of the artifacts introduced in section 3.3 by replacing the term 'Patient' with 'Information'.

1 The Digital Card metaphor has been inspired by the Higgins project [Ecl08] with its Information Cards as the foundation of an open identity framework.

Thus, a patient's health record, referred to as Personal Patient File in the previous chapter is renamed to *Personal Information File*. The replicated version of it on the subscriber side is called *Foreign Information File* in the following. These two artifacts are united under the term *dossier files*. Furthermore, the Distributed Patient Folder containing a list of Foreign Patient Files is renamed to *Distributed Information Folder* which is also referred to as *encompassing depository*.

The Personal Information File is uniquely determined by the ID of the user, that is associated with the information inside the record, who subsequently is the owner of the record. Since one user only owns a single Personal Information File, no further discriminating key is needed for this dossier artifact. It consists of a list of Digital Cards, which is outlined in figure 4.3. In the following, the key attributes of a Digital Card, depicted in figure 4.2, are omitted.

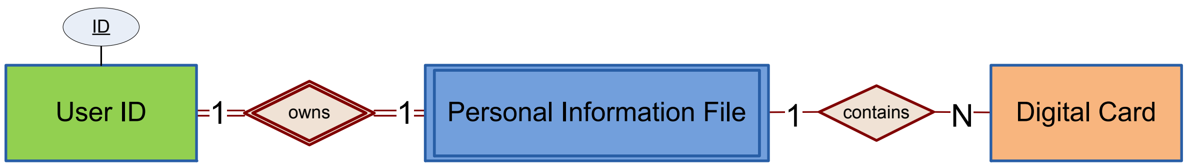


Figure 4.3: The model of the Personal Information File

Since the Foreign Information File is a replicate of the Personal Information File of a Concerned Person being held at the Information Consumer side, it also consists of a list of Digital Cards. It has a composite primary key, with the first component being the ID of the Concerned Person that is represented in the Foreign Information File. The second component is the primary key of the Distributed Information Folder that the Foreign Information File is contained in. The Distributed Information Folder in turn is identified by the user in the role of Information Consumer who owns it. Figure 4.4 outlines the model of the Foreign Information File and the Distributed Information Folder.

The Foreign Information File does not necessarily contain all the Digital Cards of the Personal Information File it mirrors but rather forms a subset of it. This subset is achieved by the Concerned Person selecting the subscribers to be notified of an update of the Personal Information File during publication. Thus, not all subscribers of the Personal Information File receive all Digital Cards. More on this can be found in section 4.4.3.

4.3.3 Types of Digital Cards

A Digital Card is an abstract concept providing only a container for arbitrary content to be included. Concrete types of Digital Cards can be derived that specify the type of embedded information. As described in section 1.3, a way to foster data integration in healthcare is the

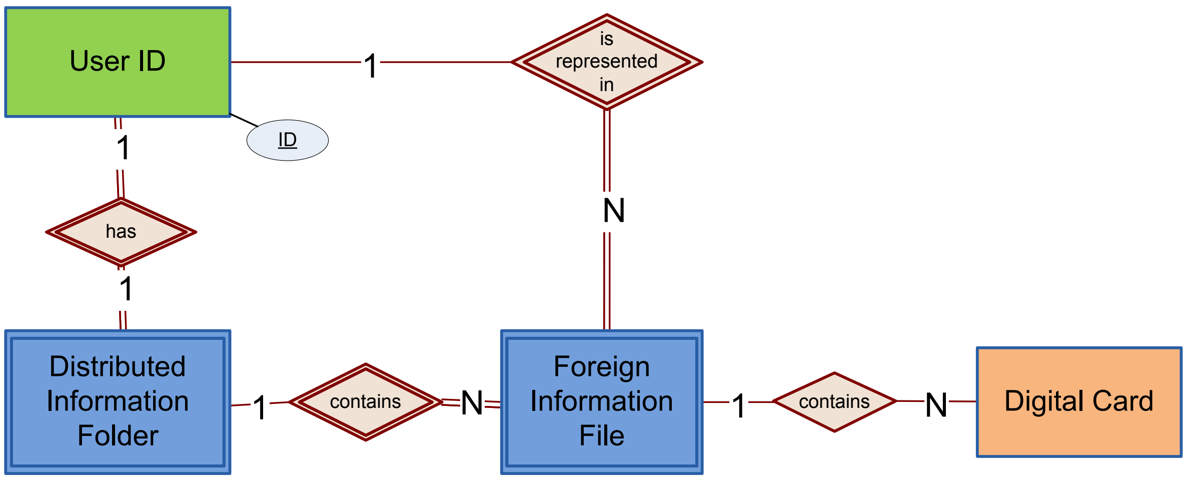


Figure 4.4: The model of the Foreign Information File and the Distributed Information Folder

interchange of documents conforming to the HL7 CDA standard. Therefore, DEUS is intended to provide a new transport infrastructure for CDA documents by exchanging Digital Cards adhering to this specification.

The focus of this thesis is on the design and architecture of a domain-unspecific system for information interchange and only to a lesser extent on the concrete type and semantics of exchanged patient-related data. Therefore, the specification of a Digital Card type containing complex CDA documents was deferred to the future. Instead, an exemplary type of Digital Card allows the incorporation of patient master data occurring in every health record. Existing standards for contact and address data are embraced with the most important and widespread one being *vCard* [DH98]. However, vCard offers no comprehensive data model, but only a list of attributes. Hence, a model for master data was created that allows for exporting address data confirming to the vCard standard. This data model is presented in section 6.3.

4.4 The DEUS Contribution-Repatriation-Publication Chain

In section 3.2, the flow of information in a basic interaction scenario was described. In the DEUS context, this information flow is called *Contribution-Repatriation-Publication Chain* and is outlined in figure 4.5.

The *contribution phase* of the chain involves bundling a self-contained information unit into a Digital Card, signing and importing it into DEUS. In the *repatriation phase*, this Digital Card is transferred to the account of the Concerned Person and the decision whether to accept the new information is posed to the patient. This transfer may involve communication over a

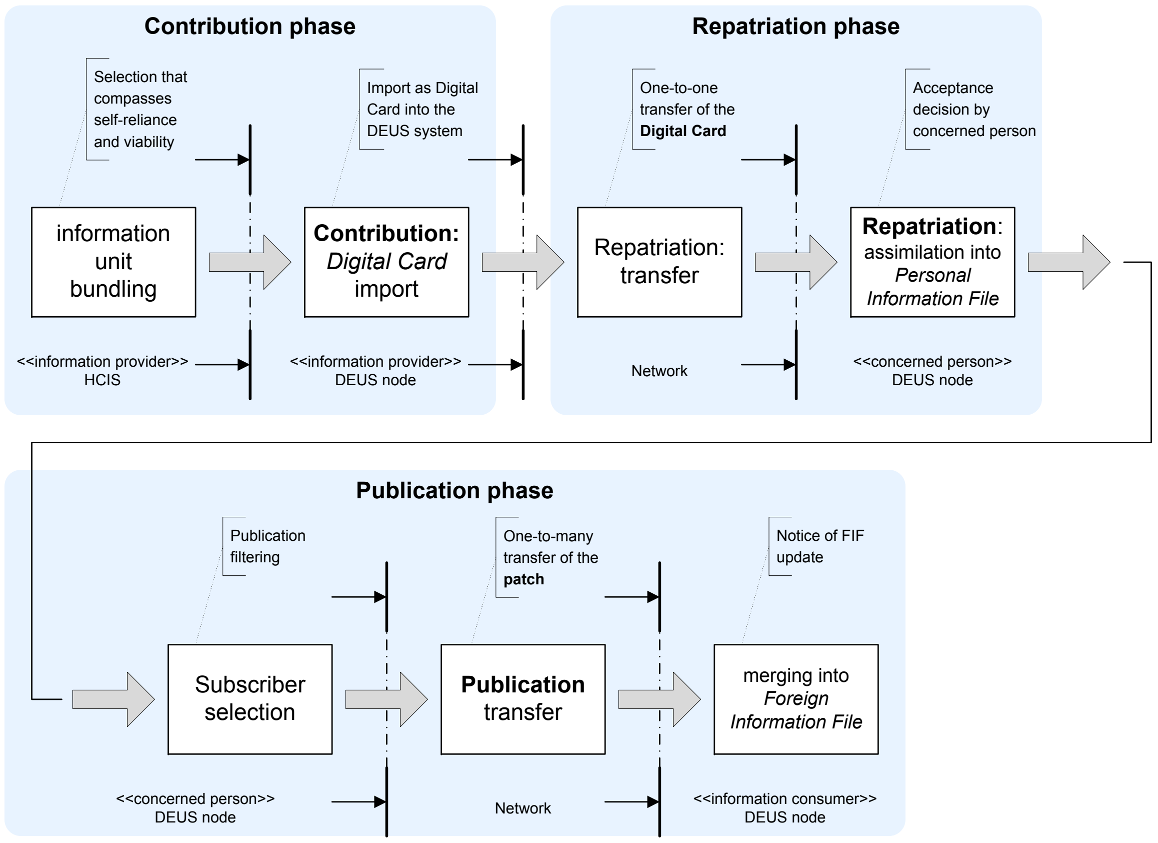

Contribution phase
Selection that compasses self-reliance and viability
Import as Digital Card into the DEUS system
information unit bundling
Contribution: Digital Card import
<<information provider>> HCIS
<<information provider>> DEUS node
Repatriation phase
One-to-one transfer of the Digital Card
Acceptance decision by concerned person
Repatriation: transfer
Repatriation: assimilation into Personal Information File
Network
<<concerned person>> DEUS node
Publication phase
Publication filtering
One-to-many transfer of the patch
Notice of FIF update
Subscriber selection
Publication transfer
merging into Foreign Information File
<<concerned person>> DEUS node
Network
<<information consumer>> DEUS node

Figure 4.5: The DEUS Contribution-Repatriation-Publication Chain

network that follows the one-to-one principle. The prior setup of a trust relationship between the Information Provider and the Concerned Person is elaborated on in section 4.4.1.2. If the patient acknowledges the repatriated Digital Card, it is assimilated into the Personal Information File. Different strategies for handling the merging semantics are described in section 4.4.2.

Subsequently, the new state of the dossier is announced in the *publication phase*. In section 4.4.3, the selection of subscribers who receive the change is explained. An unidirectional synchronization updates the local copy of each selected subscriber. This is further elaborated on in section 4.4.4. The potentially involved network communication follows a one-to-many paradigm, thus posing other requirements to the underlying network protocol than the repatriation transfer described above. The prior setup of a trust relationship between the Concerned Person and the Information Consumer is described in section 4.4.1.3.

4.4.1 Setup and Teardown of Trust Relationships

Before any communication between DEUS actors occurs, trust relationships for the repatriation and the publication phase have to be built up. Prior to repatriating a Digital Card, a *repatriation trust relationship* between the Information Provider and the Concerned Person is established. Prior to publishing changes, a *publication trust relationship* between the Concerned Person and the Information Consumer is built up. Using the notion of a 'trust relationship' reflects its conceptual nature. In the following, the term 'trust connection' is used synonymously. However, these connections don't establish low level communication channels but are rather used to avoid authentication prior to each subsequent repatriation or publication of a Digital Card.

4.4.1.1 Classification of User Accounts in the Context of Trust Relationships

Each DEUS user owns an account that assumes different distribution roles and is identified by a unique ID. From these distribution roles functional roles were deduced[1]. While trust relationships are conceptually established between distribution roles, the logical view of a relationship connects two accounts assuming functional roles.

In the following, establishing trust relationships is regarded from the perspective of a single user account that is classified as 'hosted'. While this account manages trust relationship artifacts, other accounts are only referenced by their user ID and thus classified as 'peers'. This incorporates the first dimension of a two-dimensional classification of user accounts outlined in figure 4.6 where an account is disjointly classified as peer or hosted account. The second categorization reflects the decomposition into functional roles. Since an account can potentially assume multiple functional roles, the resulting specialization is overlapping. In the following consideration about trust relationships, the two classifications of a user account are reflected by the name of an account entity.

4.4.1.2 Repatriation Relationship

A *repatriation relationship* is conceptually established between an Information Provider and a Concerned Person for authenticating the source of a repatriation. Logically, the relationship is built up between two accounts assuming the functional roles Contributor and Repatriation Authority. This is outlined by details of figure 4.1 being displayed in figure 4.7.

Assuming the functional role Repatriation Authority, a hosted DEUS account manages a list of authenticated contributors outlined in figure 4.8. An entry in this list contains describing

1 See section 4.2 for more about the relation between distribution and functional roles.

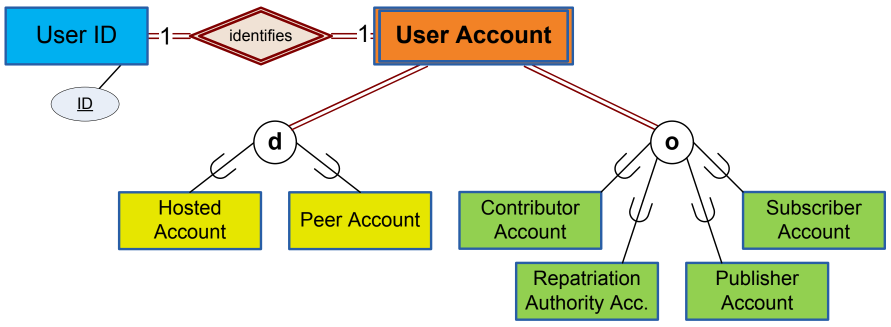


Figure 4.6: The two-dimensional classification of a user account

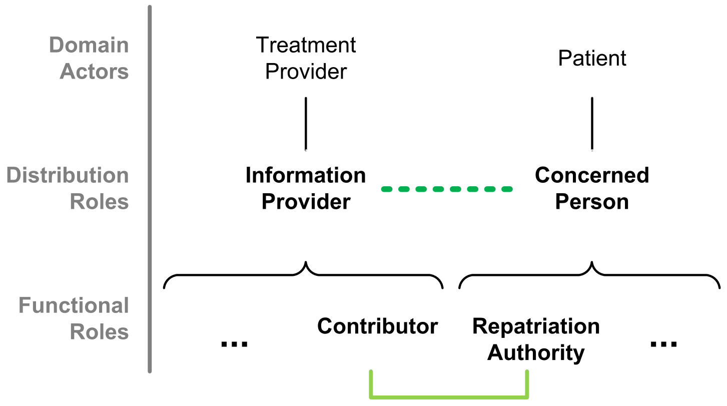


Figure 4.7: A repatriation relationship connecting the functional roles Contributor and Repatriation Authority

metadata, including the full name and gender of the contributor. This is needed to display list entries, containing the contributor's name, while the gender is used to adapt grammar. If a PKI[1] is used in order to certify communication endpoints and actors, this list will contain any attributes related to the PKI. The primary key of each list entry is composed of the primary key of the list and the primary key of a peer user account taking the role of Contributor.

If a Digital Card is received in the repatriation phase, it is checked whether the originator of the Digital Card is contained in the list of contributors. Two different *trust models* can be thought of: The more restrictive one drops all Digital Cards received by contributors not included in the list. However, if a repatriation relationship is established, a decision is presented to the user whether to accept the repatriated Digital Card. The second, less restrictive approach would automatically accept Digital Cards from the listed contributors and only display a decision to the user if the originator is unknown. Establishing repatriation trust

1 Public Key Infrastructure

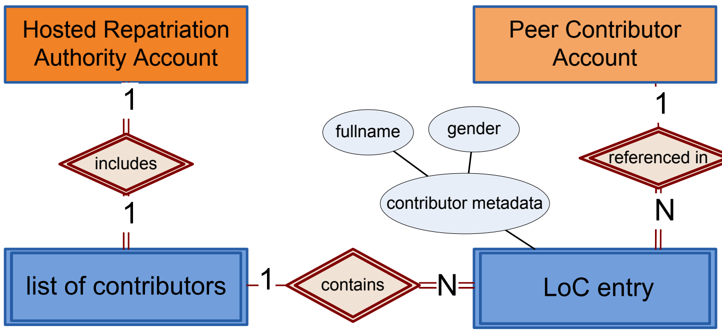


Figure 4.8: The list of contributors and their entries

connections thus prevents malicious parties from injecting false information into the Personal Information File without being noticed by the Concerned Person.

Furthermore, for not requiring the Information Provider to wait for the user accepting the repatriation relationship request, the contributed Digital Card may be piggy-backed onto the request. On confirmation of the trust connection by the Concerned Person, the Digital Card is instantly processed according to the chosen trust model.

As a first approach, the latter trust model is implemented. Establishing repatriation relationships is not provided, so that contributed Digital Cards result in always being displayed to the Concerned Person for decision.

4.4.1.3 Publication Relationship

A *publication relationship* is conceptually built between a Concerned Person and an Information Consumer while it logically connects two accounts assuming the functional roles Publisher and Subscriber. This is outlined by details of figure 4.1 being displayed in figure 4.9. While it coincides with ‘subscription’, the term ‘publication relationship’ is preferred in the following.

Establishing or terminating a publication relationship can be initiated by either party. The used terminology around the establishment and termination of publication connections is outlined in figure 4.10. Different use case names entail different conceptual messages to be sent to the communication counterpart. Messages for connection establishment are pleas requiring a decision. Different verbs are chosen for positively or negatively deciding on pleas on the publisher and subscriber side in order to clearly distinguish the two use cases.

If an Information Consumer requires access to the Personal Information File of a Concerned Person, he/she subscribes to a Publisher. Subsequently, a plea is displayed to the user of the Publisher account which requires the user to either *grant* or *deny* the request. The decision is sent to the Subscriber and in case the plea is granted, the connection is established on both

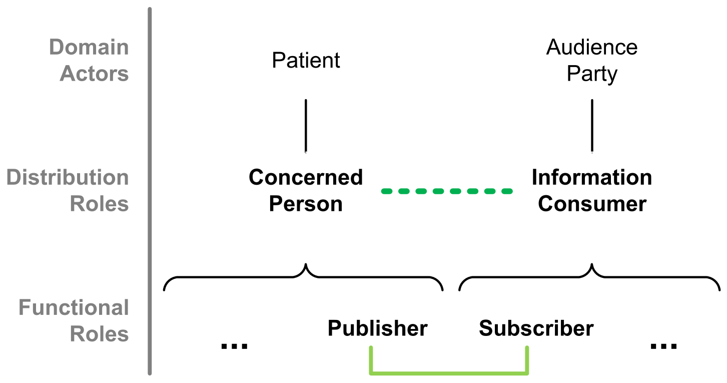
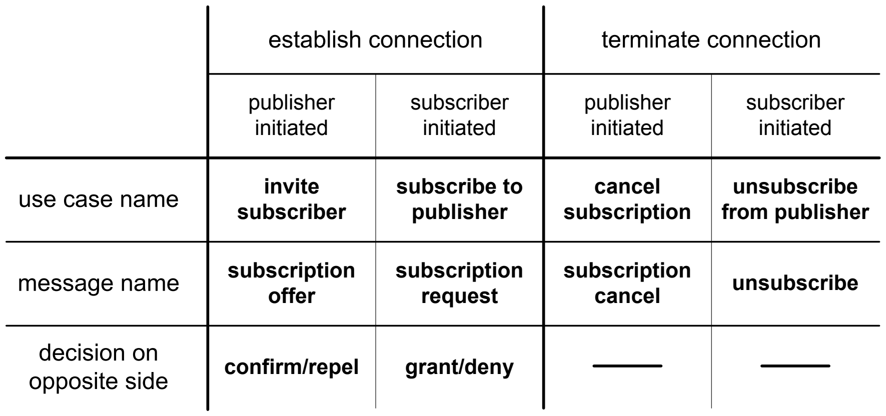


Figure 4.9: A publication relationship connecting the functional roles Publisher and Subscriber



	establish connection		terminate connection	
	publisher initiated	subscriber initiated	publisher initiated	subscriber initiated
use case name	**invite subscriber**	**subscribe to publisher**	**cancel subscription**	**unsubscribe from publisher**
message name	**subscription offer**	**subscription request**	**subscription cancel**	**unsubscribe**
decision on opposite side	**confirm/repel**	**grant/deny**	——	——

Figure 4.10: Terminology around the establishment and termination of pub/sub connections

sides. This involves adding the Subscriber to the *list of subscribers* on the Publisher and adding the Publisher to the *list of publishers* on the Subscriber side. Using the first list, outlined in figure 4.11, a Publisher knows whom to send updates of the Personal Information File to. The second artifact, outlined in figure 4.12, is used to whitelist incoming Personal Information File changes to only accept update messages from Publishers, to whom the user is subscribed to. Both lists are identified by the primary key of the account they are hosted with. Any list entry contains describing metadata of the referenced user account. An entry is identified by the list it is contained in and by the primary key of the peer user account it represents.

If the Concerned Person wants an Information Consumer to receive updates of his/her Personal Information File, he/she initiates a connection by inviting a Subscriber. Subsequently, a plea is displayed to the user of the Subscriber account, which requires the user to either

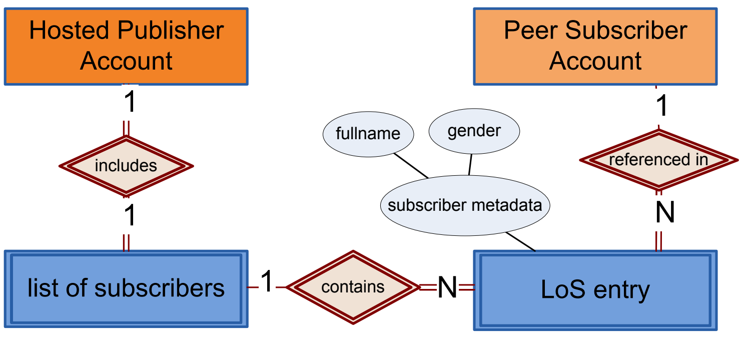


Figure 4.11: The list of subscribers

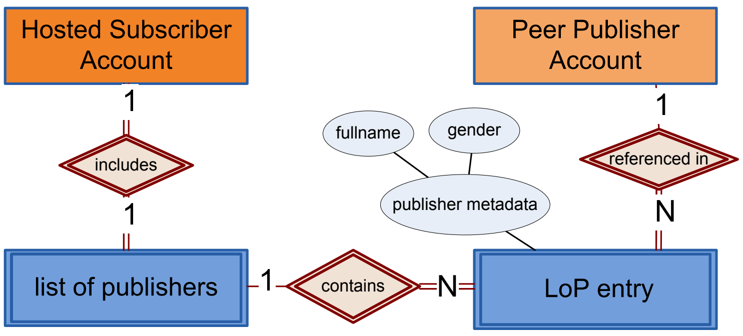


Figure 4.12: The list of publishers

confirm or *repel* the offer. The decision is sent to the Publisher and in case the plea is confirmed, the connection is established as described above.

In parallel to establishing a connection, termination can be initiated by either the Subscriber or the Publisher side. If an Information Consumer no longer requires access to the Personal Information File of a Concerned Person, he/she *unsubscribes from the Publisher*. Subsequently, a message is displayed to the user of the Publisher account, which notifies the user of the termination of the subscription relationship. The Subscriber is removed from the list of subscribers on the Publisher side as well as the Publisher is removed from the list of publishers on the subscriber side. Furthermore, the Foreign Information File related to the Concerned Person, assuming the Publisher role, is removed on Subscriber side.

If a Concerned Person decides to exclude an Information Consumer from future updates, and furthermore, demands the deletion of the Foreign Information File, he/she cancels the publication relationship. Subsequently, a message is displayed to the user of the Subscriber account, which notifies the user of the termination of the publication connection and the deletion of the Foreign Information File. The connection is torn down as described above.

4.4.2 Repatriation Semantics

If a new Digital Card is contributed by an Information Provider, it is transferred to the Concerned Person and a plea for accepting this Digital Card is displayed to the user. If the user accepts the repatriated Digital Card, it has to be added to the user's Personal Information File. Several strategies can be applied to assimilate the contributed Digital Card.

Associating Digital Cards with others allows for enriching the Personal Information File with links between Digital Cards carrying distinct semantic information. IHE XDS therefore defines *document relationships* [ACC07, 10.4.10.2]. Different types of relationships associate documents with folders and XDS submission sets and furthermore provide support for versioning. The latter option allows to mark a new document as an extension of another one (association type APND), as a replacement (association type RPLC), or as a transformation (association type XFRM or XFRM_RPLC).

Following this model, an assimilation strategy may create links between existing Digital Cards and new ones. Thus, adding a repatriated Digital Card to the Personal Information File results in potential changes not necessarily limited to appending a single Digital Card, but rather encompassing larger parts of the Personal Information File. These changes are incorporated in a *patch*[1] that manifests the difference between the old and the new state of the Personal Information File. To automate merging of new Digital Cards, extended knowledge about the content including status information, existing ontologies classifying contained data, or describing attributes is required.

Currently, DEUS neither supports any associations between Digital Cards nor has knowledge about their content. Thus the repatriation semantics narrow down to assimilation strategies that simply append a new Digital Card to the Personal Information File. If there is an existing Digital Card with the same primary key as the contributed one, an exception is thrown.

4.4.3 Publication Filtering

One issue in the publication phase is the filtering of new information that should be sent to the subscribers. Obeying the principle of minimal knowledge, each subscriber should only receive required information. Since publishing new information to all subscribers of the Personal Information File contradicts this principle, a filtering mechanism should be employed. Nonetheless, Digital Cards are always shared as a whole and not filtered internally since the inner structure is not necessarily known to the system. Furthermore, changing or filtering the

1 The notion of a patch originates from the Unix diff and patch tools.

content of a Digital Card would invalidate the existing signature issued by the Information Provider.

