JOHANNES KEPLER
UNIVERSITY LINZ

Author
Brigitte Andorfer-Plainer, BSc

Submission
Department of Business Informatics – Data & Knowledge Engineering

Thesis Supervisor
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Assistant Thesis Supervisor
Mag. Dr. Bernd Neumayr

April 2021

SemNOTAM Container Management with a Task-Based Retrieval Service

Master's Thesis

to obtain the academic degree of

Master of Science

in the Master's Program

Business Informatics

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
jku.at

SWORN DECLARATION

Linz, April 21, 2021

Signature

ABSTRACT

Ensuring smooth and safe air traffic requires a lot of information and calls for efficient and effective air traffic management (ATM). ATM consists of air traffic control (ATC), air traffic flow management (ATFM), and aeronautical information services (AIS). AIS provide relevant information to the users of the airspace, especially pilots, in the form of information messages, so-called Notices to Airmen (NOTAMs). Currently, the communication between stakeholders takes place via point-to-point connections yielding a tight coupling. Hence, information is not shared within the whole community and needs to be provided for each stakeholder separately. To overcome these drawbacks, System Wide Information Management (SWIM) reorganizes information management and communication in ATM.

The project “Achieving the BEnefits of SWIM by making smart use of Semantic Technologies” (BEST) contributes to the SWIM approach by introducing data containers that group a set of NOTAMs by different characteristics. The segmentation of the whole NOTAM set is needed to reduce the set of NOTAMs that is used for querying or filtering for relevant NOTAMs. The created data containers are provided by data producers in the SWIM environment and can be used by several stakeholders, i.e., the data consumers. To ensure that data producers and data consumers have a common understanding of the contents of the data containers, a common vocabulary is necessary. To enable the task-based retrieval of relevant NOTAMs, this thesis leverages the briefing application of the project SemNOTAM, which allows a fine-grained filtering and querying of NOTAMs by using a combination of a rule- and ontology-based approach.

This thesis covers the conceptualization, the design, and the implementation of the three main components for a SWIM environment. This includes an ontology for managing a vocabulary to describe data containers, a web application for managing these data containers, and services that serve this application and enable task-based retrieval of NOTAMs with a briefing application, e.g., SemNOTAM. Therefore, a requirements analysis is conducted, which defines the functionality, inputs and outputs, as well as the business logic for the implementation. Based on the resulting requirements, an approach for developing the three components is developed. An ontology for the common vocabulary is developed and prototypes for the Container Management Application as well as the services for task-based retrieval of relevant NOTAMs are implemented. The resulting management application is demonstrated with exemplary data containers concerning Europe. Finally, the functionality of the task-based retrieval service is exemplified based on a use case flight from Vienna to Frankfurt.

ZUSAMMENFASSUNG

Für einen reibungslosen und sicheren Ablauf des Flugverkehrs werden viele Informationen benötigt und ein effizientes und effektives Flugverkehrsmanagement wird vorausgesetzt. Flugverkehrsmanagement umfasst die Flugverkehrskontrolle, das Flugverkehrsflussmanagement und die Luftfahrtinformationsdienste. Die Luftfahrtinformationsdienste stellen den Nutzern des Luftraums, insbesondere den Piloten, relevante Informationen in Form von Nachrichten, sogenannten Notices to Airmen (NOTAMs), zur Verfügung. In der Kommunikation aller Beteiligten gibt es aktuell viele Punkt-zu-Punkt-Verbindungen, was eine enge Kopplung zur Folge hat. Daher werden Informationen nicht innerhalb der gesamten Community geteilt, sondern müssen für jeden Stakeholder separat bereitgestellt werden. Um diese Nachteile zu überwinden, wird im Rahmen von System Wide Information Management (SWIM) die Kommunikation und das Informationsmanagement im Flugverkehrsmanagement reorganisiert.

Das Projekt „Achieving the BEnefits of SWIM by making smart use of Semantic Technologies" (BEST) trägt zum SWIM-Ansatz bei, indem es Datencontainer einführt, die eine Menge von NOTAMs nach verschiedenen Merkmalen gruppieren. Die Segmentierung der gesamten NOTAM-Menge wird benötigt, um die Anzahl der NOTAMs zu reduzieren, die für die Abfrage oder das Filtern von relevanten NOTAMs verwendet wird. Die erstellten Datencontainer werden von Datenproduzenten in der SWIM-Umgebung zur Verfügung gestellt und können von allen Stakeholdern, den Datenkonsumenten, genutzt werden. Um sicherzustellen, dass Datenproduzenten und Datenkonsumenten ein gemeinsames Verständnis über die Inhalte der Datencontainer haben, ist ein gemeinsames Vokabular notwendig. Um die aufgabenbasierte Abfrage von relevanten NOTAMs zu ermöglichen, nutzt diese Arbeit die Briefing-Applikation des Projekts SemNOTAM, welche eine feingranulare Filterung und Abfrage von NOTAMs durch eine Kombination aus einem regelbasierten und einem ontologiebasierten Ansatz ermöglicht.

Diese Arbeit umfasst die Konzeptionierung, das Design und die Implementierung der drei Hauptkomponenten für eine SWIM-Umgebung. Dazu gehören eine Ontologie zur Verwaltung eines Vokabulars zur Beschreibung von Datencontainern, eine Webanwendung zur Verwaltung dieser Datencontainer, sowie Services, die diese Anwendung bedienen und eine aufgabenbasierte Abfrage von NOTAMs mit einer Briefing Applikation, z.B. SemNOTAM, ermöglichen. Dazu wird eine Anforderungsanalyse durchgeführt, die die Funktionalität, Ein- und Ausgaben sowie die Geschäftslogik für die Implementierung definiert. Basierend auf den daraus resultierenden Anforderungen wird ein Konzept zur Umsetzung der drei Komponenten entwickelt. Es wird eine Ontologie für das gemeinsame Vokabular entwickelt und es werden Prototypen für die Container Management Applikation und die Services zum aufgabenbasierten Abruf relevanter NOTAMs implementiert. Die resultierende Management Applikation wird mit exemplarischen Datencontainern zu Europa demonstriert. Abschließend wird die Funktionalität des aufgabenbasierten Abfrage-Services anhand eines Anwendungsfalls, eines Fluges von Wien nach Frankfurt, exemplarisch dargestellt.

Table of Contents

1 Introduction

This section introduces the topics Air Traffic Management (ATM), System Wide Information Management (SWIM), and the idea to bring these topics together. Furthermore, it describes the scope of this thesis, its role, and its structure.

1.1 Preface

In February 2017 the European Organization for the Safety of Air Navigation (EUROCONTROL) stated that the total air traffic in 2016 in Europe reached 10.2 million flights and is expecting a growth of 2.9 % in 2017 [1]. In September 2017 a new forecast was published which shows that in 2017 the total air traffic grew by 4.5 %. With the expectation of 2.8 % growth in 2018 and a stable annual growth of 1.7 % from 2019 to 2023, the total number of flight movements in 2023 will be 12.0 million [2]. This is a growth rate of 17 % from 2016 to 2023. This amount of air traffic requires an efficient and effective ATM, which consists of three main activities [3]:

1. **Air Traffic Control (ATC)** denotes the process of safely separating aircraft at airports and as they operate a flight. At airports, the aircraft are guided by the tower, whereas during the flight phase ATC centers guide them.
2. **Air Traffic Flow Management (ATFM)** is conducted prior to flights. For each flight, a flight plan exists which is analyzed in a central repository. Air traffic controllers have to handle a large number of flights. To enable this, the ATFM computes the positions of each aircraft at any moment.
3. **Aeronautical Information Services (AIS)** compile and distribute needed aeronautical information to airspace users. The provided information covers legal matters, technical information, or updates on the navigation.

The messages that contain the information provided by the AIS are called Notices to Airmen (NOTAM) and are mainly free text and hence loosely structured [4]. Geographical and temporal information in text-based NOTAMs can lead to inaccuracies as, for example, only the position and radius of a NOTAM can be processed automatically but not the concrete area. In addition, the free text information needs human interpretation to filter the relevant NOTAMs for a specific flight [4]. These drawbacks make clear that traditional NOTAMs cannot satisfy the increasingly automated Aeronautical Information Management (AIM) systems [5]. Therefore, Digital NOTAMs (DNOTAMs) were introduced to overcome these drawbacks. Free text information of traditional NOTAMs is represented as semi-structured information in DNOTAMs. The shift to DNOTAMs also enables the provision of the information by digital services that can be processed by systems and automated equipment [5].

The information exchange between the stakeholders of the ATM process, and therefore, the encoding for DNOTAMs is based on the Aeronautical Information Exchange Model (AIXM) [4]. This model was developed and extended by the EUROCONTROL and the Federal Aviation Administration (FAA), who created a new version (version 5.1). The semi-structured data allows machine-interpretation and -processing and thus reduces the number of irrelevant DNOTAMs, which reduces the risk of information overload in pilot briefings [6]. Furthermore, AIXM provides a data encoding specification for geographical information, that is based on the Geography Markup Language (GML) [5].

Providing and accessing relevant DNOTAMs is important for various ATM stakeholders, such as airport staff and pilots. The communication between these stakeholders is traditionally implemented through own proprietary applications [7]. The left side of Figure 1.1 shows that this approach results in numerous point-to-point connections, which leads to a tight coupling of these applications. In contrast, the vision of SWIM, depicted on the right side of Figure 1.1, follows the idea of a service-oriented architecture [8]. For example, traditionally, airports develop their own interfaces to communicate with aircraft, while the ATFM center would need a similar interface but cannot use it, as it is too specific for the airport's application. When the interfaces for communicating with different stakeholders are defined in a SWIM environment, each stakeholder can communicate over this central interface.

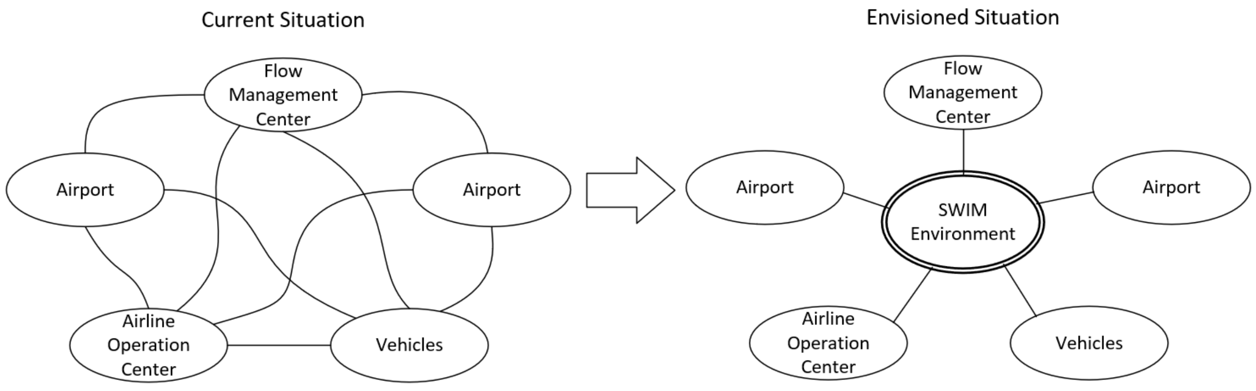


Figure 1.1: System Wide Information Management (SWIM) Vision [8]

The goal of SWIM visionaries is to completely reorganize the whole ATM communication within a SWIM environment [8]. The provision and consumption of information shall be completely decoupled. The result will be that producers, those who provide the data, and consumers, those who want to access and use the data, will no longer need to know each other. Furthermore, consumers can benefit from the information of several producers, as they are not limited to one information producer due to a single point-to-point communication [9]. As information is shared in a SWIM environment a common understanding is needed – otherwise collaborative decision making cannot be enabled. By decoupling of information producers and consumers through standardized interfaces and the provision of common semantics in a SWIM environment, a marketplace for the exchange of aeronautical information is enabled.

Concerning the ATM process there are two relevant issues. On the one hand, the large number of DNOTAMs needs to be filtered in an effective and efficient way to minimize the need of human interpretation and optimize the precision of the filter results. On the other hand, a SWIM environment is needed to decouple the stakeholders and establish a marketplace for aeronautical information. The filtering aspect is covered by the SemNOTAM project, which aims to improve temporal and spatial filtering and enable filtering based on the semantics. The BEST project focuses on a data-centric perspective, which reflects the SWIM vision, and aims to build a prototype for the mentioned marketplace for aeronautical information.

The project SemNOTAM [10], which tries to achieve a precise filtering of DNOTAMs, was launched due to the limited spatial and temporal filtering capabilities of existing services, like the Federal NOTAM Service and NOTAM Distribution Service (FNS-NDS) [11]. On the one hand, the SemNOTAM project tries to improve the temporal and spatial filtering by conducting a more fine-grained approach, e.g. filtering on the exact shapes or the actual times the DNOTAMs are active. On the other hand, the aim of SemNOTAM is to enable intelligent filtering based on concepts with predefined semantics [12]. To enable this, an ontology-based approach for the explicit specification of the used concepts combined with business rules for filtering and annotating the DNOTAMs is used. The DNOTAMs are categorized by concepts using business rules, e.g., a DNOTAM concerning a runway facility and a closure event is categorized using the concept *RunwayClosure*. Other business rules specify which DNOTAMs are relevant for a specific flight and how to determine the importance of these DNOTAMs [13]. As part of the SemNOTAM project a web service, the SemNOTAM Web Service, is developed, which provides the functionality to filter a DNOTAM set with so-called interest specifications or task descriptions. An interest specification consists of the information the user is interested in, respectively his/her specific task, and is represented using eXtensible Markup Language (XML) [14]. The SemNOTAM Web Service expects such a specification of the user's task and a DNOTAM set as input and performs the filtering and annotation based on the concepts and business rules. As an output this service returns a filtered DNOTAM set with the annotated importance for each DNOTAM [12]. Furthermore, to address the pilots' needs, to easily describe their task, i.e. define their flight plan, a web application, the SemNOTAM Briefing Application, is developed. This application allows the users to create their flight plan in a simple user interface and receive the filtered and annotated DNOTAM set from the SemNOTAM Web Service.

The main drawback of the SemNOTAM filtering approach is that a large number of DNOTAMs, for example, all DNOTAMs of Europe, are used as input for the filtering. This overload of DNOTAM data causes high costs while filtering because many irrelevant DNOTAMs have to be processed. The idea to improve the filtering process is to use less DNOTAMs as input for the filtering. Therefore, the available DNOTAMs shall be split in segments that can be organized as subsets of each other, i.e., in a subsumption hierarchy. For filtering the DNOTAMs with respect to a specific task description, only the most specific superset necessary for the specified task shall be found and shall be used as the input for the filtering process.

The segmentation of the available DNOTAMs is defined in more detail in the project BEST [15], which was started in 2016. In this project the term semantic container is introduced for one such DNOTAM set. Gringinger et al. [16] use the term data container as a synonym for semantic containers while Kovacic et al. [9] only use the term data container. Therefore, the concept of these containers will be referred to as data containers in this thesis. A data container consists of its description, including temporal, spatial, and semantic description, and the corresponding data set [9]. For example, a data container can contain all DNOTAMs of the year 2017 in Europe that are relevant for aircrafts with a wingspan of more than 120 feet. The project BEST focuses on a data-centric perspective to distribute data containers over SWIM [15]. The goal of this project is to find out which semantic technologies can be used in the context of SWIM. The data containers need to be described through an ontology-based specification, to allow to derive subsumption hierarchies between them [17]. The subsumption hierarchies define which DNOTAM sets are subsets of other DNOTAM sets, thus only a specific subset can be used to minimize the costs for filtering. Furthermore, a prototype for demonstrating the technical feasibility of data containers is developed in the BEST project and the integration within a SWIM lifecycle is shown [18].

With the SemNOTAM and the BEST project both sides of actors in the SWIM environment can be facilitated. The producers can define data containers by describing the content to be published, while consumers only specify the information necessary for a task (see Figure 1.2). As described before, it is important that consumers and producers have the same understanding of the shared information. The use of the ontological descriptions allows to derive a subsumption hierarchy between the information need and the provided data containers. In the end, the created subsumption hierarchy enables the discovering of the most specific superset for a defined information need, i.e., the consumer's task [17].

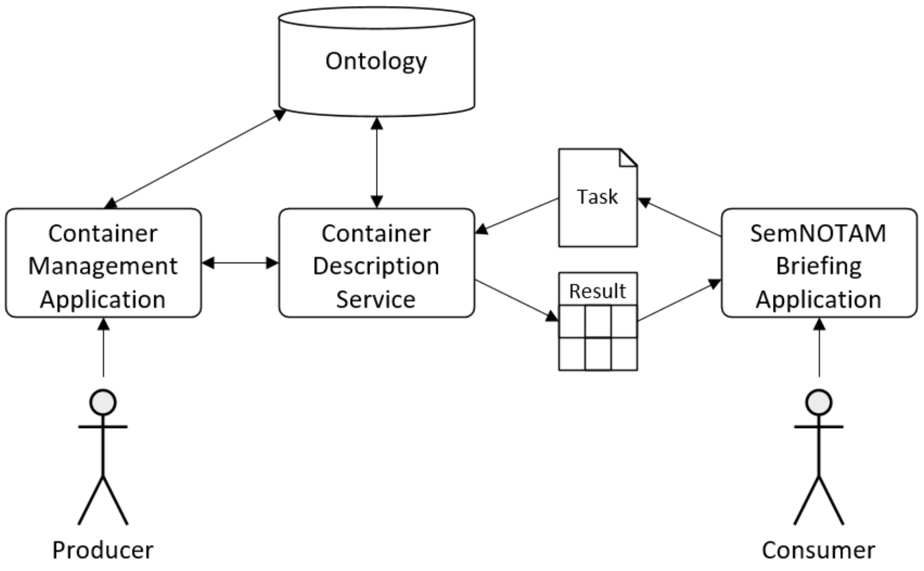

Ontology
Container Management Application
Container Description Service
Task
Result
SemNOTAM Briefing Application
Producer
Consumer

Figure 1.2: Producer vs. consumer

The producers use a management application to describe the information they want to provide, whereas the consumers describe their task in the SemNOTAM Briefing Application. As described before, both actors need to have a common understanding and therefore, use the concepts defined in the ontology for their description. As depicted in Figure 1.2, the management application for producers accesses the ontology for providing the vocabulary and it uses the Container Description Service to derive information about the existing data containers. The SemNOTAM Briefing Application accesses the vocabulary of the ontology via the Container Description Service to ensure the consumer uses the same vocabulary as the producer. The resulting task description is then forwarded to a task-based retrieval service in the SWIM environment and returns the result containing the relevant information for the specific task, i.e. the relevant DNOTAMs with the corresponding annotations. The task-based retrieval service allows pilots to find relevant DNOTAMs according to their information need, respectively their task.

1.2 Problem statement

One of the main problems for the SWIM environment is that consumers and producers must have the same understanding for describing data containers. This means that for each term, its meaning and interpretation has to be specified. Therefore, a common vocabulary has to be defined to enable the communication between producers and consumers. Such a vocabulary has to be extensible to cover new terms when needed. The definition of a vocabulary can be realized with an ontology. Studer et al. [19] define an ontology as *"a formal, explicit specification of a shared conceptualization."* Another definition of ontologies from Gruber also addresses the exchange of information between two stakeholders:

> *"Pragmatically, a common ontology defines the vocabulary with which queries and assertions are exchanged among agents. Ontological commitments are agreements to use the shared vocabulary in a coherent and consistent manner." [20]*

After ensuring the common understanding, an exchange process has to be defined. The producers shall be able to provide and manage data containers. A platform is needed where the producers can maintain their data containers by regularly refreshing the data items. The consumers, e.g., pilots, require access to these data containers and need to be able to find a concrete data container according to their specified task.

In this thesis this problem is tackled by designing and developing several systems – each of them specific to a problem to be solved. Producers and consumers need to have the same understanding (vocabulary), the producers must have the ability to manage their data containers and a service for describing the data containers using the vocabulary is needed.

- A Container Ontology is developed allowing for the management of the vocabulary in a repository. Each repository includes an ontology that contains the descriptions of the data containers and the referenced files containing the data items and task descriptions, i.e. interest specifications, of those data containers. The interest specification is an XML representation of the information covered by this container that can be used by services, for example, the SemNOTAM Web Service, to filter DNOTAMs accordingly.
- For the producers, a Container Management Application has to facilitate different tasks concerning the management of their data containers. This management system needs to be able to show the existing data containers and to enable the producer to delete, modify, and refresh their data containers.
- A Container Description Service shall enable producers to describe data containers, and consumers to describe their information need with a shared common vocabulary.

Figure 1.3 shows the overall system view, which includes the three parts described before (Container Ontology, Container Management Application, and Container Description Service) on the top. The bottom shows the existing applications (SemNOTAM Web Service and SemNOTAM Briefing Application), that shall be integrated. Furthermore, the relationships between the systems are depicted. The Container Management Application and the Container Description Service access the ontology in the repository directly, whereas the existing applications only communicate over the Container Description Service. The Container Management Application additionally uses the Container Description Service for creating new data containers. The Container Management Application, as well as the SemNOTAM Briefing Application, use the SemNOTAM Web Service to filter DNOTAMs.

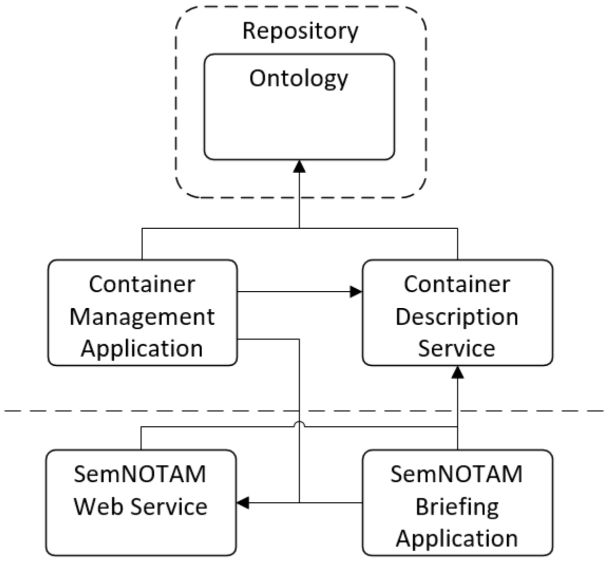

Repository
Ontology
Container Management Application
Container Description Service
SemNOTAM Web Service
SemNOTAM Briefing Application

Figure 1.3: Overview of applications

The SemNOTAM Briefing Application environment shall benefit from this prototype as the filtering process must not be applied to all available DNOTAMs anymore. The SemNOTAM Web Service can use the Container Description Service to find an adequate data container based on the described task, respectively the interest specification, which is used as input for its filtering process. The task description is based on the concepts defined in the ontology. Hence, it is compatible with other applications that use the same ontology.

This thesis covers the requirements analysis, approach, implementation, and usage of a prototype for the management of the data containers and the embedding of it into an existing briefing application environment, i.e., SemNOTAM. The aim of this thesis is to provide a SWIM environment enabling producers to manage the data containers as well as enabling consumers to determine relevant DNOTAMs based on their information need, i.e., task description. The Container Ontology and the basic functionality of the Container Management Application shall be shortly demonstrated using simple concepts and small data containers concerning Europe, and especially Austria.

1.3 Outline

The thesis is structured as follows. Section 2 gives an overview of the fundamentals. First, a theoretical background (Section 2.1) about the key terms NOTAMs and SWIM, the information structure, and the main concepts of knowledge-based systems is given. Afterwards, the basic concepts of the projects SemNOTAM (Section 2.2.1) and BEST (Section 2.2.2) are described.

Section 3 starts with the description of the existing applications, and their relationship to the Container Management Application (Section 3.1), followed by an overview of the given task (Section 3.2). This is divided into three main parts, the Container Description Service (Section 3.2.1), the Container Ontology (Section 3.2.2), and the Container Management Application (Section 3.2.3). Based on this task description, requirements are elicited and categorized (Section 3.3). At the end, Section 3.4 provides a description of challenges and problems that might occur during the implementation and possible solutions to handle them.

Section 4 describes the design process of a client-server application. The approach for the different parts of the application is described in Section 4.1. Furthermore, the architecture of the Container Management Application, which represents the client application is described in Section 4.2.2, followed by the architecture of the Container Description Service, representing the server application, in Section 4.2.3.

The implementation is detailed in Section 5. Therefore, the used technologies for the ontology, the management application, and the services are described (Section 5.1). First, the implementation of the ontology is presented (Section 5.2) followed by the implementation of the Container Management Application (Section 5.3). The last section describes the Container Description Service, which is used by the Container Management Application as well as the SemNOTAM Briefing Application and the SemNOTAM Web Service (Section 5.4).

Section 6 covers the different usages of the application. The administration of the ontology is demonstrated (Section 6.1), followed by the description of the use of the Container Management Application (Section 6.2) and a demonstration of the task-based retrieval service by the use case flight from Vienna to Frankfurt (Section 6.3).

Finally, a conclusion and an outlook on future work is given (Section 7).

2 Fundamentals

This section describes the fundamentals for understanding the purpose and goals of this thesis. Therefore, a theoretical background about important key terms, the information structure and main concepts of knowledge-based systems is given. Furthermore, the main concepts of the SemNOTAM project and its architecture are introduced. Afterwards, the purpose of the project BEST is described to foster the understanding of the concept of data containers.

2.1 Theoretical background

Important key terms and fundamentals are introduced in this section. First, the evolution of NOTAMs from textual representation to a digital, more structured representation is described. Afterwards, the concept of SWIM is explained, which focuses on centralizing the communication in a so-called SWIM environment to avoid point-to-point connections of stakeholders. Next, an overview of knowledge-based systems is given, followed by the concept of ontologies in more detail. The last subsection describes the reasoning process to infer new knowledge and the possibility to derive hierarchies between the concepts, e.g., a subsumption hierarchy of the data containers.

2.1.1 Notices to Airmen

At the moment, NOTAMs are represented by plain text that is provided over basic teletype networks [21]. Those NOTAMs, which are mainly free text and loosely structured, contain information about any changes of aeronautical facilities and are used by pilots, controllers, and other operational personnel [5]. NOTAMs can contain information of different relevance, for example, Listing 2.1 contains information about the closure of the runway 01 for aircraft with a wingspan of more than 120 feet from 30 July 2018 12 p.m. to 31 July 2018 12 p.m. This NOTAM is critical to all flight at that time with an aircraft that exceeds the wingspan and would be represented like this:

```
RWY 01 CLSD TO ACFT WINGSPAN MORE THAN 120FT
1807301200-1807311200
```

Listing 2.1: Textual NOTAM - Runway Closure

Textual NOTAMs cannot be interpreted and filtered automatically, because the information contained in a NOTAM is loosely structured [4]. In addition, geographical and temporal information often requires human interpretation, which can lead to misunderstandings and slows down the information flow.

To overcome the drawbacks of textual NOTAMs, a joint project by EUROCONTROL and FAA was launched in 2009 [6]. The aim was to develop DNOTAMs that can be processed by automated systems, thus minimizing the need of human interpretation. A more structured representation of the information in DNOTAMs simplifies both the automatic processing, such as schedule resolution, and the human readability and therefore, mitigates the risk of human misinterpretation.

EUROCONTROL defines DNOTAMs as follows:

> *"A data set made available through digital services containing information concerning the establishment, condition or change in any aeronautical facility, service, procedure or hazard, the timely knowledge of which is essential to systems and automated equipment used by personnel concerned with flight operations." [5]*

The shift from textual NOTAMs to DNOTAMs does not only concern the conversion of the textual information in a more structured format [4]. Rather, information updates are encoded the same way as long-term information and are provided over the same distribution channels. EUROCONTROL [5] describes DNOTAMs with the following six characteristics:

- **Geo-referenced:** The geographical information included in a DNOTAM can automatically be processed and plotted to a chart for human readability.
- **Temporal:** Temporal information in textual NOTAMs is hard to comprehend by humans. DNOTAMs eliminate the risk of human misunderstandings by allowing to compute and interpret the effective time automatically.
- **Linked to static data:** Temporal information is not stored redundantly anymore, as changes are referenced to the baseline information, which refers to the static information in a NOTAM.
- **Transformable:** A conversion of the information into textual or graphical representations is possible.
- **Query enabled:** Based on the users need complex queries can be executed by computer systems to select the information of interest.
- **Electronically distributable:** The distribution of one to another computer system does not need any manual intervention.

A comprehensive data model for aeronautic information was developed by EUROCONTROL in cooperation with the FAA and the support of the international AIS community to realize the concept of DNOTAMs [5]. AIXM [22] version 5.1.1 is an exchange standard for aeronautical information based on XML that can be used for several current and future aeronautical information applications, as, for example, DNOTAMs [5]. AIXM includes a temporality model [23] and a geographical model based on GML [24] to allow a more precise description of temporal and geographical information.

Due to a missing structure of the different aspects of temporal information in textual NOTAMs, it is hard for humans to comprehend temporal information [25], e.g., the relevant days or time of day. Furthermore, the same temporal information can be described in various possibilities, which can lead to human misunderstandings. Listing 2.2 shows the same textual NOTAM as before with different temporal information.

```
RWY 01 CLSD TO ACFT WINGSPAN MORE THAN 120FT TUE-FRI 0130-1300
1811161200-1901011200
```

Listing 2.2: Textual NOTAM - Additional Temporal Information

The temporal information in this NOTAM, *TUE-FRI 0130-1300 1811161200-1901011200,* means that the NOTAM is only relevant from Tuesday to Friday from 1:30 a.m. to 1 p.m. in the period from 16 November 2018 until 1 January 2019. The time information after the dates is only relevant for the period, but not for the actual time slot where the NOTAM is active, i.e. when the NOTAM is actually relevant. As the 1 January 2019 is a Tuesday, the NOTAM would be active from 1:30

a.m. to 1 p.m., but as the period ends at 12 p.m., the NOTAM is not active from 12 p.m. to 1 p.m. anymore.

AIXM includes a temporality model, which allows the definition of different time slices for information concerning permanent or temporal changes [23]. A temporary change can either be permanent or temporary, but in AIXM the temporary status of the feature does not have to replace the permanent status, by generating a new NOTAM. Instead, a temporary time slice, like the schedule *TUE-FRI 0130-1300*, is overlaid on the permanent status, and can easily be reverted back to the permanent time slice [23].

Using a defined structure for the representation of the different aspects of information increases the human readability and minimizes the risk of misunderstandings. Therefore, AIXM includes a temporality model to store all states and events of aeronautical facilities [23]. The automatic processing benefits of this model and can easily extract the relevant information for further processing and presentation to humans. This means that temporal information changes in DNOTAMs can be processed in a machine-readable way as they are encoded in a defined structure.

Addressing the geographical information of DNOTAMs, the AIXM contains a geographical model based on GML [24], which is an international standard for exchanging geographical information in XML format [26]. While textual NOTAMs only contain information about the position and influenced radius [4], GML allows DNOTAMs to describe geographical information through detailed geometries which consequently improves the filtering capabilities as it is more precisely [26]. Figure 2.1 shows the difference between the geographical information of textual NOTAMs compared to DNOTAMs.