Resulting from the collaborative scenario described in section 3.4, DEUS should support the creation of groups of subscribers. Thus, filtering is no more applied subscriber-individually but rather for a group, potentially only encompassing a single subscriber. The allocation to a subscriber group can be demanded by an Information Consumer during subscription request. The patient decides about this demand and subsequently associates the subscriber with one or more subscription groups. However, the current implementation of DEUS does not support groups of subscribers.

Filtering rules associate a repatriated Digital Card, being classified by its contained context, with subscriber groups to which the Digital Card is published. Two different approaches for this are known from existing publish/subscribe-systems: channel-based vs. content-based [EFGK03].

The *channel-based* subscription scheme, also known as topic-based, provides channels, being identified by a keyword or 'topic'. Publication to a channel involves broadcasting the event to all subscribers of this channel. Extensions of this mechanism include a hierarchical organization of the keywords using containment relationships. Furthermore, the use of wildcards supports the subscription to a collection of channels. In the DEUS context, this would require a predefined hierarchy of keywords which provide a classification of the content of repatriated Digital Cards. On accepting a Digital Card, it is classified according to the hierarchy and published to the subscribers that subscribed to this channel. Subscriber groups as mentioned above can be realized by a topic-based subscription scheme.

In contrast, the *content-based* subscription scheme, also known as property-based, uses the actual content of an event for classification. This may include any internal attributes of data structures as well as associated metadata. Filters following a name-value-operator syntax may be specified to subscribe to selected events. Logical operations may connect filters to form composite ones. In the DEUS context, this would allow repatriated Digital Cards to be published after acceptance without the user being involved in classification. This either requires a common set of attributes accompanying each Digital Card or the definition of executable predicate objects. These are defined by the Concerned Person and associated with subscriber groups. For each repatriated Digital Card the predicate objects are applied to it, in order to check if the Digital Card should be published to this subscriber group.

The current state of the DEUS system does not involve any publication filtering but rather distributes incoming Digital Cards to all subscribers in the same way. Implementing the sketched ideas requires more conceptual work on what a subscriber group is, who knows of subscriber groups existing on different accounts, and how a classification of incoming Digital Cards can be achieved.

4.4.4 Synchronization

Assimilating a repatriated Digital Card into the Personal Information File results in a patch as described above. This patch potentially incorporates changes of the Personal Information File not limited to a single Digital Card but encompassing larger parts of it. These changes need to be sent to the Information Consumers which constitutes a one-way synchronization in order to keep consistency of the Foreign Information Files. This issue is closely connected to replication techniques used in distributed systems and is elaborated on in [Len97]. Replication is furthermore only guaranteed, if the transfer mechanism provides guaranteed delivery [HW03, p. 122].

As elaborated in section 4.4.2, the current implementation just appends repatriated Digital Cards to the Personal Information File. Thus, a patch only manifests the addition of a single Digital Card which confines the synchronization to a publication of the newly added Digital Card.

4.4.5 Initial Publication

After a pub/sub relationship is established, an initial publication can be accomplished to send historic Digital Cards to the new Information Consumer. A decision needs to be made about which Digital Cards of the Personal Information File are encompassed in this initial publication. One strategy requires the user to once select a set of Digital Cards that are initially published to each new subscriber. This may, for example, include Digital Cards containing master data and basic medical information like allergies. Another option would be to require the Concerned Person to individually select the Digital Cards to initially publish for each established publication relationship. If the subscriber is added to a publication group after connection setup, a third option would be to automatically publish all Digital Cards formerly sent to this group. This involves tracking of the Digital Cards published to each publication group. The current implementation of DEUS does not initially publish anything at all after setup of a publication relationship.

4.5 The Collaborative Scenario

The collaborative scenario described in section 3.4 can also be supported by DEUS. Since patient interaction should be dropped here, the account of the Concerned Person is trimmed down. No user interaction during the repatriation and publication phases is needed any more; The account only exists for a logical purpose. Repatriated Digital Cards are automatically accepted, provided that the repatriation originates from an Information Provider being in a list of certified contributors. Also the list of Information Consumers is preconfigured and new, repatriated information is published to all Information Consumers in the same way without filtering. The list of certified Information Providers and Information Consumers is injected into the specially configured DEUS node during account generation. This account can, for example, be hosted by the breast-cancer treatment center. Using this approach, information about a treated patient can be distributed without the patient involved in decision processes.

4.6 Further Issues

Since DEUS accounts need to be addressed, an identifier scheme needs to be found which fulfills certain conditions. Information exchange between accounts residing on different DEUS nodes requires network communication. The instrumented protocols need to adhere to several requirements subsumed below. Another important issue to solve is to guarantee the identity of a communication counterpart by an authentication process.

4.6.1 Identifier Schemes

To enable communication between DEUS accounts, each account needs to be addressable. Thus, an *identifier scheme* needs to be found, that fulfills the following objectives:

1. Global uniqueness
2. Available metadata discovery mechanism
3. No dependency on a specific communication protocol
4. Maturity and widespread deployment

Since DEUS is distributed, accounts may reside on different nodes and thus need to be globally addressable with a unique ID. An established discovery protocol should be available, so that metadata about the ID and its associated resources as well as related services can be retrieved. The objective of abdicating any central infrastructure requires the identifier scheme to neither need a central instance for discovery nor for establishing the global uniqueness. Furthermore, the protocol should be agnostic of any specific communication protocol or at least not favour one in a manner that makes integration of other communication protocols difficult. The last objective is the maturity of the identifier that may be indicated by a widespread deployment of it. Two alternatives for identifier schemes fulfilling the posed requirements are evaluated in section 5.1.

4.6.2 Communication Protocols

For the whole described flow of information and the setup and teardown of trust relationships, data needs to be exchanged between different DEUS accounts. Due to the distributed nature of DEUS, this data exchange may result in communication between physical nodes over network links. Thus, network communication protocols are required that should particularly foster the principle of low coupling. A common paradigm for communication between loosely-coupled endpoints is *asynchronous message passing*. This provides decoupling of the communication entities in time, space, and synchronization [EFGK03]. Thus, used communication protocols should basically reflect the message-oriented paradigm.

Since DEUS should not be bound to specific technologies, it should not rely on a specific communication protocol. Rather, bindings to communication protocols, that fulfill a common set of prerequisites, should be specified. These bindings map the messages needed for establishment and termination of trust relationships and sending of Digital Cards to protocol specific methods. The requirements to which instrumented communication protocols must adhere sum up to the following points:

1. *Guaranteed Delivery* [HW03, p. 122] of messages, taking into account that DEUS nodes may be down over a certain period of time
2. High abstraction level to support the mapping of DEUS high level concepts like users, their accounts, subscriptions, and message routing

Furthermore, the repatriation transfer follows a one-to-one paradigm that must be supported by offering a *Point-to-Point Channel* [HW03, p. 103]. The publication transfer follows a one-to-many paradigm that must be supported by offering a *Publish-Subscribe Channel* [HW03, p. 106]. This may result in different protocols being instrumented in both phases of the described information flow. However, both phases require guaranteed delivery of messages and should provide the introduced high level concepts. Protocols complying to the above requirements are described in section 5.3.

4.6.3 Authentication

Furthermore a secure authentication mechanism is needed that guarantees the trustful authentication of the different users. This can either be provided by used communication protocols or it needs to be specified on top of them. Authentication mechanisms in healthcare information systems are out of the scope of this thesis. DEUS therefore does not yet provide authentication mechanisms. Further evaluation of the requirements of authentication in healthcare information systems and existing standards is needed.

4.7 Conclusion

The proposed system for exchanging patient-related information is called DEUS. It follows document-oriented principles by introducing the notion of a Digital Card which incorporates the subject of information exchange. Since DEUS is independent of the type of information contained in Digital Cards, terminology around actors, roles, and artifacts is abstracted to reflect the domain-unspecific orientation. As a result, DEUS can be used in arbitrary information distribution scenarios that require a mediated publish/subscribe approach. This architecture and the basic flow of information is described as Contribution-Repatriation-Publication Chain. As a summary figure 4.13 outlines the artifacts including their association to phases in the information flow. Domain-specific terms are opposed to generic terms, hence, reflecting the introduced generic abstraction layer.

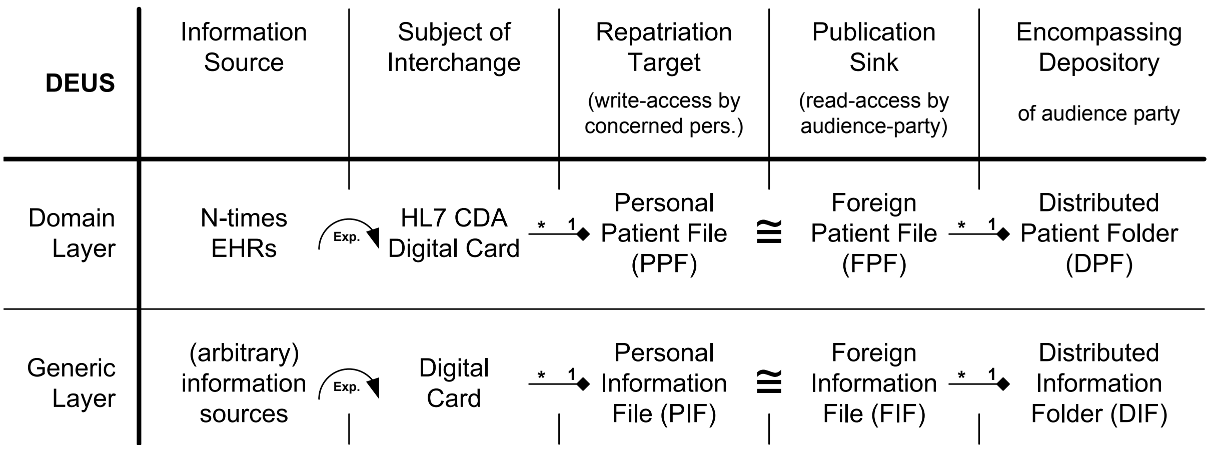

DEUS
Information Source
Subject of Interchange
Repatriation Target (write-access by concerned pers.)
Publication Sink (read-access by audience-party)
Encompassing Depository of audience party
Domain Layer
N-times EHRs
Exp.
HL7 CDA Digital Card
* 1
Personal Patient File (PPF)
≅
Foreign Patient File (FPF)
* 1
Distributed Patient Folder (DPF)
Generic Layer
(arbitrary) information sources
Exp.
Digital Card
* 1
Personal Information File (PIF)
≅
Foreign Information File (FIF)
* 1
Distributed Information Folder (DIF)

Figure 4.13: Domain-specific artifacts opposed to generic DEUS artifacts in the context of the Contribution-Repatriation-Publication Chain

In the contribution phase, information originating from arbitrary sources is bundled and imported into DEUS as a self-contained, viable Digital Card. Subsequently, this Digital Card is repatriated by transferring it to the account of the Concerned Person and assimilating it into the Personal Information File. The new information is published to subscribers where it is collected in a Foreign Information File. The Foreign Information Files of all Concerned Persons, a subscriber declared interest in, are encompassed in the Distributed Information Folder.

This chapter has provided comprehensive considerations of possible features of a system to support seamless flow of patient-related information. An initial reference implementation supports all phases of the Contribution-Repatriation-Publication Chain including all introduced artifacts. However, not all described concepts are implemented yet, though they were considered during the design phase. DEUS currently supports only a simple trust model for the repatriation phase while trust connections in the publication phase are fully implemented. With the existing implementation, assimilating a repatriated Digital Card into the Personal Information File confines to simply appending it. Thus, the publishing of dossier state changes narrows down to transferring the newly added Digital Card to subscribers. No concept of subscription groups nor initial publication of historic Digital Cards is yet implemented. The collaborative scenario was regarded during DEUS system design. However, implementing it would require a feature to inject lists of subscribers and trusted contributors.

5 Fundamentals

DEUS user accounts must be addressable by providing identifiers that fulfill the requirements described in section 4.6.1. In the following, with URI[1] and XRI[2], two types of identifiers are introduced. If a user obtains an ID of another DEUS user, it should be possible to discover metadata like available communication channels. Therefore, a stack of discovery protocols around XRD[3] is described. For DEUS inter-node communication, communication protocols are needed that fulfill preconditions introduced in section 4.6.2. Hence, REST[4] as an architectural style and XMPP[5] are elaborated, that both match the requirements to transfer protocols to a certain degree. Subsequently, a short overview of JMS[6] as another possible communication protocol is given.

5.1 Addressing Schemes and Identifiers

Two alternative identifier schemes to address DEUS accounts are examined, fulfilling the objectives described in section 4.6: URI and XRI.

5.1.1 URI

From the very beginning of the Internet, a URI has been the central identifier for any addressable resource. A URI consists of a string of characters that follows a URI scheme and identifies a single resource. This scheme defines the syntax of a URI as `<URI scheme name> + ":" + <scheme-specific part>`. URIs can be classified as either URL[7], URN[8], both or none of them.

A URN defines the identity of a resource and is often compared to the name of a person. It is agnostic of the location of a representation of the identified resource and provides no way to access this representation. The syntax of a URN is `"urn:" + <namespace identifier> +`

1 Uniform Resource Identifier
2 Extensible Resource Identifier
3 Extensible Resource Descriptor
4 Representational State Transfer
5 Extensible Messaging and Presence Protocol
6 Java Message Service
7 Uniform Resource Locator
8 Uniform Resource Name

`":" + <namespace-specific part>` with the URI scheme name `"urn"`. A famous example of a URN is `urn:isbn:0-486-27557-4`, belonging to the `isbn` namespace and identifying a certain book[1]. This URN can be used to uniquely identify the book, but it provides no location information about where and how to retrieve a representation of it.

On the other hand, a URL provides a method for retrieving a representation of a resource and is often compared to a street address of a person. A URL is a URI that 'in addition to identifying a resource, provides a means of locating the resource by describing its primary access mechanism (e.g., its network "location")'[BLFM05]. The syntax of a URL is `scheme-name://domain:port/filepathname?query-string#anchor`. The most important part is the scheme name that defines the syntax of the remaining part and the semantics around retrieval. Dereferencing a URL applies scheme-specific methods to retrieve the resource representation. An example for a URL is `http://www.gutenberg.org/dirs/etext04/8gs1610.txt`[2] that retrieves a representation of the specified resource using the HTTP[3] protocol.

A URL using the HTTP scheme[4] for the most part conforms to the objectives as outlined above. It is widely deployed and globally unique since the registration of domain names, that constitute a part of a URL, follow a unified process. While it is specific to a communication protocol, namely to HTTP, this can be tolerated since HTTP is lightweight. With the discovery stack around XRD, protocols are available to obtain metadata of a resource. More about this can be found in section 5.2. Furthermore, due to the spreading of OpenID[5], an URL gains publicity as identifier for persons.

5.1.2 XRI

A new effort in the area of universal identifiers has been made by the invention of XRIs [RM05]. The goal is to create abstract, structured identifiers that are domain-, location-, application- and transport-independent. The standard is developed under the hood of OASIS[6] and comparable to XML's role in the world of data formats, where XML constitutes a domain-unspecific, application-independent, and self-describing data format. Self-description of XRIs is achieved using the cross-reference feature of XRI that allows the nesting of XRIs using `'()'`. Thus structured identifiers are obtained, that are self-describing using tags that themselves constitute XRIs. 'Global context symbols' provide a human-friendly way to indicate the global context of an XRI.

1 The example is an edition of 'Romeo and Juliet' of William Shakespeare
2 It is a text document containing William Shakespeare's 'Romeo and Juliet'.
3 Hypertext Transfer Protocol
4 In the following, the term URL implies that the used scheme is HTTP.
5 http://www.openid.net
6 Organization for the Advancement of Structured Information Standards

If an XRI is prefixed with `'='`, it identifies a person, `'@'` prefixes XRIs identifying companies or organizations, and an XRI with `'+'` at the beginning denotes a generic concept, like a tag. Examples of XRIs are:

- `=alice`
- `=smith*alice`
- `@Acme`
- `+flower`
- `+flower*rose`
- `=Mary.Jones*(+phone.number)`

A `'*'` character denotes the delegation of resolution to another resolver, like a dot in a domain name denotes the delegation to another DNS[1] authority. XRI resolution is decentralized using 'i-brokers' and can be done in a peer-to-peer way by two institutions assigning an XRI to each other without other parties being affected. Parts of an XRI can be marked as persistent by prepending `'!'` to indicate that this part will never be reassigned. Human-friendly XRIs ('i-names') are mapped to machine-friendly XRIs ('i-numbers') during resolution. The resolution of XRIs is done using the XRDS[2] protocol.

An XRI is globally unique and a metadata discovery and resolution protocol is available with XRDS. It is independent of a specific communication protocol since XRIs are abstract. However, the drawbacks of XRI are its lack of widespread deployment and maturity together with its complexity.

5.2 Discovery

According to [HL08a], discovery is 'the process, in which machines learn how to interact with other machines'. The question to be answered during discovery is not 'Teach me how to talk to you' but rather 'Which of the languages I know do you understand?' The process is analogous to two people meeting, where they try to figure out a common language, both of them are speaking.

Discovery can further be distinguished into descriptor and service discovery. *Descriptor discovery* tries to answer the question about the attributes of a given resource, identified by a given ID, including capabilities, characteristics and relationships to other resources. The

1 Domain Name System
2 Extensible Resource Descriptor Sequence

opposite is achieved using *service discovery*, where a set of attributes is given and a resource has to be located, that matches the given set of attributes [HL09a, Appendix A]. The two types of discovery are closely related, and a typical discovery use case involves both. The focus is on descriptor discovery first since the typical DEUS use case reflects the need for more information about a user given the user's ID. The process of descriptor discovery consists of three phases:

1. Locate the descriptor for a given ID
2. Obtain the descriptor
3. Interpret the descriptor

These three phases form a protocol stack that is outlined in figure 5.1.

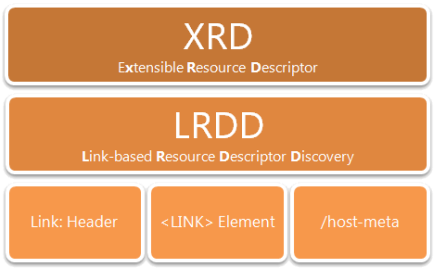

XRD
Extensible Resource Descriptor
LRDD
Link-based Resource Descriptor Discovery
Link: Header
<LINK> Element
/host-meta

Figure 5.1: The discovery protocol stack around XRD ([HL09b])

Focusing on discovery over HTTP, step 2 is rather simple and just consists of issuing an HTTP `GET` to retrieve the descriptor document. For step 1, several methods for locating the descriptor document using HTTP have been analyzed in [HL08b]. This analysis resulted in a description discovery standard called LRDD[1] [HL09a] bundling three methods of locating the descriptor to a 'link framework' [HL09b]:

1. HTTP `Link:` header [Not09]
2. HTML[2] `<link/>` element [RLHJ99, 12.3]
3. host-meta [NHL09]

The first method obtains the address for the descriptor document by instrumenting the `Link` header of the HTTP response packet that is returned after issuing an HTTP `GET` respectively an HTTP `HEAD` to the resource. The second method obtains the descriptor document by following the HTML `<link/>` element. The 'host-meta' approach locates the descriptor by retrieving a document from a well-known location, namely `/host-meta`. There, authority-wide metadata is

1 Link-based Resource Descriptor Discovery
2 Hypertext Markup Language

stored including a link to the descriptor document. The relationship type which is used for specifying the link type for all of the above methods is `describedby`.

An upcoming format for the descriptor document is XRD which is outlined in the following exemplary document:

```
<XRD>
    <Subject>http://www.deus.com/alice</Subject>
    <Alias>http://www.deus.com/alices-alias</Alias>

    <Expires>2010-01-30T09:30:00Z</Expires>

    <Type>http://www.deus.org/type/cp</Type>
    <Type>http://www.deus.org/type/ip</Type>

    <Link>
        <Rel>master-data</Rel>
        <URI>http://www.deus.com/alice/master-data</URI>
        <MediaType>text/xml</MediaType>
    </Link>
</XRD>
```

The `<Subject/>` element contains the ID of the resource, that is described in the descriptor document, with multiple possible aliases given by `<Alias/>` elements. `<Expires/>` contains an expiration date of the descriptor, which should match the expiration date of the related HTTP header. Multiple `<Type/>` elements can be used to describe the resource itself by listing URI-formatted type tags. An arbitrary number of `</Link>` elements contain links to associated resources or services. XRD was derived from XRDS by simplification, renaming of elements, and adding the facility for self-description using `<Type/>` elements. More on the history of various discovery formats around XRDS can be found in the specification of XRDS-Simple [HL08c].

5.3 Instrumented Communication Protocols

As it was elaborated in section 4.6, used communication protocols have to fulfill certain requirements. Two protocols are introduced, that are candidates for being instrumented as transfer protocols. A REST binding provides support for a point-to-point channel and may be used for transfer in the repatriation phase. An XMPP binding is offered that particularly supports publish/subscribe communication.

5.3.1 REST

Roy Fielding, one of the authors of the HTTP specification, specified a principal architectural style of hypermedia system in [Fie00]. This is called Representational State Transfer and systems adhering to this architecture are called RESTful.

REST introduces the concept of a resource as the central subject of interest, being uniquely addressable. These resources represent data being exchanged, as well as application state. All resources share a common, minimal, well-defined interface that allows the transfer of resources and the incorporated application state to a client. While a resource is an abstract concept, only representations of a resource are exchanged. These representations are identified by a well-defined set of content types. The basic exchange protocol is stateless and follows the client-server paradigm. The name Representational State Transfer reflects the concept of transferring state in form of representations of abstract resources.

The most prominent example of a RESTful architecture is the WWW[1]. Its overwhelming success and its scalability can be attributed to the key principles of a RESTful design. Each resource on the Internet is identified by a URI. Different content types of resource representations are specified by MIME[2] types with some of the most important being `text/plain`, `text/html`, `application/xhtml+xml`, `application/xml`, and `application/json`. A certain representation of a resource can be retrieved by a process called ‘Content Negotiation’ that allows the client to specify accepted content types. The used protocol is HTTP, offering only a small set of methods, including `POST`, `GET`, `PUT`, and `DELETE`. These methods can be associated with the operations `CREATE`, `READ`, `UPDATE`, and `DELETE` (`‘CRUD’`) known from database technology. The HTTP specification states, that `POST` creates ‘a new subordinate of the resource identified’ [FGM+99, p. 54]. On the other hand, `PUT` is intended to create a new resource ‘stored under the supplied Request-URI’. If a resource identified by the given URI already exists, HTTP states, that ‘the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server’ [FGM+99, p. 55]. HTTP is stateless and follows the client-server principle, with the client being a user-agent and the server being a HTTP web server. HTTP also supports the signaling of errors by its ‘status codes’.

Instrumenting HTTP and following the architectural style of REST, a web service can be offered that can provide arbitrarily complex use cases. Concepts of the problem domain are mapped to patterns of URIs. Using these URIs, HTTP can be instrumented to exchange resource representations and thus modify application state. An example would be an application offering

1 World Wide Web
2 Multipurpose Internet Mail Extensions

the URI `http://www.example.org/orders`. Issuing an HTTP `GET` to this URI would retrieve a list of all orders. `POST` would add a new order, a presentation of which is passed in the body of the HTTP message. Special orders could be retrieved by using the URI template [GHNO06] `/orders/{id}` and issuing a `GET` to it. `PUT` would be used to add a new order or update an existing one, while `DELETE` removes an existing order. The basic difference between `POST` and `PUT` can be seen here: `POST` yields the choice of the ID of the new resource to be added to the server, while `PUT` specifies an ID to be used.

DEUS may offer a RESTful API[1] to support the transfer of messages between DEUS peers. The advantage of a RESTful API is the wide-spread use of the underlying HTTP protocol that allows for the usage of existing infrastructure. HTTP supports a point-to-point communication channel as required, while delivery of messages cannot be guaranteed. Thus, an additional queue for unsent messages on sender side needs to be implemented. Messages that cannot be delivered to communication partners are appended to this queue and resending is tried periodically. Furthermore, no high level concepts are encompassed by HTTP that support the notion of users, accounts or subscriptions.

5.3.2 XMPP

The XMPP protocol suite came out of the *Jabber* instant messaging and presence protocol and constitutes its core protocol. XMPP is XML-based, follows an open development model, and allows for simple extensibility using XML namespaces. No central instance is in control over the whole system since it follows a distributed paradigm. It is specified in RFC[2] 3920 and RFC 3921. The goal in the beginning of XMPP was just to create an open and extensible protocol for Internet-wide instant messaging and presence information. In the last years, it expanded into the domain of general message-oriented middleware and is widely deployed on thousands of servers across the Internet. The core concepts are the following:

- *Decentralization* of the XMPP infrastructure by using the concept of an 'XMPP domain' that is governed by a single XMPP server. This server manages user accounts of all users in this domain and handles communication inside the domain. All XMPP domains that are able to exchange messages are combined to an 'XMPP network'.
- *Distinguishing users and resources*
  A user represents by its user account a logical message endpoint. While messages are normally directed to a user, they are actually delivered to 'XMPP resources' that constitute

1 Application Programming Interface
2 Request For Comments

different XMPP clients the user is instrumenting. Thus, a user can instrument multiple clients simultaneously, while a message directed to the user is routed to a single resource by its XMPP server.

- Each entity of the XMPP infrastructure can be addressed by an *XMPP ID*. It takes the format `[user@]server[/resource]` where `server` is an Internet domain name, `user` is a nickname and `resource` indicates a special client. If the part `user@` is omitted, the server is addressed. While messages are usually addressed to a user, a special resource of a user can be identified by appending the resource name to the XMPP ID. Typical examples for a resource would be `home`, `work`, `mobile`.
- To be able to see the online status of a user, *exchange of presence information* is provided by the XMPP protocol. Users can subscribe to the presence status of other users and thus build up a 'buddy list'.

A typical XMPP session starts with the client sending `<stream:stream>` to port 5222 of the XMPP server the user has an account on. The server answers with also starting an XML stream by returning `<stream:stream>` containing a session ID as an XML attribute. An authentication of the user to its server follows which logs in the user to its user account and registers itself as a resource. Afterwards, packets are exchanged that constitute self-contained XML fragments and are called 'stanzas'. These packets follow XMPP subprotocols and offer a wide range of possibilities, including instant message sending, presence status exchange, querying user metadata and more. To finish the conversation, the client sends an `</stream:stream>` message to the server that answers with the same message and subsequently tears down the TCP[1] connection. Thus, in the course of the conversation, two complete XML documents are exchanged between the client and the server. Error conditions occurring during the conversation are signaled using the XML element `<stream:error>`.

Communication always happens between a client and a server while client-to-client interaction is not supported in order to keep the client component as simple as possible. If a message is sent to a target account whose user is offline, the message can be stored on the XMPP server. The message is delivered to the user as soon as the user logs in again, guaranteeing message delivery. If the message is directed to another XMPP domain, the server to which the client is connected establishes a connection to the server governing the target domain. A special server-to-server protocol is instrumented to forward the message to the target domain where it is delivered to the user account. Thus, in a typical inter-domain communication scenario, a

1 Transmission Control Protocol

message is sent from a client to its server, which forwards it to the server of the target domain, where it is delivered to the target client.

The presence subprotocol is used for updating the presence state of a user and managing presence subscriptions. Most of the logic of subscription management is included in the server component keeping the client as simple as possible. A list of presence subscriptions, called 'roster' in the context of XMPP, are stored with the user account on the server. For updating a user presence status, the client sends a single presence message to the server, which is subsequently forwarded to all subscribers on the roster.

Furthermore, the presence protocol is used for supporting a basic groupchat facility. Each group chat room has its own ID following the general XMPP ID conventions: <chat-room>@<server>. Messages being sent to a chat room are addressed to this ID and forwarded by the group chat server to all members of the group chat. A user joins a group chat room by sending a presence update message of type 'available' to the group chat ID. The from field of the XMPP message packet carries the user's normal XMPP ID. On sending a 'presence unavailable' packet to the group chat ID, the user is removed from the group chat room.

XMPP IDs resemble email addresses, and the overall architecture is similar to the distributed SMTP[1] architecture for mail delivery. The concept of various physical communication endpoint known as XMPP resource has been introduced due to the problem of SMTP of accessing stored emails from different locations.

XMPP offers guaranteed delivery of messages by its 'offline messages'. Messages that cannot be delivered to the recipient are stored on the XMPP server. Concepts of users, their accounts, as well as message routing, and subscription management are encompassed by XMPP. It supports point-to-point as well as publish-subscribe communication channels. The latter can be implemented by instrumenting either the presence protocol or the pub/sub extension [MSAM09].

5.3.3 JMS

Another alternative for being adopted as transfer protocol is JMS [HBS02]. JMS is an API, that deals with asynchronous, message-oriented communication between systems. It offers two paradigms for message sending: point-to-point using message queues and topic-based publish/subscribe. The JMS API is implemented by systems called JMS providers that are either stand-alone or embeddable. Since JMS specifies no bindings of the introduced JMS messaging paradigms to specific communication protocols, it is up to the JMS provider to offer arbitrary

1 Simple Mail Transfer Protocol

bindings. One example of a JMS provider is Apache ActiveMQ[1]. Besides others, ActiveMQ offers a binding to the XMPP protocol.

The publish/subscribe paradigm of JMS is bound to XMPP methods in the following way: A JMS topic is mapped to an XMPP group chat room. A client joining this room subscribes to the associated JMS topic, leaving the room results in being unsubscribed from the topic. Messages being published to a topic are broadcasted in the group chat room. XMPP group chat support has been introduced above.

JMS offers no facility for requesting a subscription that can either be confirmed or refused. Since a subscription is a low level concept in JMS, it is always granted without intended user interaction. Other high-level concepts like user accounts are not provided, while JMS guarantees the delivery of messages by making them persistent. A point-to-point as well as a publish-subscribe communication channel are supported as described above. A drawback of JMS is its coupling to the Java programming language.

5.4 Conclusion

This chapter introduced URL and XRI as possible identifier schemes for DEUS accounts. While XRI enables addressing of real world entities, the benefit of URL is its wide-spread use and publicity. Metadata discovery for URLs is provided by the XRD standard together with surrounding protocols. Three candidates of transfer protocols to be used are introduced that fulfill the DEUS requirements to communication protocols to a certain degree. REST offers an architectural style, implemented by HTTP, that provides no high-level concepts needed by DEUS. JMS is an artifact of the Java world, while XMPP provides all the features, that are required by DEUS.

1 http://activemq.apache.org

6 Architectural Overview and Party Information Data Model

A reference implementation proves the practical feasibility of the concepts presented in chapter 4. Therefore, some decisions were made during the design phase of DEUS. Amongst others, user ID type and an associated discovery method as well as used communication protocols were chosen. Bindings to them are defined and an abstraction is introduced to decouple the DEUS core from specific protocols. The implemented system is modularized using vertical and horizontal decomposition, while further cross-cutting concerns were identified. The resulting architecture adheres to the three-tier-style, where the application logic is included in the core tier. A set of well-defined interfaces is exported to the presentation tier that also encompasses neighbor systems. Decoupling of any concrete infrastructure technology like data stores or specific communication protocols is achieved by introducing layers of abstraction. The horizontal decomposition is most visible in the Soul sublayer, where subsystems support the functional roles introduced in section 4.2. Since DEUS is agnostic of the content of exchanged Digital Cards, an exemplary Digital Card type is introduced containing general party information including contact and further master data.

6.1 DEUS System Design Decisions

DEUS adheres to a mediated publish/subscribe architecture that inherently results from the basic interaction scenario described before. User accounts are addressed by URIs, while the design of the system allows for arbitrary ID types to be adapted. URI as addressing scheme was preferred over XRI due to its widespread use, maturity and simplicity. Nonetheless, XRI is a candidate for being integrated in the future, since it offers addressing of entities like persons and organizations which is conceptually preferred over URI addressing web resources.

For discovering metadata about a user account with a given ID, a stack of discovery protocols around XRD is intended. Step 1 and 2 of a discovery process, as described in section 5.2 involves locating and obtaining a descriptor document. This is supported by LRDD instrumenting HTTP, thus requiring the user ID type to be a URI of type URL. Communication between DEUS accounts residing on different nodes is supported by a transfer protocol binding to XMPP that was described in section 5.3.2.

6.2 Decomposition of DEUS

Decomposing DEUS results in a set of modules with a well-defined scope of work. The core tier exports functionality provided by its subsystems to either be used by a user interface or by neighbor systems. Therefore, it instruments data stores and communication protocols being encompassed in the infrastructure tier.

6.2.1 Vertical, Horizontal and Lateral Decomposition

DEUS is decomposed vertically into tiers and sublayers. The horizontal decomposition into subsystems is deduced from the differentiation of the functional roles introduced in section 4.2. Thus, the modules have high functional cohesion and a distinct scope of work. As additional 'lateral' decomposition, the Barker subsystem provides cross-cutting functionality for user-system interactions. The resulting overall architecture is outlined in figure 6.1.

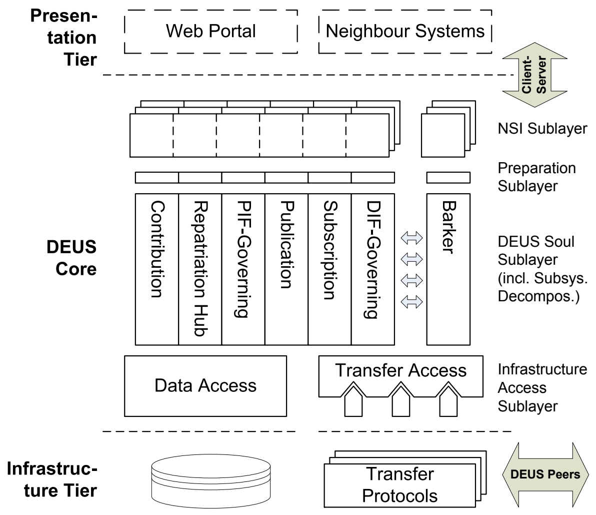


Figure 6.1: The DEUS architecture

6.2.2 User and Neighbor System Interaction

The *presentation tier* is intended to offer arbitrary interfaces for user interaction like a web portal. Alternatively, neighbor systems can access the DEUS system by several remote invocation techniques that are provided by the *NSI*[1] sublayer. These institutional neighbor systems access their DEUS extension within an intranet by client/server access. They should be clearly distinguished from DEUS peers which are nodes that constitute a part of DEUS themselves. By contrast, neighbor systems are not part of DEUS. Both, the user interface, as well as neighbor systems instrument the same services being exported by DEUS.

6.2.3 The DEUS Core

The *application tier* or *core layer* encompasses the NSI sublayer, as well as the preparation sublayer, the Soul, and the infrastructure access sublayer. The primarily disposed NSI remote invocation technology is a REST interface, since the RESTful approach fits with the document-oriented perspective. Modules that provide NSI access by SOAP[2] or any other remote invocation technology could be deployed additionally. All invocations to the subsystems that reside in the DEUS Soul sublayer are intercepted by the *preparation sublayer.* It checks for data validity and transforms the parameter types to domain data types, thereby shielding the Soul subsystems from calls with invalid data.

The main sublayer of the DEUS core is the *Soul sublayer*, incorporating the essential business logic. Here, the horizontal decompositioning and the resulting subsystems are most visible. These subsystems, their interfaces, and their cooperation are further elaborated on in section 6.2.5.

6.2.4 Access to the Infrastructure

The *infrastructure tier* encompasses data stores and DEUS data schemas as well as transfer protocols for communication with DEUS peers. The *infrastructure access sublayer* within the DEUS core decouples the Soul from specific infrastructure technology. The *data access sublayer* provides an API to the Soul that solely handles domain objects and is agnostic of any specific persistence technology. An implementation of this API provides a binding to a specific back-end data store.

The *transfer access sublayer* provides the Soul with a facility to communicate with remote DEUS accounts. Instrumenting this sublayer, it is transparent to the Soul whether the targeted account resides on another or on the same DEUS node. Thus, the transfer access sublayer incorporates a layer of abstraction between the DEUS Soul sublayer and concrete communication protocols. Hence, DEUS is kept independent of communication protocol internals and

1 Neighbor System Interface
2 Simple Object Access Protocol

furthermore allows to integrate arbitrary other protocols fulfilling the requirements posed in section 4.6.2. These protocols are instrumented by implementing bindings that map the required functionality to protocol-specific methods. These bindings follow a plug-in architecture and can be deployed into DEUS at runtime. A process of *communication protocol negotiation* chooses a protocol binding that is available to the sender of a message as well as to the receiver. This is done on a per-message basis so that protocol bindings that are plugged in during runtime can immediately participate in the protocol negotiation. A loopback binding for node-local as well as an XMPP binding for inter-node communication are defined.

6.2.5 Soul Subsystems

Figure 6.2 outlines the subsystems of the Soul sublayer in the context of the Contribution-Repatriation-Publication Chain described in section 4.4. The subsystems are grouped according to their related distribution roles[1]. For the sake of simplicity, only a selection of the interfaces exported by the subsystems are included in the figure. Also, the lateral Barker system, that provides user-system interaction is omitted.

Interfaces exported to the presentation tier are displayed above the subsystem components. These interfaces can be instrumented by a user interface and are furthermore published in the NSI sublayer to be remotely invoked by neighbor systems. Interfaces displayed below the subsystem component are exported to the transfer access sublayer and thus to DEUS peer systems.

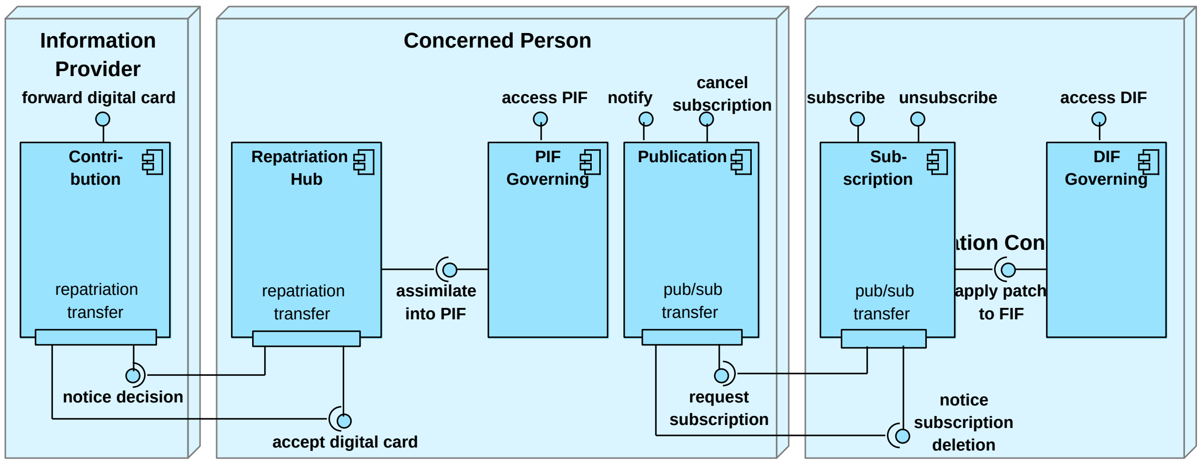


Figure 6.2: The Soul subsystems

1 See section 4.2 for more information about the distribution roles. Since a DEUS actor like a Patient may assume several distribution roles, a DEUS node needs to support all distribution roles. Thus, it always encompasses all subsystems.

The *Contribution* subsystem provides a facility to import Digital Cards into the DEUS system. Therefore, the interface `forward digital card` is exported to the presentation tier. The interface is intended to take a digital card, sign it, and forward it to the Concerned Person's DEUS account using the DEUS peer communication protocol infrastructure. Omitted here are the interfaces used to establish repatriation relationships as described in section 4.4.1.2.

The *Repatriation Hub* subsystem of the receiving account accepts the Digital Card by an `accept digital card` interface. It presents the decision whether to accept or decline the contribution to the user by delegating the event to the Barker subsystem. There, it is appended to the *attention list* that constitutes a collection of elements that require the user's attention. The repatriated Digital Card is meanwhile persisted in a staging area.

If the user confirms the plea about the Digital Card acceptance, the Digital Card is removed from the staging storage and merged into the Personal Information File. Therefore, the interface `assimilate into PIF` is offered that is only called from the Repatriation Hub subsystem and not exported to the presentation tier. The exported interface `access PIF` can be used by the presentation tier to retrieve the Personal Information File.

The functionality around the traditional publish/subscribe capabilities of DEUS is comprised in the subsystems *Publication* and *Subscription.* The Subscription subsystem offers a `subscribe` interface to the presentation layer. An invocation of it results in a remote call to `request subscription` of the publication subsystem of the addressed DEUS account. The subscription request is delegated inside the Publication subsystem to the Barker that presents the request to the user for decision. As soon as the user decides about the request, the decision is sent back and the publication relationship is established on both sides. The interface `unsubscribe` of the Subscription subsystem may be used by clients to tear publication connections down resulting in a remote invocation of methods on the Publication subsystem.

The Publication subsystem offers the interface `notify` to the presentation tier, that triggers the publication of any changes of the Personal Information File. Furthermore, the interface `cancel subscription` is used for terminating a pub/sub relationship initiated by the Concerned Person and results in a remote call to `notice subscription deletion` of the Subscription subsystem. Interfaces for publisher-initiated invite of subscribers are not displayed.

As the consequence of a received update of a Concerned Person's Personal Information File, the interface `apply patch to FIF` of the *DIF-Governing* subsystem is instrumented. This interface offers to merge a received patch into the Foreign Information File which is associated with the Concerned Person. Moreover, DIF-Governing exports an interface `access DIF` to the presentation tier that can be used to retrieve data from the Distributed Information Folder.

6.3 The Party Information Data Model

As described in section 4.3.3, the focus of this thesis is not on the semantics of exchanged patient-related data. Thus, providing a type of Digital Card incorporating CDA documents was deferred to the future. Instead, a data model encompassing patient master data is introduced that is compatible to the *vCard* standard for address and contact data [DH98]. Figure 6.3 presents an EERD[1] of a party's master data, that also allows for non-person parties like organizations to be included. The model forms a superset of the data that can be stored using vCard including features of the upcoming version 4.0.

Information about a DEUS party, either being a person or an organization, is contained in a Digital Card of type *Party Information Digital Card*. The focus of the DEUS system is on individuals, so that only the entity person is elaborated on in the EERD. A person is described by a *name* consisting of a collection of prefixes, a given name, a collection of additional (middle) names, a family name, a collection of suffixes, an optional maiden name, and an optional nick name. The attribute full name has its origin in the vCard standard, where this is the default display name of a vCard. It may be composed of several parts of the introduced name attributes. Further attributes of a person are the gender, date and place of birth, date and place of death, and spoken languages stored according to the ISO 639-2[2] standard. The attributes photo, sound, and note are again derived from the vCard standard.

A person may have multiple *addresses* which are either tagged as personal or professional and furthermore supplied with a label and a numeric priority. An address constitutes a weak entity, since, in rare cases, two persons may have the same address and thus, the primary key of the person needs to be part of the primary key of an address. It is made up by a country, a region, a locality (confining to a city in most cases), a zip code, and an arbitrary address extension. All of these attributes embody the discriminator part of the weak entity's primary key and originate from the vCard standard. Furthermore, an address can either be a street address or the address of a post office box. For the sake of simplicity, an address is not further specialized into two subtypes. The specialization rather is denoted by the keyword 'XOR' between the attribute POB and the attribute street.

A person may have multiple *phone numbers* which have the relationship attributes in common with the address relationship. Phone label as a free text attribute may be 'car', 'assistant', or 'secretary' and thus encompass information that is not contained in the schema. Since two persons in the DEUS system may have the same phone number, phone is also a weak entity

1 Enhanced Entity-Relationship Diagram
2 ISO 639-2, Codes for the representation of names of languages, http://www.loc.gov/standards/iso639-2

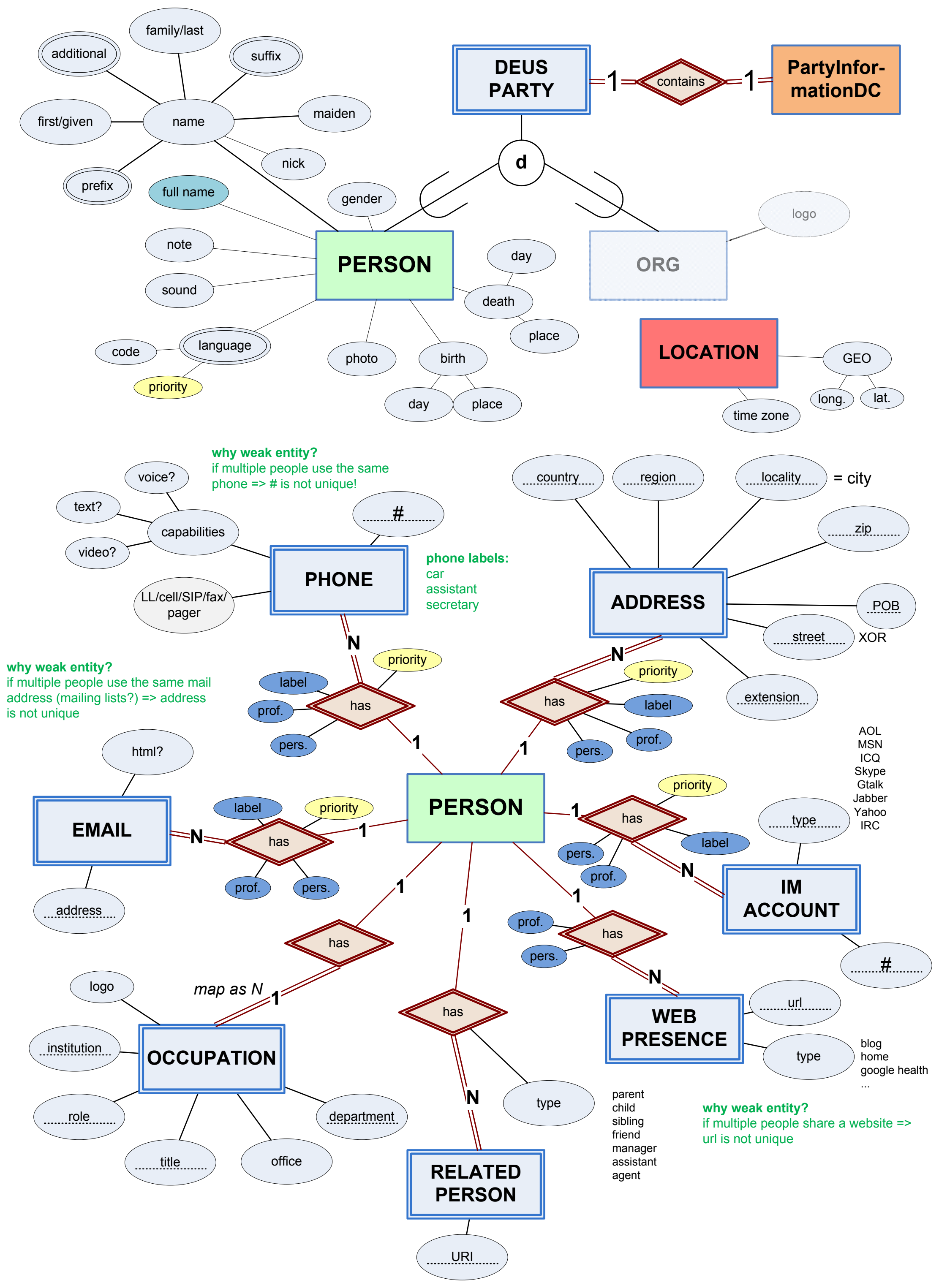


Figure 6.3: An EERD of a party's master data

with the phone number as the partial primary key. Each phone may have the capabilities voice, text and video constituting boolean flags. Moreover, a phone is either of type landline, cell, Internet phone (SIP[1]), fax, or pager.

A person may have multiple *email addresses* where the address string constitutes the partial key. Aside from the relationship attributes, it contains a flag to indicate desired delivery of emails in HTML format. The case of multiple persons using the same email address is handled by declaring email as a weak entity. The same holds for optional *instant messaging accounts*, where the partial key is the identifier of the account, together with its type which encompasses a list of instant messaging providers. A person may specify several *web presences*, where the attribute url is the discriminator. A further attribute indicates the type of web presence, e.g. being either a blog, a homepage, or even a Google health account web address. Neither a label nor a priority is attached to this relationship.

A person is restricted to have one *occupation* at most, being identified by an institution, a role, a job title, and a department. Since, by chance, two individuals in the DEUS system may have the same occupation as far as identified by these attributes, occupation again is a weak entity. Furthermore, a logo and an office description can be attached to an occupation, while it lacks any label or priority. Also, following the vCard standard, it is possible for a person to attach *related persons*, partially identified by a URI. Each relationship to a related person is attributed a type that further specifies the related person, like parent, child, sibling, friend, manager, assistant, or agent.

6.4 Conclusion

A reference implementation of the concepts introduced in the previous chapters proves the feasibility of the design. URI is thereby used as the identifier type. A decomposition in a horizontal, vertical and lateral manner results in modules with a well-defined scope of work. The core tier of the three-tier architecture exports functionality to the presentation tier and to other DEUS accounts. Communication between accounts instruments the transfer access sublayer that adheres to a plug-in architecture for adding arbitrary transfer protocol bindings. The horizontal decomposition of the Soul sublayer results in clearly separated subsystems supporting the functional roles introduced in previous chapters. An exemplary type of Digital Card contains patient master data forming a superset of the vCard address data standard.

1 Session Initiation Protocol

7 System Design

A major design goal was to embody boundaries and interfaces of system modules also during runtime. Therefore, OSGi is used as a dynamic component model that provides application life cycle management, a service registry, and an execution environment. Subsequently, all modules of DEUS are packaged as OSGi bundles exporting their functionality as OSGi services to other bundles. The domain model bundle provides other subsystems with the basic DEUS artifacts introduced in previous chapters. It is structured according to the system dissection by grouping artifacts related to subsystems.

The infrastructure access sublayer is divided into a data access part, which abstracts from a specific backing data store, and into a transfer access part. Core subsystems communicating with other DEUS accounts instrument the transfer access sublayer for sending messages. A loopback binding for node-local communication as well as a basic binding to the XMPP protocol for inter-node data exchange are implemented as plug-ins.

Business logic in the Soul sublayers is decomposed into subsystems with well-defined interfaces exported to the presentation tier, to other subsystems, and to other DEUS peers. The Barker subsystem provides system-to-user interaction by the management of attention lists and user-to-system interaction by its decision processor. The additional Gatekeeper subsystem encapsulates DEUS account registration, user login, and the setup and teardown of distribution roles assumed by accounts.