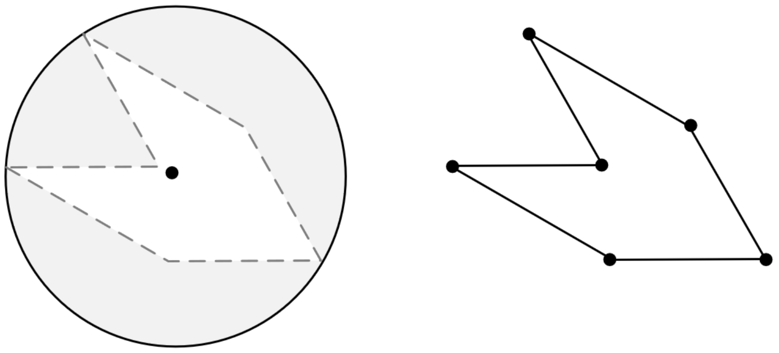


Figure 2.1: Geographic information in NOTAMs/DNOTAMs

The left part of Figure 2.1 shows the geographic information in a textual NOTAM where the center of the shape and a radius is defined. The grey part shows the irrelevant geographical information that is included in this NOTAM. On the right side, the geographical information of a DNOTAM represented with GML is shown, where each corner of the actual shape can be defined as an explicit point, and therefore the filtering is more precise. Furthermore, as GML is based on XML, points of one geometry can be easily reused by other geometries by simply referencing the XML element.

An example of a geometric surface represented in GML is shown in Listing 2.3, where the positions of the surface are directly listed, instead of referencing another XML element.

```
<ns9:ElevatedSurface
  srsDimension="2" srsName="urn:ogc:def:crs:EPSG::4326"
  xmlns:ns9="http://www.aixm.aero/schema/5.1"
  xmlns:ns5="http://www.opengis.net/gml/3.2"
  ns5:id="ESf06e533a-f75c-4b72-a49b-da596c5c5c1a">
  <ns5:patches>
    <ns5:PolygonPatch>
      <ns5:exterior>
        <ns5:LinearRing>
          <ns5:posList>
            40.50954437508608 -73.71719841801409
            41.74146679120112 -73.75080079239157
            1.83145522828321 -73.7499539336916
            40.50954437508608 -73.71719841801409
          </ns5:posList>
        </ns5:LinearRing>
      </ns5:exterior>
    </ns5:PolygonPatch>
  </ns5:patches>
</ns9:ElevatedSurface>
```

Listing 2.3: GML surface

2.1.2 System-Wide Information Management

The ATM process involves various stakeholders, e.g., operators, pilots, and airport authorities, which need to communicate and exchange information with each other [9]. Traditionally, these stakeholders have their own communication interfaces, connected over point-to-point connections, as shown on the left side of Figure 1.1 [7]. In this setting, the stakeholder providing information (the producer) communicates directly with the stakeholder that needs the information (the consumer). The communication over point-to-point connections is based on the assumption that consumers know beforehand where, that is, from which producer, they can obtain the required information, as there is no central administration available allowing consumers to find producers based on their information need [9].

The concept of SWIM envisions using a service-oriented architecture for the whole ATM communication to facilitate the exchange of data between various applications [7]. Following a service-oriented architecture allows to decouple information provisioning and consumption [9]. Ultimately, SWIM will result in a marketplace, where producers provide, and consumers consume various information via aeronautical information services [9]. Information producers provide relevant information as so-called data products for the consumers, whereas information consumers shall have the ability to search for data products according to their information need. Furthermore, these roles can be intertwined as, for example, a consumer accesses information regarding to his/her need, processes the information in some way, and afterwards, provides this information as a data product for other consumers. In this case, the user acts as a consumer as well as a producer in the marketplace.

SWIM aims to ensure a common understanding of information by providing relevant information to the right people [9]. EUROCONTROL states that the collaborative information-sharing approach of SWIM is an important driver for global interoperability and standardization [8]. DNOTAMs are one kind of information that can be exchanged in a SWIM marketplace. A data-centric approach for SWIM, concerning data products for DNOTAMs and other aeronautical information is developed as part of the BEST project [9], which is described in Section 2.2.2.

2.1.3 Knowledge-based systems

A knowledge-based system is a program that was developed to emulate the work of experts in a specific field or to support humans in their decision-making process [27]. The system uses the built-in knowledge to solve problems or make decisions, as a human would do. When compared with human expertise, a knowledge-based system is focused on a specific problem and can be more accessible than one or a few experts [27]. On the other hand, humans are creative, have a broad focus, and can use common sense knowledge – all of this currently cannot be performed by a knowledge-based system.

Kendal and Creen describe the following seven main types of knowledge-based systems [27]:

1. **Expert systems:** Expert systems are used to model algorithms that mimic the human decision-making process. The main characteristics of an expert system are that it uses knowledge and an inference mechanism to process this knowledge. Expert systems are mainly used to give advice in the decision-making process by processing existing data and utilizing rules to find the most suitable solutions.
2. **Neural networks:** A neural network consists of simplified models of the brain. Based on measured data, a neural network is able to perform classification and predictions by estimating relationships in the given data and can adopt different behavior based on new data. Neural networks are used for pattern recognition tasks and for predicting future trends.
3. **Case-based reasoning:** In case-based reasoning systems, existing cases, which consist of the definition of the problem and the corresponding solution, are used to reason via analogy. For new problems, the descriptions of the existing problems and solutions are used, to find a match for the situation and solve it accordingly, such as humans would do based on their experience.
4. **Genetic algorithms:** Methods of evolving solutions to complex problems are called genetic algorithms. The algorithms in neural networks are static, whereas a genetic algorithm tries to mimic evolution by reproduction, crossover, or mutation of the code. Genetic algorithms are mainly used for scheduling tasks, such as timetables or transportation tasks.
5. **Intelligent agents:** Intelligent agents are programs that have a specific goal or task defined, but the software can make its own decisions on how to achieve it. The purpose of intelligent agents can either be to gather information and provide reports or to make decisions based on the gathered information and perform actions.
6. **Data mining:** Data mining is a term that describes knowledge discovery, knowledge extraction, and data harvesting. The idea is that a database containing a large amount of data is used in conjunction with a data mining algorithm to uncover specific and useful knowledge and relationships. Data mining algorithms can also be used to predict future trends and use data analysis techniques, such as neural networks, decision trees, or genetic algorithms.

7. **Intelligent tutoring systems:** Tutoring systems are used to support teaching, but for an intelligent tutoring system, it is important to react to the different student's learning, as a teacher would do. In contrast to simple tutoring systems, intelligent tutoring systems provide multiple teaching methods according to the type of domain knowledge.

Often the terms knowledge-based system and expert system are used as synonyms, but it is just one type of knowledge-based systems [27]. The main characteristic of an expert system is the clear distinction between the knowledge base, where the expertise is stored, and the processing of the knowledge for solving specific problems [28]. A schematic structure of an expert system is depicted in Figure 2.2, which clearly shows the separation of the knowledge base and knowledge processing.

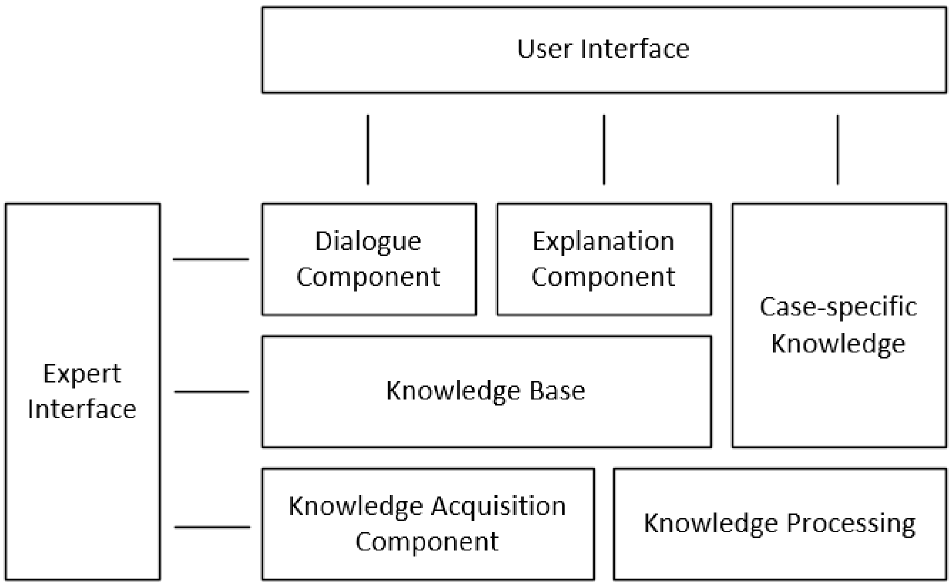


Figure 2.2: Schematic structure of expert systems [28]

As depicted in Figure 2.2, an expert system schematically consists of several components and provides different types of interfaces, i.e., an interface for the expert that needs to provide his/her knowledge, and an interface for the user of the system, who needs to have a problem solved. Rule-based knowledge that is provided by the experts over the knowledge acquisition component is stored in the knowledge base, whereas knowledge that is provided by the user, for example, a description of the problem is defined as case-specific knowledge [28]. For both interfaces, a dialogue component exists, but for the user additionally an explanation component is included, that is used to describe how the system presents how the given problem was solved by using the experts' knowledge [28].

In the SemNOTAM project, the knowledge base consists of specific knowledge and deduction rules [12]. The knowledge processing, in this case, is the use of the rules by a reasoner to derive new knowledge, which in the end, becomes part of the knowledge base. A more detailed description of the knowledge representation and processing in SemNOTAM is given in Section 2.2.1.

For the container management, the knowledge base is defined as ontology, i.e., the Container Ontology, which contains all relevant information about the data containers and their DNOTAMs. The Container Description Services are part of the knowledge processing, and the Container Management Application is used for knowledge acquisition. On the one hand, the expert interface is the Container Ontology which allows to define new concepts. On the other hand, it is realized through the Container Management Application which enables experts to create new data containers based on the existing knowledge base, i.e., the concepts in the Container Ontology. A more detailed description of the connection between the systems and the corresponding functionality is given in Section 4.

2.1.4 Ontology

The term ontology originates from the field of philosophy, but in the 1990s, it was introduced as a technical term [29]. In a philosophic field, ontology is defined as the study of being, concerning what entities exist, how they are grouped or divided according to similarities and differences, and how the entities are related to each other [30]. As already mentioned in Section 1.2, in the informatics, an ontology defines a formal vocabulary that allows to query and exchange information between actors that share the same understanding. An ontology can be defined by the following three main characteristics. An ontology is:

- an explicit, formal specification
- of the conceptualization of a defined area for a specific purpose
- that a group of actors has agreed on [31].

Staab and Studer [31] describe these three characteristics as follows. The conceptualization of a specific area refers to concepts and relations between those concepts that give a simplified view of the relevant parts of the world you want to describe. The identified concepts need to be explicitly specified in a suitable language, for example in a machine-readable way by using F-Logic or the Web Ontology Language (OWL). As the purpose of ontologies is to share it with other people, all involved actors need to have the same understanding of the specified ontology. To understand a defined ontology correctly, the communication between the machine and the human has to be enabled [31]. This communication can be represented by the semiotic triangle, which is depicted in Figure 2.3.

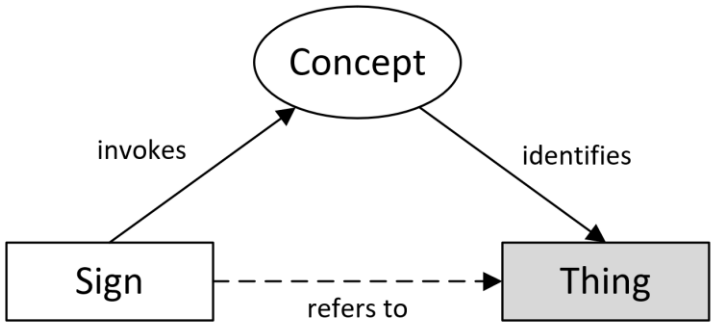


Figure 2.3: Semiotic triangle [31]

The semiotic triangle in Figure 2.3 illustrates that when somebody sends, and another one receives any kind of sign (e.g., a word, a picture, or a symbol), both actors invoke some concept in their mind represented by this sign [31]. Depending on the context of the communication, both actors identify a thing or entity based on their concept, which leads to the indirect reference of a sign to a thing.

In this thesis, all stakeholders in the SWIM environment need to have the same understanding of the concepts for describing data containers. Therefore, an ontology defining these concepts is needed to ensure the interchangeability of the data containers across various producers and consumers.

Two other important terms in the context of ontologies are the Closed World Assumption (CWA) and the Open World Assumption (OWA), which refers to whether a specific part of the world is assumed to be modeled in total or some parts are not modeled although they exist in the real world [32]. Figure 2.4 shows an example of an ontology as a graph, which has the concept *Person* and two individuals *Max* and *Mike*.

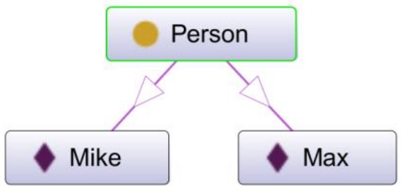


Figure 2.4: Open World Assumption vs. Closed World Assumption

Additional to the concepts in Figure 2.4, we have the concept *hasSibling* that defines one *Person* to be a sibling of another *Person*. Now we want to know if *Max* and *Mike* are siblings. When considering the CWA, the answer would be No as the negation would be assumed to be true [33], but when considering the OWA, the answer is unclear, as this relationship might exist but is not modeled. With ontologies one typically makes the OWA, which means that everything that is not modeled is unclear and cannot be inferred as true or false.

2.1.5 Reasoning and subsumption hierarchies

Reasoning describes the process of using existing knowledge and rules to infer new knowledge and is a central component of each knowledge-based system [28]. The reasoning term also comes from the philosophical field, where Peirce[34] defined the following three types of inference [34]:

1. **Deduction:** Based on the existing knowledge, a certain truth is deducted, by concluding from a general case to one specific case. For example, if we know that all birds can fly and Tweety is a bird, then we can infer that Tweety can fly.
2. **Induction:** Induction means, that based on several individual cases, general knowledge can be inferred. In most cases, the new knowledge might be correct, but it is not necessarily true for all possible cases. For example, if we know three dogs, that do bark, but do not bite, we infer that dogs that bark do not bite. However, this is an uncertain inference.
3. **Abduction:** Due to rule-based knowledge, an inference is made for an observation, which is often used in medicine. For example, with the rule that when a patient has a high temperature, then he/she most likely has the flu, and the observation that a person has a high temperature, then he/she probably has the flu. This is again only an uncertain inference.

The reasoning process can also be distinguished in monotonic and non-monotonic reasoning, which refers to whether knowledge is persistent or can be thrown away and redone [27]. In non-monotonic reasoning, it is possible that due to new knowledge, some of the inferred knowledge before becomes false and has to be removed from the knowledge base [35], whereas in monotonic reasoning newly inferred knowledge cannot be conflicting the existing knowledge [36].
The open world assumption described in Section 2.1.4 also has an impact on the results of the reasoning process. For example, when defining a concept in an ontology for *only child* as a person that has no siblings and having the individual *Max* who has no siblings defined, the reasoner will not infer that *Max* is an *only child*. Due to the open world assumption, only asserted knowledge is used in the reasoning and nothing is assumed. There might be a sibling of *Max* that is not modeled in the ontology, and therefore, it is not sure that *Max* is an *only child*.

Reasoning enables to infer new knowledge, so-called facts, by combining the given knowledge represented in the ontology. This also includes the identification of concepts that subsume other concepts, which allows to derive a so-called subsumption hierarchy. The possibility to identify such hierarchies enables to automatically find the most specific superset, the set that is directly one level above the current set. The data set of the subset can be retrieved by filtering the data items of the superset. How to find the most specific superset and the factors that are considered are further discussed in Section 3.2, where the task is defined, and in Section 5.2.3 and Section 5.4.3, where the concrete implementation is described.

2.2 Projects

This section gives an overview of the projects SemNOTAM and BEST. The SemNOTAM project is about a knowledge-based system that allows intelligent, fine-grained filtering, and annotating of DNOTAMs, which was launched in 2014 [12]. The project BEST focuses on a data-centric perspective to distribute data containers over SWIM [15].

2.2.1 SemNOTAM

The research project SemNOTAM was launched in 2014 as a cooperation between industry and university partners with the aim to filter and annotate DNOTAMs [10]. The research of the Department of Business Informatics – Data & Knowledge Engineering at the Johannes Kepler University covers inter alia business intelligence, semantic technologies, and the development of web-based and knowledge-based systems [37]. One of the main industrial partners is the Austrian company Frequentis AG, which develops safety-critical communication and information solutions in safety and transportation markets [38]. Other cooperation partners are EUROCONTROL and the FAA, who conduct research on modernizing and harmonizing the ATM systems, i.e., in the Single European Sky ATM Research Program (SESAR) or with the Next Generation Air Transportation System (NextGen) [39].

The huge and steadily growing number of DNOTAMs that results in information overload, especially for pilots, has shown that intelligent querying and filtering of DNOTAMs is an important aspect of air traffic safety [12]. Existing filtering systems, e.g., the FNS-NDS, only support basic filtering and do not fit the need of complex semantic querying of DNOTAMs [12]. According to Burgstaller et al. [12], two specific approaches exist in the literature that address issues of intelligent NOTAM processing. The first approach was for querying DNOTAMs by the use of an ontology that contains knowledge about DNOTAMs [40]. Defining when, how, and for whom the information is relevant is not possible with this approach, as no business rules are used [12]. The second approach uses business rules for notifying the cockpit crew with DNOTAMs depending on their relevance for a certain flight and their significance [41]. This approach is not knowledge-based and therefore, does not allow to use ontologies for the reasoning process [12].

2.2.1.1 Problem description and method

In the SemNOTAM project, a combination of rule- and ontology-based approach is used in a knowledge-based framework, which allows fine-grained filtering and querying of DNOTAMs [12]. Conditions of DNOTAMs are described with event scenarios, e.g., runway closure, that are then used in business rules to derive the relevance and importance of a DNOTAM [12]. Additionally to the event scenarios, SemNOTAM supports business terms for the definition of precise, understandable, and machine-processable concepts, that can again be used in business terms to derive the relevance and importance of DNOTAMs concerning these scenarios [12].

As described in Section 2.1.3, a knowledge-based system consists of two main components, respectively the knowledge base and the reasoner. The knowledge base in the SemNOTAM system consists of the knowledge and the business rules [12]. As depicted in Figure 2.5, the reasoner uses the rules to derive new knowledge out of the existing knowledge, which then becomes part of the knowledge in the knowledge base for further processing.

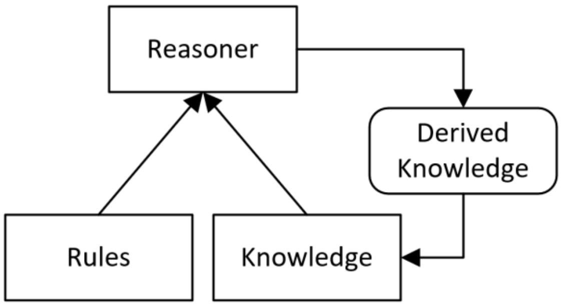


Figure 2.5: Reasoning process [12]

The knowledge-based framework of SemNOTAM accesses different data to build up its knowledge, e.g., DNOTAMs, AIXM baseline data, configuration data, e.g. aircraft characteristics, and the query interface, where specific interests can be provided [12]. Figure 2.6 shows that all data needs to be mapped from AIXM to the ontological representation to store it in the SemNOTAM ontology.

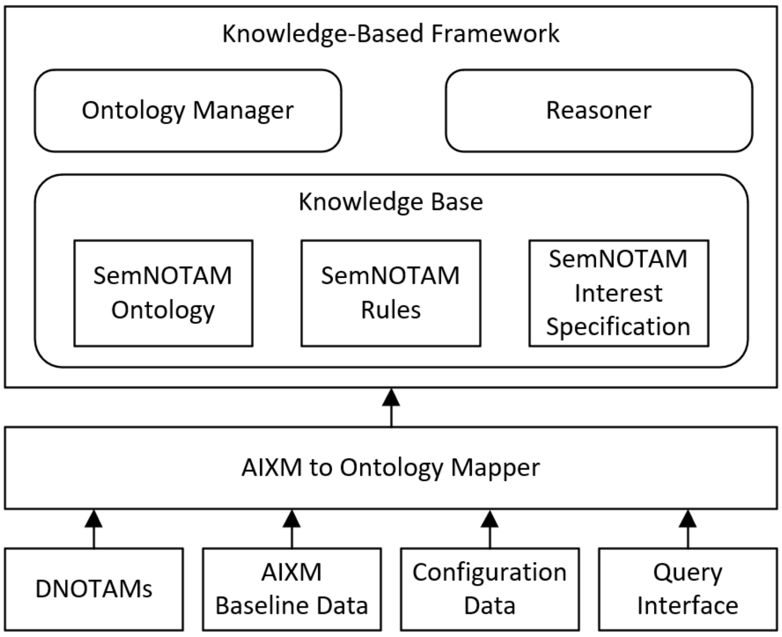


Figure 2.6: SemNOTAM Settings and Components [12]

As described before, the knowledge base contains the ontology that comprises the business terms and the hierarchical relationship between them [12]. The SemNOTAM Rules contain the business rules that define whether a DNOTAM is relevant or not and classifies its importance based on the business terms of the ontology. The SemNOTAM Interest Specification allows the specification of what the user is interested in, by defining so-called simple interests that can be combined to complex interests [12]. However, each simple interest is evaluated separately, and the results are combined to complex interests with union or intersection. The complete definition of the user's task, the combination of simple and complex interests, is called interest specification, i.e., task description. A more detailed description of the structure and use of interest specifications is given in Section 2.2.1.2.

For the integration with external applications the SemNOTAM Web Service was developed, which provides filtering functionalities of the SemNOTAM prototype. The concrete usage of this web service is described in Section 5.3 and is demonstrated in Section 6.2. Interest specifications, which are the input for the SemNOTAM Web Service, are introduced in the next section.

2.2.1.2 Interest specification

An interest specification is the definition of what the user is interested in, and affects which rules of the SemNOTAM system are applied [12]. A user can define his/her task to perform in three different interests, namely aircraft, temporal, and spatial interests [12]. These types of simple interests can be combined to complex interests using intersection or union. Listing 2.4 shows an exemplified simple interest in XML format for the aircraft type 717. By defining this interest, SemNOTAM rules that are not relevant for this aircraft type are not considered further.

```xml
<ns11:AircraftOfInterest
  xmlns="http://www.aixm.aero/schema/5.1/extensions/FAA/FNSE"
  xmlns:ns11="http://semnotam.frequentis.com/schema/1.0"
  xmlns:gml="http://www.opengis.net/gml/3.2"
  gml:id="AF_717">
  <ns11:aircraftTypeName>717</ns11:aircraftTypeName>
</ns11:AircraftOfInterest>
```

Listing 2.4: Simple Interest - Aircraft

A temporal interest, as represented in Listing 2.5, has a bit more structure, where the begin of the time interval is defined in Line 4 and the end of the time interval is defined in Line 5. This temporal interest defines that the user is only interested in DNOTAMs that are relevant in the year 2017.

```xml
1   <ns11:PeriodOfInterest
      xmlns="http://www.aixm.aero/schema/5.1/extensions/FAA/FNSE"
      xmlns:ns11="http://semnotam.frequentis.com/schema/1.0"
      xmlns:gml="http://www.opengis.net/gml/3.2"
      gml:id="TIF_Year2017">
2     <ns11:occTime>
3       <ns11:TimeInterval>
4         <ns11:beginPosition>
            2017-01-01T00:00:00.000Z
          </ns11:beginPosition>
5         <ns11:endPosition>
            2018-01-01T00:00:00.000Z
          </ns11:endPosition>
6       </ns11:TimeInterval>
7     </ns11:occTime>
8   </ns11:PeriodOfInterest>
```

Listing 2.5: Simple Interest - Period of Interest

To define an interest that has both restrictions a complex interest is used, in this case an intersection interest, shown in Listing 2.6. Intersection means that only when a DNOTAM fits both restrictions, i.e., it is relevant in 2017 and for the aircraft type 717, it is relevant for this complex interest. The evaluation is done for each simple interest and those DNOTAMs that are included in both result sets are returned to the user.

```xml
<ns11:IntersectionInterest
  xmlns="http://www.aixm.aero/schema/5.1/extensions/FAA/FNSE"
  xmlns:ns11="http://semnotam.frequentis.com/schema/1.0"
  xmlns:gml="http://www.opengis.net/gml/3.2">
  <ns11:hasMember>
    <ns11:AircraftOfInterest gml:id="AF_717">
      <ns11:aircraftTypeName>717</ns11:aircraftTypeName>
    </ns11:AircraftOfInterest>
  </ns11:hasMember>
  <ns11:hasMember>
    <ns11:PeriodOfInterest gml:id="TIF_Year2017">
      ...
    </ns11:PeriodOfInterest>
  </ns11:hasMember>
</ns11:IntersectionInterest>
```

Listing 2.6: Complex Interest - Intersection

In contrast to the intersection interest, the union interest can be seen as a logical OR, which is mostly used for spatial interests. For example, when defining two spatial interests, like in Listing 2.7 one departure aerodrome area and one destination aerodrome area, the user is interested in DNOTAMs that are in the first aerodrome area or in the second aerodrome area. It only has to fulfill one of the requirements to be part of the result set.

```xml
<ns11:UnionInterest
  xmlns="http://www.aixm.aero/schema/5.1/extensions/FAA/FNSE"
  xmlns:ns11="http://semnotam.frequentis.com/schema/1.0"
  xmlns:gml="http://www.opengis.net/gml/3.2">
  <ns11:hasMember>
    <ns11:DepartureAerodromeArea>
      <ns11:designator>KJFK</ns11:designator>
    </ns11:DepartureAerodromeArea>
  </ns11:hasMember>
  <ns11:hasMember>
    <ns11:DestinationAerodromeArea>
      <ns11:designator>KIAD</ns11:designator>
    </ns11:DestinationAerodromeArea>
  </ns11:hasMember>
</ns11:UnionInterest>
```

Listing 2.7: Complex Interest - Union

The combination of all restrictions with union and intersection interests is then called interest specification, i.e., task description. This specification is used by the SemNOTAM prototype for filtering DNOTAMs based on the corresponding rules. How the Container Management Application creates the interest specifications and why the interest specification is also relevant for the SemNOTAM Briefing Application is described in Section 4.1 and Section 3.1.2.

2.2.1.3 Filtering process

As already introduced in Figure 2.6, the filtering and annotation process is covered by the SemNOTAM Rules, which are used by a reasoner to retrieve the relevant DNOTAMs and infer their importance. For the definition of the SemNOTAM Rules the language ObjectLogic, which is based on the knowledge representation technology F-Logic, was used [12].

As already described before, business terms are used to define specific concepts, that are referenced in the filtering and annotation rules. Business terms can also be hierarchically structured, by defining so-called super-business terms, for example, the business term *aerodrome* is a super-business term of the business terms runway, taxiway and gateway [12]. The representation of this hierarchy in ObjectLogic is shown in Listing 2.8. Each DNOTAM (defined by `?n:NOTAM`) will be classified as an `AerodromeConcept` if it is of the type `RunwayConcept`, `TaxiwayConcept` or `GatewayConcept`.

```
?n:AerodromeConcept :- ?n:NOTAM, (?n:RunwayConcept OR
  ?n:TaxiwayConcept OR ?n:GatewayConcept).
```

Listing 2.8: Business Term – AerodromeConcept [12]

The defined business terms can then be used in the relevance rules to reduce the complexity of the rules and increase the readability [12]. These SemNOTAM Rules are defined similar to business terms, which either determine a DNOTAM to be relevant or irrelevant if it is a negative rule. Most of the rules will be negative because a recall of 100 % should be provided, which means that no relevant DNOTAM must be excluded [12]. An example for such a negative rule is given in Listing 2.9, which classifies all DNOTAMs of the type `RwyClosure` as irrelevant for a specific `aircraftInterest` if the aircraft width is more than 20 ft smaller than the closure width.

```
aircraftInterest[irrelevant -> ?n] :- ?n:RwyClosure,
  AircraftInterest[type -> ?type[wingspan -> ?acWSpan]],
  ?n[wingspan -> ?nWSpan, wingSpanInterpretation -> "above"],
  (?acWSpan + 20) < ?nWSpan.
```

Listing 2.9: Relevance Rules – Wingspan Rule [12]

Additionally to the filtering functionality, SemNOTAM also covers the aspect of annotating relevant DNOTAMs based on their importance [13]. For example, some DNOTAMs are only additional information for a flight, while others need to be classified as flight critical. In the SemNOTAM project the following six importance levels can be distinguished:

- Flight Critical
- Special Consideration
- Operational Restriction
- Potential Hazard
- Additional Information
- Unknown Importance

The annotated importance of the DNOTAMs depends on the importance for the flight and the probability of the DNOTAM. For example, a runway closure would be annotated as flight critical, whereas, wildlife hazards at an aerodrome would only be annotated as additional information [13].

2.2.2 BEST

The research project BEST was launched in 2016 and focuses on a data-centric perspective with the aim to identify how semantic technologies can be used in combination with the SWIM concept in aeronautical domain [9]. It shall be determined what types of ATM data can be represented in ontologies and how these ontologies can effectively be used in SWIM [42].

The BEST project is a follow-up project of the original SESAR project, that has two main results. First the SWIM concept that will lead to changes in how ATM information is provided and second ATM Information Reference Model (AIRM), which provides a standardized vocabulary for sharing information [18]. In addition to SWIM and AIRM, modelling techniques, languages and tools of semantic technologies should be used to avoid information overload and enable a truly effective information management [18].

2.2.2.1 Problem description and method

The SWIM concept that is used in the BEST project aims for an service-oriented architecture for the organization of ATM communication and achieving a shared understanding by all stakeholders [9]. As already described in Section 2.1.2, the result of SWIM will be a marketplace, where producers provide, and consumers consume various information via aeronautical information services [9]. According to EUROCONTROL, this concept of collaborative information-sharing is an important driver for global interoperability and standardization [8].

In the BEST project the AIRM ontology was separated into several ontology modules for certain topics, for example, Aircraft or Meteorology [16]. The ATM information represented by these vocabularies is then used by data containers, which describe the information provided to end-users [42]. Furthermore, to describe how the ATM information should be exchanged, in BEST strategies for data distribution and consistency management are developed [42]. Gringinger et al. [16] use the term data container as a synonym for data containers while Kovacic et al. [9] only use the term data container. Therefore, the concept of these containers will be referred to as data containers in this thesis. The next section describes in more detail how data containers are structured and which types of information they contain.

2.2.2.2 Data containers

To organize ATM information in suitable data sets the concept of data containers was introduced, which should enable an effective data exchange [42]. One data container contains data items of one specific data type, for example, DNOTAMs [9]. The data items that are included in one data container are described by the corresponding semantic label [9]. Data containers can be distinguished between Primary Data Containers and Secondary Data Containers, as depicted in Figure 2.7. Primary Data Containers have a defined data set, whereas Secondary Data Containers have another data container as data source, either a Primary Data Container or a Secondary Data Container, and filter the data set accordingly [9].

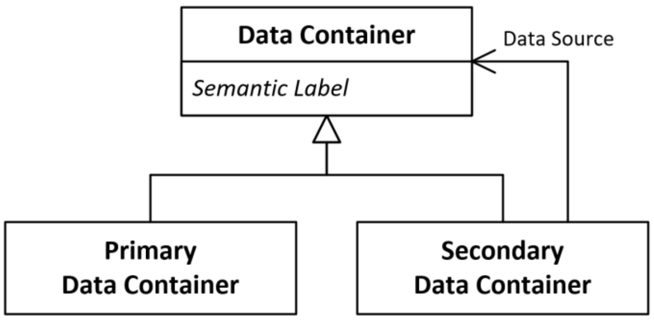


Figure 2.7: Data Container Hierarchy [9]

Each data container has a semantic label that describes the corresponding data set. A semantic label consists of several ontology-based concepts and annotations, so-called metadata [9]. Two different types of metadata can be distinguished, administrative metadata for container maintenance and descriptive metadata for the description of the data items as shown in Figure 2.8 [9].