Functional roles introduced in section 4.2 are supported by the subsystems in the Soul sublayer: The subsystem Contribution supports the functional role Contributor. The subsystems Repatriation Hub and PIF-Governing support the functional role Repatriation Authority. The subsystem Publication supports the functional role Publisher. The subsystem Subscription supports the functional role Subscriber. The subsystem DIF-Governing supports the functional role Retriever. All these subsystems together provide the functionality described in the previous chapters and are individually packaged as OSGi bundles.

7.1 From Components to OSGi Modules

OSGi introduces the notion of a *bundle* incorporating a module with a defined life cycle. This life cycle includes installing, starting, stopping, updating, and uninstalling the bundle without requiring the deployment platform to be restarted.

The OSGi *service registry* provides dynamic registration and unregistration of services. Since this is the only facility that bundles can use to interact with each other, a clear interface design is enforced. Trackers that detect the loading or unloading of bundles and services can react to these events and enforce the application to adapt accordingly. The OSGi environment fosters the principle of small, cohesive bundles. By interacting with each other only through exported services, loosely coupling is promoted. The internals of bundles are hidden by the OSGi container. Instrumenting a deployment descriptor, bundles can declare their name, version, and dependencies on other bundles.

Due to these benefits, the DEUS system is implemented as a set of OSGi bundles which incorporate the components obtained by decomposing the system. Bundles export interfaces as services by registering an implementation of an interface in the service registry. This plays especially well with a layered architecture, where bundles of a higher layer consume OSGi services offered by bundles of lower layers. Since there is no restriction enforced by OSGi which bundles may import which services, a meaningful naming of the service interfaces is introduced. This naming specifies the part of the system that is intended to consume certain services.

7.2 Domain Model

The previous chapters introduced several artifacts occurring in the DEUS system. Most of the artifacts can be grouped according to the horizontal decomposition and thus be assigned to subsystems. However, several cross-cutting model elements are identified that cannot be related to a specific subsystem. This grouping is outlined in figure 7.1.

7.2.1 Cross-Cutting Model Elements

Several model elements cannot be related to a specific subsystem, but rather are cross-cutting and needed in the whole system. These model elements include UserMetadata, that encapsulates basic information about a user including a full name and gender. This is needed to display list entries of the user, containing a name, while the gender is used to adjust grammar.

The UserId element is an abstract model element incorporating a generic user ID. It is further examined in figure 7.2. UserId requires classes deriving from it to implement the abstract methods getType and toString. UserUrl is a derivation of it including a username and the base

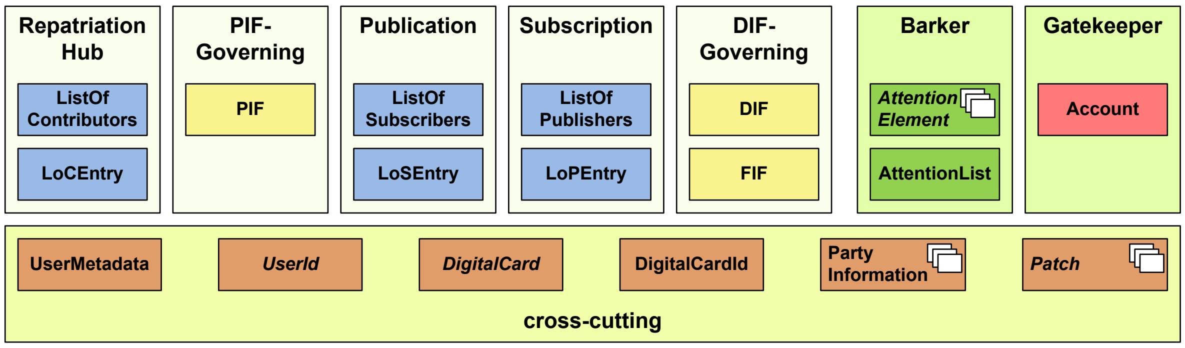


Figure 7.1: The DEUS domain model with model elements grouped by subsystems

URL of the DEUS node where the user account resides on. The method getUrl returns the URL that results by appending the username to the server base URL.

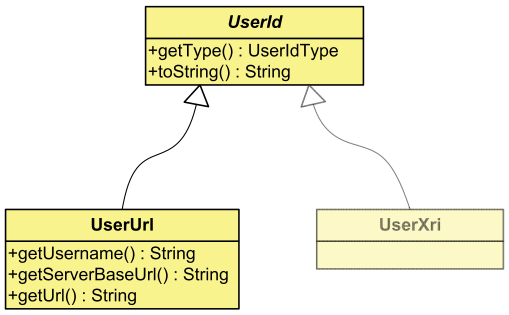


Figure 7.2: The UserId model element

The introduced concept of a Digital Card manifests itself in the abstract model element DigitalCard. A DigitalCard contains a label and its date of creation and is outlined in figure 7.3. Its primary key constitutes a separate model element of type DigitalCardId with its components being described in section 4.3.1. The single concrete Digital Card is of type PartyInformationDC and not included in the figure. It contains the model element PartyInformation that represents the whole data structure described in section 6.3. PartyInformation only constitutes the root entity of a comprehensive object graph, what is indicated by the icon on the right side.

The Patch model element incorporates a change that results from assimilating a Digital Card into the Personal Information File. Depending on the assimilation strategy, concrete patch elements inherit from the abstract Patch. These patches should be merged after publication into the Foreign Information Files of any subscribers. Since the repatriation semantics currently confine to only appending the repatriated Digital Card, the only concrete patch type contains a single Digital Card.

7.2.2 Domain Model Elements for Administration of Relationships

For the repatriation as well as the publication phase, trust relationships need to be built up prior to data exchange, as it was described in section 4.4.1. There, the concept of a *list of contributors*, a *list of subscribers*, and a *list of publishers* were introduced. These lists and their entries are incorporated in the domain model with their own model elements, painted blue in the figure. Each type of list entry contains the UserId of the contributor/subscriber/publisher and related UserMetadata.

7.2.3 Dossier and Depository Model Elements

The essential concepts of the 'dossier' artifacts Personal Information File and Foreign Information File have already been introduced in section 4.3.2. Furthermore, the encompassing Distributed Information Folder as depository artifact has also been explained. These entities are contained in the DEUS model under their abbreviations and displayed yellow in figure 7.1. Both, the PIF and the FIF model elements contain a set of DigitalCards without a specified ordering. The DIF entity contains a set of FIF entities. The introduced dossier and depository elements are outlined in figure 7.3.

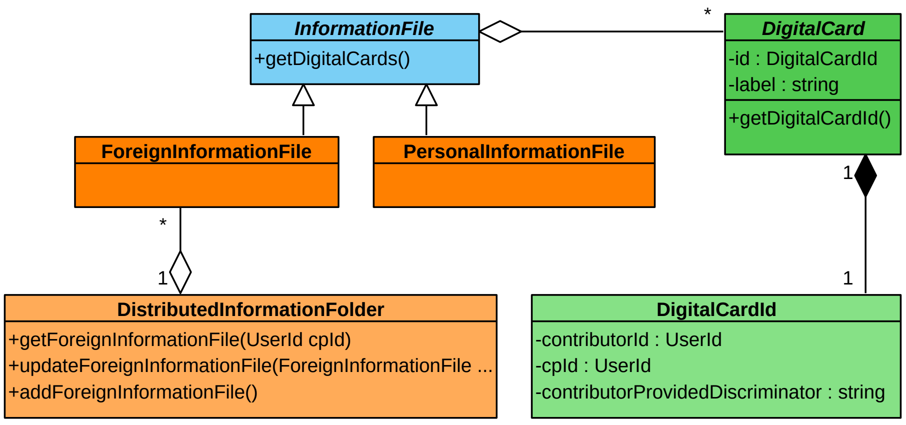


Figure 7.3: The dossier and depository model elements

7.2.4 Attention Model Elements

The lateral *Barker* system is used for system-user interaction by presenting attention elements to the user. These attention elements constitute *pleas* and *notices* that require the attention of the user. They are manifested in the domain model as derived elements of the abstract AttentionElement, painted green. The fact that AttentionElement is just the base element of a list of concrete attention elements is indicated by the icon on the right side. Each AttentionElement contains a consecutive number, its creation date and a flag, that indicates if the user already interacted with it. It is outlined in figure 7.4.

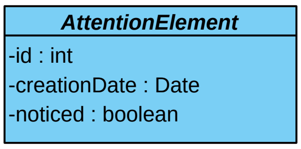


Figure 7.4: The abstract base entity AttentionElement

All attention elements are collected in an AttentionList and are outlined in figure 7.5. A basic distinction needs to be made between *notice* and *plea* attention elements. Notices, displayed blue, only inform the user of an event, while pleas, displayed orange, require a user decision.

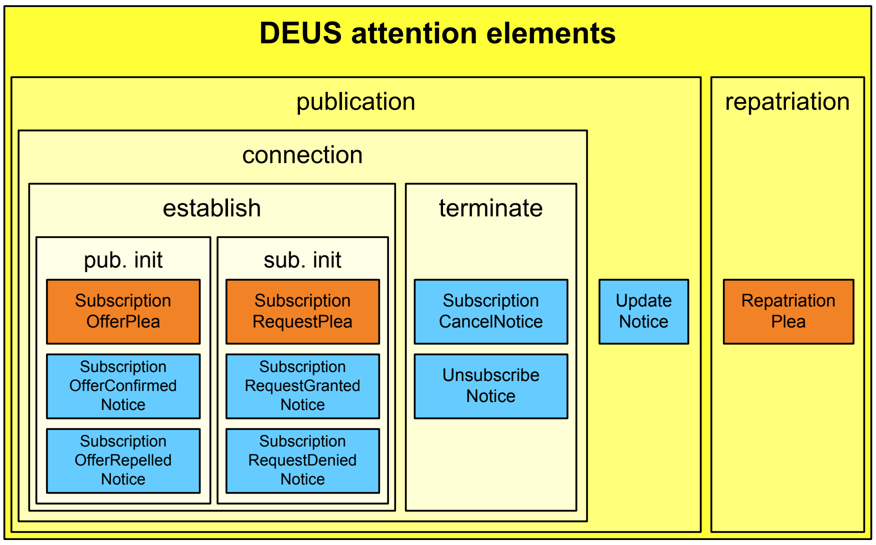


Figure 7.5: The hierarchy of attention elements

All attention elements can be grouped on top level according to the two phases of information exchange between accounts, namely repatriation and publication. The repatriation phase currently only comprises a plea about a repatriated Digital Card. The publication elements include a notice being displayed to the user when an update patch is received from any connected publisher. Furthermore, attention elements concerning trust connections are grouped by occurrence during establishment or termination. Since termination requires no decision from a user, only two notice elements display the teardown of a trust relationship either initiated by subscriber or publisher side.

The attention elements of connection establishment are further distinguished in two groups regarding the actor that initiates the connection. A publisher initiating a connection results in a SubscriptionOfferPlea issued to the subscriber. According to figure 4.10, the subscriber either confirms or repels the offer resulting in two different notices being displayed to the publisher, reflecting the decision. A subscriber initiating a connection entails a SubscriptionRequestPlea issued to the publisher. The publisher either grants or denies the request resulting in two different notices reflecting the decision to be displayed to the subscriber.

7.2.5 The Model Elements of the Lateral Gatekeeper

Another lateral system is *Gatekeeper* that is responsible for account generation and logging users in and out of their accounts. A user account is embodied by the model element Account that contains the UserId of its owner. Furthermore, a local username is included that provides a decoupling of the username during login and the ID of the user: While the ID needs to be globally unique, the username needs to uniquely identify the user only on the local node. This separation allows for flexibility in the choice of a user ID. However, simplicity benefits from a local username coinciding with a potential username part of the user ID. The login information is completed by the password, also stored in the Account element. Since a DEUS user can potentially assume all distribution roles, introduced in section 4.2, the associated roles need to be stored in the Account model element. The Account entity is outlined in figure 7.6.

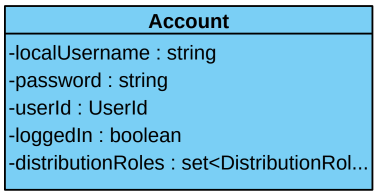


Figure 7.6: The Account entity and its properties

7.2.6 The Resulting OSGi Bundle

It would be possible to split the domain model into different bundles, according to the grouping of figure 7.1. However, quite often, model elements from different subsystems are required together. Thus, it was decided to include all model entities into a single OSGi bundle and structure the module internally according to the subsystem decomposition. The resulting bundle is a 'pure library' bundle that is only imported by other bundles and does not register any services. This reflects the basic difference between passive objects without behaviour and active, often stateless objects with well-defined behaviour[1]. This principle of dividing passive, data carrying objects and active, behaviour-rich objects is adhered to throughout the whole DEUS system architecture.

1 This differentiation is also applied to distinguish between *Entity Beans* and *Session Beans* in JEE[2].

7.3 Infrastructure Access Sublayer

In figure 7.7 details from figure 6.1 outline the infrastructure access sublayer. This layer was introduced to abstract from concrete backing data stores and used communication protocols. In the following, communication protocols being used by DEUS to transfer messages are also called *transfer protocols*. The transfer access sublayer allows for adding arbitrary bindings to transfer protocols during runtime, since it is implemented adhering to a plug-in architecture.

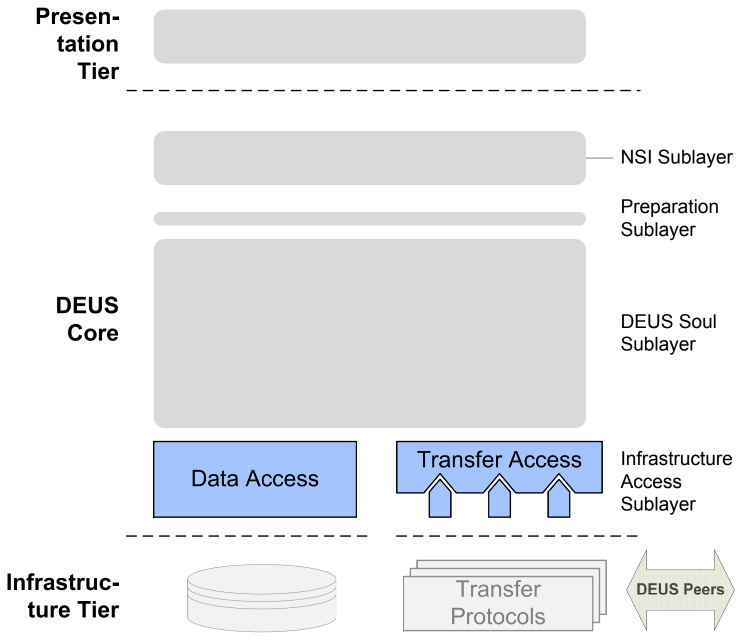
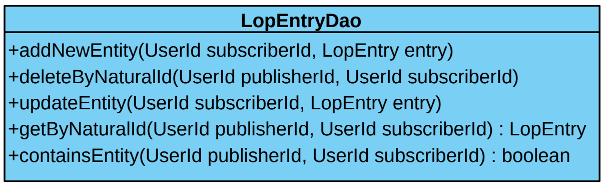


Figure 7.7: The DEUS infrastructure access sublayer architecture

7.3.1 Data Access

Keeping runtime objects persistent requires a back end data store and a mapping from runtime objects to the data model of the store, following the *Mapper* pattern in [Fow03, p. 473]. To keep DEUS independent of any particular storage technology, the mapping to the storage data model must be replaceable. This results in creating a storage-unspecific API for data storing and loading, and separating this API from concrete implementations (*Separate Interface* [Fow03, p. 476]) that integrate certain data stores. These mappings, together with the generic API, reside in the *data access* part of the infrastructure access sublayer.

The API for the storage of these domain objects is structured according to their grouping introduced above. For each domain object, an interface definition contains the necessary methods for storing an object to a data store. Further operations include retrieving, updating and deleting a domain object so that each interface contains all CRUD[1] methods. An exemplary API for loading and storing of entries of the list of publishers is outlined in figure 7.8.



```
LopEntryDao
+addNewEntity(UserId subscriberId, LopEntry entry)
+deleteByNaturalId(UserId publisherId, UserId subscriberId)
+updateEntity(UserId subscriberId, LopEntry entry)
+getByNaturalId(UserId publisherId, UserId subscriberId) : LopEntry
+containsEntity(UserId publisherId, UserId subscriberId) : boolean
```

Figure 7.8: An exemplary data access interface for loading and storing entries of the list of publishers

The collection of interfaces forming the data access API is included in a single OSGi bundle. It contains a dependency on the domain model bundle described above. Other bundles requiring data access only depend on the API bundle and not on a concrete implementation of it.

The first implementation of the data access API is an in-memory storage that doesn't permanently save objects to background storage but only keeps them in transient memory. It uses list data structures to store objects, which are not indexed at all but scanned, if an object with a given key needs to be retrieved. This implementation serves as a dummy for testing purposes. It is packaged into an OSGi bundle that requires the domain model bundle and the data access API bundle.

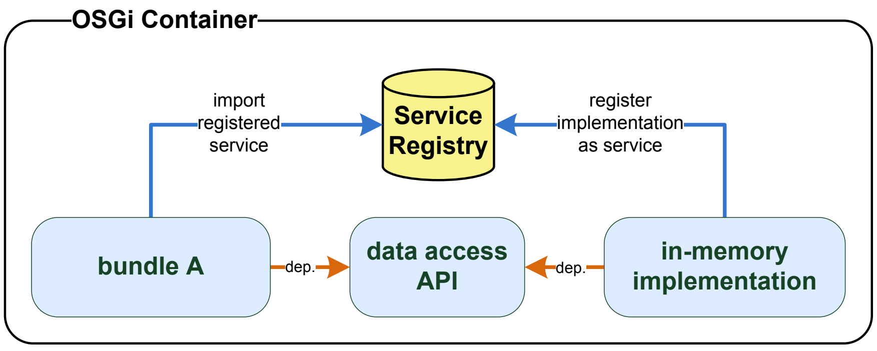


Figure 7.9: Decoupling of interface and implementation instrumenting the OSGi service registry

Furthermore, this bundle exports the implementations of the API interfaces to the OSGi service registry as depicted in figure 7.9. OSGi allows the registration of a service using the interface as a key that is implemented by the service. Other bundles, requiring data access for a domain object, import a service by passing this interface as an argument. In the example, bundle A thus obtains a reference to the service implementation without being coupled to in-memory

1 Create, Read, Update, Delete

implementation. Both bundles only depend on the data access API bundle. This constitutes a vivid example of how OSGi fosters the decoupling of bundles and thus reduces complexity.

Another implementation of the data access API might be provided by a bundle implementing object-relation mapping. For example, Hibernate[1] may be used to describe a mapping from domain objects to a relational database schema in a declarative way. Arbitrary other mapping frameworks like Toplink[2], iBatis[3] or Apache OJB[4] may be added.

7.3.2 Transfer Access

Another infrastructural part needed for DEUS are communication protocols. Since a design objective was to not be coupled to specific communication protocols, a layer of indirection is introduced. This layer takes calls from the DEUS Soul sublayer, chooses a communication protocol, and sends the call in form of a message to the DEUS account of the recipient. If a message is received over a communication channel, this message is translated to a call to the Soul instrumenting a registered callback. The artifacts exchanged between the transfer core and any protocol bindings are abstracted as *messages*. The calls between the transfer core and the Soul sublayer are abstracted as *commands*.

Figure 7.10 outlines the architecture of this transfer access sublayer instrumenting an exemplary protocol binding. The sublayer implements a plug-in architecture, where bindings to arbitrary protocols can be registered to the TP Binding Registry. Each transfer protocol binding forms its own OSGi bundle and thus can be deployed and started at runtime. During start up, it queries the service registry for the TP Binding Registry being exported as a service by the transfer core sublayer. The plug-in subsequently registers itself as a transfer protocol binding instrumenting this registry and is instantly available.

The core part of the transfer access sublayer offers the conceptual interface send command to the DEUS Soul sublayer. Different subsystems that require interaction with other DEUS accounts instrument this exported interface. Command Sender receives the calls and negotiates the transfer protocol to be used.

Therefore, the list of supported protocols is retrieved from the receiver of the command by using the remotely exported interface negotiate TP. These protocols are subsequently matched with the ones supported by the local DEUS node and a common transfer protocol is chosen. Both communication partners have priorities attached to their supported transfer protocols

1 Open Source Java Persistance Framework, http://www.hibernate.org
2 http://www.oracle.com/technology/products/ias/toplink
3 http://ibatis.apache.org
4 http://db.apache.org/ojb

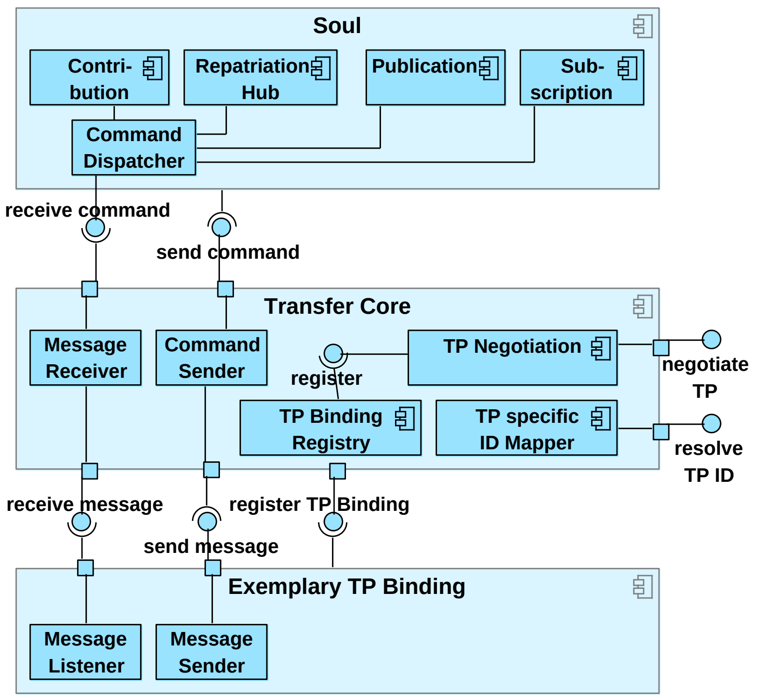


Figure 7.10: The architecture of the transfer access sublayer including an exemplary protocol binding

which are taken into account during protocol negotiation. After registration, protocol bindings are instantaneously available for protocol negotiation. Currently, transfer protocol negotiation is currently repeatedly done for each sent command. This results in the ability to register and unregister protocol bindings at arbitrary points in time.

Subsequently, the user ID of the remote DEUS account is resolved to the protocol-specific ID of the negotiated transfer protocol using the remotely offered interface resolve TP ID. Following the *Command Message* pattern in [HW03, p. 145], the command to send and its parameters are marshaled and included in a message object. The send message interface of the chosen transfer protocol binding is thereupon used to deliver the message.

Registered protocol bindings are actively listening for remote messages being sent by other DEUS accounts. If a message is received by a binding, it invokes a callback to the receive message interface of Transfer Core. The message is unmarshaled, the contained command is extracted and passed to the Soul sublayer using its interface receive command. Passing a command is implemented as a method invocation of a Soul subsystem. Sending commands and receiving messages is further elaborated in the following sections where the current implementation is outlined.

7.3.2.1 Sending Implementation

An exemplary sending of a message for requesting a subscription is shown in figure 7.11. Prior to any sending, a protocol binding must register itself at the TP Binding Registry. Subsequently, the protocol is made available for being chosen during transfer protocol negotiation.