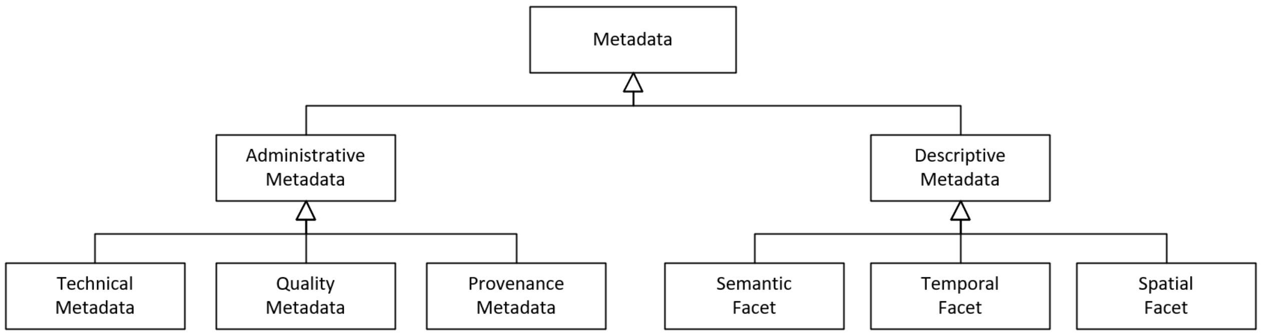


Figure 2.8: Semantic Label – Metadata [9]

As depicted in Figure 2.8 the administrative metadata as well as the descriptive metadata are each further divided into three different categories. The administrative metadata is divided in the following three groups [9]:

- **Technical metadata** describes technical characteristics of the data set of the data container, such as data format or encoding.
- Completeness, relevance or timeliness are examples for **quality metadata**, which provides information about the freshness of the data container, like the date of the last change and the last update.
- **Provenance metadata** describes the origin and data processing by specifying the data source, which is another data container and a data service that is used for filtering the data set of the data source accordingly.

The descriptive metadata either describes **semantic**, **temporal**, or **spatial** conditions of the data set of the data container [9]. Therefore, these three different types of facets are distinguished, where one or more facets belong to each category. The realization of the facets in the container ontology will then be further described in Section 5.2.1.

In the context of this thesis, reasoning is used to identify hierarchies between the concepts of a specific facet and consequently derive the subsumption hierarchy of the data containers based on their descriptive metadata. For example, the set of DNOTAMs including all DNOTAMs of Europe subsumes the set of DNOTAMs that only include the DNOTAMs for Austria and this set again subsumes the set containing the DNOTAMs for Upper Austria. By applying reasoning on the facts described by the spatial information, the subsumption hierarchy shown in Figure 2.9 is inferred.

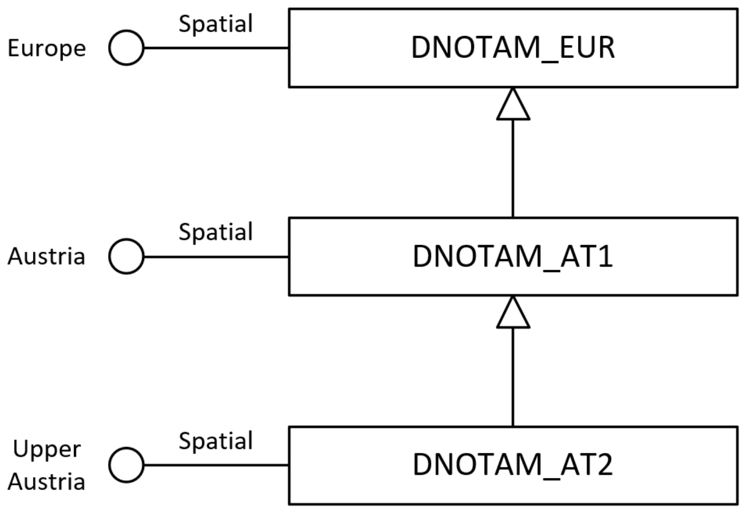


Figure 2.9: Data Container – Subsumption hierarchy

3 Requirements Analysis

This section provides a detailed task description, which clearly describes the scope of this thesis and serves as a base for the definition of requirements. At first, the existing applications of SemNOTAM are introduced to understand the integration with them. Afterwards, a detailed task description is given, which describes the implementation components of this thesis and serves as a base for the definition of requirements. Furthermore, possible problems that might occur are presented. Further sections will then refer to the identified requirements when describing the design (Section 4) and the implementation (Section 5).

3.1 Existing applications

An overview of the existing applications that are used by or use the Container Management Application and the Container Description Service is given in this section. A simple depiction of the applications is shown in Figure 1.3. Since the aim of the SemNOTAM project is to enable the intelligent and fine-grained filtering of DNOTAMs, a web service was developed, which is also used by the Container Management Application for secondary containers. As this thesis also covers a task-based retrieval service, the SemNOTAM Briefing Application is also described, which allows pilots to find relevant DNOTAMs according to their information need, respectively their task.

3.1.1 SemNOTAM Web Service

The SemNOTAM Web Service enables to use the filtering capabilities of the SemNOTAM prototype by providing two different methods for filtering DNOTAMs. Either the set of DNOTAMs is given by the user as input for the method, or the system performs the filtering based on a set of DNOTAMs that fits the user's need. In both cases, the user has to pass the information need, i.e., task description, in the form of an XML interest specification as described in Section 2.2.1.2.

The integration of the SemNOTAM Web Service aims for two goals. On the one hand, this service is used by the Container Management Application to perform the filtering of DNOTAMs accordingly to the descriptive metadata of a data container. On the other side, the identification of the most suitable set of DNOTAMs for the user's need is performed by the Container Description Service, which means that the SemNOTAM Web Service uses methods of the Container Description Service. Therefore, when the user sends a request to the SemNOTAM Web Service without a specific set of DNOTAMs, the Container Description Service will be called along with the task description i.e., the interest specification. The result of this call will provide the SemNOTAM Web Service with the corresponding most specific superset. The specific requirements concerning this service are described in more detail in Section 3.3.

3.1.2 SemNOTAM Briefing Application

Another part of this thesis is the integration of the Container Management Application with a briefing application by providing a task-based retrieval service. The SemNOTAM Briefing Application is the client web application of the SemNOTAM system, which enables pilots to filter DNOTAMs based on their task to be performed. By defining a flight plan according to their task, an interest specification is mapped and used as input for the SemNOTAM Web Service. The users of the SemNOTAM Briefing Application only pass their task description and the SemNOTAM Web

Service returns the filtered and annotated/grouped DNOTAMs. To this end, the SemNOTAM Web Service receives the most specific superset regarding specific task to be performed from the Container Description Service. To enable the Container Description Service to transform the task description into a data container, a service has to be provided that forces the users of the SemNOTAM Briefing Application to define their task by using the predefined concepts. These concepts need to be defined in the Container Ontology.

3.2 Task Description

To clarify the aim of this thesis, this section describes the task to be fulfilled. It is clearly specified what functionality shall be covered by the implementation and how the subsystems relate.

A management application for data containers should be developed. A data container can either represent a primary container which does not have another data container as a data source or a secondary container which uses other primary or secondary containers as data source. Regardless of the type, a data container consists of the data set, including the data items and a semantic label representing metadata. The metadata can either be descriptive or administrative. Descriptive metadata is used to describe the membership conditions fulfilled by the data items in the data set. Administrative metadata is used to capture other technical, quality, and provenance aspects of the data container.

The semantic label includes the descriptive metadata, such as the time period or the spatial area covered by the data set. The concepts of the descriptive metadata are assigned to semantic labels by so-called facets. The basic idea of facets is to describe the features of the content, e.g., the contained DNOTAMs. The data set must be sound, which means that each data item in a data set has to fulfill the assigned concepts. Furthermore, it must be complete, which means that at least all existing data items which fulfill the concepts must be included in the data set. For example, assigning the concept Austria over the spatial facet to a semantic label indicates that the data set contains all existing data items which can be assigned to Austria, or even more. Additionally, a facet “Specific Interest” shall be included, that has concepts assigned that, for example, represent a flight path. The difference between those concepts and the other administrative metadata is that typically the concepts for the descriptive metadata are used by several semantic labels, whereas a concept of the specific interest facet is only used by a single semantic label. The semantic label, as well as the descriptive metadata, shall be represented by concepts within an ontology to enable the creation of a hierarchy between the metadata of one facet as well as the semantic label.

The use of an ontology for the descriptive metadata should make it possible to create specialization respectively generalization hierarchies between concepts, which are used for facets of semantic labels. These hierarchies should either be defined explicitly by the user or should be derived by employing subsumption reasoning. Following this approach, it should be possible to derive specialization hierarchies between semantic labels, and since semantic labels describe the content of the data container, it also allows to draw subsumption hierarchies between the data containers. This allows finding the most specific superset for a given semantic label representing a user’s task description. Since multiple data containers can represent the most specific superset, the administrative metadata can be considered to decide which superset shall be selected, e.g., select the superset which contains most-recent data items, has the smallest size, or the one with the right encoding.

This additional information represented by the administrative metadata shall not be used for reasoning purposes, which is why it shall not be represented by concepts in the ontology; instead, it shall be annotated in some way.

The communication with the existing applications described in Section 3.1 also has to be realized. Therefore, web services shall be implemented to enable the task-based retrieval of relevant DNOTAMs with the SemNOTAM Briefing Application. The management application and the briefing application need to be able to select concepts for facets based on the Container Ontology. Furthermore, both systems need to build interest specifications based on the selected concepts and use this interest specification as input for a filtering service. This thesis only shows the filtering of DNOTAMs and therefore, uses the SemNOTAM Web Service, but the concept can also be applied to other aeronautical information, e.g., METeorological Aerodrome Report (METAR).

The difference between the Container Management Application and the SemNOTAM Briefing Application is as follows: The Container Management Application is used by information producers that have the intention to create new data containers, while the users of the SemNOTAM Briefing Application want to get the relevant data items for their information need. Therefore, the users of the Container Management Application decide which data container shall be used as the data source for a secondary data container. Users of the SemNOTAM Briefing Application shall not be confronted with this, they only describe their information need and send a request to the SemNOTAM Web Service. This service calls another service that shall find the most specific superset in the ontology, based on the subsumption hierarchy between containers and their administrative metadata. This most specific superset is then filtered by the SemNOTAM Web Service.

To support these and other tasks an ontology, a management application for data containers, and a web service shall be developed.

3.2.1 Container Description Service

To create a container description, the user shall be able to select concepts for the different facets. Therefore, a service to select a concept for one specific facet, which only provides concepts for this facet, must be provided. Furthermore, it shall be possible to select other ontologies which include other concepts for this facet.

The SemNOTAM system requires an interest specification as input to filter a set of DNOTAMs. This means that the container description needs to be translated to its corresponding interest specification representation. Therefore, a service is needed, which realizes the mapping between a concept and its interest representation. For that reason, the annotated interest representations of the concepts in the ontology are used.

Another service shall enable the user to find the most specific superset of his/her information need respectively specified interest specification. Therefore, a service for retrieving the most specific superset shall be implemented. The service needs to map the interests back to its corresponding concepts since interests itself do not store the concept information. The service shall find all possible most specific supersets and decide which one to use based on the administrative metadata.

3.2.2 Container Ontology

The semantic label of a data container includes the descriptive metadata which shall be represented by ontology concepts. The facets are also represented as concepts, and the descriptive metadata concepts are defined as their sub-concepts. Furthermore, each descriptive metadata concept has an interest representation annotated. The interest representation is the XML representation of the concept that is used to build the interest specification and can be used as input for the SemNOTAM Web Service. The administrative metadata is assigned to the punned instance of a data container. Punning means that the data container has a representation as class and instance, and both can be referenced through the same Internationalized Resource Identifier (IRI). The Container Ontology shall support create, modify, and delete operations on concepts and their annotations.

3.2.3 Container Management Application

An application for managing data containers shall be implemented. This application should enable the user to view, modify, delete, create, and refresh data containers, respectively their metadata.

- **View:** The user shall be able to view the existing containers which are available in the corresponding ontology. The list of containers shall display the descriptive metadata, and a detailed view of a single container shall display descriptive and administrative metadata.

- **Modify:** Furthermore, the user shall be able to modify some of the administrative metadata, e.g., the refresh interval and the refresh until timestamp. If other metadata shall be modified, e.g., descriptive metadata, then this shall be prevented, and it shall be indicated that a new container shall be created for this purpose.

- **Delete:** The user shall also be able to delete a container, but in this case, possible existing dependencies to another container must be considered, e.g., the user should be informed that the data container might act as a data source for another container.

- **Create:** For the creation of new containers, there shall be an interface where the user can select a concept for each facet. The user can choose between creating a primary container or a secondary container.
  - For primary containers, the corresponding data set has to be declared explicitly, since they do not have a data source. Therefore, no other primary or secondary data container shall be selected as input. The user shall ensure that the selected data set includes all the data items specified by the container facets respectively the administrative metadata, hence, primary containers can only be updated manually.
  - For the creation of a new secondary container, first, its most specific superset shall be found. The user shall be able to decide which superset should be used. Furthermore, a service that is used for filtering the data set has to be selected by the user, e.g., the SemNOTAM Web Service. The data set of the subsuming container is then used as input for the selected service. This service filters the data set based on the selected concepts and if existing on the specific interest. The output of this service represents the data set of the created container. The service and the parameters that are used for calling it are saved in the ontology to enable the refreshing of the container.

A data container shall be created within a repository. A repository includes the ontology and the data sets of the existing data containers. Repositories shall allow creating “public” containers and “private” containers, e.g., for specific flights. These repositories shall be considered during the data container search. The implementation only includes one repository, which acts as public repository. However, the possibility to select another repository shall be given.

- **Refresh:** The refreshing of containers shall only be implemented as a manual task. The automatic refreshing, however, shall be considered as well. Furthermore, different variants of refreshing shall be analyzed, such as the decentral/central or the push/pull refresh process. The basic steps of the refresh process can be summarized as follows: When a refresh is initiated the administrative data of the container and its data source container must be checked, whether there has been a refresh of the data source and whether the data source contains new data. If a refresh is needed, then the new data items must be filtered with the service and the parameters of the container and the data container with its metadata shall be updated.

3.3 Requirements

The task description of Section 3.1 is used to specify the requirements. These requirements are first grouped according to which of the three subsystems (Container Description Service, Container Ontology, and Container Management Application) they belong. A second grouping is done according to the part or functionality of the subsystem they describe.

1 Container Ontology
  1.1 Container
    1.1.1 A data container must be represented by a class and an individual, which have the same IRI (punning).
    1.1.2 It must be possible to assign descriptive and administrative metadata to a data container over facets.
    1.1.3 It must be possible to create a subsumption hierarchy of the existing data containers by utilizing their semantic labels.
  1.2 Descriptive Metadata
    1.2.1 A single feature of the content of a container is represented by a concept in the corresponding facet.
    1.2.2 The different types of descriptive metadata must be assigned to facets.
    1.2.3 A data containers’ descriptive metadata must be described by the combination of one concept of each facet.
    1.2.4 The concepts of the facets must be organized hierarchically to enable subsumption reasoning. Some hierarchies must be specified explicitly whereas; other hierarchies shall be derived through the characteristics of a facet, e.g., the start and end of a time period.
  1.3 Administrative Metadata
    1.3.1 The administrative metadata of the data container must be assigned to the data container.
    1.3.2 For secondary data containers, the data source container must be referenced.
    1.3.3 Each data container needs to have several timestamps assigned to enable the comparison of their quality.

2 Container Management Application
  2.1 View container
    2.1.1 The application must enable the user to see a list of all containers in an ontology.
    2.1.2 The list of containers must show the descriptive metadata.
    2.1.3 The application must enable to view the details of a specific container, which includes its descriptive and administrative metadata.
  2.2 Create container
    2.2.1 The application must enable the user to create new containers.
    2.2.2 The user must be able to specify the descriptive metadata using concepts for the facets.
    2.2.3 The user must be able to define parts of the administrative metadata, i.e., data format, data encoding, refresh interval, and refresh until timestamp.
    2.2.4 For the creation, the user must be able to decide if a primary or secondary container should be created.
    2.2.5 For primary containers, the user must be able to upload the data items for the new container.
    2.2.6 For secondary containers, the user must be able to select the superset data container to be used.
    2.2.7 The user must be able to decide which service shall be used for filtering the data items of the superset data container.
    2.2.8 The user must be able to define whether a public or a private repository should be used.
  2.3 Modify container
    2.3.1 The user must be able to modify the refresh interval and the refresh until timestamp.
  2.4 Delete container
    2.4.1 The user must be able to delete containers of an ontology.
    2.4.2 The application should inform the user that containers may serve as data source for other containers.
  2.5 Refresh container
    2.5.1 The application must enable the user to initiate a refresh for a container.
    2.5.2 For secondary data containers, the refresh shall only be started when it is needed. This means that the refresh until timestamp is not reached yet and the data source container contains new data items.
    2.5.3 When refreshing a primary data container, the user must be able to upload the file with the current data items for this data container.
3 Container Description Service
  3.1 Service for selecting a concept for one specific facet
    3.1.1 This service has a facet IRI as input parameter and returns a list of concept IRIs.
    3.1.2 This service must return all possible concepts for the given facet.
  3.2 Service for the mapping between concepts and interest representations
    3.2.1 This service has the ontology IRI and a concept IRI as input parameter.
    3.2.2 This service must return the interest representation for the given concept in the given ontology.
  3.3 Service for finding the most specific superset
    3.3.1 This service has an interest specification as input parameter and returns a feature collection with DNOTAMs.

3.3.2 The interests in the interest specification must be mapped to its corresponding concepts.
3.3.3 For the given interest specification, a data container must be created, described by concepts in the facets.
3.3.4 This service must find the most specific superset in the main ontology and return its feature collection.
3.3.5 If multiple most specific supersets are available, the administrative metadata shall be considered for the decision which data container to use.

The further sections, especially the approach and the implementation, reference these requirements when describing the corresponding parts of the applications.

3.4 Challenges and Problems

Requirement 2.2.8 defines the possibility to create data containers in private or public repositories. However, when a user has a specific information need and wants to retrieve the relevant DNOTAMS, which is a functionality provided by the SemNOTAM Briefing Application, the data container is always created in his/her private repository. The challenge for creating the link to the identified superset data container is, that at first, the superset data container shall be searched in the corresponding private repository. Only if no suitable data container can be found, the public repository is searched for a superset data container. In the implementation of this thesis only one repository is defined which acts as a public repository and the application always searches the superset in this repository. Therefore, the differentiation between private and public repositories is not considered in the implementation.

Another challenge concerning the connection between data containers and their superset data containers is the type of implementation that can be used. The automatic refreshing of data containers can either follow a decentralized or centralized approach.
The decentralized approach can be realized either with the push or the pull principle. Figure 3.1 shows the two possibilities for the decentralized refresh strategy, where the data container DNOTAM_EUR is the superset data container of the data container DNOTAM_AT1.

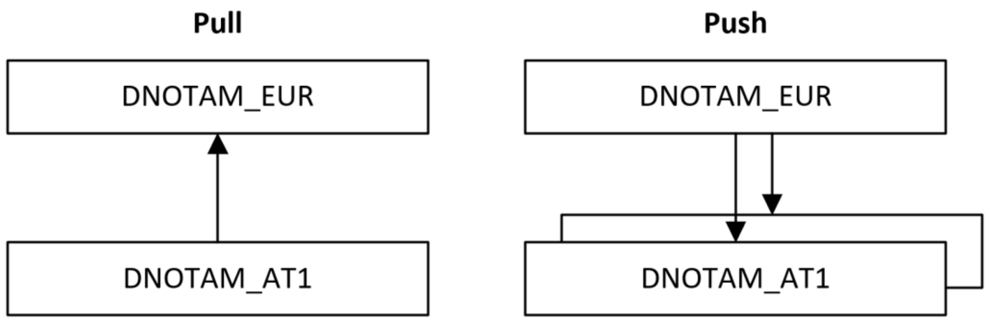


Figure 3.1: Decentralized refresh strategy – Pull vs. Push

When using the pull principle, as depicted in the left part of Figure 3.1, the secondary data container requests new data items of its superset data container. The secondary data container stores the information of when (refresh interval) to request data of which data container (data source) over which service (data service).

On the other side, when using the push principle, the data source data container, i.e., the data container DNOTAM_EUR, needs to store this information of all the data containers for which it acts as data source, in Figure 3.1, two data containers are indicated. The data container DNOTAM_EUR stores for each dependent data container when (refresh interval) to forward new data, according to the data container's interest specification.

The last refreshment possibility is the centralized approach where a central administration manages all the data source dependencies. For the above example with two data containers, this approach is depicted in Figure 3.2.

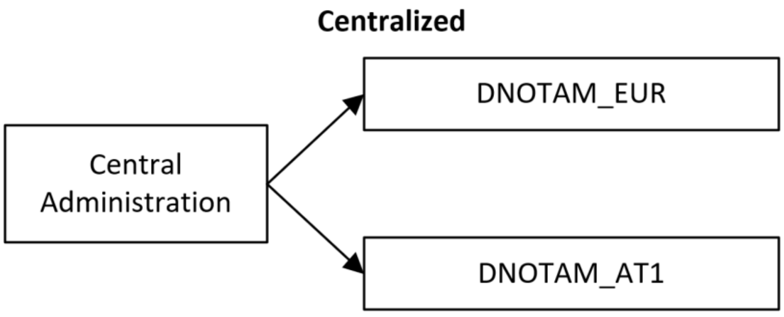


Figure 3.2: Centralized refresh strategy

The central administration stores the refreshment information for each secondary data container, respectively its data source, refresh interval and interest specification. As shown in Figure 3.2, the central administration performs a pull of new data items according to the interest specification of a specific data container and forwards the data items to this data container. Using this approach, a single point of truth is created, that has all the information about the freshness of all data containers and their refresh cycles.

However, as the automatic refreshing of data containers is not part of the implementation of this thesis, there is no decision for one of these strategies. To enable the manual refreshing of data containers, each secondary data container stores the data source, refresh interval and interest specification, and the manual refresh is implemented using the pull principle.

4 Design

In this section, the design process, including the basic approach, the underlying data model, and the architecture is detailed. The approach comprises the whole system, which consists of the components Container Ontology, Container Management Application, and Container Description Service. The data model gives an overview of the relationships between data of the different subsystems, and finally, the application architecture of the client and the server is detailed.

4.1 Approach

Three main components have to be developed, performing the task, and fulfilling the requirements identified in Section 3. First, the overall approach, the relationships, and the communication between the components, including the existing applications, are described. Section 4.1.1 details the approach of creating a suitable ontology for the definition of data containers along with the modeling of their relationships and dependencies. Section 4.1.2 describes the organization of the Container Management Application, comprising the design and functionality that should be provided. Finally, Section 4.1.3 presents the approach for the Container Description Service, which provides the task-based retrieval service and therefore, enables the integration with the SemNOTAM Briefing Application.

The overall task is divided with regard to the three introduced components Container Ontology, Container Management Application, and Container Description Service, as shown in Figure 4.1. At the bottom of Figure 4.1, the existing applications to be integrated are depicted. Above them the three components, that have to be implemented as part of this thesis are depicted, comprising the repository, including the Container Ontology and its data items, the Container Description Service and the Container Management Application. The Container Management Application and the Container Description Service both directly communicate with the Container Ontology (green uses-relationship). The Container Description Service is used by the Container Management Application as well as the existing applications, the SemNOTAM Web Service and the SemNOTAM Briefing Application (red uses-relationship). The SemNOTAM Web Service provides methods for evaluating an interest specification and a corresponding set of DNOTAMs for the Container Management Application and the SemNOTAM Briefing Application (blue uses-relationship).

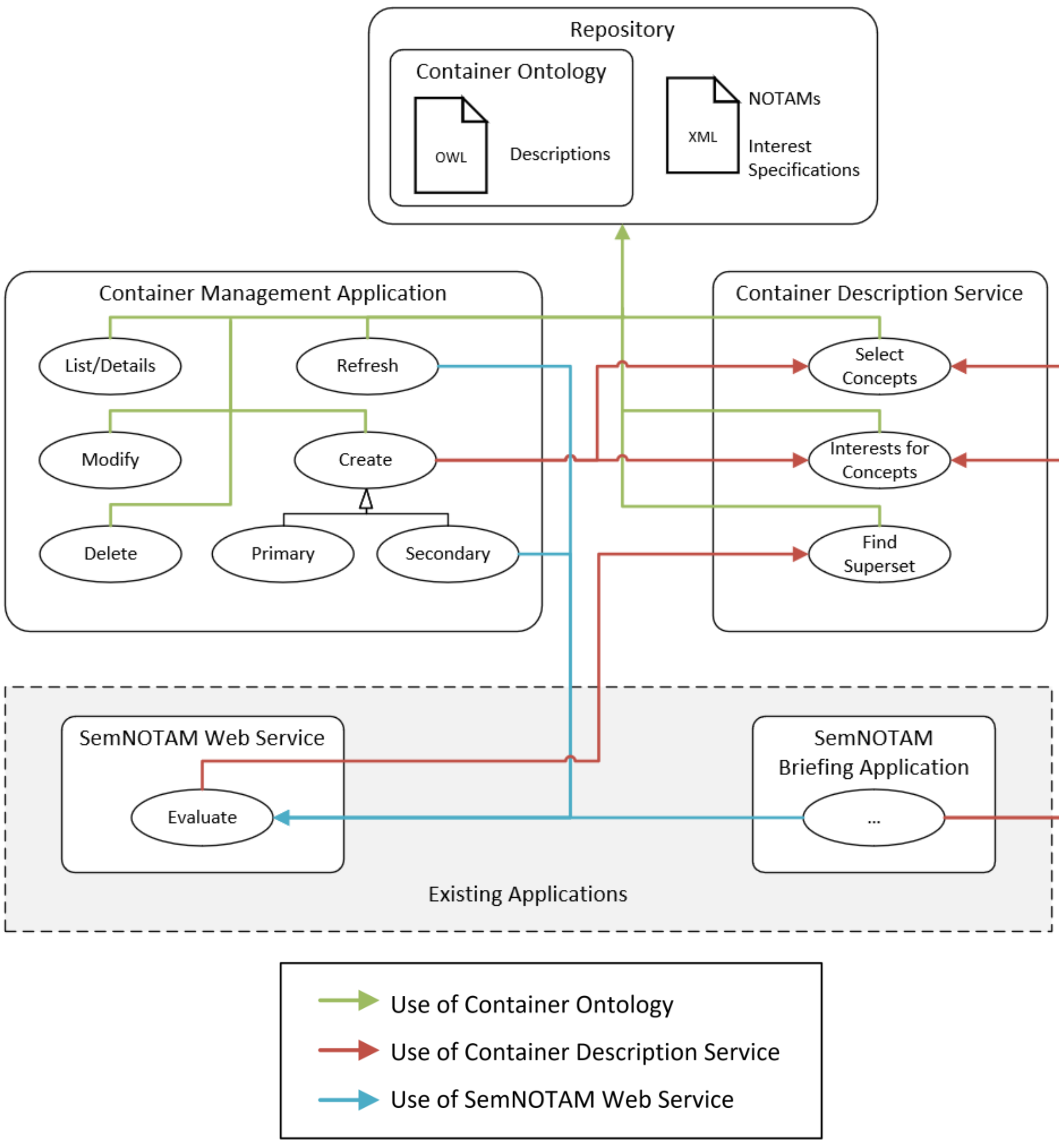


Figure 4.1: Overview of Application Functionality

The communication with the Container Ontology is directly implemented using the OWL Application Programming Interface (OWL API), which is detailed in the implementation part of this thesis (Section 5). The red and blue relationships are detailed as follows: There are seven relationships at all (five red relationships and two blue relationships) between the Container Management Application, the Container Description Service, the SemNOTAM Briefing Application, and the SemNOTAM Web Service. The two services for selecting a specific concept and for receiving the interest representation for the concepts of the Container Description Service are used by the Container Management Application and the SemNOTAM Briefing Application, therefore, this communication pattern is only described once as it works the same for each of those applications. This means that the following five different communication patterns shown in Figure 4.2 exist.

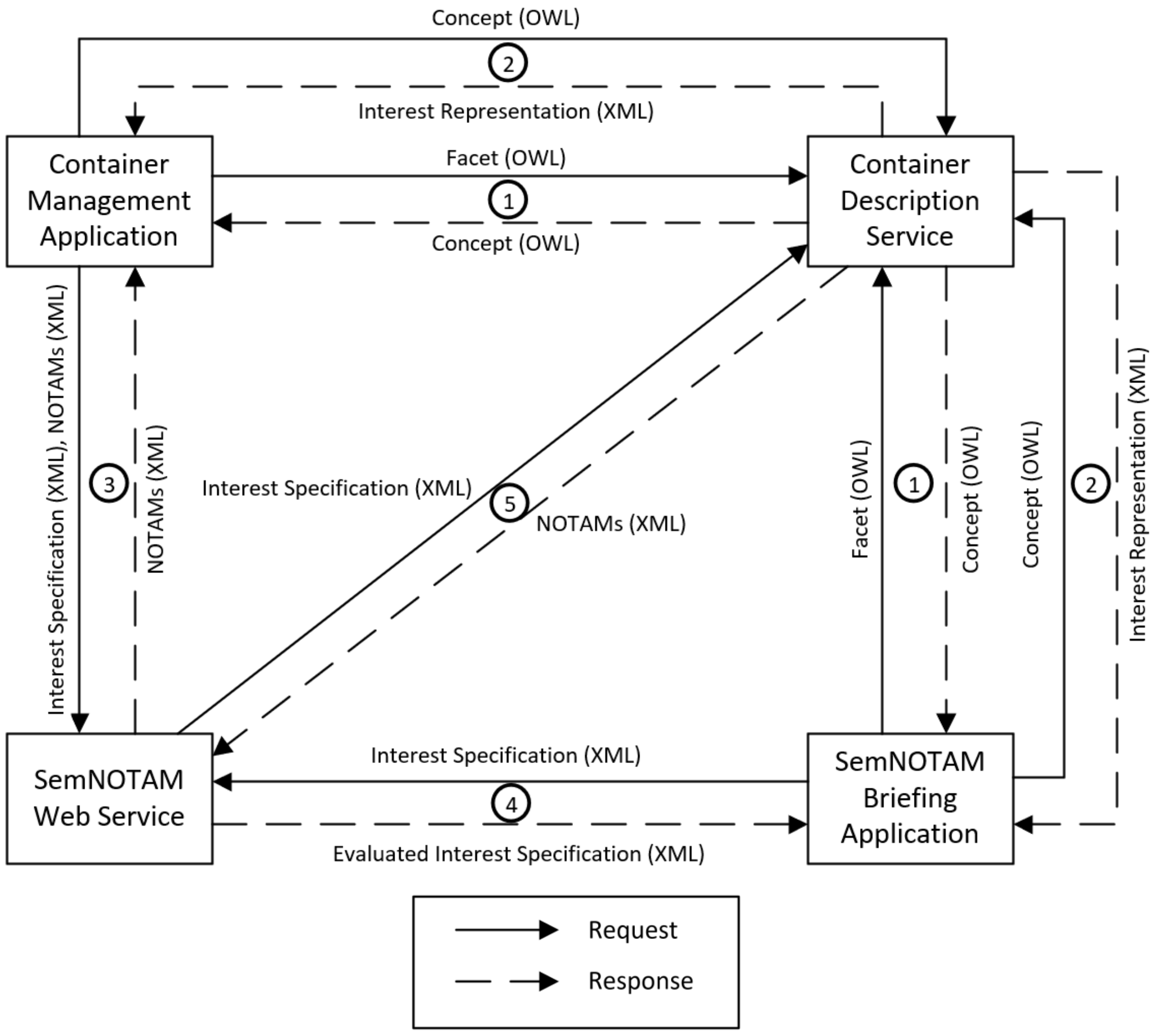

Concept (OWL)
2
Interest Representation (XML)
Container Management Application
Facet (OWL)
1
Concept (OWL)
Container Description Service
Interest Specification (XML), NOTAMs (XML)
3
NOTAMs (XML)
Interest Specification (XML)
5
NOTAMs (XML)
Facet (OWL)
1
Concept (OWL)
Concept (OWL)
2
Interest Representation (XML)
SemNOTAM Web Service
Interest Specification (XML)
4
Evaluated Interest Specification (XML)
SemNOTAM Briefing Application
Request
Response

Figure 4.2: Communication patterns between the systems

A communication pattern always consists of a request and a response between two systems. In Figure 4.2, the continuous arrows represent the requests, and the dashed arrows represent the corresponding responses. The different communication patterns are numbered in Figure 4.2, where communication pattern 1 and 2 occur between various components. As depicted, the communication between the Container Management Application and the Container Description Service follows the same communication pattern, i.e., the same sequence of request and response as the communication between the SemNOTAM Briefing Application and the Container Description Service (communication pattern 1 and 2 in Figure 4.2). This yields the following five communication patterns:

1. The first communication pattern is the call of the Container Description Service for receiving all possible concepts for one specific facet. Users of the Container Management Application invoke this service during the creation of a new data container, whereas users of the SemNOTAM Briefing Application invoke this service during the definition of their information need. The calling application, either the Container Management Application or the SemNOTAM Briefing Application, provides one specific facet in OWL representation as input and receives a list of the IRIs all possible concepts for this facet of the Container Description Service.