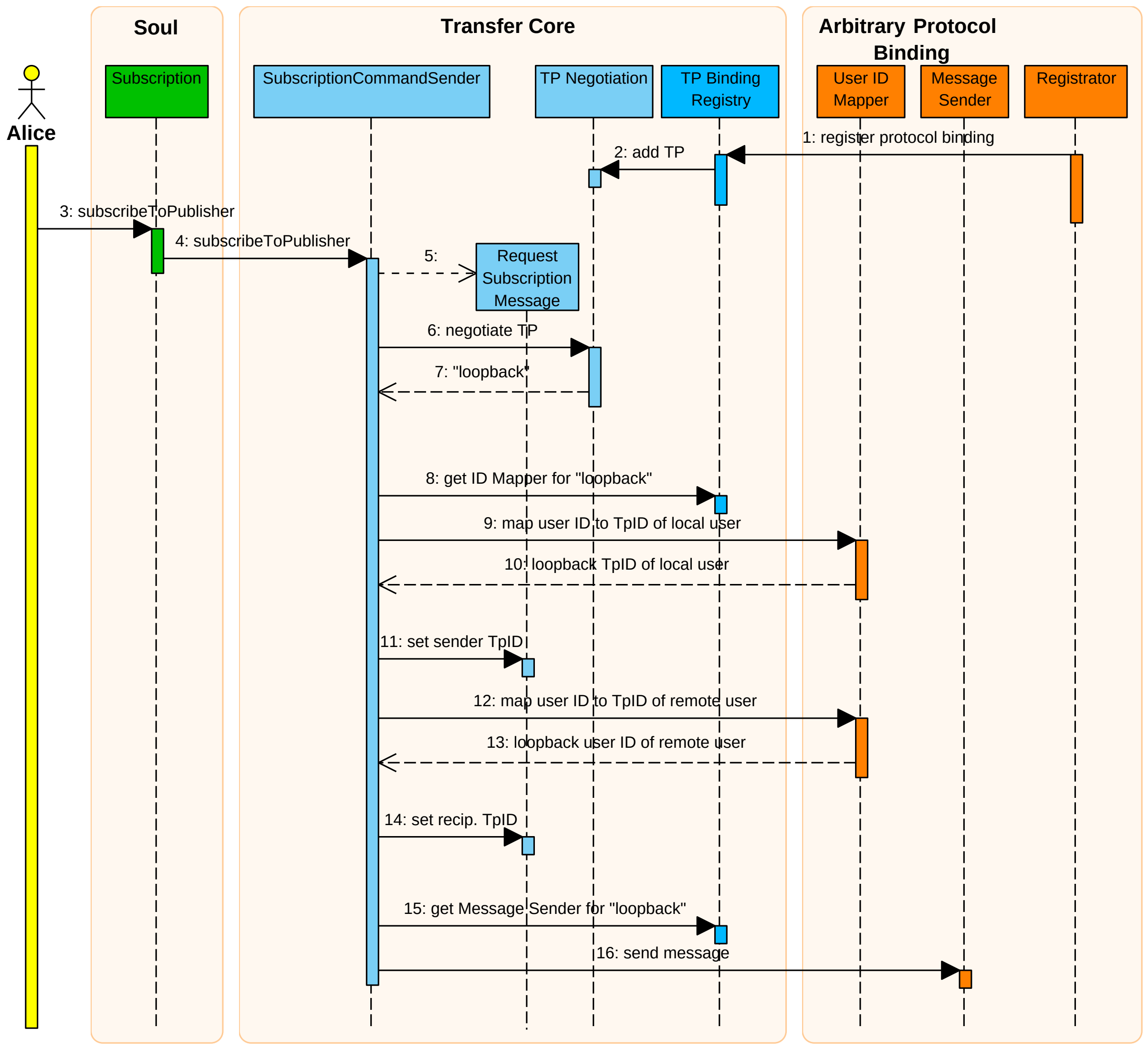

Soul
Transfer Core
Arbitrary Protocol Binding
Alice
Subscription
SubscriptionCommandSender
TP Negotiation
TP Binding Registry
User ID Mapper
Message Sender
Registrator
1: register protocol binding
2: add TP
3: subscribeToPublisher
4: subscribeToPublisher
5:
Request Subscription Message
6: negotiate TP
7: "loopback"
8: get ID Mapper for "loopback"
9: map user ID to TpID of local user
10: loopback TpID of local user
11: set sender TpID
12: map user ID to TpID of remote user
13: loopback user ID of remote user
14: set recip. TpID
15: get Message Sender for "loopback"
16: send message

Figure 7.11: Exemplary sending a message for requesting a subscription

The sending is initiated by an actor that calls subscribeToPublisher on the Subscription subsystem. This triggers a call to a command sender that is responsible for sending commands on behalf of the Subscription subsystem. This component creates a new RequestSubscriptionMessage which is agnostic of the transfer protocol over which it will be sent. Subsequently, the transfer protocol is negotiated by using the component TP Negotiation as described above. The result of the depicted negotiation is the choice of using the apb[1] protocol.

In the next phase, the DEUS user IDs need to be mapped onto IDs specific to the chosen protocol. This is supported by the component User ID Mapper that is registered at the TP Binding Registry during protocol binding registration. First, the sender's user ID is mapped to its transfer protocol specific ID (TpID) and injected into the constructed message. Mapping the recipient's user ID to a protocol-specific ID requires a resolution process involving remote communication with the recipient's DEUS account. This can be accomplished using the metadata discovery process introduced in section 5.2. Furthermore, the protocol-specific ID can already be obtained during protocol negotiation. However, this is not implemented yet. After setting the received transfer protocol ID, the Message Sender of the negotiated protocol is obtained from the TP Binding Registry. It has been registered there during protocol binding registration and provides a protocol-specific way of sending the message.

In conclusion, message sending is initiated by a call from the Soul sublayer to a Command Sender which creates a message. Subsequently, a message is created, a transfer protocol binding chosen, and the user ID is mapped to transfer-protocol specific ones. For sending the message, it is passed 'further down the stack' to the chosen protocol binding.

7.3.2.2 Receiving Implementation

Receiving of an exemplary Subscription Request Granted Message is outlined in figure 7.12. Prior to any communication, a protocol binding must be started which results in registering it at the TP Binding Registry. Subsequently, a callback reference to Message Sender residing in transfer core is obtained. Furthermore, the subsystems that export methods to peers must register themselves as callbacks at the Soul Callback Registry of the transfer core sublayer.

Any protocol binding actively listens for incoming messages. If a message is delivered to the binding, the Message Receiver reference is obtained from the Registrator where it has been stored during registration. Subsequently, the method receive of Message Receiver is called and the received message is passed. The Message Receiver then retrieves the callback to the Subscription subsystem that has been registered previously. The reference to Subscription is subsequently used to dispatch the message to a call to the subscriptionRequestGranted method. This is one of the methods that the Subscription subsystem exports to peers.

1 This is just an acronym for 'arbitrary protocol binding'.

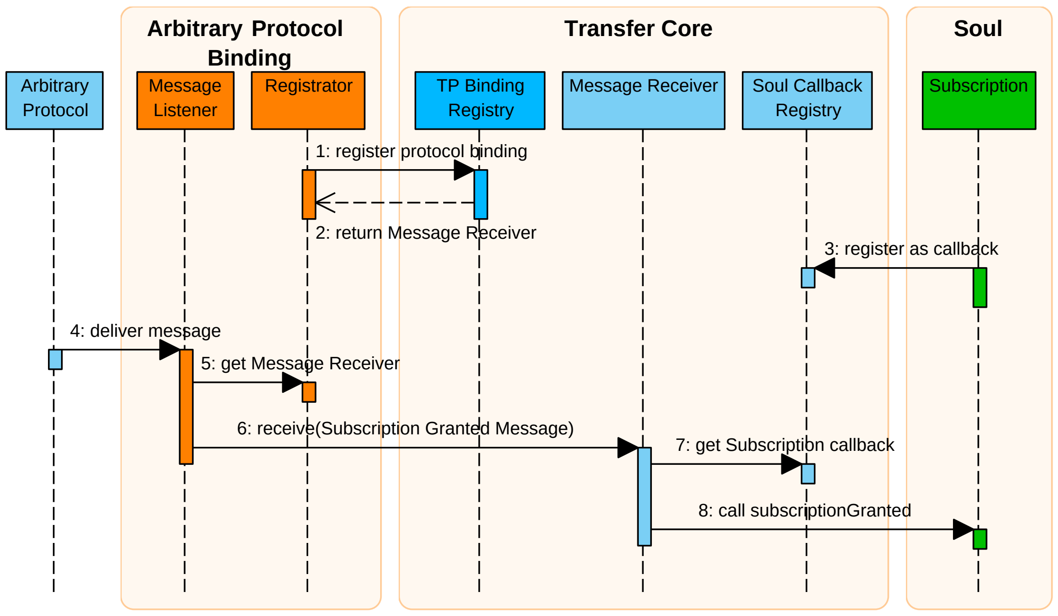


Figure 7.12: Receiving an exemplary Subscription Request Granted Message

To conclude, Soul subsystems can export methods to peers by registering itself as callback in the transfer core subsystem. Message Receiver subsequently dispatches incoming messages and instruments the registered callback for invocations to the exported methods of Soul subsystems.

7.3.2.3 Loopback Binding

During transfer protocol negotiation, a detection reveals whether the DEUS account of the communication counterpart resides on the same DEUS node. This detection compares the server base URL of the sender and the receiver, provided that both user IDs are of type UserURL. If both base URLs match, the accounts reside on the same node since a node is uniquely identified by its server base URL.

In this case, a simple protocol binding is chosen by the negotiation component that implements *node-local loopback* communication. As with more complex bindings, the loopback binding is registered as a plug-in. It is employable during the repatriation phase, if the DEUS accounts of the Information Provider and the Concerned Person reside on the same node. Just as well, it can be used during the publication phase, in case the accounts of the Concerned Person and the Information Consumer share a node. The basic way the loopback binding works is outlined in figure 7.13.

On choosing the loopback protocol binding for message sending, the Loopback Message Sender is retrieved from the TP Binding Registry. Subsequently, the message is sent to the Loopback Message Sender which forwards it to Loopback Message Forwarder. This component resembles the

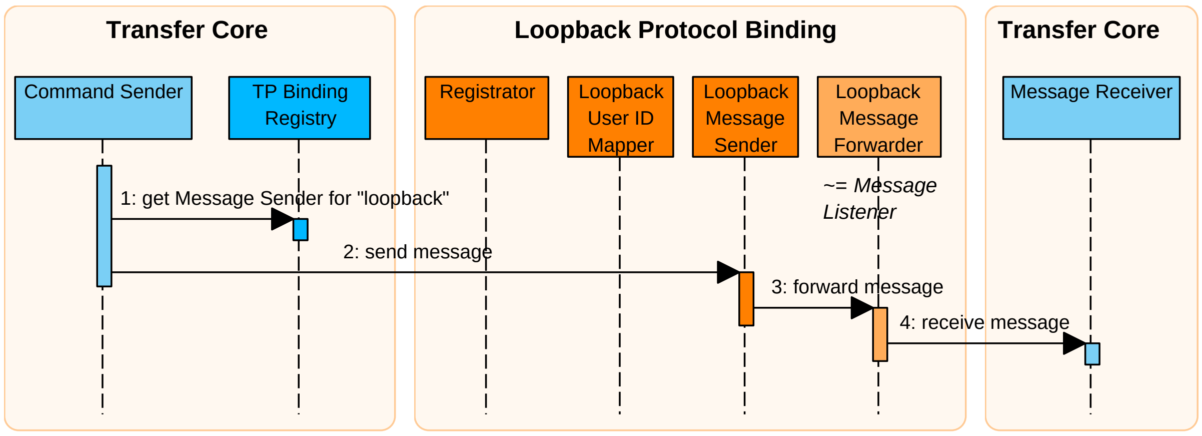


Figure 7.13: The basic functionality of the loopback protocol binding

conceptual component Message Listener in the above description of receiving a message. Loopback Message Forwarder retrieves the reference to Message Receiver from the Registrator what is omitted in the figure. It then calls Message Receiver residing in transfer core. The transfer protocol ID of the loopback protocol binding simply consists of a node-locally unique username. It not necessarily has to coincide with the login username or the username part of a UserURL.

7.3.2.4 XMPP Binding

A more complex transfer protocol binding is needed if communication bridges different DEUS nodes. One candidate fulfilling the requirements, described in section 4.6.2, is XMPP. The basics of XMPP were already described in section 5.3.2. For implementing trust relationship management, the presence subprotocol of XMPP with its subscription management facility was used. On requesting the establishing of a trust relationship, a presence packet of type 'subscribe' is sent to the communication counterpart. Offering of a relationship initiated by the Concerned Person is not yet supported by the XMPP binding. On granting the establishing of a relationship, another presence packet of type 'subscribed' is sent. Otherwise, the presence packet type is 'unsubscribed'. Sending Digital Cards instruments the messaging subprotocol of XMPP but requires XML serialization of the Digital Card.

However, the XMPP binding requires a closer investigation of the XMPP protocol. Work on this binding is subsequently deferred to the future, as described in chapter 9.

7.3.2.5 Conclusion

The described approach guarantees the complete decoupling of the DEUS Soul from concrete communication protocols. Arbitrary bindings to communication protocols can be plugged in during runtime and are immediately available for usage. The registration of bindings, marshaling of commands to send and unmarshaling of received messages is done in a communication abstraction sublayer called transfer core.

It delegates sending of messages to registered transfer protocol bindings after negotiating the protocol with the communication counterpart. All registered bindings listen on the related communication channels for incoming messages, which are relayed to transfer core to be delivered to the Soul.

A local loopback protocol binding is defined for communication between DEUS accounts on the same node. For transferring messages to accounts residing on another DEUS node, a binding to XMPP is provided. Instrumenting the plug-in architecture, bindings to arbitrary other communication protocols can be defined and registered during runtime. The defined bindings together with the transfer core sublayer constitute the transfer access subsystem.

7.4 Soul Sublayer

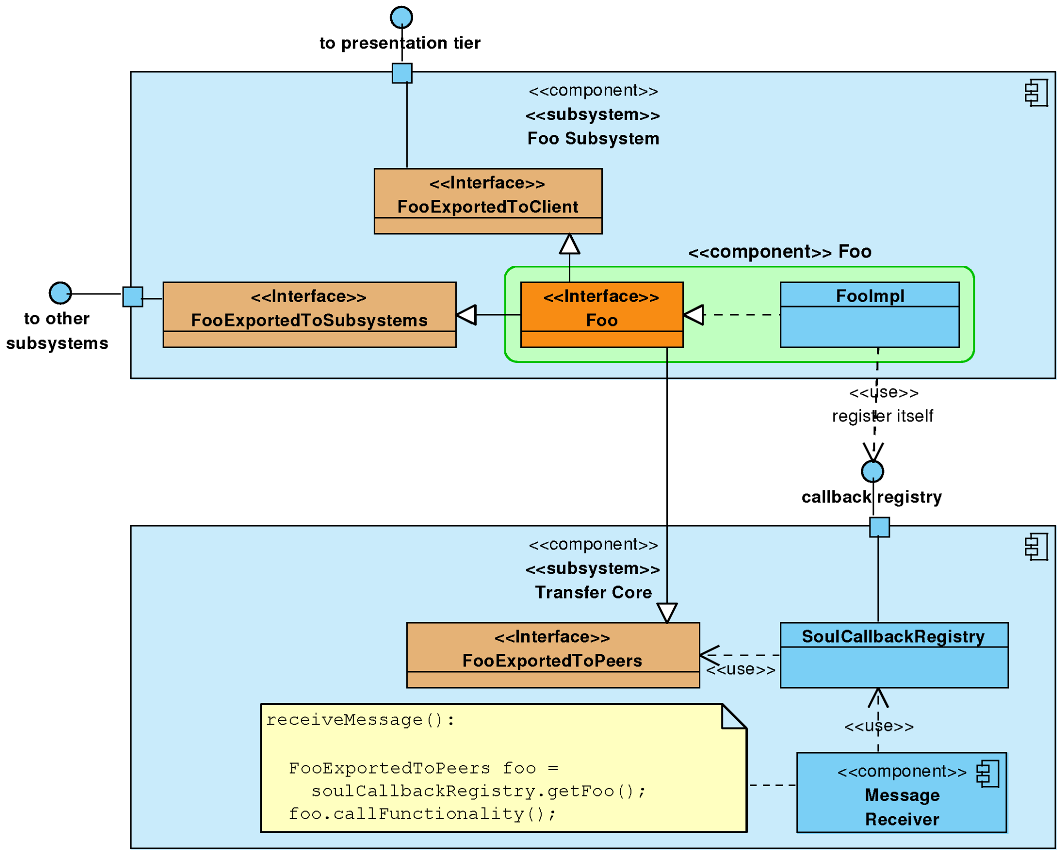
The Soul sublayer clearly manifests the horizontal decomposition into subsystems with a well-defined scope of work. In addition to decomposing DEUS in a horizontal and a vertical manner, several cross-cutting concerns have been identified resulting in lateral subsystems.

Soul subsystems export functionality to the presentation tier including user interfaces and neighbor systems. Furthermore, Soul subsystems export functionality to other subsystems and to other DEUS peers[1]. The described adjacent parts only gain access to required functionality by separating it into interfaces and exporting them to the distinct parts. These interfaces are exported as OSGi services.

OSGi cannot restrict bundles to only import specific services. Rather, all services registered in the service registry may be used by any bundle. However, to at least indicate the intended importing bundle, a naming convention for services is introduced. Service interfaces exported by a Soul subsystem to the presentation tier are named <Subsystem>ExportedToClient. Functionality exported to other subsystems is bundled in an interface named <Subsystem>ExportedToSubsystems. Methods offered to be remotely called by other DEUS peers are collected and named <Subsystem>ExportedToPeers. These interfaces are used by the presentation tier, respectively other subsystems to import needed services from the service registry. Interfaces exported to peers are registered with the transfer access sublayer as callbacks to the subsystems. Thus, a dependency from the transfer access sublayer bundle to any Soul subsystem bundle is avoided.

1 DEUS peers constitute accounts residing on the same or a different node.

An example Soul subsystem called ‘Foo’ is outlined in figure 7.14 together with the three types of exported functionality. Furthermore, the registration of a subsystem as a callback to the transfer access core sublayer is depicted. It therefore instruments the SoulCallbackRegistry and registers itself under the interface FooExportedToPeers. If a message is received by any transfer protocol binding plug-in, the component MessageReceiver retrieves the registered subsystem. Subsequently, the received message is dispatched to a call to the Foo subsystem.



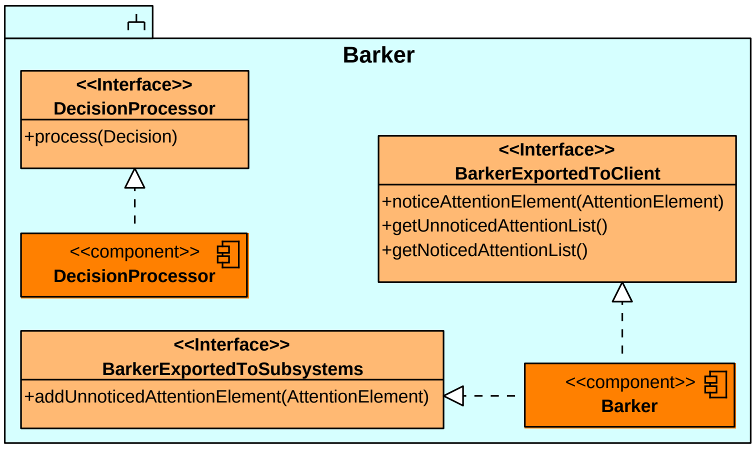
```
receiveMessage():

  FooExportedToPeers foo =
    soulCallbackRegistry.getFoo();
  foo.callFunctionality();
```

Figure 7.14: The exemplary subsystem Foo and its exported interfaces

In the following figures describing the Soul subsystems, the callback mechanism to the transfer access sublayer, any ports, and explicit interface exports are omitted. The subsystem’s main interface together with the implementation are abstracted to a UML component. The interface exported to clients is depicted above the main interface and the interface exported to subsystems to the left. The interface exported to peers is contained in the transfer access sublayer.

7.4.1 Lateral Barker

System-user interaction was decomposed into the lateral *Barker* subsystem. The component Barker encapsulates system-to-user interaction by providing the presentation layer with elements, that require the user's attention. Two lists of attention elements are managed: The first list collects attention elements not yet noticed by the user, the second list contains already noticed ones. For keeping the elements persistent, the Barker subsystem instruments the data access sublayer. The attention elements and the basic distinction between notice elements and pleas is described in section 7.2.4.



Barker
<<Interface>>
DecisionProcessor
+process(Decision)
<<component>>
DecisionProcessor
<<Interface>>
BarkerExportedToClient
+noticeAttentionElement(AttentionElement)
+getUnnoticedAttentionList()
+getNoticedAttentionList()
<<Interface>>
BarkerExportedToSubsystems
+addUnnoticedAttentionElement(AttentionElement)
<<component>>
Barker

Figure 7.15: The lateral Barker subsystem for user-system-interaction

The Barker component provides other subsystems with a method to add attention elements to the unnoticed attention list. This method is included in the interface BarkerExportedToSubsystems. To the presentation tier, methods to retrieve the two attention lists are offered. If a user marks an attention element as noticed, the method noticeAttentionElement moves it to the list of noticed attention elements. These methods are contained in the interface BarkerExportedToClient.

A further distinction needs to be made between noticing a plea and deciding about it. While noticing of it is done by the Barker component, decision making is processed by the component DecisionProcessor. It offers the method process(Decision) to the presentation tier and thus embodies user-to-system interaction. A Decision constitutes a plea together with a boolean value that reflects the decision made about the plea. DecisionProcessor delegates the requested processing to decision processors that were previously registered by other Soul subsystems. Since decision processing is closely related to certain subsystems, decision processors reside at these subsystems to achieve high cohesion. DecisionProcessor constitutes a *facade* [GHJV95, p. 185] for processing various kinds of decisions. Decision processing is explained together with the subsystems that handle related decisions.

7.4.2 Lateral Gatekeeper

By bundling methods for registration of users and logging them in and out of the DEUS system, the *Gatekeeper* subsystem was created. It depends on the data model module and on the data access sublayer for storing registered account information and login state.

7.4.2.1 Registration of User Accounts

For user sign up, the Registrator component offers methods for registering and unregistering a user. It is outlined in figure 7.17. The method register takes an instance of RegistrationInformation as an argument. This object contains the local username, a password, the desired type of user ID, and user metadata that was described in section 7.2.1. Furthermore a set of distribution roles[1] is included, that should be assumed by the account to register. These can either be chosen by the user during sign up or the DEUS node is preconfigured with a collection of roles to be assumed by new accounts.

On calling the method register, a user ID of the desired type is generated by delegating to an implementation of the strategy [GHJV95, p. 315] interface UserIdGenerator. This is outlined in figure 7.16 where the user Alice registers a DEUS account. Currently, the generation of UserUrls is supported by using a preconfigured server base URL and appending the local username chosen during registration. Subsequently, an account is created and persisted by instrumenting the data access sublayer.

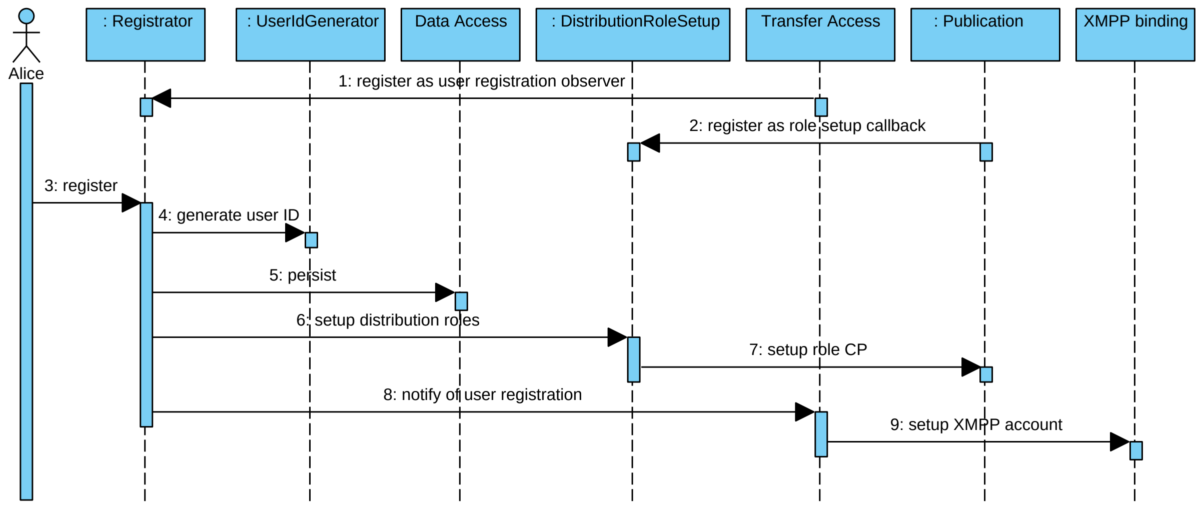


Figure 7.16: The registration of a user and subsequent actions of role setup and observer notification

1 See section 4.2 for more information about distribution roles.

Setting up a distribution role may require rather sophisticated operations like the creation of DEUS artifact data structures. Thus this functionality is factored out to DistributionRoleSetup which is presented in figure 7.17. Artifacts associated with distribution roles are related to specific Soul subsystems. Thus, role initialization procedures are bundled with these subsystems to maintain high cohesion. Subsequently, callbacks are registered by these subsystems on start up that are used by DistributionRoleSetup.

Accordingly, the subsystem Contribution is responsible for setting up the distribution role Information Provider. Each of the subsystems Repatriation Hub, PIF-Governing, and Publication offers a callback for collectively setting up the role Concerned Person. Finally, the role Information Consumer is governed and set up by the subsystems Subscription and DIF-Governing. Figure 6.2 outlines the relation of subsystems to distribution roles.

The distribution role setup callbacks are exemplarily outlined by the subsystem Publication registering a callback in step 2 in the above diagram. After account creation, each registered callback for each desired distribution role is used to setup the required data structures.

Subsequently any observers [GHJV95, p. 293] that previously subscribed to user sign up events (in step 1 in the above figure) are notified and passed the ID of the new user account. One of these observers is a component of the transfer access core layer that broadcasts this event to callbacks of all transfer protocol bindings. Notification of user sign ups may be needed by transfer protocols that require a protocol-specific account to be created, like the XMPP binding does. On adding a new transfer protocol binding that requires user accounts, these accounts must be created for all users of the DEUS node. While this is not included in the reference implementation, it is deferred to the future.

7.4.2.2 Unregistration of User Accounts

Unregistration of a user removes the Account entity from the data store and tears down any distribution roles the account assumed. This again involves the callbacks registered by the Soul subsystems which also provide support for distribution role teardown. It furthermore may include the termination of existing trust relationships being conceptually connected to the torn down distribution role. Subsequently, all observers listening for account unregistration are notified, among other things resulting in transfer protocol accounts to be removed.

7.4.2.3 Changing of Account Data

The component AccountManager of the gatekeeper subsystem provides support for changing user accounts after sign up and is outlined in figure 7.17. This includes changing of user metadata and password as well as adding and removing roles assumed by the account. The same mechanisms as during account registration and unregistration are applied for distribution role setup and teardown.

7.4.2.4 Logging Users In and Out of DEUS

The Cerberus[1] component provides the system with a facility for logging users in and out of their accounts. It is outlined in figure 7.17. After checking the login credentials, the logged in state of the Account entity is updated in the backing data store. As with the Registrator, the Cerberus component provides interested subsystems with the ability to listen for users logging in and out. This can be used by transfer protocol bindings to log in the users into their protocol-specific accounts after DEUS login. For logging a user out of its DEUS account, the login state is updated and persisted again and, in turn, login state listeners are notified.

7.4.2.5 Elements of the Gatekeeper Subsystem

The functionality of the Gatekeeper subsystem can be divided into four main scopes of work. The registration part copes with the registration of users and the setup of their accounts. Observers can be added to be noticed of user sign up events. This part is outlined in figure 7.17 in the upper left corner. Interfaces being implemented by other subsystems like observer interfaces are displayed yellow. The second part of the Gatekeeper functionality copes with the setup of distribution roles, outlined in the lower left corner. Logging users in and out of DEUS is done by a the login part and its Cerberus component. The management of accounts after sign up is accomplished by the account management functionality of the Gatekeeper subsystem.

7.4.3 The Contribution Subsystem

The Soul subsystem *Contribution* supports the functional role Contributor[2] and is outlined in figure 7.18. It offers a facility to import Digital Cards into the DEUS system by exporting the method forwardToCp to clients. The parameters of the method include the Digital Card to contribute and the user IDs of the Information Provider and the Concerned Person to whom the contribution is addressed. Subsequently, the Digital Card is forwarded to the DEUS account of the Concerned Person which constitutes the repatriation phase. Communication during this phase is supported by the transfer access sublayer.

For receiving the decision, whether the Digital Card to repatriate was accepted by the Concerned Person, the component Contributor exports an interface to DEUS peers. This interface

1 Cerberus is the name given to the entity which, in Greek and Roman mythology, is a multi-headed dog which guards the gates of Hades, to prevent those who have crossed the river Styx from ever escaping.

2 See section 4.2 for more information about functional roles.

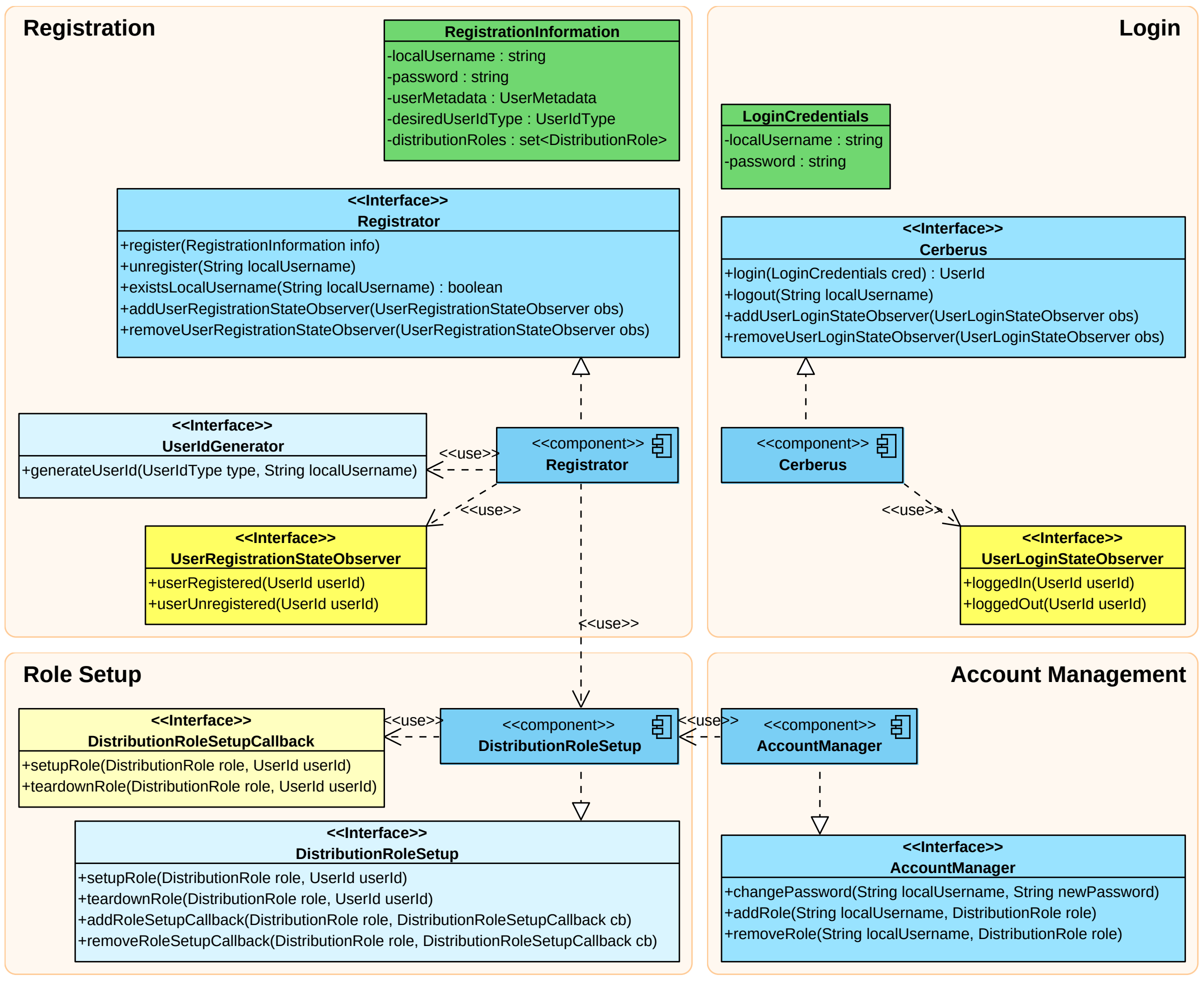

Registration
RegistrationInformation
-localUsername : string
-password : string
-userMetadata : UserMetadata
-desiredUserIdType : UserIdType
-distributionRoles : set<DistributionRole>
<<Interface>>
Registrator
+register(RegistrationInformation info)
+unregister(String localUsername)
+existsLocalUsername(String localUsername) : boolean
+addUserRegistrationStateObserver(UserRegistrationStateObserver obs)
+removeUserRegistrationStateObserver(UserRegistrationStateObserver obs)
<<Interface>>
UserIdGenerator
+generateUserId(UserIdType type, String localUsername)
<<use>>
<<component>>
Registrator
<<use>>
<<Interface>>
UserRegistrationStateObserver
+userRegistered(UserId userId)
+userUnregistered(UserId userId)
<<use>>
Login
LoginCredentials
-localUsername : string
-password : string
<<Interface>>
Cerberus
+login(LoginCredentials cred) : UserId
+logout(String localUsername)
+addUserLoginStateObserver(UserLoginStateObserver obs)
+removeUserLoginStateObserver(UserLoginStateObserver obs)
<<component>>
Cerberus
<<use>>
<<Interface>>
UserLoginStateObserver
+loggedIn(UserId userId)
+loggedOut(UserId userId)
Role Setup
<<Interface>>
DistributionRoleSetupCallback
+setupRole(DistributionRole role, UserId userId)
+teardownRole(DistributionRole role, UserId userId)
<<use>>
<<component>>
DistributionRoleSetup
<<use>>
<<Interface>>
DistributionRoleSetup
+setupRole(DistributionRole role, UserId userId)
+teardownRole(DistributionRole role, UserId userId)
+addRoleSetupCallback(DistributionRole role, DistributionRoleSetupCallback cb)
+removeRoleSetupCallback(DistributionRole role, DistributionRoleSetupCallback cb)
Account Management
<<component>>
AccountManager
<<Interface>>
AccountManager
+changePassword(String localUsername, String newPassword)
+addRole(String localUsername, DistributionRole role)
+removeRole(String localUsername, DistributionRole role)

Figure 7.17: The basic elements of the Gatekeeper subsystem

includes the methods contributionAcknowledged and contributionDenied. Both take the ID of the repatriated Digital Card as an argument and are remotely called by the DEUS account of the Concerned Person after deciding about the acceptance.

7.4.4 The Repatriation Hub Subsystem

The described forwarding of a contributed Digital Card results in a call to the method accept of the subsystem *Repatriation Hub* that is exported to peers. This subsystem supports the functional role Repatriation Authority with an entry point for contributed Digital Cards and is outlined in figure 7.19. Two different cases must be distinguished when receiving a Digital Card by the accept method: A user may contribute information about himself/herself instrumenting the Contribution subsystem. The loopback binding subsequently forwards the Digital Card to the Repatriation Hub subsystem. Here, the RepatriationHub component detects that the

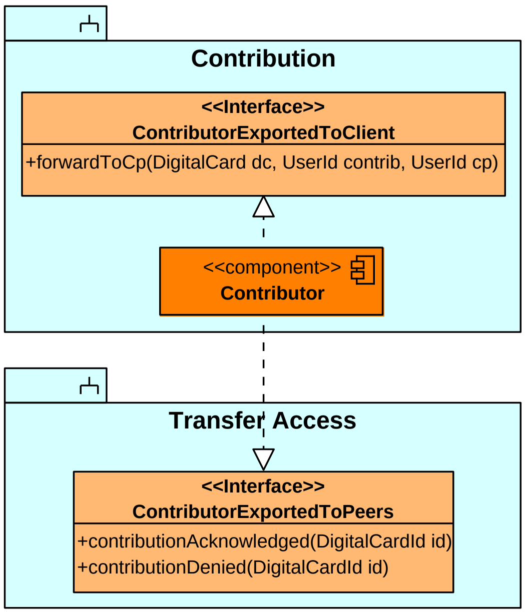

Contribution
<<Interface>>
ContributorExportedToClient
+forwardToCp(DigitalCard dc, UserId contrib, UserId cp)
<<component>>
Contributor
Transfer Access
<<Interface>>
ContributorExportedToPeers
+contributionAcknowledged(DigitalCardId id)
+contributionDenied(DigitalCardId id)

Figure 7.18: The Contribution subsystem and its exported interfaces

Information Provider of the Digital Card matches the Concerned Person by inspecting the Digital Card's primary key. Thus, the Digital Card is directly added to the Personal Information File as it will be described in the next section.

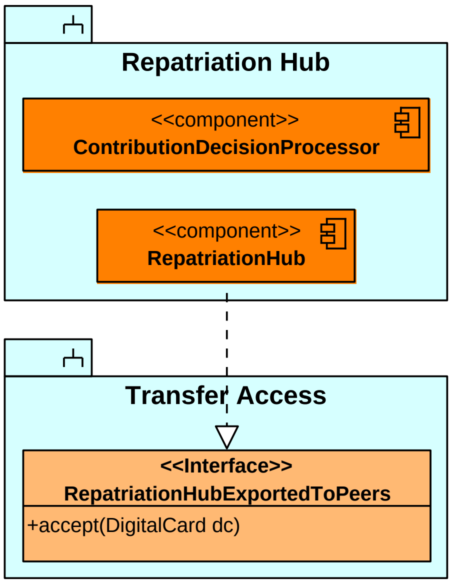


Figure 7.19: The Repatriation Hub subsystem and its exported interfaces

If the Digital Card is contributed by another DEUS account, the Information Provider of the Digital Card does not coincide with the Concerned Person. Thus, a decision about the acceptance of the repatriated Digital Card is required. It depends on the implemented trust model, introduced in section 4.4.1.2, if a plea for approving the Digital Card is displayed to the user. Currently, establishing trust relationships for the repatriation phase is not possible. Thus, each Digital Card contributed by a foreign account is presented to the user for acceptance by instrumenting the Barker subsystem. A plea attention element for accepting the contribution is added to the unnoticed attention list.

The processing of the decision made about this plea is contained in the component ContributionDecisionProcessor. On a positive acceptance decision, the contributed Digital Card is added to the Personal Information File and the method contributionAcknowledged of the Contribution subsystem is invoked. On a negative acceptance decision, the method contributionDenied is remotely called. Both invocations instrument the transfer access sublayer for communication with the other DEUS peers.

7.4.5 The PIF-Governing Subsystem

The subsystem *PIF-Governing* is responsible for managing the Personal Information File. As depicted in figure 7.20, it exports the method getPersonalInformationFile to the presentation layer. This allows a client to get read access to the Personal Information File. The method assimilateRepatriatedDigitalCard is offered to subsystems and consumed by the subsystem Repatriation Hub. On calling this method, a strategy [GHJV95, p. 315] is used that encapsulates different ways of merging new Digital Cards. The current implementation is SimpleAppendAssimilationStrategy that just appends the Digital Card to the Personal Information File. In case it already exists, an exception is thrown. The assimilation strategy returns a patch that manifests the difference between the old and the new Personal Information File state. Subsequently, the Personal Information File is updated in the backing data store.

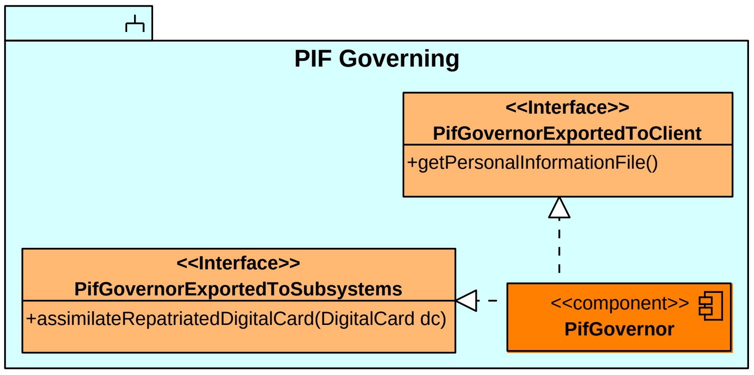

PIF Governing
<<Interface>>
PifGovernorExportedToClient
+getPersonalInformationFile()
<<Interface>>
PifGovernorExportedToSubsystems
+assimilateRepatriatedDigitalCard(DigitalCard dc)
<<component>>
PifGovernor

Figure 7.20: The PIF-Governing subsystem and its exported interfaces

7.4.6 The Publication Subsystem

The *Publication* subsystem offers support for publishing changes and building up pub/sub trust relationships. It exports the method getListOfSubscribers to the presentation tier, displayed in figure 7.21. The second method being called by clients is notifySubscribers. Different models for the *point in time of publication* can be thought of. Either, publication is done automatically after accepting a contributed Digital Card or it is only manually triggered by the user. In the latter case, a *list of patches to publish* must be maintained. Instrumenting notifySubscriber an individual subscriber group can be chosen to publish patches to. Since this method allows for notifying individual groups, patch queues need to be administrated per subscriber group. DEUS may process patch lists prior to publication to condense them by removing redundant changes. Currently DEUS supports the manual triggering of publication by invoking notifySubscribers. No maintaining of patch queues is supported.

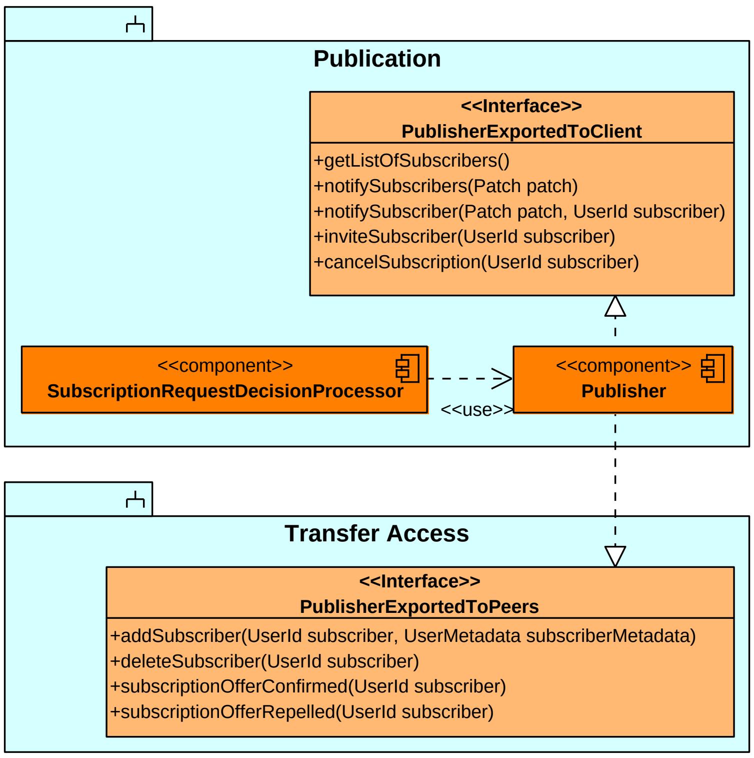

Publication
<<Interface>>
PublisherExportedToClient
+getListOfSubscribers()
+notifySubscribers(Patch patch)
+notifySubscriber(Patch patch, UserId subscriber)
+inviteSubscriber(UserId subscriber)
+cancelSubscription(UserId subscriber)
<<component>>
SubscriptionRequestDecisionProcessor
<<use>>
<<component>>
Publisher
Transfer Access
<<Interface>>
PublisherExportedToPeers
+addSubscriber(UserId subscriber, UserMetadata subscriberMetadata)
+deleteSubscriber(UserId subscriber)
+subscriptionOfferConfirmed(UserId subscriber)
+subscriptionOfferRepelled(UserId subscriber)

Figure 7.21: The Publication subsystem and its exported interfaces

Furthermore, DEUS supports the setup and teardown of publication relationships either initiated by the publisher or by the subscriber[1]. For the former, the Publication subsystem exports the method inviteSubscriber as well as cancelSubscription to the client. Inviting a subscriber

1 See section 4.4.1.3 for further information about publication trust relationships

results in a message to be sent to him/her, offering a subscription, which can either be confirmed or repelled by the subscriber. Subsequently the subscriber notices the publisher of this decision by remotely calling the method subscriptionOfferConfirmed or subscriptionOfferRepelled. In the former case, a publication relationship is built up by adding the subscriber to the list of subscribers. This inter-peer communication is accomplished using the send command interface of the transfer access sublayer. Canceling a subscription entails a notice message to be sent to the subscriber potentially demanding the deletion of the Foreign Information File. On publisher side the subscriber is removed from the list of subscribers.

A publication connection can also be requested by the subscriber. Therefore, the subscriber remotely invokes addSubscriber on the Publication subsystem. This results in a plea displayed to the user instrumenting the Barker subsystem. A decision made by the user is handled in the SubscriptionRequestDecisionProcessor. This component either calls grantSubscriptionRequest resulting in the subscriber being added to the list of subscribers. If the decision is negative, denySubscriptionRequest of the Publisher component is called. Both methods subsequently result in signaling the decision to the subscriber account instrumenting the transfer access sublayer.

7.4.7 The Subscription Subsystem

Publication trust relationships are logically built up between the functional roles Publisher and Subscriber, embodied by the Publication and the Subscription subsystems. While the part of the Publication subsystem in pub/sub connection management has been elaborated on before, this section describes the Subscription subsystem, outlined in figure 7.22.

In parallel to the Publication subsystem, it offers the method getListOfPublishers to the presentation tier. Furthermore, the method subscribeToPublisher for requesting a subscription is exported to clients. A call to this method entails the sending of a message to the publisher instrumenting the send command interface of the transfer access sublayer. After the decision about the subscription request is made on publisher side as described above, the decision is sent to the subscriber. This results in an invocation of either the method subscriptionRequestGranted or subscriptionRequestDenied the Subscription subsystem exports to peers. On confirmation of the connection, it is built up on subscriber side by adding the publisher to the list of publishers. Teardown of a subscription relationship initiated by the subscriber is triggered by a client calling unsubscribeFromPublisher.

A publisher offering a subscription results in a remote call to the method addPublisher on subscriber side. A plea is added to the Barker subsystem and thus presented to the user. The decision about this plea is handled by the SubscriptionOfferDecisionProcessor. Depending on the decision, it either calls confirmSubscriptionOffer or repelSubscriptionOffer on the Subscriber

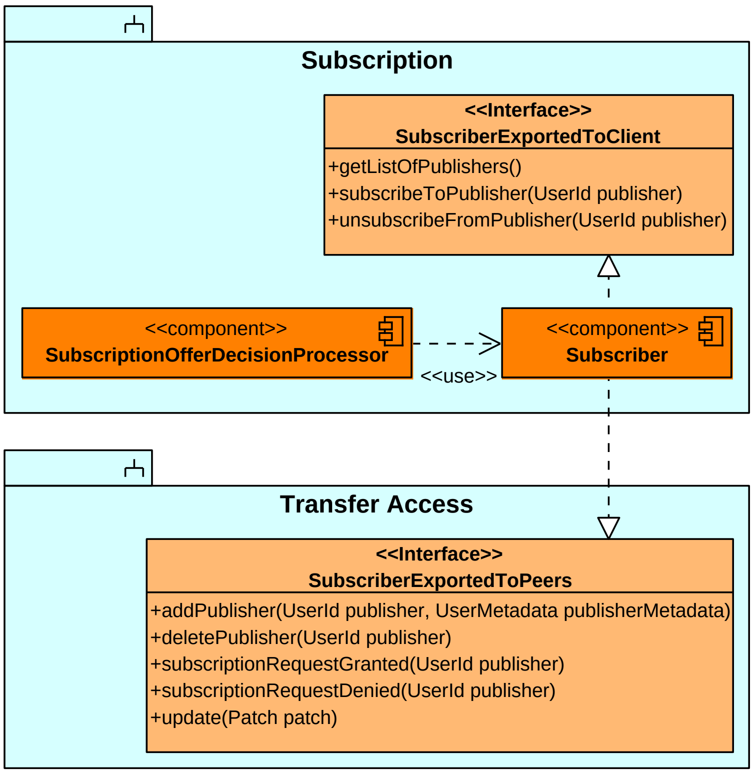

Subscription
<<Interface>>
SubscriberExportedToClient
+getListOfPublishers()
+subscribeToPublisher(UserId publisher)
+unsubscribeFromPublisher(UserId publisher)
<<component>>
SubscriptionOfferDecisionProcessor
<<use>>
<<component>>
Subscriber
Transfer Access
<<Interface>>
SubscriberExportedToPeers
+addPublisher(UserId publisher, UserMetadata publisherMetadata)
+deletePublisher(UserId publisher)
+subscriptionRequestGranted(UserId publisher)
+subscriptionRequestDenied(UserId publisher)
+update(Patch patch)

Figure 7.22: The Subscription subsystem and its exported interfaces

component. The former method adds the publisher to the list of publishers, and both methods report the decision to the publisher by sending a notifying message. The publisher demanding the termination of the subscription results in a call to the remotely exported method deletePublisher.

Furthermore, the Subscription subsystem exports the method update to other DEUS peers. This method is called by the transfer access sublayer after receiving an update message containing a patch that promotes a change to a Personal Information File. The Subscription subsystem checks if the originator of the message is contained in the list of publishers. Subsequently, the transferred patch is merged to the publisher's Foreign Information File using the DIF-Governing subsystem.

7.4.8 The DIF-Governing Subsystem

The *DIF-Governing* subsystem is responsible for managing the Distributed Information Folder of an Information Consumer and depicted in figure 7.23. By its functionality, it supports the functional role Retriever while only a part of it is yet implemented. Therefore, it offers the method getDigitalCardIdsInFif that returns a list of the IDs of all Digital Cards contained in a Foreign Information File. By furthermore instrumenting the method getDigitalCardInFif, a client can access a Digital Card with a given ID. Currently, methods for searching a Foreign Information File are not implemented but deferred to future work.

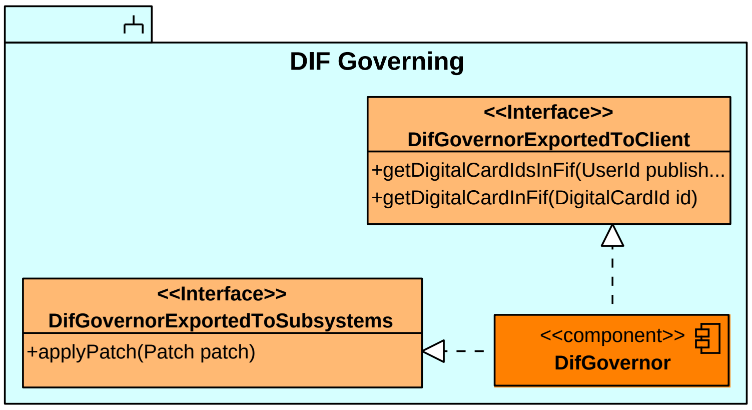

DIF Governing
<<Interface>>
DifGovernorExportedToClient
+getDigitalCardIdsInFif(UserId publish...
+getDigitalCardInFif(DigitalCardId id)
<<Interface>>
DifGovernorExportedToSubsystems
+applyPatch(Patch patch)
<<component>>
DifGovernor

Figure 7.23: The DIF-Governing subsystem and its exported interfaces

To other subsystems, the method `applyPatch` is exported which is called after a patch from a publisher is obtained. This method applies the transferred patch to the Foreign Information File.

7.5 Conclusion

By packaging the modules obtained by decomposing DEUS as OSGi bundles, clear module boundaries and interfaces can be enforced during runtime. Functionality offered to other subsystems, to clients or to other DEUS peers is exported as OSGi services. The DEUS system follows a three-tier architecture, where the business logic and techniques for integrating data stores and communication protocols are encompassed in the core tier. The subsystems contained in the Soul sublayer collaborate in providing the full functionality of the DEUS Contribution-Repatriation-Publication Chain. The scope of work of each subsystem as well as its interfaces and exported services were described using UML class and component diagrams.