2. The Container Description Service further provides the functionality to transform a specific concept into its corresponding interest representation. This is used by the Container Management Application as well as the SemNOTAM Briefing Application. The Container Management Application or the SemNOTAM Briefing Application send the OWL representation of one concept to the Container Description Service, which returns the corresponding XML interest representation. Both applications need those interest representations to build their interest specification, which is in turn needed by the SemNOTAM Web Service to filter DNOTAMs.
3. The third communication pattern shows the use of the SemNOTAM Web Service by the Container Management Application for filtering DNOTAMs. The Container Management Application passes the interest specification and the DNOTAMs that shall be filtered as XML documents to the service. The SemNOTAM Web Service filters the set of DNOTAMs according to the interest specification and returns a new XML document including the set of relevant DNOTAMs. This service is invoked by the Container Management Application to receive the relevant DNOTAMs for a specific secondary data container.
4. The filtering service of the SemNOTAM Web Service is used by the SemNOTAM Briefing Application. The SemNOTAM Web Service filters a set of DNOTAMs (the selection of the set of DNOTAMs that is used for filtering is described in communication pattern 5) and returns the evaluated interest specification to the SemNOTAM Briefing Application. The SemNOTAM Briefing Application, therefore, forwards the created interest specification in XML format to the SemNOTAM Web Service and receives an evaluated interest specification in XML format containing all relevant DNOTAMs. An evaluated interest specification includes all the information of the provided interest specification, such as the geometric points of the flight route, and additionally all relevant and annotated DNOTAMs.
5. The last communication pattern described here regards the communication between the SemNOTAM Web Service and the Container Description Service. As described before, the SemNOTAM Web Service requires to find the most specific superset for the information need, i.e., XML interest specification, provided by the SemNOTAM Briefing Application. Therefore, the Container Description Service is invoked by the SemNOTAM Web Service by passing the XML interest specification. The Container Description Service isolates several parts of the interest specification to map them back to corresponding OWL concepts, to define a temporary data container. These parts are contained in the general interest specification and consist of interest representations of the previously selected OWL concepts. By using a reasoner, the most specific superset of the existing data containers is chosen, and its corresponding set of DNOTAMs in XML format is sent back to the SemNOTAM Web Service. This set contains all relevant DNOTAMs for the user's task. The temporary data container is deleted from the ontology afterwards, as this container is only used for the specific information need and no administrative metadata exits.

4.1.1 Container Ontology

As defined in the task description (Section 3.2), a vocabulary is required to provide concepts during the description of data containers. Each concept thereby represents specific characteristics which can be assigned to corresponding data containers. Each data container, however, represents a concept in that ontology. This allows to utilize reasoning capabilities of ontologies to derive hierarchies between the data containers. This is essential for the identification of the most specific superset for the Container Management Application as well as the retrieval of relevant

DNOTAMs based on a task description provided by the SemNOTAM Briefing Application. Typically, knowledge-based systems are built incrementally as not all the requirements can be specified at the beginning, and therefore, an iterative implementation with the use of prototypes is needed [43]. For ontologies, this methodology cannot be applied, as the ontology shall be shareable. Fernández et al. [43] defined a method of how to build ontologies, called the Methontology. It is based on the experience from modeling an ontology of chemicals and consists of the following seven phases:

1. **Specification:** In the first phase, a document that specifies the purpose of the ontology should be written in natural language. The level of formality and the scope shall be included, which means the set of terms, the characteristics, and the granularity that will be represented by the ontology.
2. **Knowledge Acquisition:** Most of the knowledge acquisition is already done in the specification phase. However, the purpose of the ontology has to be clearly defined throughout this phase. Knowledge Acquisition is usually performed by brainstorming and conducting interviews. The aim of this phase, on the one hand, is to understand what the purpose of the ontology is and on the other hand, gather knowledge about what shall be included in the ontology.
3. **Conceptualization:** This phase includes the creation of the conceptual model, which corresponds to the definition of the Glossary of Terms. This comprises of the definition and grouping of the needed classes, instances, verbs, and properties. Furthermore, these concepts need to be set into relation to each other to define the modeling possibilities that are needed by the ontology.
4. **Integration:** In this phase, existing ontologies shall be reviewed. Existing ontologies should be reused only if the concepts are coherent with the identified concepts during the conceptualization phase.
5. **Implementation:** For implementing the ontology, an ontology environment shall be chosen, which meets the already defined requirements.
6. **Evaluation:** After completing the ontology implementation, the whole ontology needs to be evaluated, to ensure the ontology is correct, and the goal is reached. Depending on whether any concepts are missing or not clearly defined, or the overall task cannot be fulfilled by the current state of the ontology, the process needs another cycle of the previous phases.
7. **Documentation:** The last phase in the Methontology consists of the documentation of the ontology. The documentation should rather be an ongoing process during the whole development.

For the development of the ontology in this thesis, not all the phases are relevant, as, for example, the ontology shall be developed independently and therefore, no existing ontologies shall be used. Furthermore, apart from annotations directly defined for the concept in the ontology, there will be no additional documentation as the ontology as well as its usage are described in this thesis. The specification of the purpose is documented in the task description (Section 3.2.2), which already required knowledge acquisition. The third phase is partly covered in the following subsections, where the main structure of the ontology is described. The next three sections describe the approach followed to build the ontology including the concept of data containers and their metadata. The implementation of the concepts is detailed in Section 5.2, which covers the whole implementation phase, including facets, data containers, and the discovery of subsumption hierarchies through reasoning.

4.1.1.1 Data Container

The main concept in the ontology is the data container, which represents a set of DNOTAMs, that are characterized by multiple facets. As subsumption hierarchies between data containers have to be derived by a reasoner (Requirement 1.1.3), the data containers need to be modeled as classes (Requirement 1.1.1). Some information, however, shall not influence the reasoning, e.g., administrative metadata such as the size of data items contained in the data container or the refresh interval. Therefore, data containers need to be modeled as individuals additionally to allow the assignment of the administrative metadata (Requirement 1.1.1). The definition of one concept as class and individual with the same IRI is called punning [44]. This clearly indicates that both, the class and the individual, represent the same concept. A data container consists of descriptive metadata, which describes the data items contained by the data container and administrative metadata, which, amongst others, describes the quality of the data container. The connection between the data container and its descriptive and administrative metadata is modeled either by using object properties or data properties (Requirement 1.1.2).

4.1.1.2 Descriptive Metadata

Descriptive metadata describes the content of the data containers, i.e., the data items. The concepts are defined as classes (Requirement 1.2.1) and are modeled as subconcepts of so-called facets, that are classes themselves (Requirement 1.2.2). A facet is, for example, the temporal restriction of the data container and a corresponding descriptive metadata concept would be the *Year_2017*. Either, the concepts in the facets are explicitly hierarchized, or the reasoner can infer the hierarchy based on the description of the concepts (Requirement 1.2.4). The facet for describing the spatial or temporal filter level needs to be specified explicitly since there is no property for inferring these hierarchies. For example, it needs to be explicitly defined that the spatial filter level *Shape* is more precise than the filter based on the *BoundingBox*, which corresponds to a rectangle from the top left point to the bottom right point. The inferring of such a hierarchy is enabled through descriptions of the concepts via data properties, e.g., the wingspan or weight of an airplane. A data container needs to be described by always one concept of each facet (Requirement 1.2.3). To assign the descriptive metadata concepts to a data container, object properties are needed.

4.1.1.3 Administrative Metadata

In addition to the descriptive metadata, each data container is characterized by administrative metadata. Since administrative metadata is not used for reasoning, it needs to be assigned to the corresponding data container individuals. For assigning the administrative metadata, data properties are used (Requirement 1.3.1), except for the data source, which requires an object property referencing the data container concept that is used as superset (Requirement 1.3.2). The administrative metadata comprises, amongst others, several timestamps that enable to conclude the freshness of the data (Requirement 1.3.3). These timestamps, furthermore, allow to make decisions for the most specific superset, as they also define the quality of the data set. For example, let us assume we are looking for relevant DNOTAMs for a flight from Linz to Salzburg on the 31st of January 2017 and two data containers exist. The first one contains all DNOTAMs of Austria in January 2017, whereas the second data container contains all DNOTAMs of Upper Austria and Salzburg in the year 2017. Simply by using the subsumption reasoning of the spatial and temporal restrictions, it is not clear which data container shall be used. The reasoner would return both data containers as direct supersets as none of them is the superset of the other. Let us further assume that the first data container has last been updated on January the first and the second data container has been updated each day. Considering these timestamps, the second data container shall be used as the first one is not as fresh as the second and therefore might miss relevant DNOTAMs.

The conceptualization of the needed classes, individuals, object properties, and data properties is done in Section 5.2, as well as the implementation and evaluation phase. As described before, the integration phase of the Methontology is not in the scope of this thesis.

4.1.2 Container Management Application

The Container Management Application will be developed as a web application. To simplify the development of web applications, it is recommended to follow design patterns [45]. Design patterns are abstract templates that can be applied for several problems, especially the communication of the application parts. The most common design pattern for web applications is the Model-View-Controller (MVC) pattern, which distinguishes the three different component classes, the model, which stores and manages the data, the view, which creates the graphical interface, and the controller, which comprises the business logic [46].

The Java framework Vaadin [47], used for the SemNOTAM Briefing Application, allows to develop a web application similar to a desktop application. To avoid the introduction of new design paradigms and to foster reuse of the design, the Vaadin framework is also used for the implementation of the Container Management Application. It is typical for applications created in the Vaadin framework to use a similar design pattern as the MVC pattern, the so-called Model-View-Presenter (MVP) [48]. The third component, the presenter, is responsible for presenting the view throughout an interface and connecting it with the data provided by the model. This means that the model is only for the data perspective, and the view is only responsible for the visualization independent of the data. The presenter is the connection between the model and the view to present the model's data in the view's interface. There is no direct communication between the model and the view anymore. However, the Container Management Application has a higher focus on the business logic, and therefore, the original MVC design pattern is used. The concrete organization of those three components and its implementation is described in more detail in Section 5.3.

4.1.3 Container Description Service

The Container Description Service is designed as a web service which provides three different functions. The Container Description Service needs to enable the access to the concepts of a facet, to their interest representation and must be able to find the most specific superset for the given information need. The decision for the most specific superset is primarily based on the subsumption hierarchy, and secondarily based on the administrative metadata. Therefore, the Container Description Service needs to have access to the Container Ontology and should act as a middleware between the Container Management Application or the SemNOTAM Briefing Application and the Container Ontology.

4.2 Architecture

This section introduces the architecture of the client application represented by the Container Management Application, and the server application, which consists of the Container Description Service and the SemNOTAM Web Service. The visualization of the architecture is based on the modeling language ArchiMate [49], which was specified by the Open Group for modeling enterprise architectures. In this section, parts of the whole architecture model, especially the most detailed ones, are shown and described, while a general view of the complete model is provided in the Appendix.

4.2.1 ArchiMate

The Open Group specified the modeling language ArchiMate to enable users to model complex structures of enterprise architectures [50]. Therefore, ArchiMate separates the model into different layers, the business layer, the application layer, and the technology layer [51].

1. The business layer contains the business services that are provided to customers. The customers only know which services are available but do not know the underlying business processes and functions.
2. The application layer describes the different applications that are used to fulfill the business processes.
3. In the technology layer, the services for storing and processing information are depicted. Additionally, the communication and system software that is needed to realize the application are modeled.

Figure 4.3 shows the connection between those three layers, where the yellow elements represent the business layer, the blue elements represent the application layer, and the green elements belong to the technology layer.

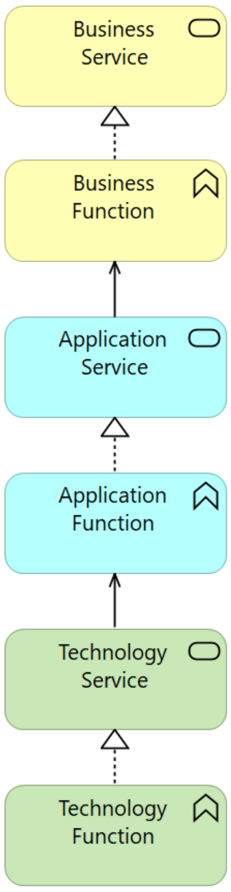


Figure 4.3: Service-oriented structure in ArchiMate [51]

Figure 4.3 exemplary shows that each layer consists of a service and a function. The function realizes the service of the same layer, whereas, a function at the next higher layer can use a service of an underlying layer, i.e., the used service serves this function [51]. For example, the Technology Function realizes the functionality offered by the Technology Service, and the Application Function can use this service. The Business Function can only access services of the application layer, but not of the technology layer.

The main elements of ArchiMate are services, functions, and processes [52]. As described above, typically functions or processes realize a service that either is provided to the customer, which is the user of the application or serve the functions or processes in the next layer [51]. Processes in ArchiMate represent a collection or a sequence of functions depending on if the functions in the process are linked with process flow or not. Figure 4.4 shows the symbols of those three elements that are used in ArchiMate.

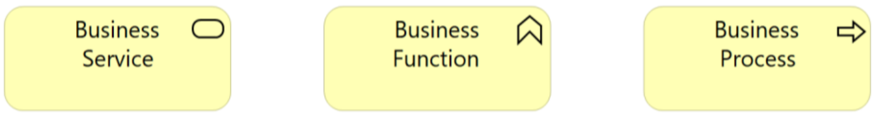


Figure 4.4: ArchiMate Elements – Service, Function, Process [52]

The color of the element represents the corresponding layer, which is, in this case, the business layer. In the following sections, other elements are introduced when they are used in the architecture model.

4.2.2 Client-Application

The Container Management Application is the client application that is implemented as part of this thesis. It enables the management of data containers in the SWIM environment. The next sections show its architecture divided into the three parts, technology layer, application layer, and business layer.

4.2.2.1 Technology Layer

The technology layer of the client application defines the technical set-up, and needs to model all the services that are provided to the next layer, the application layer [51]. To describe the technical set-up, five additional ArchiMate elements of the technology layer need to be introduced, which are shown in Figure 4.5.

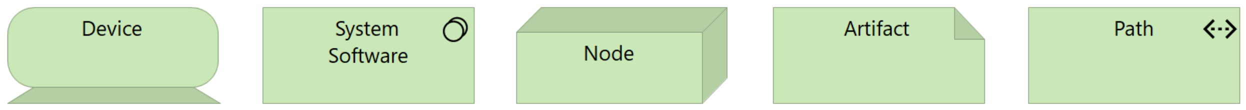


Figure 4.5: ArchiMate Elements – Device, System Software, Node, Artifact, Path [52]

The five elements in Figure 4.5 have the following meaning [52]:

- A device can represent any device, e.g., a personal computer.
- The system software can be used inside of a device, to show that the device has this software installed.
- A node can be used for itself to represent any component that is not described in more detail. Inside of a device, it can be used to depict that a server is running.
- Any file is represented by an artifact. By modeling some elements inside of the artifact, it means that the artifact consists of them or uses them.
- The path element can be used for the description of any communication path between elements in the technology layer.

Figure 4.6 visualizes the technical infrastructure and provided services of the client application. Apart from the newly introduced elements, also the services of the technology layer that enable the connection to the application layer are used.

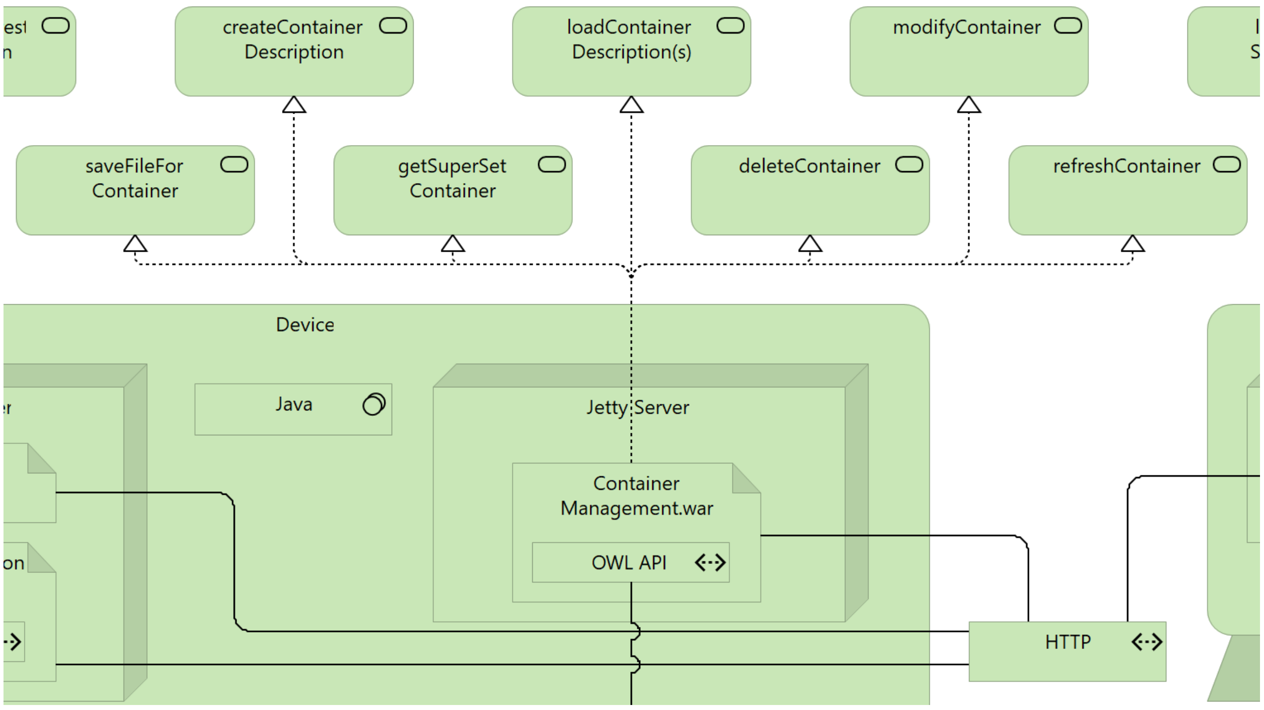


Figure 4.6: Client Application – Part of the Technology Layer

In Figure 4.6, the Container Management Application is modeled as a WAR file artifact. This artifact uses the OWL API for the communication with the Container Ontology and the Hypertext Transfer Protocol (HTTP) to communicate with the SemNOTAM Web Application and the SemNOTAM Briefing Application. This artifact is modeled inside of the Jetty Server node, which represents that this WAR file is running on this server. The server node is modeled inside of the device where it runs, and on this device, there is the system software Java running. Additional to the technical set-up, Figure 4.6 also shows which services are realized by the Container Management Application. All the provided services serve an element in the application layer, which is described in the next section.

4.2.2.2 Application Layer

In the application layer, all applications that are needed for the business purpose are modeled. In some cases, only the connection between the technology layer and the business layer needs to be enabled with a corresponding function and service. In other cases, the application layer may contain some logic or processes that are not visible for the user and therefore, shall be modeled in this layer. For the application layer, one additional element needs to be introduced, shown in Figure 4.7.

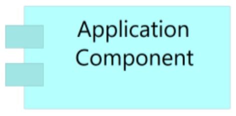


Figure 4.7: ArchiMate Elements – Application Component [52]

This element represents a component of the overall application or can also represent the whole application. As already described in Section 4.2.1, a service of the technology layer can serve a function or process in the application layer. However, to simplify the model, the services can be merged into one application component, and then the component is connected to the functions and processes [52]. A part of the application layer for the Container Management Application is depicted in Figure 4.8.

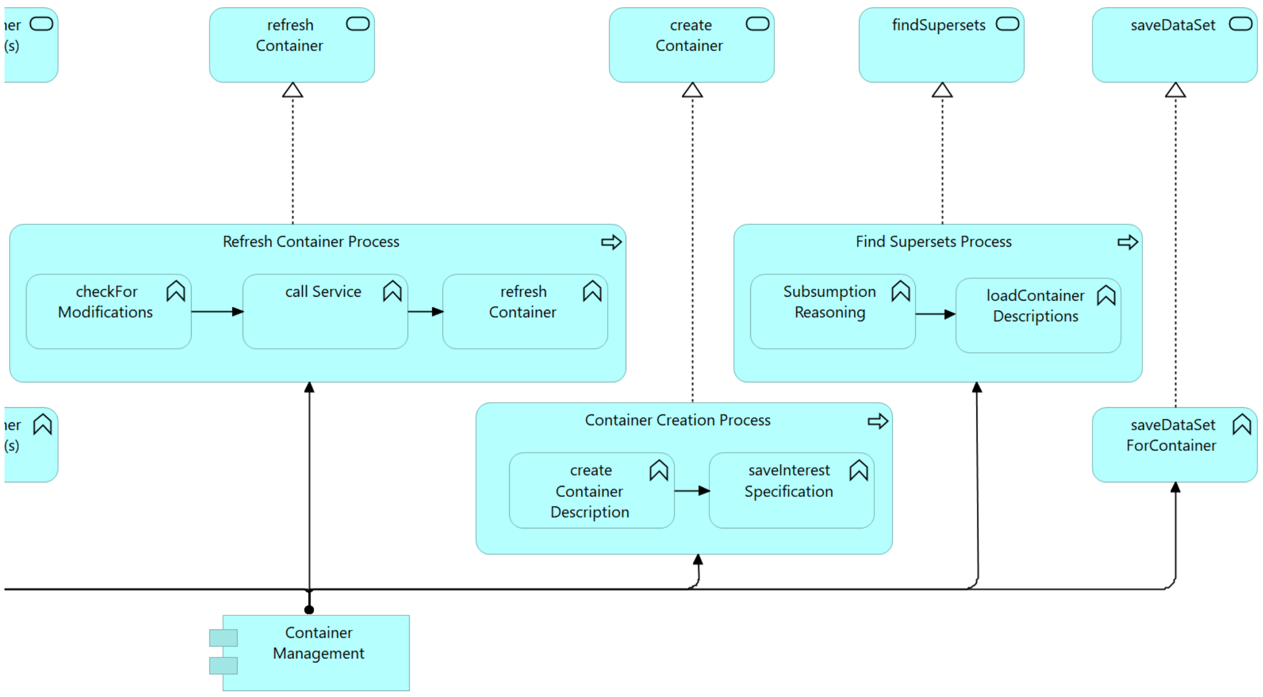


Figure 4.8: Client Application – Part of the Application Layer

In the bottom of Figure 4.8, the Container Management is modeled as an application component to merge all the services of the technology layer. This component is then used by all the functions or processes. The simplest part of the application layer are those services that are passed from the technology layer to the business layer by adding an application function and an application service. For the saving of a data set, this is shown in Figure 4.8. For modifying or deleting a container and loading the container descriptions this works the same, and therefore, it is not shown separately. The more complex parts, such as refreshing and creating a container as well as finding the superset, are modeled as processes. In the *Refresh Container Process*, firstly it needs to be checked if there are modifications of the data set of the data source. Afterwards, a service needs to be called, and finally, the container needs to be refreshed. The *Container Creation Process* consists of two functions, first creating the container description and second saving the corresponding interest specification. In the *Find Supersets Process* first the subsumption reasoning is performed and then the container descriptions of the superset data containers are loaded. Each of these processes are realized by a corresponding service, that again enables the connection to the business layer.

4.2.2.3 Business Layer

The business layer is used to model the business logic of the whole application. For the business layer, two new elements need to be introduced as this layer also describes the user of the application, shown in Figure 4.9.

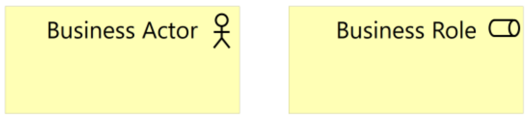
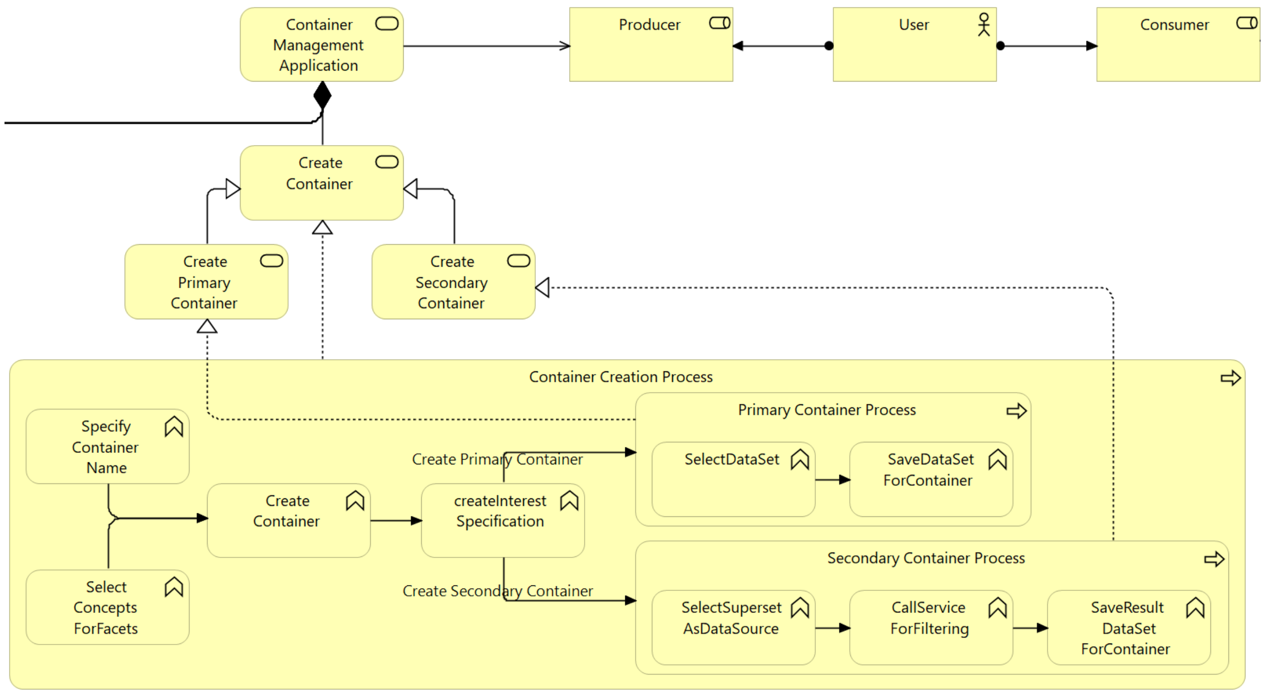


Figure 4.9: ArchiMate Elements – Business Actor, Business Role [52]

The business actor describes a person or an organization that somehow interacts with the system, and the business role is used to describe the role a business actor can have [52]. One business actor can have several roles, depending on which services the actor uses. As this layer contains the whole business logic, only a part of it is shown in Figure 4.10.



Container Management Application
Producer
User
Consumer
Create Container
Create Primary Container
Create Secondary Container
Container Creation Process
Primary Container Process
Specify Container Name
Create Primary Container
SelectDataSet
SaveDataSet ForContainer
Create Container
createInterest Specification
Secondary Container Process
Select Concepts ForFacets
Create Secondary Container
SelectSuperset AsDataSource
CallService ForFiltering
SaveResult DataSet ForContainer

Figure 4.10: Client Application – Part of the Business Layer

In the middle top of Figure 4.10, the user of the application is modeled as a business actor. Two roles are connected to this user, the business role producer, and the business role consumer. For the Container Management Application, only the producer is relevant, but as this ArchiMate model also contains the architecture of the SemNOTAM Briefing Application (see Figure 4.11), and one user can be a producer and a consumer, the consumer role is also modeled. The Container Management is modeled as a service that consists of all the other services below, e.g., the container creation service. As the container creation service is the most complex one, Figure 4.10 only shows this part of the business layer. Services in ArchiMate can be modeled as specializations of another service; for example, the service *Create Secondary Container* is a specialization of the service *Create Container*. The process *Container Creation Process* depicts the whole creation process, whereas the processes *Primary Container Process* and *Secondary Container Process* show the specific process depending on which container shall be created. ArchiMate allows to model triggers that are shown on arrows. After the function *createInterestSpecification* depending on which container shall be created, different flows will be taken. The functions inside of the processes are at the end connected to the corresponding application service; for example, the function *SelectSupersetAsDataSource* uses the application service *findSupersets*.

The relevant part of the business layer of the SemNOTAM Briefing Application, the part concerning the evaluation of interest specifications, is shown in Figure 4.11. As mentioned before, the users of this application act as a consumer in the SWIM environment.