8 Implementation Issues

One of the objectives of DEUS was to be independent of any platform. This was achieved by choosing Java as the programming language resulting in the possibility to run the program wherever a Java virtual machine is installed. While a specific programming language was chosen DEUS itself is not dependent on Java. The DEUS functionality can also be implemented in a different language. All used techniques for inter-peer communication are also available for other programming languages. The proposed reference implementation is compatible with implementations written with different languages running on other nodes.

For supporting the build process, Maven is used. Maven provides a declarative configuration of a predefined build life cycle and adheres to 'convention over configuration'. Furthermore, the Spring Framework was instrumented to introduce Inversion of Control and, thus, fosters loose coupling. The Spring sub-project Spring Dynamic Modules provides an integration of OSGi with declarative importing and exporting of OSGi services. The whole application is deployed as OSGi modules into the Spring DM application server.

8.1 Build Environment

In the following, it is outlined how Maven as the used build tool was applied for supporting the build process of the DEUS system with its multiple modules.

8.1.1 Maven as Build System

During build time, *Maven 2*[1] was used as a build tool. The principles of Maven are adoption of a predefined build process and 'convention over configuration' [vZB07]. A Maven project is declaratively configured by using the Maven *Project Object Model*. Projects are identified by Maven coordinates, their dependencies are managed by Maven. A default project structure is provided by Maven, together with a build cycle that can be extended by plug ins. Multi-module projects and project inheritance support the creation of enterprise-level software with multiple modules.

1 http://maven.apache.org

Further information about Maven can be found in chapter A in the appendix. There, more details are provided about Maven coordinates, versioning of projects, dependency management, project folder structure, the Maven build cycle, plug ins, the command line interface, and multi-module projects.

8.1.2 DEUS Maven Project Layout

DEUS uses Maven as a build tool and all the features described in chapter A. The group ID of all DEUS Maven modules is inf6.promed. Since Maven does not support hierarchical artifact IDs, dashes were used to separate hierarchy levels. An exemplary coordinate of one of the DEUS modules is inf6.promed:deus-core-access-transfer-plugins-xmpp:bundle:0.3-SNAPSHOT. This coordinate addresses the XMPP protocol binding plug-in, packaged as OSGi bundle in version 0.3-SNAPSHOT.

Dependencies common to all submodules of DEUS include JUnit[1] and the Spring Framework Test component. Dependencies managed on behalf of the submodules, as described above, include all intra-project dependencies. Furthermore the Spring Framework is included in the managed dependencies.

The decomposition of the system into sublayers and subsystems is reflected by a special folder structure. Inner nodes incorporate pom packaged projects, leafs incorporate OSGi bundles with packaging type bundle. Each Maven module reflects its position in the folder tree by its artifact ID. It constitutes the folder names on the path to the project root folder separated by dashes. The DEUS project structure is outlined in figure B.1. All of the modules described in chapter 7 are bundled as OSGi modules which is reflected by the project folder structure. Furthermore, the root POM[2] file and an exemplary POM file of the Gatekeeper subsystem can be found in chapter B.

8.2 Spring Framework and Spring DM Server

The *Spring Framework*[3] provides a lightweight Inversion-of-Control-container. Further features include support for AOP[4], abstraction of enterprise technologies like JMS, JMX[5], and JCA[6] and an abstraction layer for data access. By its dependency injection facility, the Spring framework

1 http://www.junit.org
2 Project Object Model
3 http://www.springframework.org
4 Aspect-Oriented Programming
5 Java Management Extensions
6 Java Connector Architecture

fosters low coupling between system components which are embodied by Spring beans. It eases the development by providing autowiring of dependencies and declarative component detection using Java annotations.

8.2.1 Spring Dynamic Modules

The Spring sub project *Spring Dynamic Modules (DM)* provides an integration for OSGi. Spring beans can be exported as OSGi services in a declarative way. Importing of services from the OSGi service registry creates Spring beans that can be directly injected into other beans.

Furthermore, the creation of a Spring application context is automated. Therefore, a bundle from the Spring DM project is loaded into the OSGi container prior to application deployment. This bundle detects Spring powered OSGi bundles by searching for Spring XML configuration files in the folder /META-INF/spring/ of the JAR[1] file. Any XML files that are found are subsequently loaded and used to create an application context for this bundle.

Using Spring DM not only eases development, but also avoids any dependency on OSGi interfaces by exporting POJOs[2] as OSGi services in a declarative way. Furthermore, Spring DM provides the facility of calling specially annotated methods directly after start up of the bundle. Thus, the concept of a OSGi bundle activator is replaced by arbitrary classes implementing start up and teardown methods.

8.2.2 Spring Framework Applied

For each OSGi bundle, the Spring configuration resides in the JAR file at /META-INF/spring/, where several individual Spring configuration files are included in all.xml. The file /META-INF/spring/deus/context.xml contains the context configuration. This includes the element <context:annotation-config /> that enables the declarative configuration of dependencies using Java annotations.

Get- or set-methods and class fields can be annotated with @Required to declare a required dependency. Furthermore, the automatic wiring of dependencies without any necessary declarations is enabled with the annotation @Autowired. Methods annotated with @PostConstruct or @PreDestroy are executed by the Spring container after the object is configured, respectively before it is destroyed. The configuration element <context:component-scan base-package="deus.core.soul"/> enables the active detection of Java classes annotated with @Component. Annotated classes below the given base package are added to the list of Spring beans without explicitly declaring

1 Java Archive
2 Plain Old Java Objects

them as beans in an XML configuration. These two configuration elements dramatically reduce the need for XML configuration in favor of annotation-based configuration.

The following listing provides the class RegisterLoopbackTransferProtocol that registers the loopback transfer protocol binding after the OSGi bundle is loaded. The class is automatically added as Spring bean since it is annotated with @Component. Furthermore, its two dependencies are autowired. Due to their annotations, the method register is called after class setup, the method unregister prior to class tear down.

```java
@Component
public class RegisterLoopbackTransferProtocol {

    @Autowired
    private ExportedTransferProtocolRegistry registry;

    @Autowired
    private TransferProtocol loopbackProtocol;


    @PostConstruct
    public void register() {
        registry.registerTransferProtocol(loopbackProtocol);
    }


    @PreDestroy
    public void unregister() {
        registry.unregisterTransferProtocol(loopbackProtocol.getId());
    }

}
```

The file /META-INF/spring/deus/osgi.xml is also included in each bundle and specifies imported and exported OSGi services. The following listing shows one exported and one imported OSGi service. The first declaration exports the bean publisher under the given interface into the service registry. The second statement imports a service with a given interface from the service registry and makes it available as Spring bean with the name pubDao.

```xml
<osgi:service id="publisherOsgiService" ref="publisher" interface="deus.core.
    soul.publication.PublisherExportedToClient" />

<osgi:reference id="pubDao" interface="deus.core.access.storage.api.sub.
    LopEntryDao"/>
```

8.2.3 Spring DM Server

Since DEUS is composed by several OSGi bundles, an OSGi container is needed for deploying the application. For this purpose *Spring DM server* is used. It builds on top of the OSGi runtime Equinox[1] and provides further support for Spring powered OSGi bundles. Bundles to be deployed are dropped into a pickup folder and started by the server immediately.

8.3 Conclusion

By instrumenting Maven as build tool, the build process of DEUS follows conventions established by Maven. The benefits are the use of Maven's versioning mechanism, the facility for dependency management, usage of the default Maven project structure, as well as support for enterprise-level software development. The folder layout of DEUS reflects the decomposition of the system into modules, where each module incorporates its own Maven project. The Spring Framework supports loose coupling by instrumenting its dependency injection container. Declarative configuration using Java annotations reduces complexity during development. Spring Dynamic Modules was used for supporting the creation of Spring-powered OSGi modules that eventually were deployed into the Spring DM application server.

1 http://www.eclipse.org/equinox

9 Future Work

While designing DEUS, many points of extensibility were discovered. Thus, on the one hand, this chapter provides an overview of crucial features that were conceptually elaborated but not yet implemented. On the other hand, issues requiring deeper investigation and further conceptual work are outlined.

9.1 Implementational Features

While the concepts of *repatriation relationships* have been described in section 4.4.1.2, they are not yet included in the reference implementation of DEUS. Since implementing these relationships is analogous to the relationships in the publication phase, the effort is manageable. Two different trust models have been described that introduce a plea for repatriating a Digital Card being presented to the user. These trust models can be added to the reference implementation in the course of implementing repatriation relationships. Furthermore, a *history* of all contributed Digital Cards for each Concerned Person can be managed at the Information Provider. Since this only involves a data structure that collects contributed Digital Cards and associated Concerned Persons, implementation is straightforward. This would allow the Information Provider to obtain an overview over contributed Digital Cards.

In section 7.3.2, the negotiation of a used transfer protocol on sending a message has been described. This involves discovering transfer protocols being available to the target DEUS peer. Subsequently, a simple algorithm may be implemented that chooses one protocol out of the common subset of both communication partners, taking attached priorities into account. For discovering metadata of URI-based resources over the Internet, the discovery protocol stack around XRD has been described in section 5.2. The current reference implementation takes transfer protocol negotiation into account by externalizing related code into strategies [GHJV95, p. 315]. However, the described *discovery mechanisms* have not been integrated yet. The reference implementation is prepared for transfer protocol negotiation and the concepts behind discovery have been elaborated.

Plugging in a new transfer protocol binding may involve protocol-specific account generation for all registered users of the DEUS node. Currently, transfer protocol accounts are generated during user sign up for all available protocol bindings. If a new binding is added while user accounts already exist, these DEUS accounts need to be provided with accounts specific to the new transfer protocol. The reference implementation is lacking this feature of *supplementary protocol-specific account generation.*

The emphasis of this thesis was more on creating an extensible system design that allows for plugging in arbitrary transfer protocols than on concrete binding implementations. Thus, the current XMPP binding awaits completion by implementing an *XML binding* for the party information model described in section 6.3. Furthermore, other possibilities offered by XMPP and its extensions need to be evaluated to complete the implementation of trust relationship establishment and termination. While currently the presence protocol part of XMPP is used to manage subscriptions, a shift to the publish/subscribe extension of XMPP [MSAM09] or to group chats may provide better support. Another possible binding can be created by instrumenting a RESTful API including the implementation of message queues to guarantee message delivery. Besides regarding REST as a protocol binding for transferring messages to DEUS peers, it can also be instrumented for exporting DEUS functionality to neighbor systems. Interfaces that are exported by the core tier to clients, thus, could be remotely invoked by non-DEUS systems using a RESTful API. These interfaces are also instrumented to control DEUS using an intended web interface that also awaits its realization. Currently, no user interface is included with the reference implementation.

XRI has been described in section 5.1.2 as another type of user ID that can be integrated. In the course of this thesis, XRI has been evaluated as too complex and not wide-spread enough. Nonetheless, its concept of addressing real-world entities like persons and organizations instead of web resources fits well into the concept of DEUS. Thus, XRI may be integrated in the future depending on its maturity and the acceptance obtained among Internet users.

Furthermore, a standard named *Portable Contacts* [Sma08] provides an API that allows access to existing address books. This API can be implemented in order to export address information of all publishers included in the Distributed Information Folder as an address book.

9.2 Conceptual Issues

While the previous features can be implemented in a straightforward way, the following concepts require deeper evaluation and more conceptual work. In the following, future conceptual work motivated by the healthcare problem domain is distinguished from concepts whose adaption is motivated by technical issues.

9.2.1 Future Work Motivated by the Problem Domain

A local *search* could empower users of a DEUS node to search for other users. Any DEUS user may decide to include arbitrary information in an index that is used for searching. That solves the 'first contact problem' of local users getting known to each other. Since each user is uniquely identified by its user ID, a request to establish a connection can be issued as soon as this ID is known. Without possessing the ID of the communication counterpart, methods like the local search need to be employed to discover IDs. While the same applies for distributed search, this is even more difficult since no central index is available. The problem of searching a network without a central infrastructure also appears with peer-to-peer file sharing protocols. XMPP also offers an extension for distributed search, that may be evaluated [SA04]. Thus, further investigation is needed for implementing a comprehensive, local and global search for DEUS account.

More conceptual work is also required on various topics that have already been touched on briefly. This includes *filtering* during the publication phase by the creation of subscriber groups, as described in section 4.4.3, and the *initial publishing* of historic Digital Cards, elaborated on in section 4.4.5. Furthermore, requirements of *authentication mechanism* in healthcare systems need to be investigated. Although, DEUS system design has considered the collaboration scenario of section 3.4, concrete support for it requires the implementation of the *injection of lists of trusted peers.*

A modular way of providing DEUS with knowledge about the content of exchanged Digital Cards is to introduce a *plug-in model for different kinds of Digital Cards.* This is a precondition for *adding other assimilation strategies*, described in section 4.4.2. Since these strategies would be aware of the content of digital cards, automatic assimilation becomes possible. This could be accompanied by the introduction of a model of associations between Digital Cards. The result of these more complex assimilation strategies would be a patch incorporating a more complex data structure, subsequently being sent to subscribers.

The described Contribution-Repatriation-Publication Chain requires the patient to govern information exchange. This could become problematic since the patient may not always be able to decide which subscribers should receive which repatriated Digital Card. A possible solution for this issue may include 'hints' being provided by the Information Provider during repatriating the Digital Card. These hints could indicate which subscriber groups should receive this Digital Card. The patient subsequently either follows these hints or decides to chose other subscribers. This would require the Information Provider to be aware of the subscriber groups existing at the Concerned Person.

Besides obtaining Digital Cards with a given ID, the functional role Retriever should also enable searching a Foreign Information File. This requires knowledge of the content of Digital Cards and methods for scanning it. As soon as this is specified, searching in a Distributed Information Folder can be implemented and exported to existing HCIS.

9.2.2 Future Work Motivated by Technical Issues

Together with the advent of the XRI standard, a method for semantic markup of relationships known as *XDI*[1] has been developed [RS04]. XDI embodies a generic service for sharing, linking and synchronizing data using XML documents and XRIs. It includes the high level concept of link-contracts that enable control over authorizing and securing shared data. Together with XRI as identifier scheme, XDI can be introduced as another transfer protocol binding.

Furthermore, the focus during the implementation of trust relationships was on the publication phase. Here, an XMPP binding is employed to map 1:N communication. As described in section 4.6.2, the requirements of transfer protocol bindings in the repatriation phase are different from the ones in the publication phase. Therefore, REST as an architectural style implemented by HTTP has been proposed in section 5.3.1 that partially fulfills the requirements. Thus, a *RESTful API* may be specified that serves as another transfer protocol binding.

Another issue is the *relocation of existing DEUS accounts.* A patient may wish to change the provider of his/her DEUS account, for example, after changing the health insurance company, where the account was hosted. Physicians assuming the roles of Treatment Provider and Audience Party may change their user IDs due to renaming. Thus, a mechanism needs to be established that either changes existing entries of the list of contributors, subscribers and publishers, described in section 4.4.1, or forwards requests to the new account. This heavily depends on the type of identifier: While XRIs are abstract per definition another service endpoint may be returned by the resolution process [WRC+06]. Since URLs lack this abstract definition, either existing list entries need to be updated or the old account needs to be further maintained to redirect requests to the new one.

Currently, logic for the establishment and termination of trust connections is distributed over several subsystems. The subsystems Publication and Subscription encompass methods exported to clients and peers for connection management for the publication phase. Connection setup of repatriation relationships is eventually distributed over the subsystems Contribution and Repatriation Hub. Both types of relationships share facilities for establishing and terminating them, potentially initiated by both communication partners. Thus concepts of relationship

1 XRI Data Interchange

management can be factored out to a *generic relationship component* that depends on the Barker subsystem for user-system interaction. Callbacks by specific subsystems may be injected that provide notifications of relationship events. A method may be provided by this component to retrieve the list of communication partners. This may, for example, be needed by the Publication subsystem in order to publish Personal Information File state changes or by the Repatriation Hub subsystem to implement both trust models described in section 4.4.1.2. The ability to initiate establishment or termination of a connection may be restricted by parameterizing the relationship component. A generic scheme for piggy-backing arbitrary payload onto establishing and termination messages may be provided. This may, for example, be used for piggy-backing a first Digital Card to contribute onto the repatriation relationship request, as explained in section 4.4.1.2. By introducing this relationship management component, logic for connection setup and teardown would no longer be distributed over several subsystems. By following the separation of concerns design principle, it would be easier to add extensions to the mechanism of trust relationship establishing like certificate checking or mutual validation of email addresses.

10 Conclusion

In Germany, patients are treated by office-based physicians constituting the primary care and hospitals, laboratories, and pharmacies of the secondary care. If the spectrum of diagnosis or treatment provided by an institution is not sufficient, the patient is referred to another institution. This often involves repeated anamnesis interviews since the patient's health record is not available to the referred institution. Patient-related information exchange is confined to the postal delivery of paper-based documents that must explicitly be requested. A system that supports comprehensive exchange of patient-related information between institutions is missing. Existing healthcare information systems focus on the secondary care and require a central infrastructure such as indexes or federated databases.

With DEUS, an acronym for Distributed Electronic Patient File Update System, a solution has been proposed that provides seamless, inter-institutional, cross-organizational flow of information. Originally, the topic was motivated by a lack of standards for synchronizing contact data on the Internet. While protocols for accessing address books exist, the synchronization problem has not yet been solved thoroughly. With DEUS a system has been created that follows a publish/subscribe architecture to distribute user-centric information to interested third parties assuming the role of subscribers. A collection of information published by various users is managed on the subscriber side forming the abstract concept of a Distributed Information Folder. In the context of address and contact data, this folder is named Distributed Address Book. Following the DEUS approach, information about foreign parties is no more managed by the owner of the address book, but by the owner of the address entry. A publication of any changes to a user's contact information results in connected address books being updated. The distributed nature of DEUS allows for arbitrary parties to provide DEUS nodes for the management of address data. While a de facto standard of exporting address books named Portable Contacts provides an API for accessing address books, DEUS tackles the problems of updates and synchronization that are explicitly deferred by Portable Contacts.

With the focus on contact data, publishing information solely contributed by the account owner and allowing subscribers to receive notifications is sufficient. To also be applicable in healthcare, DEUS furthermore allows for the contribution of information by third parties which is incorporated by the process of 'repatriation'.

By the abdication of any central infrastructure, local autonomy of healthcare institutions is preserved. DEUS follows the current working practice by adhering to a document-oriented design principle. No special platform or integration technique is required, so that it can be deployed to the current, heterogeneous infrastructure. As an extension to existing healthcare information systems, it provides the contribution of Digital Cards, constituting viable documents, to patient files. Current systems employ DEUS by either bundling information into a Digital Card and contributing it or by retrieving Digital Cards from subscribed patient files. The patient governs his health record and the distribution of parts of it to other institutions that require access. The feasibility of the concept is proven by a reference implementation of DEUS. A Digital Card containing patient master data has been chosen as the first type of information handled by DEUS. While DEUS is agnostic of the content of exchanged Digital Cards, this decision is due to the fact that patient master data occurs in all health records. By adhering to the described approach, DEUS offers a solution for the comprehensive problem of generic, user-centric data interchange which can be deployed and built upon in every domain requiring this kind of architecture.

Appendices

Bibliography

[ACC07] H. ACC. RSNA.(2005). Integrating the Healthcare Enterprise-IT Infrastructure; Technical Framework, Volume 1,(ITI TF-1): Integration Profiles. from http://www.ihe.net. *Technical_Framework/upload/ihe_iti_tf_2.0_vol1_FT _2005-08-15.pdf*, 1:157, 2007.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform resource identifier (URI): Generic syntax. *The Internet Society*, 2005.

[DH98] F. Dawson and T. Howes. RFC 2426: vCard MIME directory profile, September 1998.

[Ecl08] Eclipse project. Higgins Open Source Identity Framework. `http://www.eclipse.org/higgins/`, 2008.

[EFGK03] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *ACM computing Surveys*, 35(2):114–131, 2003.

[eHe03] eHealth. Ministerial Declaration, Brussels. `http://europa.eu.int/information_society/eeurope/ehealth/conference/2003/doc/min_dec_22_may_03.pdf`, May 2003.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol–HTTP/1.1, June 1999. *Status: Standards Track*, 1999.

[Fie00] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, 2000.

[Fow03] M. Fowler. *Patterns of enterprise application architecture.* Addison-Wesley Professional, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of reusable software.* Addison-Wesley Professional, January 1995.

[GHNO06] J. Gregorio, M. Hadley, M. Nottingham, and D. Orchard. URI Template. *Network Working Group, Internet Draft*, 2006.

[HBS02] Mark Hapner, Rich Burridge, and Rahul Sharma. Java Message Service, April 2002.

[HL08a] Eran Hammer-Lahav. Beginner's Guide to Discovery. `http://www.hueniverse.com/hueniverse/2008/07/beginners-guide.html`, July 2008.

[HL08b] Eran Hammer-Lahav. Discovery and HTTP. `http://www.hueniverse.com/hueniverse/2008/09/discovery-and-h.html`, September 2008.

[HL08c] Eran Hammer-Lahav. XRDS-Simple specification. `http://xrds-simple.net/core/1.0/`, March 2008.

[HL09a] Eran Hammer-Lahav. Link-based Resource Descriptor Discovery. *IETF*, 2009.

[HL09b] Eran Hammer-Lahav. The Discovery Protocol Stack. `http://www.hueniverse.com/hueniverse/2009/03/the-discovery-protocol-stack.html`, March 2009.

[HW03] G. Hohpe and B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[LBK07] R. Lenz, M. Beyer, and K.A. Kuhn. Semantic integration in healthcare networks. *International journal of medical informatics*, 76(2-3):201–207, 2007.

[Len97] R. Lenz. Adaptive Datenreplikation in Verteilten Systemen, volume 23 of Teubner-Texte zur Informatik (TTzI). *Teubner Verlag, Leipzig, Germany,*, 1(0):46, 1997.

[LSLG00] K. Lorig, D. Sobel, D. Laurent, and V. Gonzalez. *Living a Healthy Life With Chronic Conditions: Self-management of Heart Disease, Arthritis, Diabetes, Asthma, Bronchitis, Emphysema & Others.* Bull Publishing Company, 2000.

[MSAM09] Peter Millard, Peter Saint-Andre, and Ralph Meijer. XEP-0060: Publish-Subscribe, February 2009.

[NHL09] M. Nottingham and Eran Hammer-Lahav. Host Metadata for the Web. *IETF*, 2009.

[Not09] M. Nottingham. Link Relations and HTTP Header Linking. *IETF*, 2009.

[Pat03] Nandish Patel. Deferred system's design: countering the primacy of reflective IS development with action-based information systems. *Adaptive Evolutionary Information Systems*, pages 1–28, 2003.

[PB05] J. Powell and I. Buchan. Electronic Health Records Should Support Clinical Research. *Journal of Medical Internet Research*, 7(1), 2005.

[RL03] S.K. Rothschild and S. Lapidos. Virtual Integrated Practice: Integrating Teams and Technology to Manage Chronic Disease in Primary Care. *Journal of Medical Systems*, 27(1):85–93, 2003.

[RLHJ99] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 Specification. *W3C Recommendation REC-html401-19991224, World Wide Web Consortium (W3C)*, pages 154–156, 1999.

[RM05] Drummond Reed and Dave McAlpin. Extensible Resource Identifier Syntax 2.0, OASIS Committee Specification, OASIS XRI Technical Committee, November 2005.

[RS04] Drummond Reed and Geoffrey Strongin. The dataweb: an introduction to XDI. *White paper of the OASIS XDI Technical Committee*, 2, 2004.

[SA04] Peter Saint-Andre. XEP-0055: Jabber Search, March 2004.

[Sam04] Josie Samers. Report on Integrated Care in Advanced Cancer Project. Technical report, Inner and Eastern Melbourne BreastCare Consortium, mar 2004.

[Sma08] Joseph Smarr. Portable Contacts 1.0 Draft C. `http://www.portablecontacts.net/draft-spec.html`, August 2008.

[vZB07] J. van Zyl and E. Bjørsnøs. *Maven: The Definitive Guide.* O'Reilly Media, 2007.

[WRC+06] G. Wachob, D. Reed, L. Chasen, W. Tan, and S. Churchill. Extensible Resource Identifier (XRI) Resolution v2.0, March 2006.

[WVM01] M.H. Williams, G. Venters, and D. Marwick. Developing a regional healthcare information network. *Information Technology in Biomedicine, IEEE Transactions on*, 5(2):177–180, 2001.