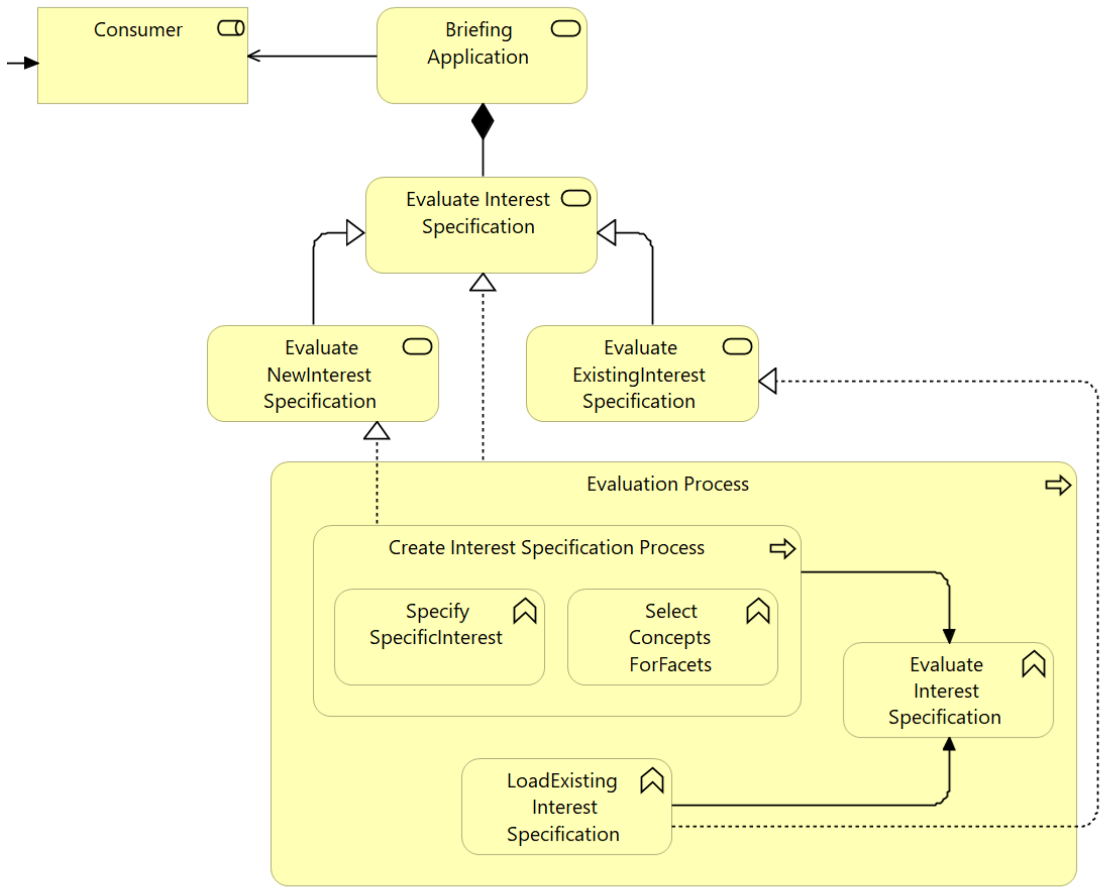

Consumer
Briefing Application
Evaluate Interest Specification
Evaluate NewInterest Specification
Evaluate ExistingInterest Specification
Evaluation Process
Create Interest Specification Process
Specify SpecificInterest
Select Concepts ForFacets
Evaluate Interest Specification
LoadExisting Interest Specification

Figure 4.11: Client Application – Part of the Business Layer (Briefing Application)

The evaluation process in Figure 4.11 shows that it is either possible to specify a new interest specification or to load an existing interest specification. For the creation of a new interest specification, the services of the Container Description Service for selecting a concept for each facet is used. Furthermore, when the SemNOTAM Briefing Application forwards the interest specification to the SemNOTAM Web Service, this service calls another service of the Container Description Service to find the most specific superset in the existing data containers for the evaluation of the given interest specification.

4.2.3 Server-Application

This thesis also covers the implementation of the Container Description Service, which is one of the server applications for the Container Management Application. The second server application is the SemNOTAM Web Service, which is reused as it was already developed as part of the SemNOTAM project and provides the functionality to filter and annotate DNOTAMs. The next three sections show the architecture of these two services, again divided into the three parts, technology layer, application layer, and business layer.

4.2.3.1 Technology Layer

The technology layer describes the technical set-up and infrastructure of the server applications. This set-up and the provided services of the server applications are visualized in Figure 4.12.

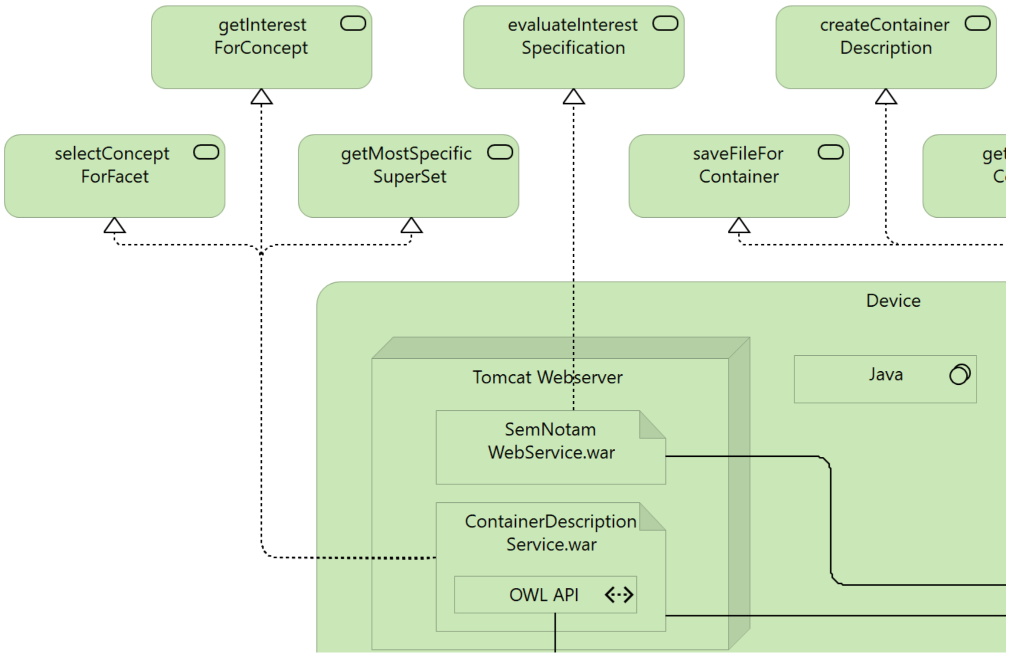


Figure 4.12: Server Application – Part of the Technology Layer

Figure 4.12 shows that the SemNOTAM Web Service and the Container Description Service are modeled as WAR file artifacts, that are running inside of a web server, in this case, a Tomcat web server. As for the client application, the server is running on a device with the system software Java. The Container Description Service artifact includes the OWL API, which is needed for the communication with the Container Ontology. The technology services provided by the Container Description Service are *selectConceptForFacet*, *getInterestForConcept* and *getMostSpecificSuperSet*, and the SemNOTAM Web Service has one relevant service, the *evaluateInterestSpecification*. The services are then referenced by the application layer, which is described in the next section.

4.2.3.2 Application Layer

The application layer is used to model all the applications that are needed for the business purpose. Therefore, again, application services and processes are modeled in this layer. Sometimes only a function and a service are modeled to enable the connection of the technology layer and the business layer. In Figure 4.13, the application layer for the Container Description Service is modeled like this, as the exact implementation of these services is not relevant in the architecture model.

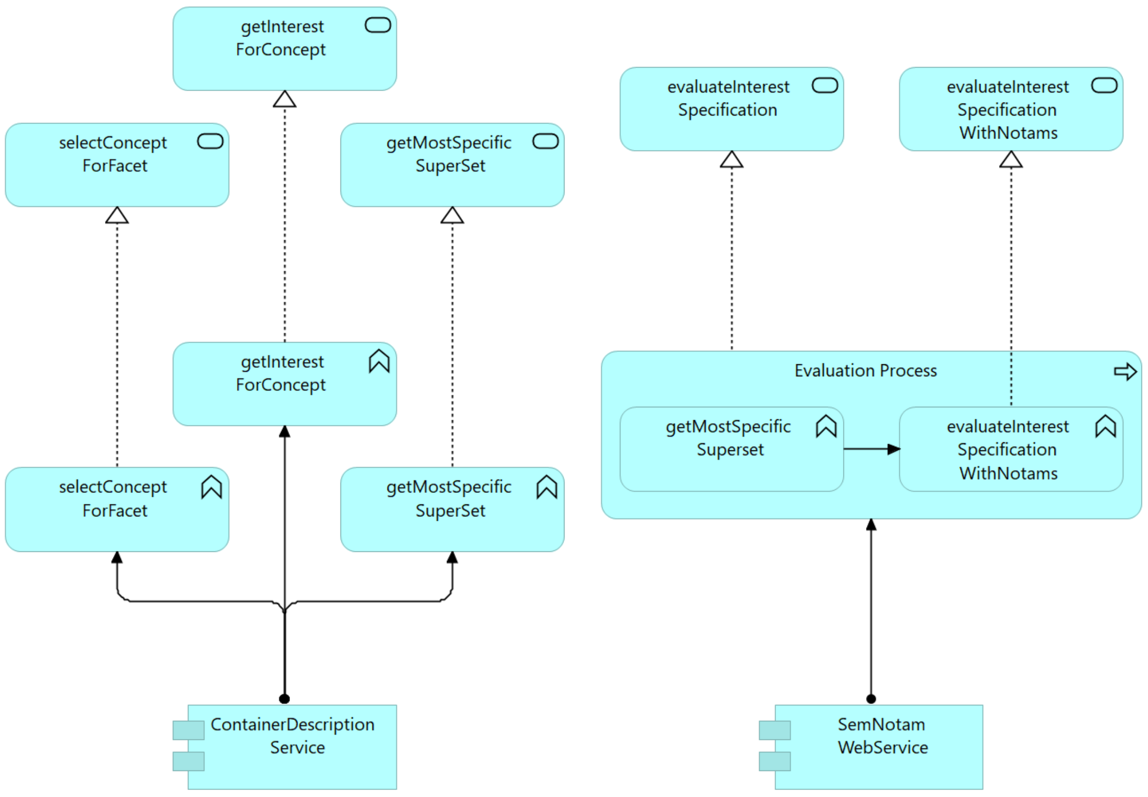


Figure 4.13: Server Application – Part of the Application Layer

The right side of Figure 4.13 shows the application layer of the SemNOTAM Web Service, which in contrast to the Container Description Service has a process defined. On the technology layer, only one service for evaluating an interest specification was modeled, but on the application layer, we need to distinguish between the two services *evaluateInterestSpecification* and *evaluateInterestSpecificationWithNotams*. When the set of DNOTAMs for filtering is given by the business layer, the service *evaluateInterestSpecificationWithNotams* is used, and the interest specification can be directly evaluated. Whereas, when no set of DNOTAMs is given, the evaluation process starts with finding the most specific superset, which is modeled in Figure 4.13 with the function *getMostSpecificSuperset*. This function directly refers to the service *getMostSpecificSuperSet* of the Container Description Service.

4.2.3.3 Business Layer

For the server applications, it is not useful to have a business layer as they are not supposed to be directly accessed by a user. The provided application services are used by the business functions of the client application, which can be seen in the Appendix.

5 Implementation

The implementation of the defined requirements (Section 3.3) and the resulting architecture (Section 4.2) is described in this section. First, an overview of the technologies, which are used for the ontology, the client application, and the server application, is given. Section 5.2 covers the implementation of the Container Ontology. The Container Management Application, which is the client application, is described in Section 5.3. The last section, Section 5.4, documents the implementation of the server application, respectively the Container Description Service.

5.1 Technologies

This section gives an overview of the technologies that are used for the implementation. The Web Ontology Language (Section 5.1.1) is used for the realization of the Container Ontology and the tool Protégé (Section 5.1.2) is used for its creation and administration. For the Container Management Application and the Container Description Service, the programming language Java is used (Section 5.1.3). Furthermore, the Container Management Application uses the framework Vaadin (Section 5.1.4), whereas the Container Description Service uses the web service framework Apache CXF (Section 5.1.5).

5.1.1 OWL

The Web Ontology Language (OWL) is a language to describe ontologies and was specified by the World Wide Web Consortium (W3C) [53]. Technically, OWL is based on the Resource Description Framework (RDF), which is one of the main components of the Semantic Web. However, OWL has a wider range of language constructs and therefore allows to represent rich and complex knowledge about things, groups of things, and relations between things.
For the implementation of the Container Ontology OWL is used to define classes, object properties, data properties, and instances. As described in Section 4.1.1, the language constructs are used in different ways, e.g., for the definition of facets and DNOTAM characteristics, as well as for the definition of data containers. To guarantee an easy and comfortable way of managing the ontology, different tools for the ontology definition exist, for example, Protégé.

5.1.2 Protégé

Protégé is the most widely used ontology editing environment in the world. It provides a Desktop and a Web Client for the development of complex ontologies [54]. Furthermore, this tool includes visualization tools, several plug-ins, and an Application Programming Interface (API) that allows to access and alter an ontology from other systems.
As mentioned in Section 4.2, the Container Management Application and the Container Description Service use this API to read from and write to the Container Ontology. In Section 5.2, the interface of Protégé is described in more detail, where the development of the Container Ontology is documented.

5.1.3 Java

Java is one of the most popular programming languages that is used for the development of applications and services [55]. It enables to provide applications over heterogenous systems; hence, the communication and cooperation of the end users can be increased.
The existing systems, respectively SemNOTAM Web Service and SemNOTAM Briefing Application, also use Java as a programming language. Since OWL provides a Java API to access the ontology, the Container Management Application and the Container Description Service are implemented with the programming language Java in version 8. For the implementation of the Container Management Application, additionally, a framework for developing Java web applications is used, which is described in the next section. The SemNOTAM Web Service and the Container Description Service use CXF for the implementation of the services, which is introduced in Section 5.1.5

5.1.4 Vaadin

Vaadin is a framework for the development of Java web applications, which allows to implement powerful web applications without the need of knowledge of additional web technologies, like Hypertext Markup Language (HTML) or JavaScript [56]. The advantage of the usage of Vaadin is the simplicity and maintainability by having simple Java code, that is transformed into HTML and JavaScript code during runtime [57]. This allows to develop a web application in the same way as a desktop application, without worrying about web technologies. Furthermore, themes can be used in Vaadin to have a similar styling of all the web pages in the application with no Cascading Style Sheet (CSS) knowledge needed [58]. However, for additional style options, a custom CSS code can be added to the theme. The SemNOTAM Briefing Application was also developed with the Vaadin framework. To ensure a consistent look and feel, the same Vaadin framework and theme is also used for the development of the Container Management Application.

5.1.5 Apache CXF/JAX-WS

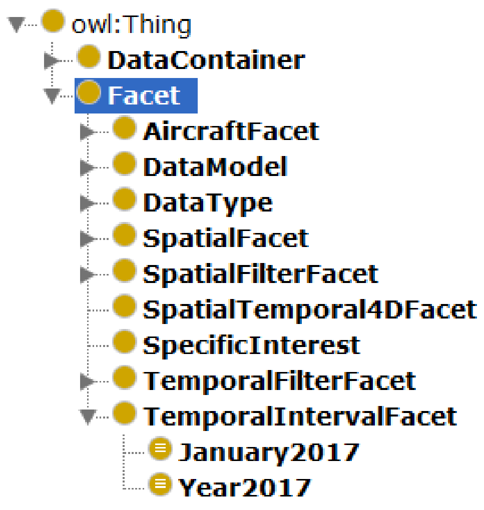
Apache CXF is an open source services framework that helps to develop services with frontend programming APIs [59]. One of these APIs is Java API for XML Web Services (JAX-WS), that allows to develop web services with Java [60] easily. The calls and responses of the web services are coded in XML and are communicated as Simple Object Access Protocol (SOAP) messages over HTTP [61]. The development of the Container Description Service with these technologies is described in Section 5.4.

5.2 Container Ontology

The implementation of the ontology is divided into three parts. The first part (Section 5.2.1) describes how the facets are represented in the ontology. This also includes modeling the hierarchy between the concepts of the different facets. Afterwards, Section 5.2.2 covers the realization of the data containers, which consists of their representation as classes and individuals, and the definition of the descriptive and administrative metadata. Finally, data discovery is explained in Section 5.2.3, which enables the creation of a hierarchy between data containers.

5.2.1 Facets

This section describes the created facets and their concepts, as well as the data properties that are used to describe these concepts. As defined in Requirement 1.2.1 each feature of the descriptive metadata of a data container shall be realized through concepts and facets. Therefore, each facet and concept is modeled as OWL class, as shown in Figure 5.1.



```
owl:Thing
    DataContainer
    Facet
        AircraftFacet
        DataModel
        DataType
        SpatialFacet
        SpatialFilterFacet
        SpatialTemporal4DFacet
        SpecificInterest
        TemporalFilterFacet
        TemporalIntervalFacet
            January2017
            Year2017
```

Figure 5.1: Container Ontology – Facets

The main OWL classes *DataContainer* and *Facet* are used for the distinction between the data containers and the facets that are used for the description of the data containers. Figure 5.1 shows the full list of facets. The *AircraftFacet* is used to define for which type of aircraft the DNOTAMs of a data container are relevant, for example, for aircraft of the type A380. The facets *DataModel* and *DataType* describe the type of data items and the according data model. In this thesis data containers are used for data items of the type DNOTAM and the data model AIXM, but the ontology can also be used for other data items, for example, the Weather Information Exchange Models and Schema (WXXM) that is used for meteorological information.

The spatial metadata of a data container can be defined by several facets. The *SpatialFacet* describes the geographical shape defining the relevance of the data items, whereas the *SpatialFilterFacet* defines the granularity of the spatial relevance. This facet has exactly three concepts, *SpatialNone*, *BoundingBox*, and *GmlShape*. *SpatialNone* means that there is no geographical filter applied to the data items of this data container. When *BoundingBox* is used, a rectangle from the top left corner and the bottom right corner is used for filtering, whereas *GmlShape* filters the data items based on the exact shape.

For temporal information also two facets, one for the granularity and one for the temporal relevance, exist. The *TemporalIntervalFacet* defines the interval during which the data items of the data container are relevant. The TemporalFilterFacet again has three concepts, *ActiveTime*, *ValidTime*, and *TemporalNone*. *TemporalNone*, similar to the spatial filter *SpatialNone*, means that no temporal filter is applied to the data items. The difference between *ValidTime* and *ActiveTime* is that each DNOTAM has a time period in which it is valid and can have a more precise definition of when the DNOTAM is actually activated. For example, a DNOTAM concerning a runway closure due to construction work can be valid for several weeks, which would be used for the filtering based on *ValidTime*. But let us assume that the construction work is done each week from Monday to Wednesday, which means that from Thursday to Sunday this DNOTAM is not active and therefore not relevant for flights on these days. This can be filtered by the use of the *ActiveTime* concept. The spatial and temporal relevance can also be specified with the *SpatialTemporal4DFacet* instead of the *SpatialFacet* and the *TemporalIntervalFacet*. Concepts of this facet contain both temporal and spatial information about the relevance of the data items. Furthermore, this facet allows to specify a third spatial dimension, the flight height. The last facet is the *SpecificInterest*, which can contain a whole flight plan or some other specific information. This facet is primarily used for data containers that are built for specific tasks, e.g., specific flights, or broadly specified flight plans which can be used for several flights.

Requirement 1.2.2 states that each concept has to be classified by one of the described facets. Therefore, the concepts that belong to one facet are modeled as direct or indirect subconcepts of this facet. A direct subconcept is modeled as a concept one hierarchy level after the facet itself, whereas an indirect subconcept is a subconcept of another concept that can again be a direct or an indirect subconcept of the facet. Furthermore, the concepts in one facet can be organized hierarchically to allow subsumption between the data containers (Requirement 1.2.4). The hierarchical structure can either be modeled explicitly by the definition of subconcepts, as shown in Figure 5.2, or can be described through data properties used in equivalent axioms, like in Figure 5.3 and Figure 5.4.

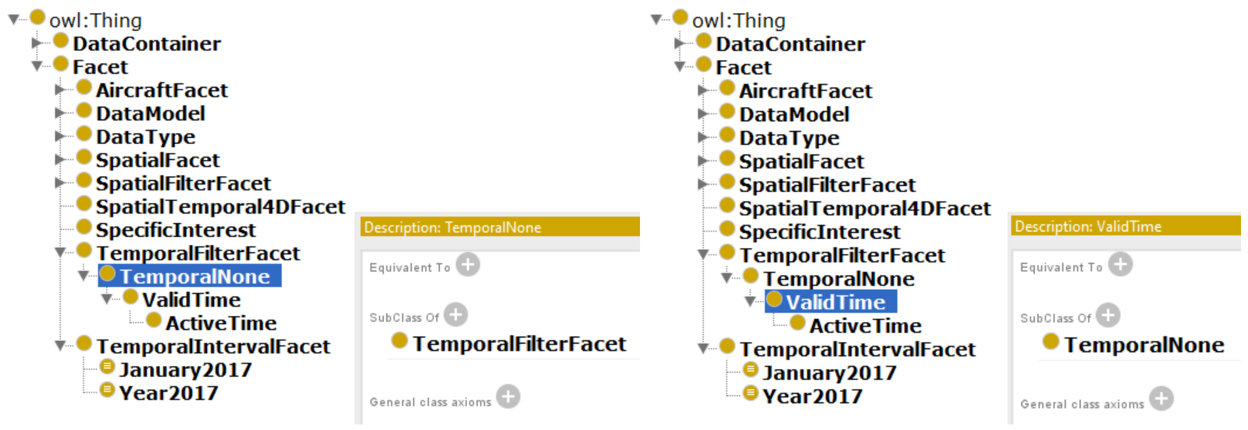
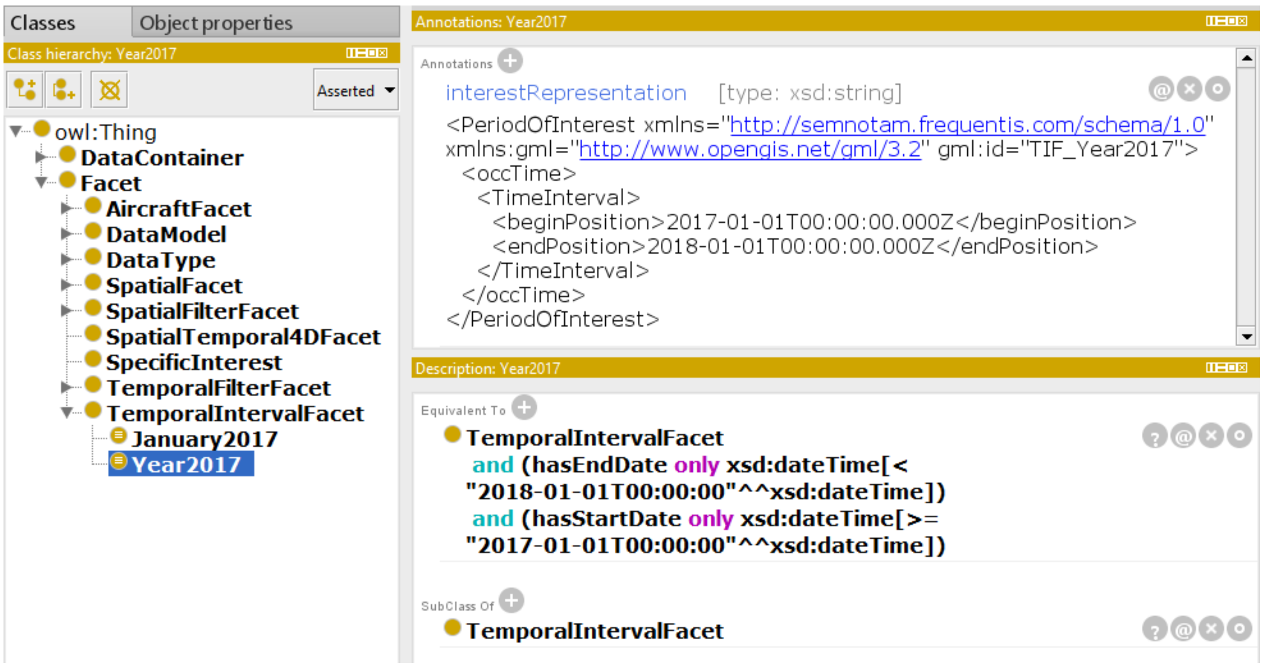


Figure 5.2: Container Ontology – Subconcepts

The left part of Figure 5.2 shows the direct definition of a concept to be the subconcept, i.e., the subclass of the facet it belongs to, in this case, that the *TemporalNone* concept is a subconcept of the facet *TemporalFilterFacet*. The right side shows the explicit definition of the hierarchy between the concepts in one facet. As described before, the concept *ValidTime* is more precise than the concept *TemporalNone* and therefore, is modeled as subconcept. The hierarchy view shows that *ValidTime* consequently is also a subconcept of the *TemporalFilterFacet*.



interestRepresentation [type: xsd:string]

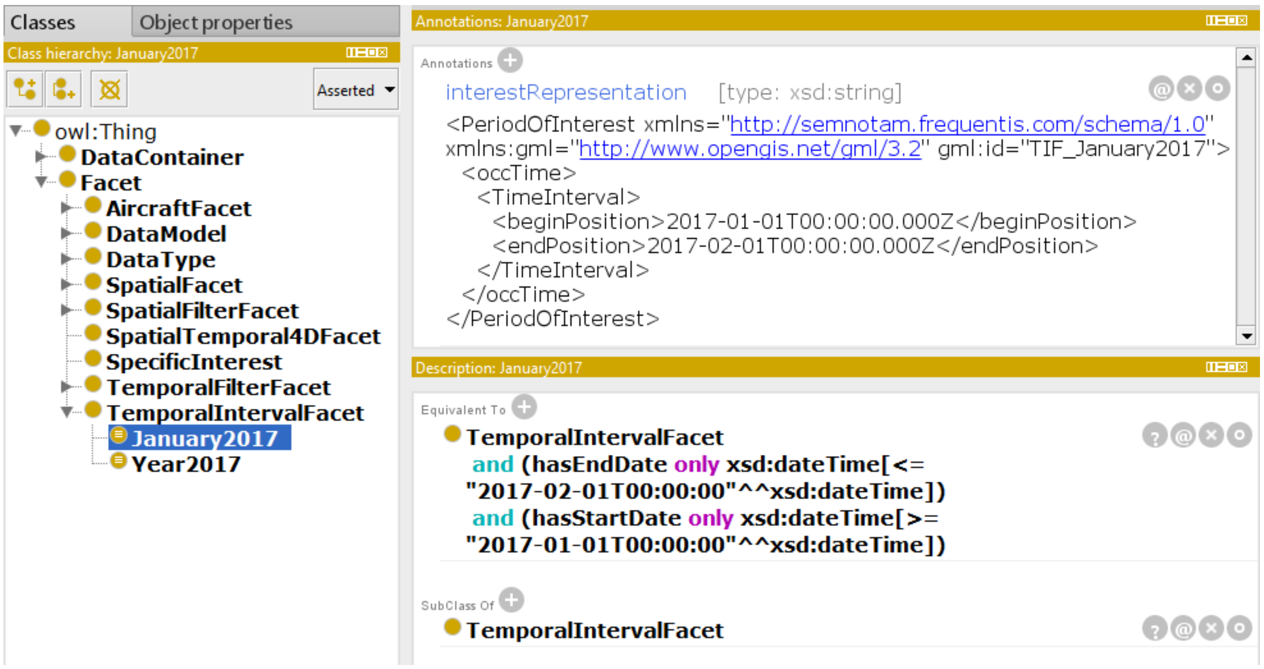
```
<PeriodOfInterest xmlns="http://semnotam.frequentis.com/schema/1.0"
xmlns:gml="http://www.opengis.net/gml/3.2" gml:id="TIF_Year2017">
  <occTime>
    <TimeInterval>
      <beginPosition>2017-01-01T00:00:00.000Z</beginPosition>
      <endPosition>2018-01-01T00:00:00.000Z</endPosition>
    </TimeInterval>
  </occTime>
</PeriodOfInterest>
```

```
TemporalIntervalFacet
 and (hasEndDate only xsd:dateTime[<
"2018-01-01T00:00:00"^^xsd:dateTime])
 and (hasStartDate only xsd:dateTime[>=
"2017-01-01T00:00:00"^^xsd:dateTime])
```

SubClass Of: TemporalIntervalFacet

Figure 5.3: Container Ontology – Year2017

Concepts of facets can also be described with data properties to specify the characteristics of the belonging data items. In the “Equivalent To” section of Figure 5.3, the concept *Year2017* is described with the data properties *hasStartDate* and *hasEndDate*, to define the corresponding time interval, in this case from the first January in 2017 to the first January in 2018. For these data properties, the data type *dateTime* was specified, which requires to set a *dateTime* when using them. As described in Section 2.1.5, the descriptions of a concept can be used by the reasoner to infer further knowledge, whereas the annotated interest representation is only used for building the whole interest specification. In Figure 5.4, another concept of the facet *TemporalIntervalFacet* is shown, which is also described with the data properties *hasStartDate* and *hasEndDate*.



interestRepresentation [type: xsd:string]

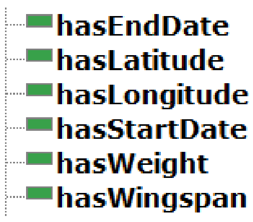
```
<PeriodOfInterest xmlns="http://semnotam.frequentis.com/schema/1.0"
xmlns:gml="http://www.opengis.net/gml/3.2" gml:id="TIF_January2017">
  <occTime>
    <TimeInterval>
      <beginPosition>2017-01-01T00:00:00.000Z</beginPosition>
      <endPosition>2017-02-01T00:00:00.000Z</endPosition>
    </TimeInterval>
  </occTime>
</PeriodOfInterest>
```

```
TemporalIntervalFacet
 and (hasEndDate only xsd:dateTime[<=
"2017-02-01T00:00:00"^^xsd:dateTime])
 and (hasStartDate only xsd:dateTime[>=
"2017-01-01T00:00:00"^^xsd:dateTime])
```

SubClass Of: TemporalIntervalFacet

Figure 5.4: Container Ontology – January2017

The concept *January2017* has no explicit definition as a subclass of the concept *Year2017*. However, Figure 5.4 shows that the time interval for this concept is defined from the first January in 2017 to the first February in 2017. As this time interval is part of the time interval defined by the concept *Year2017*, the reasoner will infer that *January2017* is a subconcept of *Year2017*.



hasEndDate
hasLatitude
hasLongitude
hasStartDate
hasWeight
hasWingspan

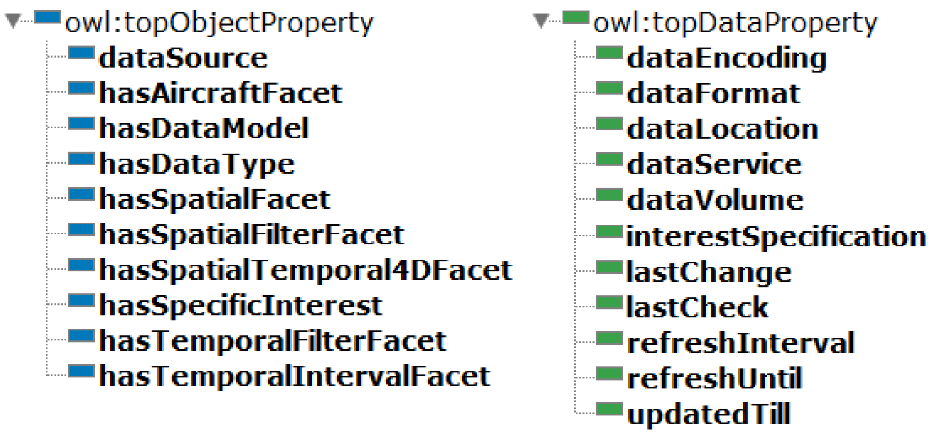
Figure 5.5: Container Ontology – Data properties for concept description

The list of data properties for the description of concepts is shown in Figure 5.5. The above example already showed the data properties *hasEndDate* and *hasStartDate*, for the description of time intervals. Figure 5.5 also shows the data properties *hasLatitude* and *hasLongitude* for describing spatial information and the data properties *hasWeight* and *hasWingspan* for the description of aircraft. These data properties are needed, as the *AircraftFacet* can contain not only concepts of specific aircraft types, but also more wide concepts concerning the size or weight of an aircraft. This can be relevant for the filtering of DNOTAMs concerning a closed runway for aircraft with a higher wingspan than 100 feet.

As mentioned before, facets are used to describe the content of data containers. A data container has one characteristic for each facet (Requirement 1.2.3). The realization of data containers in the Container Ontology and their description with concepts of each facet are described in the next section.

5.2.2 Data Container

A data container in the Container Ontology is represented as a class and as an individual using the so-called punning [44]. This means that the class and the individual can be referenced with exactly the same IRI (Requirement 1.1.1). The descriptive metadata, that is represented by concepts of the specific facets can be assigned to the data container class with object properties, whereas, administrative metadata is assigned to the individuals with object properties or data properties (Requirement 1.1.2). The possible object properties and data properties concerning descriptive or administrative metadata are shown in Figure 5.6.



owl:topObjectProperty
dataSource
hasAircraftFacet
hasDataModel
hasDataType
hasSpatialFacet
hasSpatialFilterFacet
hasSpatialTemporal4DFacet
hasSpecificInterest
hasTemporalFilterFacet
hasTemporalIntervalFacet

owl:topDataProperty
dataEncoding
dataFormat
dataLocation
dataService
dataVolume
interestSpecification
lastChange
lastCheck
refreshInterval
refreshUntil
updatedTill

Figure 5.6: Container Ontology – Object Properties, Data Properties

Except of the object property *dataSource*, which is part of the administrative metadata, all object properties are used for the description of the data items contained in a data container. The combination of these object properties is the descriptive metadata, i.e., semantic label, of the data container that describes the data items contained in this data container (Requirement 1.2.3). Facets that have no constraints directly refer to the facet itself, whereas restrictions refer to one specific concept of the facet. The descriptive metadata has to specify a concept for each existing facet to enable the complete subsumption reasoning. Listing 5.1 describes the semantic label of the data container *DNOTAM_AT_2017* that contains data items of Austria in the year 2017.

```
1   (hasAircraftFacet some AircraftFacet)
2     and (hasDataModel some AIXM)
3     and (hasDataType some NOTAM)
4     and (hasSpatialFacet some FirAustria)
5     and (hasSpatialFilterFacet some GmlShape)
6     and (hasSpatialTemporal4DFacet some SpatialTemporal4DFacet)
7     and (hasSpecificInterest some SpecificInterest)
8     and (hasTemporalFilterFacet some ActiveTime)
9     and (hasTemporalIntervalFacet some Year2017)
```

Listing 5.1: Data Container *DNOTAM_AT_2017* – Semantic Label

The data container *DNOTAM_AT_2017* has for example no constraint for the *AircraftFacet*, as shown in Line 1 in Listing 5.1. The definition of the spatial constraint for Austria is shown in Line 4 in Listing 5.1. By using the semantic labels of each data container, a reasoner can infer a subsumption hierarchy (Requirement 1.1.3). How the semantic label is used for the reasoning, is further described in Section 5.2.3.

In contrast to descriptive metadata representing the semantic label, administrative metadata is assigned to the individual by data properties or the object property *dataSource* (Requirement 1.3.1). For secondary data containers the data container that acts as data source, needs to be stored to enable the refreshing of the secondary data container (Requirement 1.3.2). For example, the data container *DNOTAM_AT_Jan2017* can use the data container *DNOTAM_AT_2017* as its data source and therefore, references the data container with the object property *dataSource*. For enabling the refreshing of data containers, the two data properties *dataService* and *interestSpecification* need to be defined for secondary data containers.

Other administrative metadata concerning primary as well as secondary data containers, such as the data format or the data encoding, are stored as literals. Therefore, they are represented as data properties in the Container Ontology. The relevant timestamps, that are needed to define the freshness of the data container, are also implemented as data properties (Requirement 1.3.3). Two examples of such timestamps are *lastChange*, which refers to the timestamp of the last added data item, or *lastCheck*, which is the timestamp of when the last check for new data items was performed.

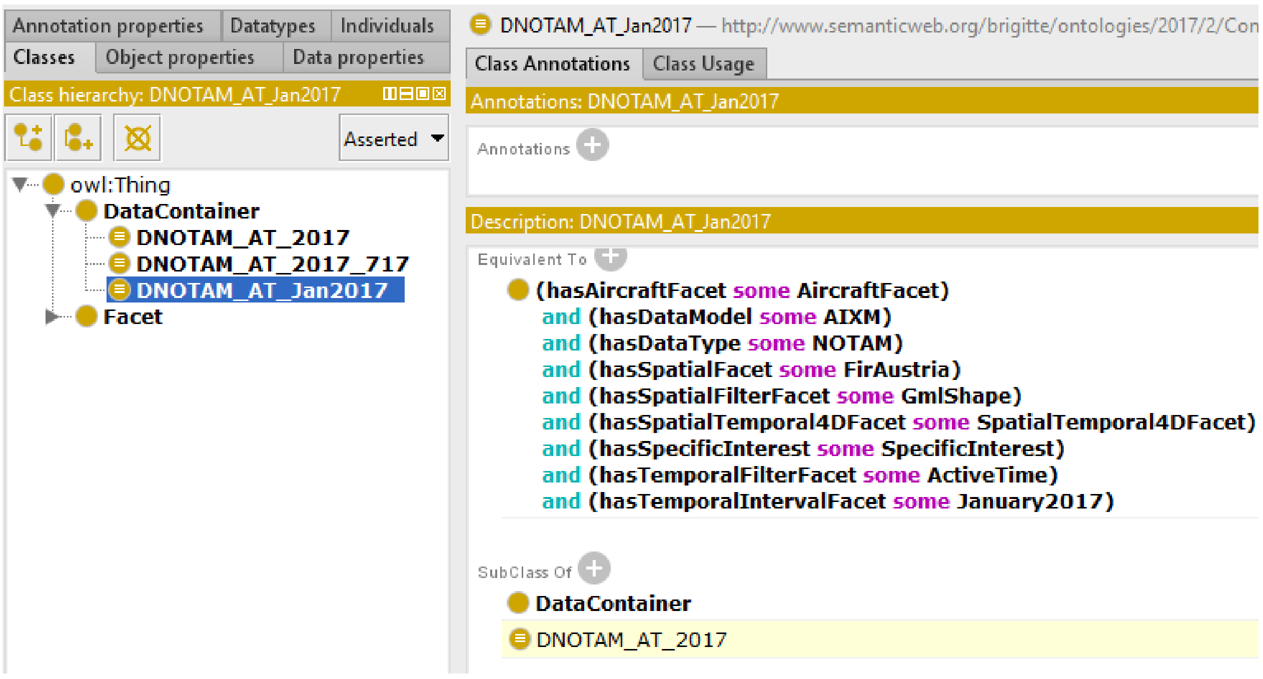
5.2.3 Discovery

As already mentioned before, the semantic labels are used to create a subsumption hierarchy between the data containers (Requirement 1.1.3). The reasoner evaluates each part of the semantic label and then combines the results to get the information about the subclass relationships. The reasoner checks for each object property if the following concept is either the same as for the other data container, or if it is defined or can be inferred as subconcept. Additionally to the data container *DNOTAM_AT_2017*, where the semantic label is shown in Listing 5.1, the data container *DNOTAM_AT_Jan2017* is specified in Listing 5.2.

```
1  (hasAircraftFacet some AircraftFacet)
2    and (hasDataModel some AIXM)
3    and (hasDataType some NOTAM)
4    and (hasSpatialFacet some FirAustria)
5    and (hasSpatialFilterFacet some GmlShape)
6    and (hasSpatialTemporal4DFacet some SpatialTemporal4DFacet)
7    and (hasSpecificInterest some SpecificInterest)
8    and (hasTemporalFilterFacet some ActiveTime)
9    and (hasTemporalIntervalFacet some January2017)
```

Listing 5.2: Data Container *DNOTAM_AT_Jan2017* – Semantic Label

In this example, only the concept for the object property *hasTemporalIntervalFacet* differs (Listing 5.1, Line 9 and Listing 5.2, Line 9). As described in Section 5.2.1 the concepts *Year2017* and *January2017* are defined by start and end dates, which allow the reasoner to infer that *January2017* is a subconcept of *Year2017*. Hence, the semantic label of the data container *DNOTAM_AT_Jan2017* describes a part of the semantic label of the data container *DNOTAM_AT_2017*, and therefore, the subsumption hierarchy between these two data containers can be inferred. The result of the reasoner in Protégé is shown in Figure 5.7, which shows the additional subclass relationship with the concept *DNOTAM_AT_2017*.



DNOTAM_AT_Jan2017 — http://www.semanticweb.org/brigitte/ontologies/2017/2/Con

```
owl:Thing
  DataContainer
    DNOTAM_AT_2017
    DNOTAM_AT_2017_717
    DNOTAM_AT_Jan2017
  Facet
```

Equivalent To

```
(hasAircraftFacet some AircraftFacet)
  and (hasDataModel some AIXM)
  and (hasDataType some NOTAM)
  and (hasSpatialFacet some FirAustria)
  and (hasSpatialFilterFacet some GmlShape)
  and (hasSpatialTemporal4DFacet some SpatialTemporal4DFacet)
  and (hasSpecificInterest some SpecificInterest)
  and (hasTemporalFilterFacet some ActiveTime)
  and (hasTemporalIntervalFacet some January2017)
```

SubClass Of

DataContainer
DNOTAM_AT_2017

Figure 5.7: Data Container *DNOTAM_AT_Jan2017* – Reasoning Result

5.3 Container Management Application

For managing the data containers, a web application that has access to the described Container Ontology and that can use the SemNOTAM Web Service and the Container Description Service is developed. As mentioned before, the Container Management Application is developed with the use of the Java framework Vaadin. The Java project is set up as a Maven project, therefore, the needed Vaadin dependencies are set in the corresponding pom.xml file. Furthermore, the Container Management Application needs to have access to the Container Ontology and needs to make use of a reasoner to receive the subsumption hierarchy. Therefore, also the OWL API and the Hermit reasoner are referenced in the pom.xml file.

Five packages are used to structure the implementation of the Container Management Application:

- *aero.aixm*
- *at.jku.dke*
- *com.frequentis.semnotam*
- *net.opengis*
- *org*

The package *at.jku.dke* contains the source code written during this thesis, whereas the other packages were provided for enabling the usage of the SemNOTAM Web Service and to make use of the Java representations of interest specifications (task descriptions) and DNOTAMs. The package *at.jku.dke* is again split in two packages where the package *containerdescription.ws* contains the relevant classes for communicating with the Container Description Service, and the package *containerManagement* contains the source code for the Container Management Application. The structure of this package is shown in Figure 5.8.

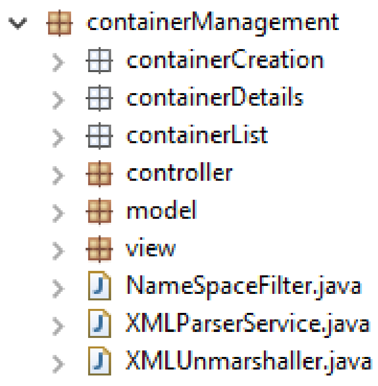

containerManagement
containerCreation
containerDetails
containerList
controller
model
view
NameSpaceFilter.java
XMLParserService.java
XMLUnmarshaller.java

Figure 5.8: Package Structure containerManagement

Source code that only provides supporting functionality, like the *NameSpaceFilter*, *XMLParserService*, and *XMLUnmarshaller*, are put directly in the *containerManagement* package. As already described in Section 4.1.2, the implementation uses the MVC design pattern, as shown in Figure 5.8, with the packages *model*, *view*, and *controller*. These MVC packages contain general functionality of the Container Management Application, such as the navigation in the application, the communication with the ontology, or the hierarchical view of data containers. Functionality that is developed for specific requirements, as the creation of data containers, is capsulated in separate packages, i.e., *containerCreation*, *containerDetails*, and *containerList*. The following paragraphs describe the MVC packages in more detail, whereas the functionality-specific packages are described in more detail in Section 5.3.1, Section 5.3.2, and Section 5.3.3.

Table 5.1 provides a short description of each class in the MVC packages. Furthermore, important aspects of the source code that are either relevant for the whole application or are a special programming type, are described afterwards.

MVC packages – containerManagement

Package/Class	Description
model	The *model* package contains all classes concerning the data that is needed for the Container Management Application as well as interfaces to other systems, like the Container Ontology.
OntologyService	The connection to the ontology via the OWL API is handled in this class. Whenever information of the ontology is requested or needs to be changed, methods of this class are called. For example, receiving a list of all data containers, creating a new data container, or editing an existing container, is performed with methods of this class.
Container	This class is used to represent a data container as a Java object and to easily use the metadata of a data container without the need of requesting the ontology for each visualization.
view	The *view* package contains classes for the visualization and navigation of the Container Management Application. Classes with visualizations that are reused in specific functionalities are also located in this package.
HomeView	The Container Management Application has a start view that shows some information of how to work with the application and where to find the information needed. This class mainly contains HTML code.
NavigatorUI	The navigation in the Container Management Application is implemented in the *NavigatorUI* class, which extends the user interface (UI) provided by Vaadin.
OntologySelectWindow	The Container Management Application allows to switch between ontologies. The window that pops up when switching the ontology is designed in this class.
DataItemsUploadWindow	For creating or refreshing a primary data container, a file containing the data items has to be uploaded. The class implements the functionality to upload such a file. (Requirements 2.2.5, 2.5.3)
HierarchyView	The subsumption hierarchy of data containers in the ontology is represented as a tree in the Container Management Application. The tree representation is also provided by the Vaadin framework.

controller	The *controller* package contains only one class for the general functionality of the application.
ManagementController	The controller implements all the listeners for button clicks, selections in the hierarchy view, and success and failure of file uploads. Furthermore, this class builds the connection between the model, the views and has a reference to the specific controller classes, i.e. *ListController*, *DetailsController*, and *CreateController*, which are described in the following sections.

Table 5.1: MVC packages – containerManagement

The UI is the entry point for the application. When opening the application in the browser window the *init* method of the class *NavigatorUI* is called. The source code of this method is shown in Listing 5.3, where at first a new instance of the *ManagementController* is created (Line 2). The constructor of the *ManagementController* establishes the connection to the default ontology and initializes all other controllers.

```
1   protected void init(VaadinRequest vaadinRequest) {
2     controller = new ManagementController(this);
3     VerticalLayout layout = new VerticalLayout();
4
5     tabs = new TabSheet();
6     tabs.setSizeFull();
7     tabs.addTab(new HomeView(), "Home", VaadinIcons.HOME);
8     tabs.addTab(controller.getListView(),
        "List of containers", VaadinIcons.LINES);
9     tabs.addTab(controller.getCreateView(),
        "Create a new container", VaadinIcons.PLUS);
10    tabs.addTab(controller.getDetailsView(),
        "Container details", VaadinIcons.INFO);
11    tabs.addSelectedTabChangeListener(controller);
12
13    layout.addComponent(tabs);
14    layout.setMargin(new MarginInfo(true, true, false, true));
15
16    setContent(layout);
17  }
```

Listing 5.3: NavigatorUI – init

As shown in Listing 5.3, the remaining part of the *init* method only concerns visualization aspects, where the *TabSheet*, a Vaadin specific UI component, is created and the tabs with corresponding icons are defined. The functionality of each tab is implemented in the *ManagementController*, as it is set as listener for the *TabSheet* in Line 11. The *ManagementController* checks to which tab the user has changed and calls the according methods, for loading the list of data containers, creating the view for creating new data containers, or showing the details page.

As the class *OntologyService* implements the connection to the ontology, this class contains most relevant source code. As already mentioned in Table 5.1, the data containers of the ontology are also stored as Java objects to allow a faster processing and minimize the number of requests to the ontology. Therefore, the class *OntologyService* contains the method *loadContainer* which loads all data containers and stores them as Java objects in a *HashMap*. In Listing 5.4, the code snippets of the *loadContainer* methods are shown. The *loadContainer* method without parameters (Lines 1-9) is called for loading all data containers, whereas, the *loadContainer* method with the *OWLClass* as parameter (Lines 11-18) creates and saves the Java object representation for a specific data container.

```
1   private void loadContainer() {
2     OWLClass dataContainerClass =
        factory.getOWLClass(currentBaseIRI + "#DataContainer");
3     Stream<OWLClass> containers =
        reasoner.subClasses(dataContainerClass, false);
4     containers.forEach(c -> {
5       if (!"owl:Nothing".equals(c.toString())) {
6         loadContainer(c);
7       }
8     });
9   }
10
11  private void loadContainer(OWLClass c) {
12    Container container = new Container();
13    container.setName(c.getIRI().getShortForm());
14    loadDescriptiveMetadata(container, c);
15    OWLNamedIndividual individual =
        factory.getOWLNamedIndividual(c);
16    loadAdministrativeMetadata(container, individual);
17    saveContainer(container);
18  }
```

Listing 5.4: OntologyService – loadContainer

In Line 2 in Listing 5.4, the *OWLClass DataContainer* which is the superclass of all data containers in the ontology is stored in the variable *dataContainerClass*. All subclasses of this class, i.e. all data containers, are then stored as a stream for further processing (Line 3). The second parameter of the *subClasses* method indicates that all subclasses shall be returned, independent of whether they are direct or indirect subclasses. For each of this classes, except for the class *owl:Nothing*, the second *loadContainer* method is called. In Line 12, the Java object for the data container is created and in Line 14, its descriptive metadata is loaded from the ontology. For loading the administrative metadata, which is stored for the individuals in the ontology, the same IRI can be used to retrieve the *OWLNamedIndividual*. After the administrative metadata has been loaded, the data container is saved to the *HashMap*.

Another important functionality of the *OntologyService* class is loading the container hierarchy for the hierarchical view. The visualization part of the hierarchy is implemented in the class *HierarchyView*, whereas loading the data containers accordingly is implemented in the *OntologyService* in the method *loadContainerHierarchy* shown in Listing 5.5.

```
1   public void loadContainerHierarchy(Tree hierarchy) {
2     OWLClass dataContainerClass =
        factory.getOWLClass(currentBaseIRI + "#DataContainer");
3     reasoner.subClasses(dataContainerClass, true).forEach(c -> {
4       if (!"owl:Nothing".equals(c.toString())) {
5         hasChildren = false;
6         long id = nextHierarchyId;
7         hierarchy.addItem(id);
8         hierarchy.setItemCaption(id, c.getIRI().getShortForm());
9         nextHierarchyId++;
10        reasoner.subClasses(c, true).forEach(sc -> {
11          if (!"owl:Nothing".equals(sc.toString())) {
12            addChildItem(hierarchy, id, sc);
13            hasChildren = true;
14          }
15        });
16        if (!hasChildren) {
17          hierarchy.setChildrenAllowed(id, false);
18        }
19      }
20    });
21  }
```

Listing 5.5: OntologyService – loadContainerHierarchy

The *OWLClass* for data container is retrieved and then each subclass, in this case only direct subclasses, is processed, as shown in Line 3 in Listing 5.5. In Lines 7-8, the subclasses are added to the hierarchy with an appropriate name. In Lines 10-15, the direct subclasses of the current class are processed and for each subclass the method *addChildItem* is called, which is shown in Listing 5.6. In Listing 5.5, Lines 16-18 are only needed to disable the expand arrow in the visualization, when there are no further subclasses.

The source code for the method *addChildItem* in Listing 5.6 is similar to the previous method. In Lines 4-5, the subclass is added to the hierarchy, while in Line 6, the parent relationship to the previously added item is set. By setting this relationship the tree is visualized correctly in the application. The remaining code in Lines 8-13 again processes the next hierarchy level, by calling the method *addChildItem* with the new parent and child classes. By recursively calling this method the whole hierarchy tree is visualized, even if one data container is a direct subclass of several data containers. Data containers that have several direct superclasses appear several times in the hierarchy view. The differentiation between the used data source and the possible superset data containers is shown later in Section 6.2, where the Container Management Application is demonstrated.

```
1   private void addChildItem(Tree hierarchy, long parentId,
    OWLClass child) {
2     hasChildren = false;
3     long id = nextHierarchyId;
4     hierarchy.addItem(id);
5     hierarchy.setItemCaption(id, child.getIRI().getShortForm());
6     hierarchy.setParent(id, parentId);
7     nextHierarchyId++;
8     reasoner.subClasses(child, true).forEach(sc -> {
9       if (!"owl:Nothing".equals(sc.toString())) {
10        addChildItem(hierarchy, id, sc);
11        hasChildren = true;
12      }
13    });
14    if (!hasChildren) {
15      hierarchy.setChildrenAllowed(id, false);
16    }
17  }
```

Listing 5.6: OntologyService – addChildItem

The content of the packages *containerList*, *containerDetails*, and *containerCreation* is described in the next sections. First, an overview of the packages and classes is given, followed by some specific source code snippets.

5.3.1 Container List

As described in Requirement 2.1.1 the Container Management Application must enable the user to see a list of all data containers in an ontology. Therefore, the package *containerList* again contains the three packages *model*, *view*, and *controller*. Table 5.2 lists and shortly describes the content of these packages.

MVC packages – containerList

Package/Class	Description
model	The *model* package for representing the list only contains one class to access the ontology.
ListModel	This class stores the connection to the *OntologyService* and loads all data containers with the corresponding method.
view	The *view* package contains the classes for visualizing the list and creating the connection to the corresponding controller.
ListView	A list for showing all data containers and their descriptive metadata (Requirement 2.1.2) is implemented in the class *ListView*. The items of this list can be filtered and are clickable to show the details of the chosen data container (Requirement 2.1.3).
controller	The *controller* package contains one class that handles all button clicks and selections in the list.
ListController	The *ListController* processes each button click and reloads the list when a filter text is entered by the user. Furthermore, this class contains the implementation for the functionality when the user selects a data container.

Table 5.2: MVC packages – containerList

As mentioned in Table 5.2, the *ListController* implements the listener for selecting a data container of the list. The source code for this method is shown in Listing 5.7.

```
1   public void select(SelectionEvent event) {
2     Object selected = view.getGrid().getSelectedRow();
3     if (selected != null) {
4       TabSheet tabs = navUI.getTabSheet();
5       tabs.setSelectedTab(3);
6       String containerName = ((Container) selected).getName();
7       view.getGrid().deselectAll();
8       managementController.loadDetails(containerName);
9     }
10  }
```

Listing 5.7: ListController – select

In Line 2 in Listing 5.7, the selected object of the list, which is implemented with the Vaadin component *Grid*, is retrieved. After checking if the selection is valid, the view is changed to the third tab, the details tab, in Line 5. The selection is removed again in Line 7 to reset the selection status, and the name of the selected data container is provided to the *loadDetails* method of the *ManagementController*. The descriptive and administrative metadata of the data container is loaded and visualized in the details view, which is described in the next section.

5.3.2 Container Details

As described before, the descriptive and administrative metadata of a specific data container can be viewed in a separate tab (Requirement 2.1.3). Furthermore, the details page allows the user to delete (Requrement 2.4.1) and refresh (Requirement 2.5.1) a data container, and to modify the refresh interval and the refresh until timestamp (Requirement 2.3.1). The packages and classes of the *containerDetails* package are listed in Table 5.3 below.

MVC packages – containerDetails

Package/Class	Description
model	The *model* package for showing the details of a data container only contains one class to access the ontology.
DetailsModel	This class stores the connection to the *OntologyService* and forwards changed data of a data container or the deletion of a data container.
view	The *view* package contains all relevant classes for visualizing the details of a specific data container, i.e., the descriptive and administrative metadata. Furthermore, it contains one class for the window that pops up when deleting a container.
DetailsView	The *DetailsView* class sets up the main visualization frame, which consists of a hierarchical view of all data containers, the *HierarchyView*, and separate views for the descriptive and administrative metadata (Requirement 2.1.3). Buttons for deleting (Requirement 2.4.1), refreshing (Requirement 2.5.1) or modifying (Requirement 2.3.1) the container are also defined in this class, and linked to the corresponding controller, the *DetailsController*.
DescriptiveMetadataView	This class contains the visualization of the descriptive metadata of a data container, i.e., the concepts for each facet, which is realized by a set of not editable text fields.
AdministrativeMetadataView	The administrative metadata is visualized the same way as the descriptive metadata, with not editable text fields, except for the timestamps, like the last change timestamp. The timestamps are visualized with date fields, that allow to select a date of a calendar, when the user is modifying the data container.
DeleteWindow	When the user wants to delete the selected data container, a separate window opens that informs the user, that the data container may act as data source for other data containers (Requirement 2.4.2). This class gives the user the opportunity to go on with the deletion or to cancel it.

controller	The *controller* package contains one class that handles all button clicks for each view or window described above.
DetailsController	The class *DetailsController* sets all the text fields and date fields correctly whenever a data container's details are viewed. When initiating a refresh of the data container, this class checks whether a primary or secondary data container needs to be refreshed. Either a window to upload the file containing the current data items for the primary data container (Requirement 2.5.3) is opened, or the defined data service is called with the secondary data container's interest specification and the data items of the data source. Furthermore, this class handles deletions of data containers and changes to the refresh interval and the refresh until timestamp and forwards it to the Container Ontology.

Table 5.3: MVC packages – containerDetails

The details page provides a lot of functionality and therefore, has to handle many different button clicks. Hence, the most interesting method of the *DetailsController* is the *buttonClick* method, where each button click is processed accordingly. The source code of this method is shown in Listing 5.8 below.

```java
public void buttonClick(ClickEvent event) {
  containerName = ((Label) view.getContainerView()
    .getComponent(0)).getValue();
  switch (event.getButton().getId()) {
  case "delete":
    deleteWindow = new DeleteWindow(containerName, this);
    navUI.addWindow(deleteWindow);
    break;
  case "delete2":
    model.deleteContainer(containerName);
    navUI.removeWindow(deleteWindow);
    managementController.loadHierarchy();
    disableContainerView("<p>This container does not exist
      anymore.</br>" + "Please select one of the containers in
      the hierarchy.</p>");
    break;
  case "cancelDelete":
    navUI.removeWindow(deleteWindow);
    break;
  case "modify":
    preserveOldValues();
    view.setModifyToolbar();
    break;
  case "save":
    saveNewValues(containerName);
    view.setDefaultToolbar();
```

```
      break;
    case "cancelModify":
      setOldValues();
      view.setDefaultToolbar();
      break;
    case "refresh":
      Container container =
        managementController.getContainer(containerName);
      Container superSet = container.getDataSource();
      if (superSet == null) {
        uploadWindow = new DataItemsUploadWindow(this);
        navUI.addWindow(uploadWindow);
      } else {
        String superSetContainerName = superSet.getName();
        Date now = new Date();
        Date refreshUntil = container.getRefreshUntil();
        if (now.before(refreshUntil)) {
          if (container.getLastCheck().before(
            superSet.getLastChange())) {
            String serviceName = container.getDataService();
            String interestSpecPath =
              model.getInterestSpecificationPath(containerName);
            File interestSpecFile = new File(interestSpecPath);
            InterestSpecificationType interestSpec =
              XMLUnmarshaller.unmarshalInterestSpecification
              (interestSpecFile);
            String superSetFilePath = superSet.getDataLocation();
            File superSetFile = new File(superSetFilePath);
            FeatureCollectionType featureCollection =
              XMLUnmarshaller.unmarshalFeatureCollection
              (superSetFile);
            String resultFilePath =
              managementController.callService(serviceName,
              interestSpec, featureCollection);
            model.saveAdministrativeMetadata(containerName,
              "dataLocation", resultFilePath);
            model.saveAdministrativeMetadata(containerName,
              "dataVolume", new File(resultFilePath).length()
              + " Bytes");
            managementController.setDates(containerName,
              superSetContainerName, resultFilePath);
          } else {
            managementController.setDates(containerName,
              superSetContainerName, null);
          }
          initDetailsView(containerName);
          Notification.show("The container has been successfully
            refreshed!");
```

```
57        } else {
58          calendar.setTime(refreshUntil);
59          String refreshUntilStr = calendar.get(Calendar.YEAR)
              + "-" + df.format(calendar.get(Calendar.MONTH) + 1)
              + "-" + df.format(calendar.get(Calendar.DATE));
60          Notification.show("This container shall only be
              refreshed until " + refreshUntilStr + ".",
              Type.ERROR_MESSAGE);
61        }
62      }
63      break;
64    default:
65    }
66  }
```

Listing 5.8: DetailsController – buttonClick

The main part of the *buttonClick* method in Listing 5.8 is a switch statement (Lines 3-65) that handles the different buttons based on their id. The first three cases (*delete*, *delete2*, and *cancelDelete*) are needed for the deletion of a data container, modifying a data container is covered in the next three cases (*modify*, *save*, *cancelModify*), and the last case (*refresh*) implements the possibility to initiate a refresh of a specific data container.

When the delete button is clicked (Lines 4-16), the *DeleteWindow* opens where the user gets the information that this data container may be a data source for another data container (Requirement 2.4.2). In this window the user, has the possibility to either delete the data container or to cancel the deletion. In Line 9, the deletion of the specific data container is forwarded to the corresponding model, which has the connection to the ontology via the *OntologyService* to permanently delete this data container. In Lines 11-12, the hierarchical representation of the data containers needs to be reloaded and the details view is disabled as the data container does not exist anymore and the user has to select another data container to view its details. If the user decides against the deletion of the data container the *DeleteWindow* disappears and the details view looks the same as before (Lines 14-16).

The details view allows the user to modify (Lines 17-28) the refresh interval and the refresh until timestamp of a data container (Requirement 2.3.1). When the user clicks the modify button, the existing values of these two fields are temporarily saved and the fields are enabled for changes (Listing 5.8, Line 18). The toolbar is then changed (Line 19), so that two buttons appear, with one button for saving the changes and one button for discarding the changes. By clicking the save button the new values are permanently stored in the ontology (Line 22) and the toolbar is set back to default (Line 23), with the buttons delete, modify, and refresh. If the user decides to cancel the modification, the two fields are set back to their old values (Line 26) and the toolbar is set back to default (Line 27).

The refreshing of the data container has the most complex source code (Listing 5.8, Lines 29-63). First, the container object that shall be refreshed and the data source container, if one exists, is selected. Primary data containers do not have a data source from which they can be refreshed. Instead, the user can upload the file with the current data items. In Line 33, a *DataItemsUploadWindow* is created and forces the user to upload such a file (Requirement 2.5.3). From Line 35 ongoing, the refreshing of secondary data containers is covered. The refresh until timestamp of the secondary data container is compared to the current date (Line 39). If the refresh until timestamp is in the past (Lines 57-61) a message is shown to the user that this data container shall only be refreshed to the corresponding timestamp (Requirement 2.5.2). In Line 40, the last change timestamp of the data source data container is compared to the last check timestamp of the secondary data container. If the data source contains no new data items, i.e., the last change is prior the last check, the data container needs no refresh (Requirement 2.5.2). However, the last check timestamp and the updated till timestamp are changed accordingly. If the data container needs to be refreshed, the defined data service needs to be called with the secondary data container's interest specification and the data items of the data source data container (Lines 41-48). The new data item file for the secondary data container is saved and the timestamps are changed accordingly.

5.3.3 Container Creation

The Container Management Application allows the user to create new data containers and to decide between the creation of primary or secondary data containers (Requirement 2.2.1). Table 5.4 describes the content of the MVC packages, followed by the description of some important source code snippets. The *containerCreation* package contains one more package that contains all classes concerning descriptive or administrative metadata. This package is again structured in the packages *model*, *view*, and *controller*, which are described in Table 5.5.