List of Figures

List of Abbreviations

AOP	Aspect-Oriented Programming
API	Application Programming Interface
CDA	Clinical Document Architecture
CRUD	Create, Read, Update, Delete
DNS	Domain Name System
EERD	Enhanced Entity-Relationship Diagram
EHR	Electronical Health Record
ERD	Entity-Relationship Diagram
FTP	File Transfer Protocol
GNU	GNU[1] is Not Unix
HCIS	Healthcare Information System
HL7	Health Level 7
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IHE	Integrating the Healthcare Enterprise
JAR	Java Archive
JCA	Java Connector Architecture
JEE	Java Platform, Enterprise Edition

1 GNU[1] is Not Unix

JMS	Java Message Service
JMX	Java Management Extensions
LOINC	Logical Observation Identifiers Names and Codes
LRDD	Link-based Resource Descriptor Discovery
MIME	Multipurpose Internet Mail Extensions
NSI	Neighbor System Interface
OASIS	Organization for the Advancement of Structured Information Standards
OSGi	Open Services Gateway Initiative
PKI	Public Key Infrastructure
POJO	Plain Old Java Object
POM	Project Object Model
REST	Representational State Transfer
RHIN	Regional Healthcare Information Networks
RFC	Request For Comments
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
SNOMED	Systematized Nomenclature of Medicine
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UI	User Interface
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name

WWW	World Wide Web
XDI	XRI Data Interchange
XDS	Cross Enterprise Document Sharing
XML	Extended Markup Language
XMPP	Extensible Messaging and Presence Protocol
XRDS	Extensible Resource Descriptor Sequence
XRD	Extensible Resource Descriptor
XRI	Extensible Resource Identifier

A An Overview of Maven

An XML representation of the POM, with which a Maven project is configured, can be found in the file pom.xml in the root folder of a Maven project. There, the *Maven project coordinates* are defined, including the artifact ID, the group ID, the version, and the packaging type. The group ID of a project reflects the organization working on it. Maven uses a dot notation to denote a hierarchical path in the group ID. In contrast to the group ID, hierarchical dissection in the artifact ID is not provided. The packaging type specifies the type of Maven project and is set to jar for projects being bundled as JAR files.

The version number is structured by Maven adhering to the following scheme: <major version>.<minor version>.<incremental version>-<qualifier>. An exemplary project version is 0.3.1-alpha. Maven furthermore offers the concept of snapshot projects, that manifest a version containing the qualifier SNAPSHOT. This indicates a project version under active development and inhibits deploying this project to public Maven repositories.

Another major feature of Maven is its declarative dependency management. Dependencies of a project are declared using their project coordinates in the POM file. During build time, Maven downloads required artifacts from public Maven repositories on the Internet. Required dependencies may themself declare their own dependencies in their POM files, so that transitive dependencies occur which are managed automatically. For each dependency, a scope may be declared that specifies during which phase the dependency is needed. The scope compile indicates dependencies only needed during compile time, the default scope runtime indicates dependencies also needed during runtime, the scope test marks dependencies as only needed during test time.

By adhering to the convention over configuration principle, Maven proposes a default project directory structure. The root folder of a project contains the POM file. Below the folder src, all source files can be found. This folder is further divided by the subfolders main and test. Maven strictly distinguishes between main project sources and sources only needed during test time. Below each folder, the directories java and resources separate source code from static configuration resources. Another folder called spring was introduced on this level to contain Spring configuration files.

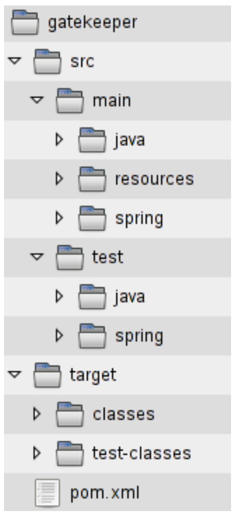


Figure A.1: The default Maven project folder structure for the Gatekeeper subsystem

Below the root folder, the directory target is taken as the output folder of all build artifacts. After Maven packaged the project, it can be found as JAR file in the target folder. All compiled classes contained in the src/main/java folder are included in target/classes. Also, the resources under src/main/resources are copied to target/classes. In parallel all compiled test classes of src/test/java and any test resources are found in target/test-classes. The Maven clean command results in just deleting the target folder. The described folder structure adheres to the Maven convention and is displayed in figure A.1. It can be changed by declaring other paths to the described folders in the POM file.

In comparison to procedural build tools like Ant[1] or GNU[2] Make[3], Maven adheres to a declarative approach. A default build life cycle is provided by Maven, that can be declaratively customized in the POM file. The basic phases include validate, compile, test-compile, test, package, install, and deploy. The phases are passed through in a sequential order and the build is canceled if one of the phases fail. Before compiling the source code and the test classes, the project is validated. After executing any tests, the compiled classes are packaged into a JAR file. The install phase copies the artifacts to a local repository that provides other local Maven projects with the generated artifact. In the last phase, the artifact is deployed to a configured remote repository, so that the artifact is available for other projects and developers.

The Maven core is only aware of the POM file, the life cycle, and a *plug-in mechanism*. Maven plug-ins, offering so called *goals*, are plugged into various phases of the life cycle. The

1 http://ant.apache.org
2 GNU is Not Unix
3 http://www.gnu.org/software/make

life cycle can be customized by changing the configuration of these plug-ins or adding other plug-ins. The plug-ins themself are retrieved from Maven plug-in repositories on the Internet. Additional plug-ins that can be instrumented include *Maven Eclipse Plug-in* for Eclipse project file generation, *Maven Dependency Plug-in* for copying all dependencies into a specified folder, *Maven Antrun Plug-in* for running Ant tasks, and *Maven Bundle Plug-in* for the creation of OSGi manifest files for projects with packaging type bundle.

The latter plug-in therefore scans all compiled Java classes of the generated artifact and extracts package imports not fulfilled by the package itself. These packages are included into the OSGi specific Import-Package: header in the MANIFEST.MF file of the JAR file.

Deploying the bundle into an OSGi container, all package imports must be available by other deployed bundles. If all package imports are fulfilled, the bundle state is 'Resolved'. On subsequently starting it, its services are exported to the service registry and it changes state to 'Active'. By default, the Maven Bundle Plug-in also adds the manifest header Export-Package: and includes each package of the JAR file as export. However, the set of exported packages can be restricted by configuration of the plug-in in the POM file. Furthermore, several OSGi specific manifest headers are written by the Maven Bundle Plug-in. These include the headers Bundle-SymbolicName, Bundle-Name, and Bundle-Version. The symbolic name is obtained by appending the artifact ID to the group ID. The bundle name is an mnemonic name taken from the Maven project property <name />. The OSGi version of the bundle reflects the Maven project version.

Maven is controlled by a command line interface and tries to locate a POM file in the folder it is called. Either a life cycle phase or a Maven plug-in goal can be passed as an argument. An example call to request Maven to generate Eclipse project files out of the POM file is 'mvn eclipse:m2eclipse'. While eclipse is the name of the plug-in, m2eclipse is the name of a goal of this plug-in. The execution of any plug-in goal can be configured through the POM file. An exemplary call that triggers the execution of all life cycle phases including package is 'mvn package'.

Another feature of Maven is the support for *multi-module projects*. Declaring a project as multi-module is done by setting the packaging type to pom instead of jar. Furthermore a list of submodules must be specified, where the module names correspond to subfolders of the root project folder. On executing any Maven goal, Maven delegates the command to all modules of this project. This is done recursively so that the modules form a tree with normal Maven projects as leafs. This can be done in order to separate subprojects, e.g. to distinguish the core logic in the module core from the web UI[1] in the module web.

1 User Interface

Maven furthermore supports the inheritance of POM files. Configuration of parent POM files is inherited by child POM files, including the group ID and the version. While all configuration elements may optionally be overwritten, the artifact ID must be provided. Multi-module support and inheritance are often used together by modules inheriting their POM file from the POM file of the parent project. In this scenario, common dependencies of all modules can be declared in the root POM file and inherited to child POM. Furthermore, Maven supports the management of dependency version numbers in the root project on behalf of the child project. Dependencies managed in this way are declared in the root POM and activated by submodules when needed. This guarantees consistent dependency versions throughout the whole project. Using a combination of module-submodule and inheritance relationships between projects results in arbitrary complex project structures. An example of a possible structure is outlined in figure A.2.

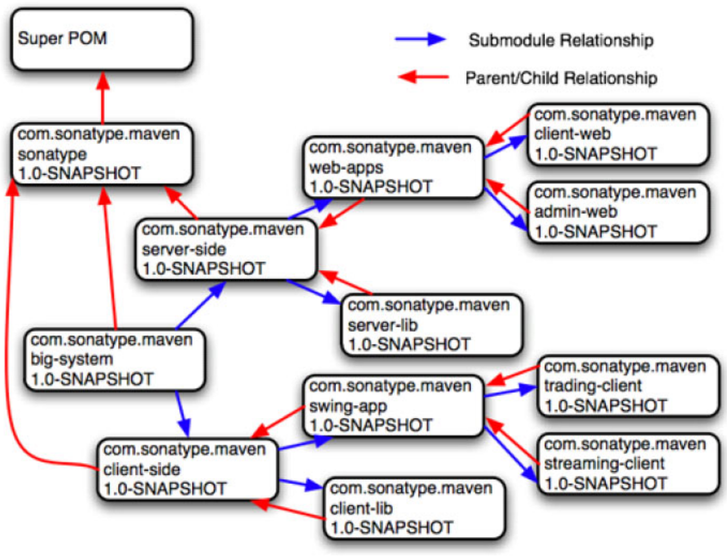
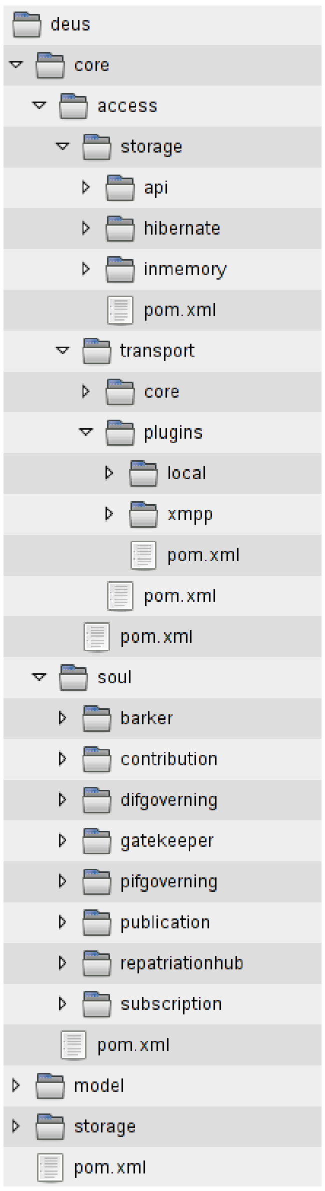

Super POM
Submodule Relationship
Parent/Child Relationship
com.sonatype.maven sonatype 1.0-SNAPSHOT
com.sonatype.maven web-apps 1.0-SNAPSHOT
com.sonatype.maven client-web 1.0-SNAPSHOT
com.sonatype.maven admin-web 1.0-SNAPSHOT
com.sonatype.maven server-side 1.0-SNAPSHOT
com.sonatype.maven server-lib 1.0-SNAPSHOT
com.sonatype.maven big-system 1.0-SNAPSHOT
com.sonatype.maven trading-client 1.0-SNAPSHOT
com.sonatype.maven swing-app 1.0-SNAPSHOT
com.sonatype.maven client-side 1.0-SNAPSHOT
com.sonatype.maven streaming-client 1.0-SNAPSHOT
com.sonatype.maven client-lib 1.0-SNAPSHOT

Figure A.2: An exemplary Maven project structure using module-submodule and inheritance relationships

B DEUS Maven Project

B.1 The Folder Structure of the DEUS Maven Project



```
deus
  core
    access
      storage
        api
        hibernate
        inmemory
        pom.xml
      transport
        core
        plugins
          local
          xmpp
          pom.xml
        pom.xml
      pom.xml
    soul
      barker
      contribution
      difgoverning
      gatekeeper
      pifgoverning
      publication
      repatriationhub
      subscription
      pom.xml
  model
  storage
  pom.xml
```

Figure B.1: The DEUS project directory structure

B.2 The DEUS Root POM File

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
   org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/
   POM/4.0.0␣http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>inf6.promed</groupId>
    <artifactId>prototype-pom</artifactId>
    <version>0.3-SNAPSHOT</version>
    <packaging>pom</packaging>
    <name>DEUS root pom</name>


    <modules>
        <module>deus</module>
        <module>dacus</module>
    </modules>

    <properties>
        <spring.version>2.5.6.A</spring.version>
        <aspectj.version>1.6.1</aspectj.version>
        <spring.agent.version>2.5.6</spring.agent.version>
        <spring.osgi.version>1.1.2.B</spring.osgi.version>
        <application.traceLevels>*=warn,deus.*=verbose,dacus.*=verbose</
           application.traceLevels>
    </properties>


    <developers>
        <developer>
            <id>cpn</id>
            <name>Christoph Neumann</name>
            <email>christoph.neumann@informatik.uni-erlangen.de</email>
            <url>http://www6.informatik.uni-erlangen.de/people/cpn/</url>
            <organization>Uni Erlangen - Informatik 6</organization>
            <organizationUrl>http://www6.informatik.uni-erlangen.de/</
               organizationUrl>
        </developer>
        <developer>
            <id>siflramp</id>
            <name>Florian Rampp</name>
            <email>Florian.Rampp@informatik.stud.uni-erlangen.de</email>
            <organization>Uni Erlangen - Informatik 6</organization>
            <organizationUrl>http://www6.informatik.uni-erlangen.de/</
                organizationUrl>
        </developer>
    </developers>

    <scm>
        <connection>scm:svn:https://svn.origo.ethz.ch/dacus/tags/dacus-0.2</
            connection>
        <developerConnection>scm:svn:https://svn.origo.ethz.ch/dacus/tags/
            dacus-0.2</developerConnection>
        <tag>HEAD</tag>
    </scm>

    <issueManagement>
        <system>Redmine</system>
        <url>faui6p15.informatik.uni-erlangen.de:3000</url>
    </issueManagement>

    <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
    <!--+++ DEPENDENCIES ++++++++++++++++++++++++++++++++++++++++++++++++++-->
    <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->

    <dependencyManagement>

        <dependencies>

            <!--+++ INTRA-PROJECT DEPENDENCIES ++++++++++++++++++++++++++-->

            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>deus-core-access-storage-api</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>deus-core-access-storage-inmemory</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
```

```
                <groupId>inf6.promed</groupId>
                <artifactId>deus-core-access-storage-hibernate</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>deus-core-access-transfer-core</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>deus-core-access-transfer-plugins-local</
                    artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>deus-core-access-transfer-plugins-xmpp</artifactId
                    >
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>deus-core-puddle</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>deus-gatekeeper</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
```

```
                <groupId>inf6.promed</groupId>
                <artifactId>deus-model</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>



            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>dacus-core-puddle</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <dependency>
                <groupId>inf6.promed</groupId>
                <artifactId>dacus-ui-puddle</artifactId>
                <version>${project.version}</version>
                <type>bundle</type>
            </dependency>

            <!--+++ INTER-PROJECT DEPENDENCIES +++++++++++++++++++++++++-->

            <!--+++ SPRING +++-->
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>org.springframework.context</artifactId>
                <version>${spring.version}</version>
                <scope>provided</scope>
            </dependency>


            <dependency>
                <groupId>org.springframework.osgi</groupId>
                <artifactId>org.springframework.osgi-library</artifactId>
                <version>${spring.osgi.version}</version>
                <type>libd</type>
                <scope>provided</scope>
            </dependency>



            <dependency>
```

```
                <groupId>org.springframework</groupId>
                <artifactId>org.springframework.aspects</artifactId>
                <version>${spring.version}</version>
            </dependency>

            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-agent</artifactId>
                <version>${spring.agent.version}</version>
                <scope>test</scope>
            </dependency>


            <!--+++ ASPECTJ +++-->
            <dependency>
                <groupId>org.aspectj</groupId>
                <artifactId>com.springsource.org.aspectj.weaver</artifactId>
                <version>${aspectj.version}</version>
                <scope>provided</scope>
            </dependency>

        </dependencies>
    </dependencyManagement>



    <dependencies>
        <!--+++ INTRA-PROJECT DEPENDENCIES ++++++++++++++++++++++++++++++-->

        <!--+++ INTER-PROJECT DEPENDENCIES ++++++++++++++++++++++++++++++-->

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.test</artifactId>
            <version>${spring.version}</version>
            <scope>test</scope>
        </dependency>

        <!--+++ JUNIT +++-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.4</version>
            <scope>test</scope>
        </dependency>


        <!--+++ LOGGING +++-->
        <!-- Logback ist used as logging backend! -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>com.springsource.ch.qos.logback.classic</artifactId>
            <version>0.9.9</version>
            <scope>test</scope>
        </dependency>

        <!-- We use the slf4j api, against which logback is programmed -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>com.springsource.slf4j.api</artifactId>
            <version>1.5.0</version>
        </dependency>

    </dependencies>



    <!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
    <!--+++ BUILD +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
    <!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->

    <build>

        <!--+++ PLUGINS +++++++++++++++++++++++++++++++++++++++++++++++++-->

        <plugins>

            <!-- MAVEN PROJECT FILE GENERATION -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-eclipse-plugin</artifactId>
                <configuration>
                    <wtpversion>2.0</wtpversion>
                    <additionalProjectFacets>
                        <com.springsource.server.bundle>1.0</com.springsource.
                            server.bundle>
                    </additionalProjectFacets>
                    <additionalBuildcommands>
```

```
                        <buildcommand>org.eclipse.wst.common.project.facet.core.builder</buildcommand>
                        <!--
                            since mvn eclipse:m2eclipse which should add m2eclipse support to eclipse project ignores all other natures and
                            builders, we just use mvn eclipse:eclipse and add m2eclipse nature and builder manually here
                        -->
                        <buildcommand>org.maven.ide.eclipse.maven2Builder</buildcommand>
                        <buildcommand>org.springframework.ide.eclipse.core.springbuilder</buildcommand>
                    </additionalBuildcommands>
                    <additionalProjectnatures>
                        <projectnature>org.maven.ide.eclipse.maven2Nature</projectnature>
                        <projectnature>org.springframework.ide.eclipse.core.springnature</projectnature>
                        <projectnature>org.eclipse.wst.common.project.facet.core.nature</projectnature>
                        <projectnature>com.springsource.server.ide.facet.core.bundlenature</projectnature>
                    </additionalProjectnatures>
                    <!-- added this, so that automatically, the spring dm server is the default runtime! -->
                    <additionalConfig>
                        <file>
                            <name>.settings/org.eclipse.wst.common.project.facet.core.xml</name>
                            <content>
                                <![CDATA[
<?xml version="1.0" encoding="UTF-8"?>
<faceted-project>
    <runtime name="SpringSource dm Server (Runtime) v1.0" />
    <installed facet="com.springsource.server.bundle" version="1.0" />
</faceted-project>
                                ]]>
                            </content>
                        </file>
                    </additionalConfig>
                    <classpathContainers>
                        <classpathContainer>org.eclipse.jdt.launching.JRE_CONTAINER</classpathContainer>
                        <classpathContainer>com.springsource.server.ide.jdt.core.MANIFEST_CLASSPATH_CONTAINER</classpathContainer>
                    </classpathContainers>
                </configuration>
            </plugin>


            <!-- BUNDLE MANIFEST GENERATION -->
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>1.4.3</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <!--
                            export all packages below the root package (except non exported packages), and do not reimport them! more on
                            reimporting exported packages here: http://www.osgi.org/blog/2007/04/importance-of-exporting-nd-importing.html
                            If this behaviour is desired, add ;-noimport:=true at the end!
                        -->
                        <Export-Package>${osgi.nonExportedPackages},${osgi.rootPackage}.*</Export-Package>

                        <!--
                            include all other packages (also the non exported packages!) as private. Export-Package takes precedence over
                            Private-Package!
                        -->
                        <Private-Package>${osgi.rootPackage}.*;-split-package:=merge-first</Private-Package>

                        <!-- remove the BND internal header, that just states, which packages are ignored by BND -->
                        <_removeheaders>Ignore-Package</_removeheaders>

                        <!-- add additional package imports -->
                        <Import-Package>${osgi.additionalPackageImports},*</Import-Package>

                        <!-- Spring DM server configuration for the level of trace output -->
                        <Application-TraceLevels>${application.traceLevels}</Application-TraceLevels>
                    </instructions>
                </configuration>
            </plugin>


            <!-- DEPENDENCY COPYING -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <configuration>
                    <!--    <outputDirectory>${springdmserver.location}/repository/bundles/usr</outputDirectory>-->
                    <outputDirectory>${project.build.directory}/dependencies</outputDirectory>
                    <overWriteReleases>false</overWriteReleases>
                    <overWriteSnapshots>false</overWriteSnapshots>
                    <overWriteIfNewer>true</overWriteIfNewer>
                    <includeScope>runtime</includeScope>
                    <!--    <includeTypes>jar</includeTypes>-->
                </configuration>
            </plugin>

            <!-- COMPILING -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
                    <debug>true</debug>
                </configuration>
            </plugin>


            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
```

```
                    <configuration>
                        <includes>
                            <include>**/*Test.java</include>
                            <include>**/*TestCase.java</include>
                        </includes>
                    </configuration>
                </plugin>

                <!-- COPY ARTIFACT TO APP SERVER ON PHASE pre-integration-test -->
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-antrun-plugin</artifactId>
                    <executions>
                        <execution>
                            <phase>pre-integration-test</phase>
                            <configuration>

                                <tasks>
                                    <taskdef resource="net/sf/antcontrib/antcontrib.properties" />

                                    <!-- copy the artifact only, if it is of 'bundle' packaging -->
                                    <if>
                                        <equals arg1="${project.packaging}" arg2="bundle" />
                                        <then>
                                            <copy file="${project.build.directory}/${project.build.finalName}.jar" todir="${springdmserver.location}/pickup" />
                                        </then>
                                        <else>
                                            <echo message="this␣Maven␣project␣is␣not␣deployed␣to␣app␣server,␣since␣it␣is␣not␣of␣packaging␣'bundle'␣but␣'${project.packaging}'" />
                                        </else>
                                    </if>
                                </tasks>
                            </configuration>
                            <goals>
                                <goal>run</goal>
                            </goals>
                        </execution>
                    </executions>
                    <dependencies>
                        <dependency>
                            <groupId>ant-contrib</groupId>
                            <artifactId>ant-contrib</artifactId>
                            <version>20020829</version>
                        </dependency>
                    </dependencies>
                </plugin>

            </plugins>

            <!--+++ RESOURCES +++++++++++++++++++++++++++++++++++++++++++++++-->

            <resources>
                <resource>
                    <directory>${basedir}/src/main/resources</directory>
                </resource>
                <resource>
                    <targetPath>META-INF/spring</targetPath>
                    <filtering>false</filtering>
                    <directory>${basedir}/src/main/spring</directory>
                    <includes>
                        <include>**/*.xml</include>
                    </includes>
                </resource>
            </resources>
            <testResources>
                <!--<testResource>
                    <directory>${basedir}/src/test/resources</directory>
                </testResource>-->
                <testResource>
                    <filtering>false</filtering>
                    <directory>${basedir}/src/main/spring</directory>
                    <includes>
                        <include>**/*.xml</include>
                    </includes>
                </testResource>
                <testResource>
                    <filtering>false</filtering>
                    <directory>${basedir}/src/test/spring</directory>
                    <includes>
                        <include>**/*.xml</include>
                    </includes>
                </testResource>
            </testResources>

        </build>

        <!--+++ REPOSITORIES +++++++++++++++++++++++++++++++++++++++++++++++-->

        <repositories>
            <repository>
                <id>com.springsource.repository.bundles.release</id>
                <name>SpringSource Enterprise Bundle Repository - SpringSource
                    Bundle Releases</name>
                <url>http://repository.springsource.com/maven/bundles/release</url
                    >
            </repository>
            <repository>
                <id>com.springsource.repository.bundles.external</id>
                <name>SpringSource Enterprise Bundle Repository - External Bundle
                    Releases</name>
                <url>http://repository.springsource.com/maven/bundles/external</
                    url>
            </repository>
            <repository>
                <id>com.springsource.repository.libraries.release</id>
                <name>SpringSource Enterprise Bundle Repository - SpringSource
                    Library Releases</name>
                <url>http://repository.springsource.com/maven/libraries/release</
                    url>
            </repository>
            <repository>
                <id>com.springsource.repository.libraries.external</id>
                <name>SpringSource Enterprise Bundle Repository - External Library
                     Releases</name>
                <url>http://repository.springsource.com/maven/libraries/external</
                    url>
            </repository>
        </repositories>

    </project>
```

B.3 The POM File of the Maven Gatekeeper Submodule

```xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>inf6.promed</groupId>
        <artifactId>deus</artifactId>
        <version>0.3-SNAPSHOT</version>
    </parent>

    <artifactId>deus-gatekeeper</artifactId>
    <packaging>bundle</packaging>
    <name>DEUS gatekeeper</name>

    <!-- don not remove properties, even if they are empty!!! -->
    <properties>
        <osgi.rootPackage>deus.gatekeeper</osgi.rootPackage>
        <!--
            org.springframework.orm.jpa.support: this is needed for <context:component-scan />. This BeanPostProcessor requires
            PersistantAnnotationBeanPostProcessor which is in the given package.
        -->
        <osgi.additionalPackageImports>org.springframework.orm.jpa.support</osgi.additionalPackageImports>

        <osgi.nonExportedPackages />
    </properties>


<!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<!--+++ DEPENDENCIES ++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->

    <dependencies>
        <!--+++ INTRA-PROJECT DEPENDENCIES ++++++++++++++++++++++++++++++-->
        <dependency>
            <groupId>inf6.promed</groupId>
            <artifactId>deus-model</artifactId>
            <type>bundle</type>
        </dependency>

        <dependency>
            <groupId>inf6.promed</groupId>
            <artifactId>deus-core-access-storage-api</artifactId>
            <type>bundle</type>
        </dependency>

        <!--+++ INTER-PROJECT DEPENDENCIES +++++++++++++++++++++++++++++-->

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.context</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.osgi</groupId>
            <artifactId>org.springframework.osgi-library</artifactId>
            <type>libd</type>
        </dependency>

    </dependencies>


<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<!--+++ BUILD +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->

    <build>

        <plugins>

        </plugins>

    </build>


</project>
```