MVC packages – containerCreation

Package/Class	Description
model	For the creation of data containers, the model package contains one class for the connection with the ontology.
CreateModel	This class stores the connection to the *OntologyService* and forwards the information, including the descriptive and administrative metadata, of the newly created data containers to the ontology.

view	The *view* package contains all relevant classes for visualizing the form for the creation of a new data container and provides separate views for creating primary and secondary data containers.
CreateView	The *CreateView* class builds the main layout of this page to enable the creation of new data containers (Requirement 2.2.1). The view consists of a hierarchical view of all data containers, the *HierarchyView*, and the metadata views. The metadata views, one for the descriptive metadata and one for the administrative metadata, are located in a separate package and are described in Table 5.5. Furthermore, the *CreateView* provides two buttons for creating either a primary data container or a secondary data container (Requirement 2.2.4). For primary data container a window to upload the data items, the *DataItemsUploadWindow*, opens where the user can upload a file with the corresponding data items (Requirement 2.2.5).
SupersetWindow	When creating secondary data containers, the user has to choose an existing data container that shall be used as data source (Requirement 2.2.6). This *SupersetWindow* shows the most specific supersets and the corresponding administrative metadata, so that the user can base his/her decision on the freshness of the data containers.
ServiceWindow	After choosing the superset data container, the user has to decide which service shall be used to filter the data items accordingly (Requirement 2.2.7). The *ServiceWindow* provides a dropdown menu where the user can choose between different services. In this thesis only the SemNOTAM Web Service is available.
controller	The *controller* package contains one class that handles all clicks and actions performed of the views or windows described above.
CreateController	This class implements the different behavior when creating a primary or secondary data container. Either the *DataItemsUploadWindow* or the *SupersetWindow* followed by the *ServiceWindow* is shown. Furthermore, the *CreateController* checks if all inputs are valid, e.g. the refresh interval is a numeric value, and the container name is available for use. If all input values are checked, the creation of a new data container with all relevant information is forwarded to the model where the corresponding methods of the *OntologyService* are called.

Table 5.4: MVC packages – containerCreation

To create a new data container the method *createClick* of the *CreateController* is called. Based on the id of the clicked button, it is decided which type of data container shall be created. The source code for this method is shown in Listing 5.9, where the different behavior for primary and secondary data containers is implemented in Lines 8-17.

```
1   private void createClick(String buttonId) {
2     if (validContainerName()) {
3       if (validDoubleValue()) {
4         createContainer();
5         metadataController.buildInterestSpec();
6         metadataController.setInterestSpecificationId
            (containerName);
7         switch (buttonId) {
8         case "createPrimary":
9           uploadWindow = new DataItemsUploadWindow(this);
10          navUI.addWindow(uploadWindow);
11          break;
12        case "createSecondary":
13          supersetWindow = new SupersetWindow(this);
14          navUI.addWindow(supersetWindow);
15          List<Container> supersets =
              model.loadSupersets(containerName, true);
16          supersetWindow.setSupersets(supersets);
17          break;
18        default:
19        }
20      } else {
21        Notification.show("For the refresh interval only numeric
            values are allowed!", Type.ERROR_MESSAGE);
22      }
23    } else {
24      Notification.show("Please enter a valid container name!
          (must be unique)", Type.ERROR_MESSAGE);
25    }
26  }
```

Listing 5.9: CreateController – createClick

Before the data container is created, the uniqueness of the defined container name (Line 2) and the value of the refresh interval (Line 3) are checked. If a data container with the same name already exists, or if the refresh interval is not a numeric value, a corresponding notification is shown to the user, and the container is not created. If the entered values are valid, a data container with the defined name is created in the ontology (Line 4) and the corresponding interest specification is built based on the selected concepts for the descriptive metadata (Line 5). Depending on the type of data container that shall be created, Listing 5.9 shows that either a *DataItemsUploadWindow* for the file upload (Line 9) or a *SupersetWindow* with a list of data containers that can be used as data source (Lines 13-16) is created.

The upload of data items for the creation of a primary data container is the same as for the refresh of a primary data container, where the user has to upload a file containing all data items for the current data container (Requirement 2.2.5). For secondary data containers, the *SupersetWindow* shows a list with the available data containers that can be used as supersets. Therefore, the reasoner finds all available superset data containers based on the descriptive metadata. By clicking on one of these data containers, the user selects the data container as data source (Requirement 2.2.6). The source code for the *itemClick* method of the class *DetailsController* is provided in Listing 5.10.

```java
1   public void itemClick(ItemClickEvent event) {
2     superSetContainerName =
        event.getItem().toString().split(" ")[0];
3     Container superSet =
        managementController.getContainer(superSetContainerName);
4     String filePath = superSet.getDataLocation();
5     try {
6       supersetFile = new File(filePath);
7     } catch(NullPointerException e) {
8       superSetContainerName = null;
9       superSet = null;
10      filePath = null;
11      Notification.show("The data set for this container
          cannot be found. Please use another container.",
          Type.ERROR_MESSAGE);
12      return;
13    }
14    model.setDataSource(containerName, superSet);
15    supersetWindow.setVisible(false);
16    serviceWindow = new ServiceWindow(this);
17    navUI.addWindow(serviceWindow);
18  }
```

Listing 5.10: CreateController – itemClick

In Listing 5.10, first the selected data container is retrieved (Lines 2-3), the path to the file containing the data items is stored (Line 4) and the file with this file path is referenced (Line 6). If no file exists for this path, a notification is shown to the user that the data set is not available, and another data container shall be selected. If the file was located successfully, the data container is saved as data source and the *ServiceWindow* opens, where the user has to select the service that shall be used for filtering the data items according to the prior created interest specification.

The *ServiceWindow* shows all possible services that can be used to filter the data set. As mentioned before, this thesis only provides the SemNOTAM Web Service in the implementation of the Container Management Application. When the user has chosen the service that shall be used for filtering (Requirement 2.2.7), the method *callService* of the class *DetailsController*, which is shown in Listing 5.11, is called.

```
1   private void callService() {
2     String serviceName =
        (String) serviceWindow.getSelect().getValue();
3     saveInterestSpecification();
4     FeatureCollectionType featureCollection =
        XMLUnmarshaller.unmarshalFeatureCollection(supersetFile);
5     String resultFilePath =
        managementController.callService(serviceName,
        interestSpec,featureCollection);
6     model.saveAdministrativeMetadata(containerName,
        "dataLocation", resultFilePath);
7     model.saveAdministrativeMetadata(containerName, "dataVolume",
        new File(resultFilePath).length() + " Bytes");
8     managementController.setDates(containerName,
        superSetContainerName, resultFilePath);
9     model.saveAdministrativeMetadata(containerName,
        "dataService", serviceName);
10    createSucceded();
11    serviceWindow.setVisible(false);
12  }
```

Listing 5.11: CreateController – callService

To call the selected service three parameters, the service, the interest specification and the data set to filter, need to be set. In Listing 5.11, the name of the selected service (Line 2), the interest specification that shall be used (Line 3), and the data set to filter (Line 4) are stored in the needed formats. These parameters are forwarded to the *ManagementController*, where the service is called, and the file path of the resulting data set is returned (Line 5). After completing the service call, the administrative metadata, i.e., data location, data volume, relevant timestamps, and data service, are set accordingly (Lines 6-9). At the end of Listing 5.11, the method *createSucceded* is called, which shows a notification to the user that the container creation was successful and sets the view in the browser back to default.

As mentioned before, a separate package for the metadata exists, which is again structured based on the MVC pattern. These packages and classes for the descriptive and administrative metadata are described in Table 5.5 below.

MVC packages – metadata

Package/Class	Description
model	The *model* package of the metadata package contains one class that reads and combines the descriptive and administrative metadata.
MetadataModel	In this class the selected concepts for the descriptive metadata are combined to an intersection interest that is needed to create the ontological representation and the interest specification of a data container.
view	This package contains views for the descriptive and administrative metadata, and one window for the selection of a concept of one specific facet.
DescriptiveCreateView	The *DescriptiveCreateView* contains a button and a text field for each facet. The buttons can be used to select a concept for the corresponding facet (Requirement 2.2.2), and the text fields show the currently selected value. The possible concepts for a chosen facet, are shown in a separate window, the *ElementLoadWindow*.
AdministrativeCreateView	This class provides the input fields for the administrative metadata. The user can define the data format, data encoding, refresh interval in corresponding text fields and choose a date for the refresh until timestamp with a date picker component (Requirement 2.2.3).
ElementLoadWindow	The *ElementLoadWindow* is opened when the user clicks one of the buttons in the *DescriptiveCreateView*. This class shows all available concepts for one facet and allows the user to select one of them.
controller	The *controller* package contains one class that processes each click and selection made in the described views.
MetadataController	This class includes the implementation of the connection between the described views and the model to create the interest specification based on the selected values. Furthermore, each button click for choosing a concept for one specific facet or for removing the chosen concept is processed in this class.

Table 5.5: MVC packages – metadata

Some code snippets of the classes in the metadata MVC packages are described below, to show how the selection of concepts and building up an interest specification with the use of the Container Description Service works.

When the user clicks on one of the search buttons for one of the facets the *ElementLoadWindow* is opened, which shows a list of all available concepts for this specific facet. Therefore, the method *searchForConcept* is called, for which a part of the source code is shown in Listing 5.12. The switch statement only contains a case for the data model, as the code for the other facets is the same, except for the last part of the unified resource identifier (URI).

```
1   private void searchForConcept(String id, String ontoUri) {
2     List<String> conceptList = null;
3     switch (id) {
4     case "searchDataModel":
5       conceptList = model.selectConceptForFacet(ontoUri
          + "#DataModel");
6       break;
7     ...
8     default:
9     }
10    if (conceptList != null && !conceptList.isEmpty()) {
11      elw.setTable(conceptList);
12    }
13  }
```

Listing 5.12: MetadataController – searchForConcept

Based on the given id parameter the corresponding facet URI is constructed by adding the ontology URI with the symbol # and the facet's name. Line 5 in Listing 5.12 shows that the last part of the URI for the data model facet is *DataModel*, whereas, for example, for the aircraft facet it would be *AircraftFacet* or *SpatialFacet* for the spatial facet. The facet URI is used as paramenter for the method *selectConceptForFacet* of the *MetadataModel*, which then calls the corresponding method of the Container Description Service. The returned list of concepts is then, in case it is not empty, shown as a table in the *ElementLoadWindow* to enable the user to select one (Lines 10-12).

By clicking on one of the items shown in the created table in the *ElementLoadWindow*, the method *itemClick* of the *MetadataController* is called. As before, a part of the source code is given in Listing 5.13, as the code for each of the facets is similar. The table in the *ElementLoadWindow* has an id which is the same as the button id before, to know the facet the concept belongs to.

```
1   public void itemClick(ItemClickEvent itemClickEvent) {
2     Table table = (Table) itemClickEvent.getComponent();
3     TextField field = null;
4     Button removeButton = null;
5     String value = itemClickEvent.getItem().toString();
6     switch (table.getId()) {
7     case "searchDataModel":
8       dataModelConcept = value;
9       field = descriptiveCreateView.getTfDataModel();
10      removeButton = descriptiveCreateView.getRemoveDataModel();
11      break;
12    ...
13    default:
14    }
15    if (field != null) {
16      field.setReadOnly(false);
17      String valueShortForm = value.substring(
          value.lastIndexOf("#") + 1,value.length()-1);
18      field.setValue(valueShortForm);
19      field.setDescription(value.substring(1,value.length()-1));
20      field.setReadOnly(true);
21      checkFilter();
22      removeButton.setEnabled(true);
23    }
24    navUI.removeWindow(elw);
25  }
```

Listing 5.13: MetadataController – itemClick

Lines 7-11 in Listing 5.13 show the source code for the data model facet, where the chosen concept (variable value), the appropriate text field, and the button to remove the concept of this facet are stored. After the switch statement the value of the text field, is set to the short form of the selected concept and the whole URI is set as description (Lines 16-20). To allow the user to remove previously selected concepts, each facet has a button to remove the concept again. This button is only available when there is already a concept for the specific facet. Therefore, this button is enabled when a concept is selected (Line 22) and will be disabled again if the user removes the concept. The *checkFilter* method in Line 21 is only relevant for the spatial and temporal facets. The user is not allowed to select a concept for the temporal interval facet as long as he/she has not selected a concept for the temporal filter facet, analogous for the spatial facet and the spatial filter facet.

After the user has selected all the relevant concepts and creates a data container, the *createClick* method of the CreateController (Listing 5.9) is called, which then needs the interest specification. In Line 5 in Listing 5.9, the method *buildInterestSpec* of the *MetadataController* is called, which processes each of the facets to build a complete interest specification. Depending on the facet, different methods of the *MetadataModel* are called, as the concepts have to be integrated in different positions of the interest specification.

An interest specification consists of a *NotamSetMetaInformationPropertyType* containing meta information and a *IntersectionInterestType* containing semantic, spatial, and temporal interests. The *NotamSetMetaInformationPropertyType* contains meta information, such as the data model, data type, or temporal and spatial filter types, whereas the other facets are part of the *IntersectionInterestType*. The source code for the method *setGeneralInterestData*, that is used for semantic, spatial, and temporal facets, is shown in Listing 5.14.

```java
1  public void setGeneralInterestData(String ontologyUri, String
   conceptUri) {
2    if (ontologyUri != null && conceptUri != null) {
3      String generalInterestPropertyXML =
         ContainerDescriptionWS.getInterestFromConcept(ontologyUri,
         conceptUri.substring(1,conceptUri.length()-1));
4      if(generalInterestPropertyXML!=null){
5        InterestPropertyType interestProp =
           XMLUnmarshaller.unmarshalGeneralInterestData(
             generalInterestPropertyXML);
6        intersectionInterest.getHasMember().add(interestProp);
7      }
8    }
9  }
```

Listing 5.14: MetadataModel – setGeneralInterestData

In Line 3 in Listing 5.14, the interest representation for the given concept is stored, which is then transformed to an *InterestPropertyType* by the *XMLUnmarshaller*. With the method *getHasMember* called for the intersection interest the list of interests is retrieved and the interest property can be added to the list. By calling the method *setGeneralInterestData* for each facet, an intersection of all interests for the selected concepts is created.
For adding the meta information, i.e., data model, data type, temporal and spatial filter, to the interest specification the method *setGeneralMetadata* is called, which is shown in Listing 5.15.

```java
public void setGeneralMetadata(String prop, String conceptUri) {
  String value = null;
  if(conceptUri!=null) {
    value = conceptUri.substring(conceptUri.lastIndexOf("#")
      + 1,conceptUri.length()-1);
  }
  switch(prop) {
  case "dataModel":
    if(value!=null) {
      metaInfo.setDataFormat(value);
    }
    break;
  case "dataType":
    if(value!=null) {
      metaInfo.setDataType(value);
    }
    break;
17    case "temporalFilter":
18      CodeTemporalRelevanceType temporalRelevance =
          CodeTemporalRelevanceType.NONE;
19      if(value!=null) {
20        if("ActiveTime".equals(value)) {
21          temporalRelevance =
              CodeTemporalRelevanceType.ACTIVE_TIME;
22        } else if("ValidTime".equals(value)) {
23          temporalRelevance =
              CodeTemporalRelevanceType.VALID_TIME;
24        }
25      }
26      relevanceOption.setTemporalRelevanceRules(
          temporalRelevance);
27      break;
28    case "spatialFilter":
29      CodeSpatialRelevanceType spatialRelevance =
          CodeSpatialRelevanceType.NONE;
30      if(value!=null) {
31        if("GmlShape".equals(value)) {
32          spatialRelevance = CodeSpatialRelevanceType.SHAPE;
33        } else if("BoundingBox".equals(value)) {
34          spatialRelevance =
              CodeSpatialRelevanceType.BOUNDING_BOX;
35        }
36      }
37      relevanceOption.setSpatialRelevanceRules(spatialRelevance);
38      break;
39    default:
40    }
41  }
```

Listing 5.15: MetadataModel – setGeneralMetadata

In Lines 7-16 in Listing 5.15, the selected concepts of the two facets data model and data type are added to the meta information by calling the corresponding method of the class *MetaInfoType*. This class is part of the package *com.frequentis.semnotam*, where the data model term of this thesis is referred to as data format. The meta information for the temporal and spatial filter is a bit different as these facets are realized with the enumerations *CodeTemporalRelevanceType* and *CodeSpatialRelevanceType*. If the user did not select a concept for the temporal filter the default relevance type is *NONE*, which is shown in Line 18. If a concept is selected it is checked if the valid time or the active time should be used, and the according enumeration value, *VALID_TIME* or *ACTIVE_TIME*, is set (Lines 19-24). For the spatial filter facet the according enumeration value is set analogously, with the values *NONE*, *BOUNDING_BOX,* and *SHAPE*.

As already mentioned above, the creation of a new data container uses methods of the Container Description Service. The three services of the Container Description Service, which are also used for the integration of the SemNOTAM Briefing Application, are described in the next section.

5.4 Container Description Service

To enable the Container Management Application and the SemNOTAM Briefing Application to use the same interfaces for selecting concepts of an ontology and creating a corresponding interest specification the Container Description Service is developed. This project is set up as Maven project with according dependencies for Apache CXF, OWL API, and Hermit reasoner in the pom.xml file.

The implementation of the Container Description Service is structured in five main packages:

- *aero.aixm*
- *at.jku.dke.containerDescription.ws*
- *com.frequentis.semnotam*
- *net.opengis*
- *org*

The package *at.jku.dke.containerDescription.ws* contains the source code written during this thesis, whereas the other packages were provided for enabling the usage of the SemNOTAM Web Service and to make use of the Java structure of interest specifications and DNOTAMs. As described before, the Container Management Application has a package that allows the communication with the Container Description Service. The classes in this package are generated by Apache CXF 3.1.8 and provide all the services of the Container Description Service. The three services *selectConceptForFacet*, *getInterestFromConcept*, and *getMostSpecificSuperset* are developed in the class *ContainerDescriptionService* and are described in the next sections.

5.4.1 selectConceptForFacet

Listing 5.16 shows the method *selectConceptForFacet*, which has one input parameter that defines the facet for which the list of concepts shall be returned (Requirement 3.1.1). As described in Section 5.3.3, each time the user clicks on the search button of one facet this method is called to provide the user the list of available concepts.

```java
public List<String> selectConceptForFacet(String facetUri) {
  String ontoUri = facetUri.substring(0,
    facetUri.lastIndexOf('#'));
  loadOntology(ontoUri);
  List<String> result = new ArrayList<>();
  OWLClass facetClass = factory.getOWLClass(facetUri);
  Stream<OWLClass> concepts = reasoner.subClasses(
    facetClass, false);
  concepts.forEach(c -> {
    if (!"owl:Nothing".equals(c.toString())) {
      result.add(c.toString());
    }
  });
  return result;
}
```

Listing 5.16: ContainerDescriptionService – selectConceptForFacet

In Line 2 in Listing 5.16, the URI of the ontology for the given facet is stored to load the corresponding ontology in Line 3. Via the OWL API the *OWLClass* for the facet is retrieved in Line 5 and all available subclasses, i.e., the concepts for this facet, are stored in a stream to process them iteratively. To receive all possible concepts (Requirement 3.1.2) the second parameter of the method *subClasses* is set to false, so that also indirect subclasses are included in the result. Each of the concepts, except the class *owl:Nothing*, is added to the result list (Lines 7-11) and in the end, the list with all concepts is returned.

5.4.2 getInterestFromConcept

As described before, the concepts of the facets are part of the interest specification for the data container. Therefore, each concept stores its interest representation as annotation in the ontology, to enable the creation of an interest specification based on the concepts. The source code to get the corresponding interest representation for a specific concept is implemented in the method *getInterestFromConcept*, which is shown in Listing 5.17. This method has two input parameters, first the URI of the ontology and second the URI of the concept for which the interest representation shall be returned (Requirement 3.2.1).

```
1   public String getInterestFromConcept(String ontologyUri,
      String conceptUri) {
2     loadOntology(ontologyUri);
3     interest = null;
4     OWLClass concept = factory.getOWLClass(conceptUri);
5     Stream<OWLAnnotationAssertionAxiom> annotations =
        owl.annotationAssertionAxioms(concept.getIRI());
6     annotations.forEach(annotation -> {
7       if ("interestRepresentation".equals(
          annotation.getProperty().getIRI().getShortForm())) {
8         interest = annotation.getValue().asLiteral()
            .get().getLiteral();
9       }
10    });
11    return interest;
12  }
```

Listing 5.17: ContainerDescriptionService – getInterestFromConcept

Listing 5.17 shows that first the ontology, where the concept is stored, needs to be loaded (Line 2) and the corresponding *OWLClass* needs to be retrieved (Line 4). With the method *annotationAssertionAxioms* all annotations of this *OWLClass* are stored in the stream in Line 5. All annotations are processed and the value of the annotation concerning the interest representation is stored in the variable *interest*, which is returned at the end (Requirement 3.2.2). An interest representation contains the information of the concept described in XML, to include it in the whole interest specification and enable the use of the SemNOTAM Web Service, which expects an interest specification as input.

5.4.3 getMostSpecificSuperset

As described in the approach in Section 4.1, the SemNOTAM Web Service provides an evaluation method to the SemNOTAM Briefing Application where only an interest specification is expected. The data set that shall be used for filtering needs to be discovered using a reasoner. Therefore, the SemNOTAM Web Service calls the method *getMostSpecificSuperset* of the Container Description Service, which returns the data set of the most specific superset for the given interest specification. The method *getMostSpecificSuperset* needs an interest specification as input and returns the feature collection of the identified superset data container (Requirement 3.3.1). The source code of this method is shown in Listing 5.18

```
1   public FeatureCollectionType getMostSpecificSuperset(
      InterestSpecificationType interestSpecification) {
2     loadOntology();
3     OWLClass container = factory.getOWLClass(DEFAULT + "#"
        + interestSpecification.getId());
4     if (owl.containsClassInSignature(container.getIRI())) {
5       OWLIndividual dataSource = getDataSource(container);
6       if (dataSource != null) {
7         return getDataSet(dataSource);
8       }
9     } else {
10      mapInterestSpecification(container, interestSpecification);
11    }
12    saveOntology();
13    OWLIndividual mostSpecificSuperset = null;
14    try {
15      mostSpecificSuperset = findMostSpecificSuperset(container);
16    } catch (Exception e) {
17      e.printStackTrace();
18    } finally {
19      removeContainer(container);
20    }
21    return getDataSet(mostSpecificSuperset);
22  }
```

Listing 5.18: ContainerDescriptionService – getMostSpecificSuperset

At first the default ontology is loaded (Line 2) and a temporary data container for the given interest specification, i.e., with the corresponding id, is created (Requirement 3.3.3). As this temporary data container might already exist in the ontology, Listing 5.18 shows that the existence of the data container is checked (Line 4) and if there is already a data source defined for this data container (Line 6), the feature collection of this data container is returned. If the data container does not exist, the interest specification needs to be mapped back to the concepts of the facets to enable the reasoning based on the descriptive metadata. Therefore, the method *mapInterestSpecification* is called, which finds the corresponding concept for each facet based on parts of the interest specification (Requirement 3.3.2). The meta information, i.e., data model, data type, temporal filter, and spatial filter, can be easily mapped from the enumerations back to the concepts.

The other facets must be identified based on the annotated interest representations in the ontology. Therefore, the single interests are extracted of the interest specification. For each facet the method *getConceptForInterest*, that is shown in Listing 5.19, is called with the corresponding *InterestPropertyType* that needs to be transformed to a string representation (Line 2).

```
1   private OWLClass getConceptForInterest(String facet,
      InterestPropertyType interest) {
2     String interestString = getStringForInterest(facet, interest);
3     conceptForInterest = null;
4     OWLClass facetClass = factory.getOWLClass(DEFAULT + "#"
        + facet);
5     reasoner.subClasses(facetClass, false).forEach(concept -> {
6       owl.annotationAssertionAxioms(concept.getIRI())
          .forEach(annotation -> {
7         if ("interestRepresentation".equals(
            annotation.getProperty().getIRI().getShortForm())) {
8           String annotatedInterest =
              annotation.getValue().asLiteral().get().getLiteral();
9           if (idEquals(interestString, annotatedInterest)) {
10            conceptForInterest = concept;
11          }
12        }
13      });
14    });
15    return conceptForInterest;
16  }
```

Listing 5.19: ContainerDescriptionService – getConceptForInterest

Based on the *OWLClass* representing the facet, the subclasses, i.e., concepts of this facet, are processed. The id of the string representation of the given interest is compared to the id of the annotation concerning the interest representation (Line 9), and if the same id is found, this concept is returned. The returned concepts of the facets are used to create an axiom for the temporary data container as description of the descriptive metadata (Requirement 3.3.3). The ontology is saved (Listing 5.18, Line 12) so that the reasoner has the needed information for the reasoning process. In Line 15 in Listing 5.18, the method *findMostSpecificSuperset* is called which returns the *OWLIndividual* representing the most specific superset. The source code for the method *findMostSpecificSuperset*, which needs the temporary data container as input, is shown in Listing 5.20.

```
private OWLIndividual findMostSpecificSuperset(
  OWLClass container) {
  Object[] superClasses =
    reasoner.equivalentClasses(container).toArray();
  OWLClass[] toBeChecked;
  if (superClasses.length == 2) {
    if (container.equals(superClasses[0])) {
      return factory.getOWLNamedIndividual(
        (OWLClass) superClasses[1]);
7       } else {
8         return factory.getOWLNamedIndividual(
            (OWLClass) superClasses[0]);
9       }
10    } else if (superClasses.length > 2) {
11      toBeChecked = new OWLClass[superClasses.length - 1];
12      int j = 0;
13      for (int i = 0; i < superClasses.length; i++) {
14        if (!container.equals(superClasses[i])) {
15          toBeChecked[j] = (OWLClass) superClasses[i];
16          j++;
17        }
18      }
19      return checkAdministrativeMetadata(toBeChecked);
20    }
21    superClasses = reasoner.superClasses(
        container, true).toArray();
22    if (superClasses.length == 1) {
23      return factory.getOWLNamedIndividual(
          (OWLClass) superClasses[0]);
24    } else {
25      return checkAdministrativeMetadata(
          (OWLClass[]) superClasses);
26    }
27  }
```

Listing 5.20: ContainerDescriptionService – findMostSpecificSuperset

The most specific superset is either a data container with exactly the same descriptive metadata, i.e., an equivalent class, or it is a direct superclass (Requirement 3.3.4). In Line 2 in Listing 5.20, an array of all equivalent classes of the given *OWLClass* are selected, which has at least one entry as the given *OWLClass* will also be returned. If the array contains two entries, the entry that is not the given data container is the most specific data container (Lines 4-9), whereas if the array has more than two entries (Lines 10-20), the administrative metadata needs to be compared (Requirement 3.3.5). If the array only contains one entry, the direct superclasses are selected in Line 21, indicated by the second parameter of the method *superClasses* set to true. If there is only one direct superclass, the individual of this class is returned, whereas if more superclasses exist, again the administrative metadata is checked (Requirement 3.3.5).

For the decision which data container to use as superset, the administrative metadata is considered in the following order: updatedTill, lastChange, lastCheck, and dataVolume. The data container with the highest freshness, indicated by the three timestamps, or with the minimal data volume is returned as the most specific superset. After the identification of the most specific superset the corresponding data set is returned as shown in Line 21 in Listing 5.18 (Requirement 3.3.4).

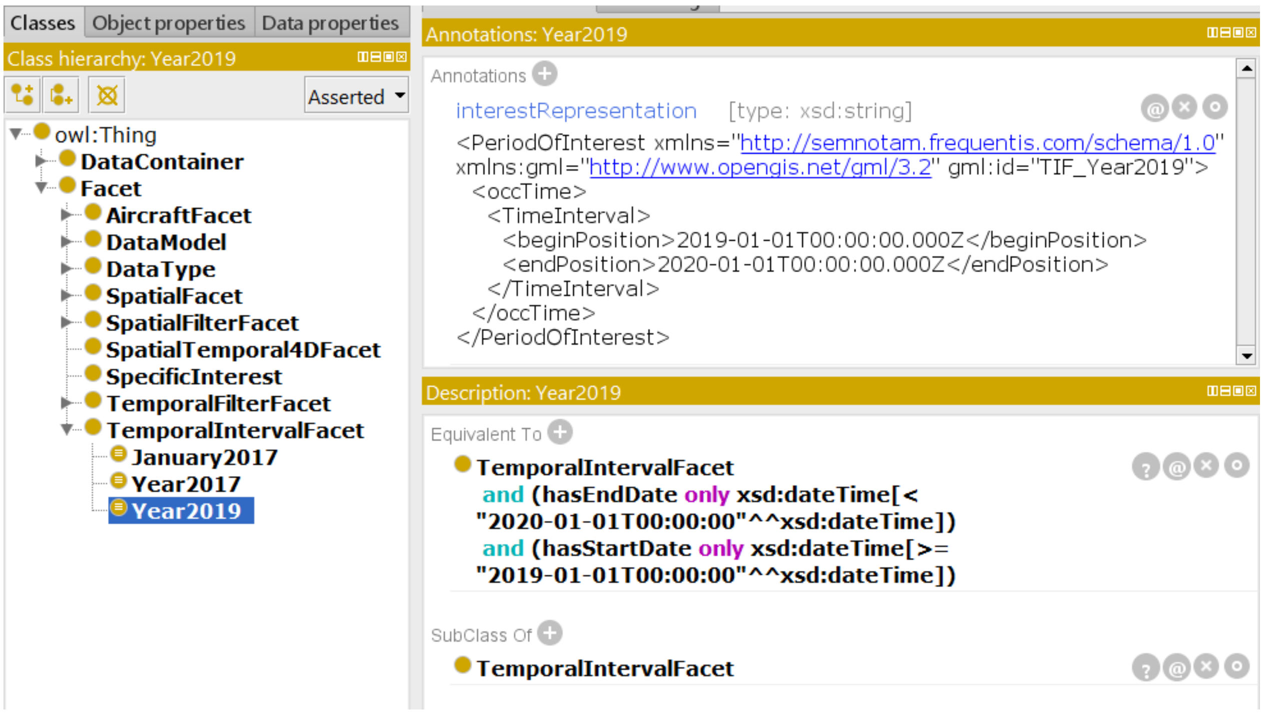
6 Demonstration

In this section, the usage of the implemented system is demonstrated. As described in Section 1.1, two different roles exist, producers and consumers. Producers can define data containers by describing the content to be published, while consumers specify the information necessary for a task to be fulfilled. The applications that are used by these two roles are shown in Figure 1.2. To ensure the producers and consumers have the same understanding of the shared information, the common vocabulary needs to be defined in the Container Ontology. The producers can describe and provide their data containers by using the Container Management Application by using the provided concepts of the Container Ontology. The consumers describe their tasks in the SemNOTAM Briefing Application by also using the concepts of the Container Ontology, which is then evaluated by the SemNOTAM Web Service based on the most specific superset the Container Description Service infers.

This section is structured as followed. Section 6.1 describes the usage of the Container Ontology, which consists of the management of the concepts for the different facets, e.g., the definition of the temporal concept to describe data containers with DNOTAMs of a specific year. Afterwards, Section 6.2 shows the usage of the Container Management Application, which can be used by producers to manage the existing data containers and also to create new data containers based on existing ones or on new data sets. In Section 6.3, the briefing application, i.e., the task-based retrieval service, is demonstrated, which allows a consumer to receive relevant DNOTAMs for a specific task, in this case a flight from Vienna to Frankfurt.

6.1 Container Ontology

The Container Ontology is used to define concepts for the description of data containers. When defining a new concept, the user must add a subclass to the corresponding facet. For example, to define the concept *Year2019* a new subclass of the *TemporalIntervalFacet* is created. Furthermore, an equivalent axiom needs to be defined to describe the concept and the interest representation of the concept is annotated. The whole definition of the concept as it looks like in Protégé is shown in Figure 6.1.



interestRepresentation [type: xsd:string]

```
<PeriodOfInterest xmlns="http://semnotam.frequentis.com/schema/1.0"
xmlns:gml="http://www.opengis.net/gml/3.2" gml:id="TIF_Year2019">
  <occTime>
    <TimeInterval>
      <beginPosition>2019-01-01T00:00:00.000Z</beginPosition>
      <endPosition>2020-01-01T00:00:00.000Z</endPosition>
    </TimeInterval>
  </occTime>
</PeriodOfInterest>
```

Equivalent To

```
TemporalIntervalFacet
 and (hasEndDate only xsd:dateTime[<
"2020-01-01T00:00:00"^^xsd:dateTime])
 and (hasStartDate only xsd:dateTime[>=
"2019-01-01T00:00:00"^^xsd:dateTime])
```

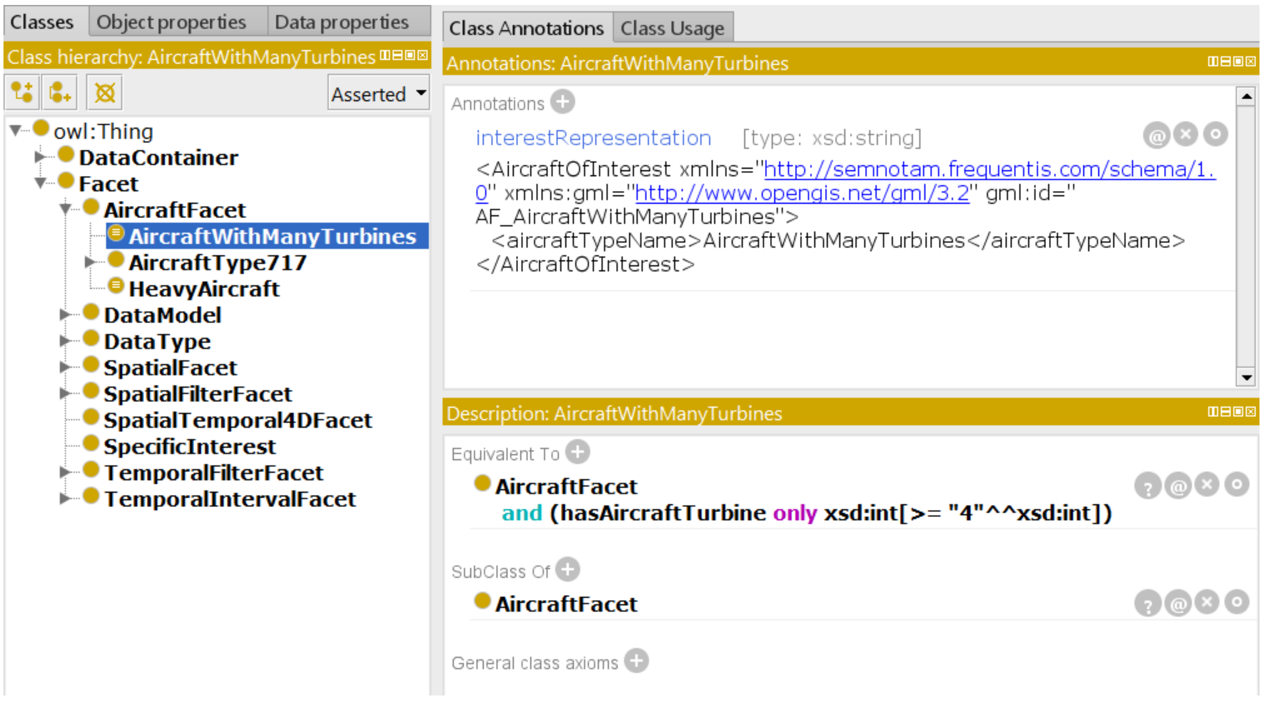
SubClass Of

TemporalIntervalFacet

Figure 6.1: Container Ontology – Year2019

The equivalent axioms allow the reasoner to perform the subsumption reasoning accordingly. Alternatively, the hierarchical relationships between concepts can be directly modeled in the ontology, by adding the new concept as subclass of an existing concept of the facet. For example, the concept *January2017* could have been defined as subclass of *Year2017* explicitly. In Section 5.2.1, it was shown that the subsumption hierarchy between these two concepts is defined with the time intervals defined as equivalent axioms.

Figure 5.6 lists the predefined data properties that can be used for the description of concepts. When the user wants to create a new concept that needs to be described by other characteristics, or if an existing concept should be described in more detail, a new data property needs to be defined. For example, if a new concept for aircrafts with four or more aircraft turbines should be created, then the data property *hasAircraftTurbine* has to be defined first. Afterwards, the concept *AircraftWithManyTurbines* can be created, as shown in Figure 6.2, which is defined that it has at least four aircraft turbines.



interestRepresentation [type: xsd:string]

```
<AircraftOfInterest xmlns="http://semnotam.frequentis.com/schema/1.0" xmlns:gml="http://www.opengis.net/gml/3.2" gml:id="AF_AircraftWithManyTurbines">
  <aircraftTypeName>AircraftWithManyTurbines</aircraftTypeName>
</AircraftOfInterest>
```

Equivalent To: AircraftFacet and (hasAircraftTurbine only xsd:int[>= "4"^^xsd:int])

SubClass Of: AircraftFacet

Figure 6.2: Container Ontology – AircraftWithManyTurbines

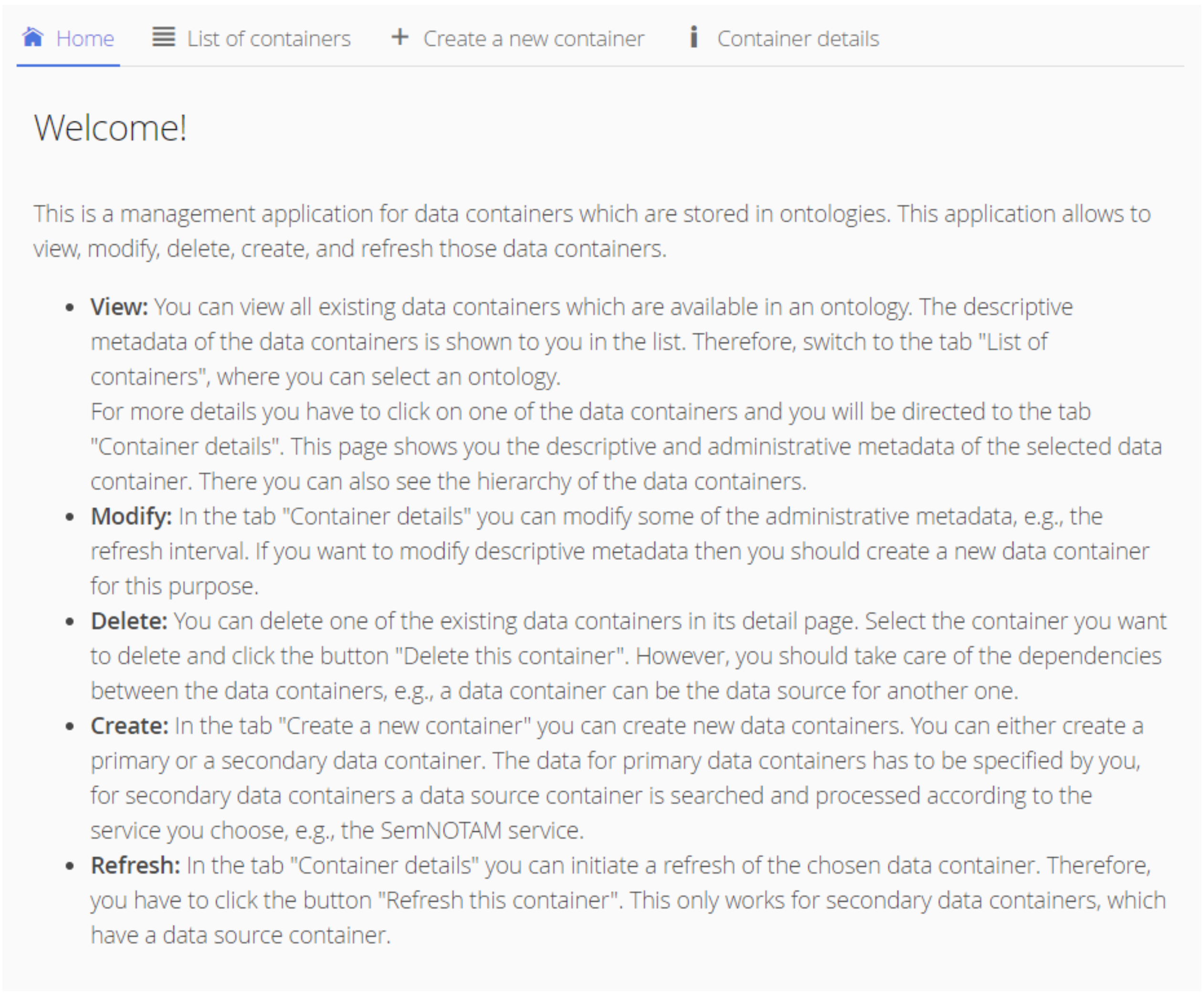
In the Container Ontology it is also possible to define object properties and individuals, however, this is not needed for defining concepts for facets. Data containers are not created directly in the Container Ontology, as it is more complex to define a data container. The user has to define the concepts for each facet explicitly and create the punned individual with all the possible timestamps as well as the corresponding files, i.e., the interest specification and the data set. For creating and maintaining data containers the Container Management Application is developed, which is demonstrated in the next section.

6.2 Container Management Application

In Section 4.2 the technology layer of the application was shown. The Container Management Application is deployed on a Jetty server and the Container Description Service and the SemNOTAM Web Service are deployed on a Tomcat 9. The port of the Tomcat server needs to be set to 8081 and the Jetty server is running at port 8080. As the Container Management Application and the Container Description Service are developed as Maven projects, Maven needs to be installed, and the system variables JAVA_HOME and MAVEN_HOME need to be set correctly. Furthermore, the Container Management Application and the Container Description Service contain a settings file, where the path of the repository needs to be configured.

When importing the two projects, which are part of this thesis, the Maven repository needs to be cleaned and the projects need to be updated, so that all needed dependencies are up to date. After deploying the Container Description Service and the SemNOTAM Web Service on the Tomcat server at port 8081 and starting the Container Management Application with a Maven build and the goal set to *jetty:run*, the application is accessible at localhost:8080.

The Container Management Application has a home view that provides the user all the information about the application. It describes where to find specific information and on which tabs the user can view, modify, delete, create, or refresh data containers. This home view is shown in Figure 6.3.



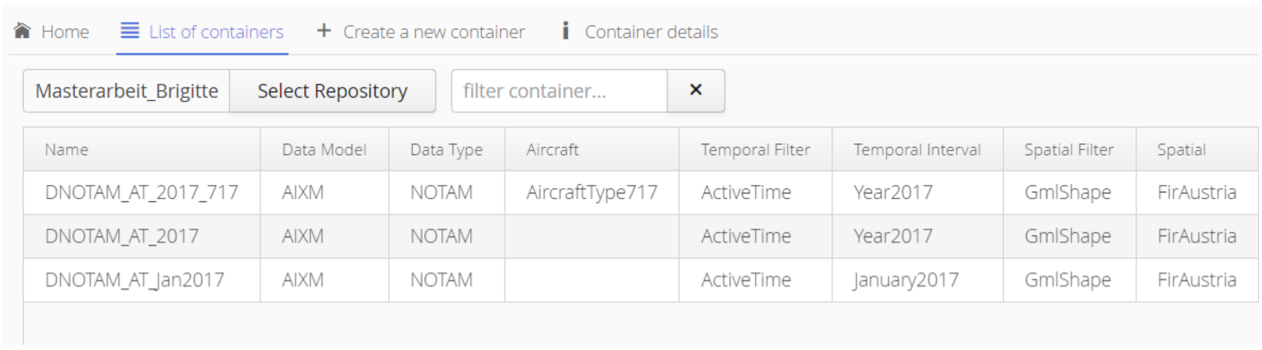
Welcome!

This is a management application for data containers which are stored in ontologies. This application allows to view, modify, delete, create, and refresh those data containers.

- **View:** You can view all existing data containers which are available in an ontology. The descriptive metadata of the data containers is shown to you in the list. Therefore, switch to the tab "List of containers", where you can select an ontology.
  For more details you have to click on one of the data containers and you will be directed to the tab "Container details". This page shows you the descriptive and administrative metadata of the selected data container. There you can also see the hierarchy of the data containers.
- **Modify:** In the tab "Container details" you can modify some of the administrative metadata, e.g., the refresh interval. If you want to modify descriptive metadata then you should create a new data container for this purpose.
- **Delete:** You can delete one of the existing data containers in its detail page. Select the container you want to delete and click the button "Delete this container". However, you should take care of the dependencies between the data containers, e.g., a data container can be the data source for another one.
- **Create:** In the tab "Create a new container" you can create new data containers. You can either create a primary or a secondary data container. The data for primary data containers has to be specified by you, for secondary data containers a data source container is searched and processed according to the service you choose, e.g., the SemNOTAM service.
- **Refresh:** In the tab "Container details" you can initiate a refresh of the chosen data container. Therefore, you have to click the button "Refresh this container". This only works for secondary data containers, which have a data source container.

Figure 6.3: Container Management Application – Home view

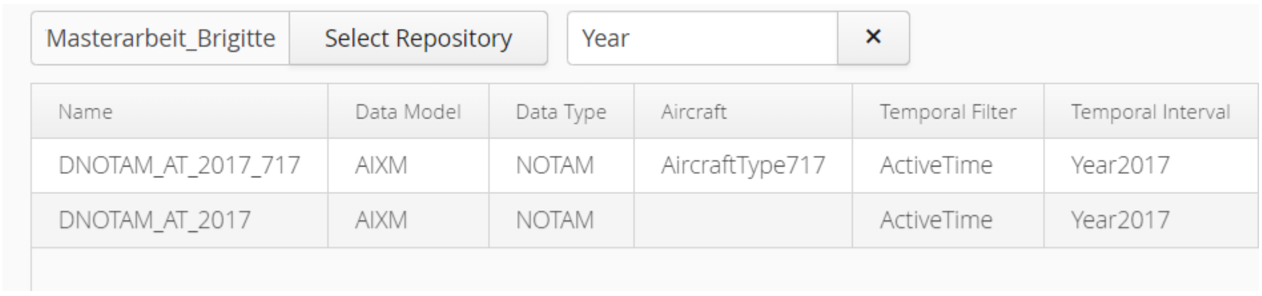
By switching to the tab *List of containers* a table with the descriptive metadata of all existing data containers in the ontology is shown. The list of possible data containers in the current ontology is shown in Figure 6.4.



Name	Data Model	Data Type	Aircraft	Temporal Filter	Temporal Interval	Spatial Filter	Spatial
DNOTAM_AT_2017_717	AIXM	NOTAM	AircraftType717	ActiveTime	Year2017	GmlShape	FirAustria
DNOTAM_AT_2017	AIXM	NOTAM		ActiveTime	Year2017	GmlShape	FirAustria
DNOTAM_AT_Jan2017	AIXM	NOTAM		ActiveTime	January2017	GmlShape	FirAustria

Figure 6.4: Container Management Application – List of containers

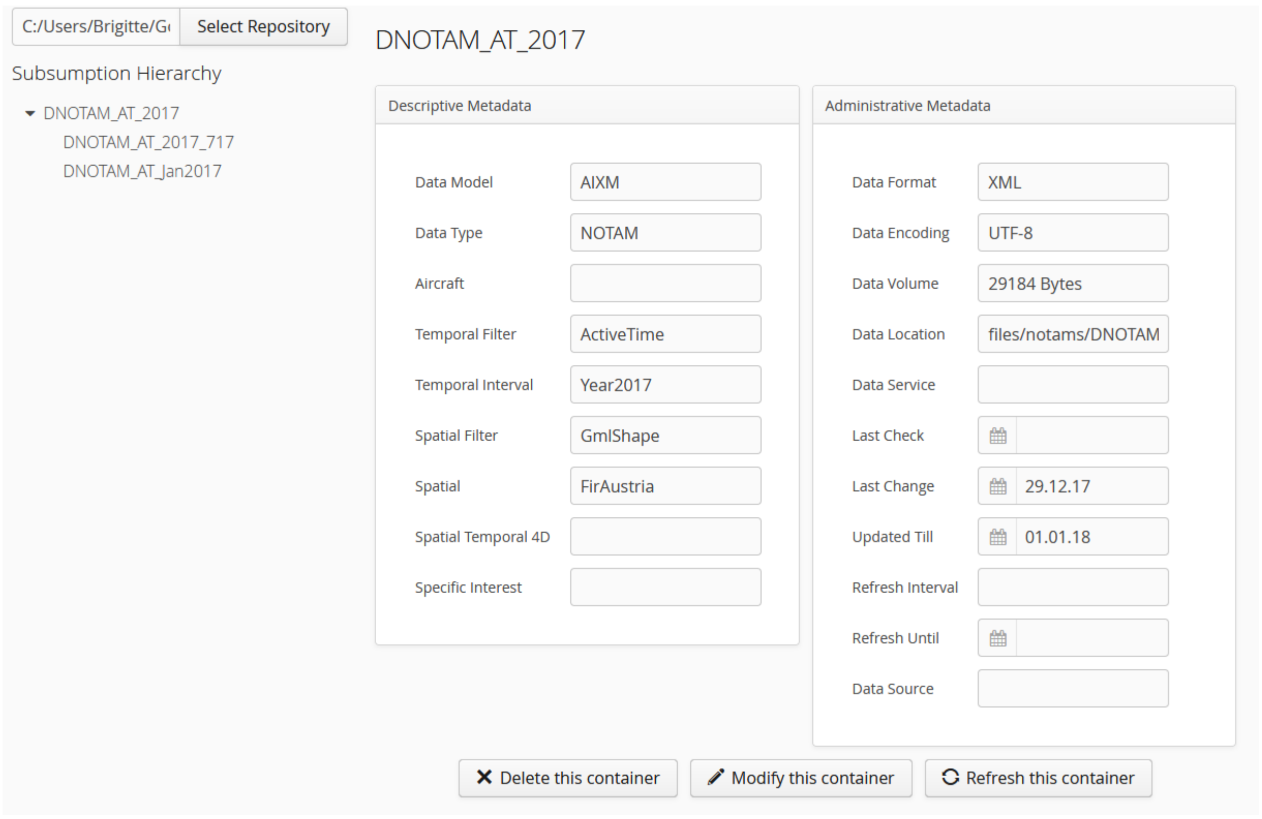
In the top of Figure 6.4, the selection of the repository can be made. However, as described in Section 3.4, in this thesis only one repository is available. Next to the repository, a filter possibility is provided to the user. The entered text does not only filter the data container’s name, but also all descriptive metadata. For example, when the user enters Year in this field, the table is filtered to those entries that have the temporal interval with the term Year in it, as shown in Figure 6.5.



Name	Data Model	Data Type	Aircraft	Temporal Filter	Temporal Interval
DNOTAM_AT_2017_717	AIXM	NOTAM	AircraftType717	ActiveTime	Year2017
DNOTAM_AT_2017	AIXM	NOTAM		ActiveTime	Year2017

Figure 6.5: Container Management Application – Filtered list of containers

By clicking on one of the entries in the table, the user can view the details of this data container. Therefore, the application switches to the tab *Container details* and loads all metadata of the ontology. Figure 6.6 shows the details of the data container *DNOTAM_AT_2017*, which is a primary data container. Furthermore, the user has three buttons to either delete, modify, or refresh the data container.


C:/Users/Brigitte/G | Select Repository

Subsumption Hierarchy

- DNOTAM_AT_2017
  - DNOTAM_AT_2017_717
  - DNOTAM_AT_Jan2017

DNOTAM_AT_2017

Descriptive Metadata

Data Model	AIXM
Data Type	NOTAM
Aircraft	
Temporal Filter	ActiveTime
Temporal Interval	Year2017
Spatial Filter	GmlShape
Spatial	FirAustria
Spatial Temporal 4D	
Specific Interest	

Administrative Metadata

Data Format	XML
Data Encoding	UTF-8
Data Volume	29184 Bytes
Data Location	files/notams/DNOTAM
Data Service	
Last Check	
Last Change	29.12.17
Updated Till	01.01.18
Refresh Interval	
Refresh Until	
Data Source	

Figure 6.6: Container Management Application – Container details

When the user clicks the delete button a message opens that warns the user that this data container may act as data source for other data containers. As shown in Figure 6.7, the user once more needs to click the delete button to permanently delete this data container.

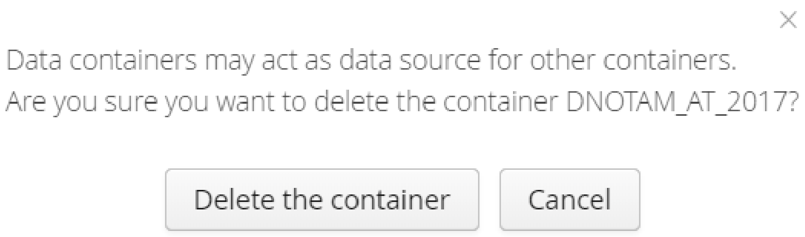
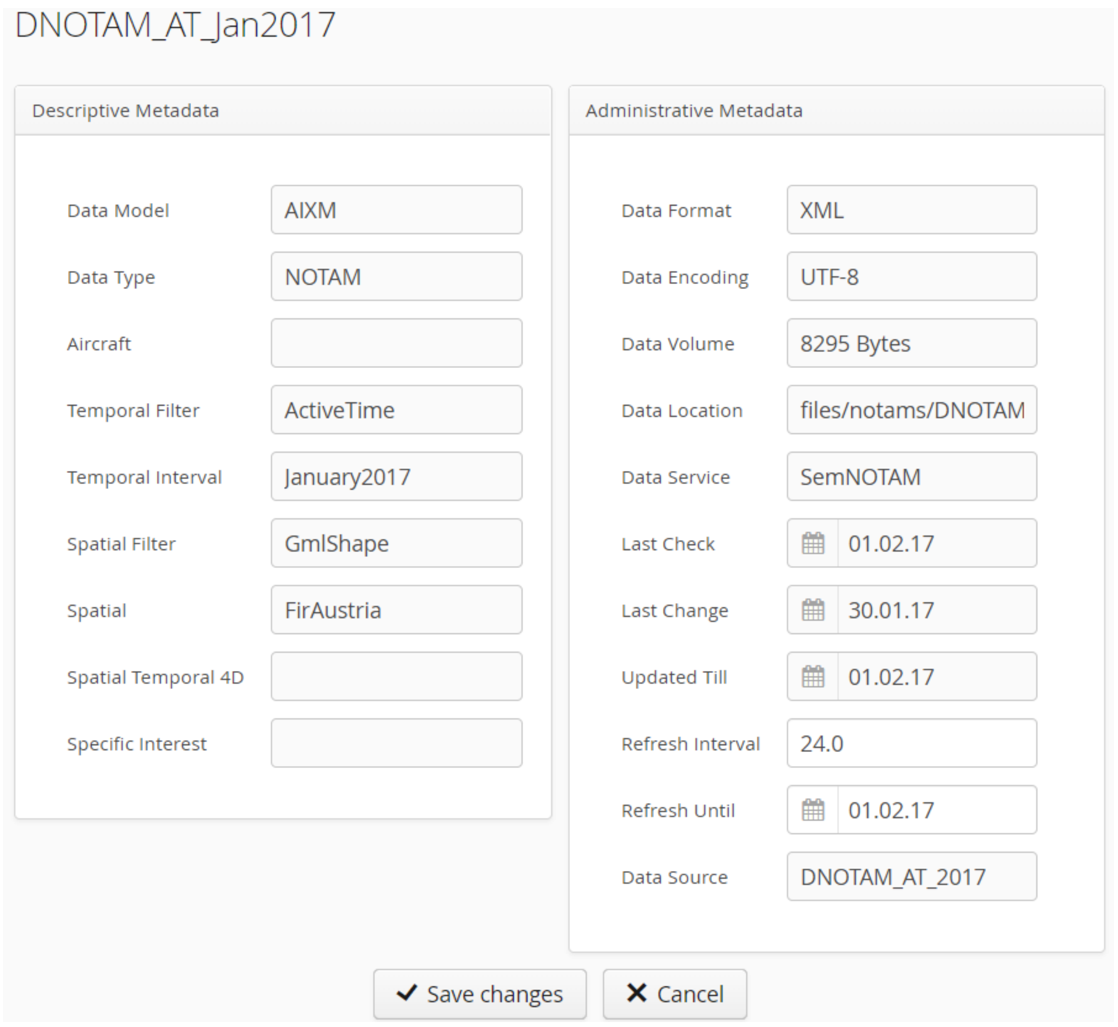

Data containers may act as data source for other containers.
Are you sure you want to delete the container DNOTAM_AT_2017?

Figure 6.7: Container Management Application – Delete a data container

The second button in Figure 6.6 shows, that the user can modify some of the information. As defined in Requirement 2.3.1, the user is able to modify the refresh interval and the refresh until timestamp. By clicking the modify button, the corresponding fields are enabled, and the user can change the values. Furthermore, the toolbar at the bottom changes, as shown in Figure 6.8, so that the user can either save or cancel the changes.


DNOTAM_AT_Jan2017

Descriptive Metadata

Data Model	AIXM
Data Type	NOTAM
Aircraft	
Temporal Filter	ActiveTime
Temporal Interval	January2017
Spatial Filter	GmlShape
Spatial	FirAustria
Spatial Temporal 4D	
Specific Interest	

Administrative Metadata

Data Format	XML
Data Encoding	UTF-8
Data Volume	8295 Bytes
Data Location	files/notams/DNOTAM
Data Service	SemNOTAM
Last Check	01.02.17
Last Change	30.01.17
Updated Till	01.02.17
Refresh Interval	24.0
Refresh Until	01.02.17
Data Source	DNOTAM_AT_2017

Figure 6.8: Container Management Application – Modify a data container

The last button allows the user to initiate a refresh of the selected data container. For primary data containers the user needs to upload the file with the new data items, and therefore, the upload window in Figure 6.9 is shown to the user.

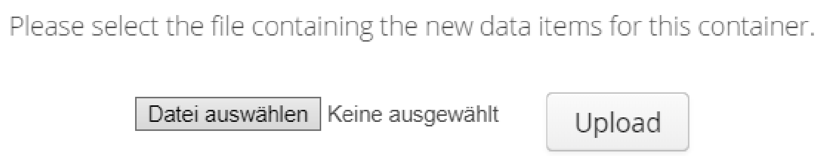

Please select the file containing the new data items for this container.

Figure 6.9: Container Management Application – Refresh a data container

When a secondary data container is refreshed, the corresponding service of the Container Description Service to build the interest specification is called and forwarded to the SemNOTAM Web Service. The SemNOTAM Web Service returns the new data set, which is stored in the repository. When the refresh of a data container, primary or secondary data container, was successful, the success message of Figure 6.10 is shown to the user.

The container has been successfully refreshed!

Figure 6.10: Container Management Application – Successful refresh

The left side of the details view (Figure 6.6) shows a hierarchical representation of all existing data containers. This subsumption hierarchy is independent of the defined data sources of the secondary data containers. It merely shows how data containers are subsumed by each other based on the descriptive metadata. The current subsumption hierarchy is shown in Figure 6.11. By clicking on one of the data containers in the subsumption hierarchy the details of this data container are loaded and presented.

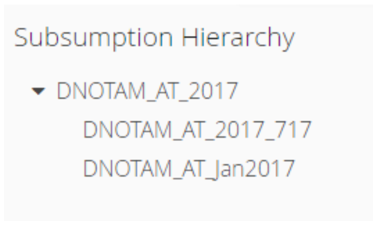


Figure 6.11: Container Management Application – Subsumption hierarchy

The last tab *Create a new container* provides the user the functionality to create new data containers, primary as well as secondary data containers. On the left side, again the hierarchical view of the existing data containers is shown. The remaining part of the tab *Create a new container* is separated in descriptive and administrative metadata. For the descriptive metadata a concept can be chosen for each facet, whereas for the administrative metadata three text fields and one date field can be filled. The structure of this page is shown in Figure 6.12.

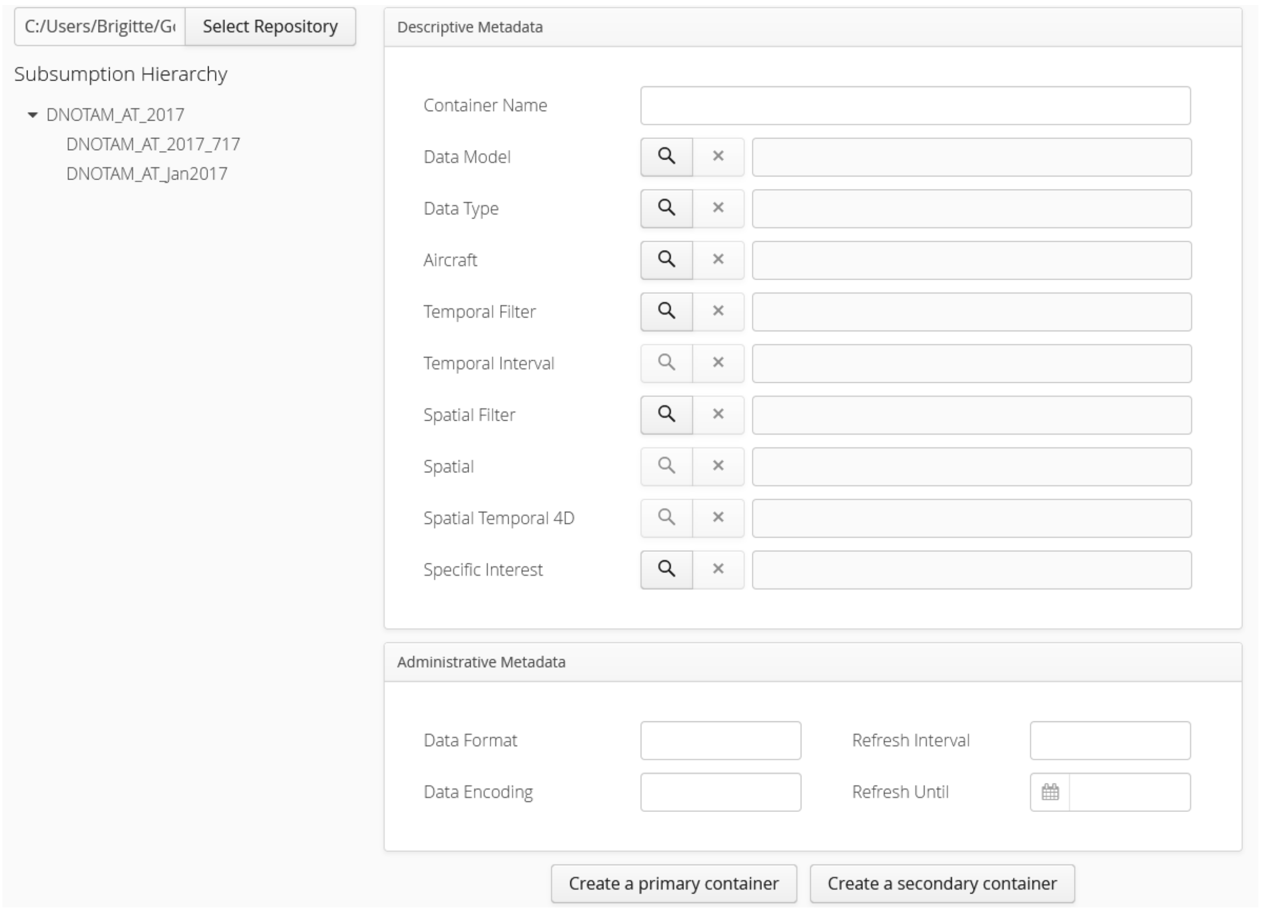
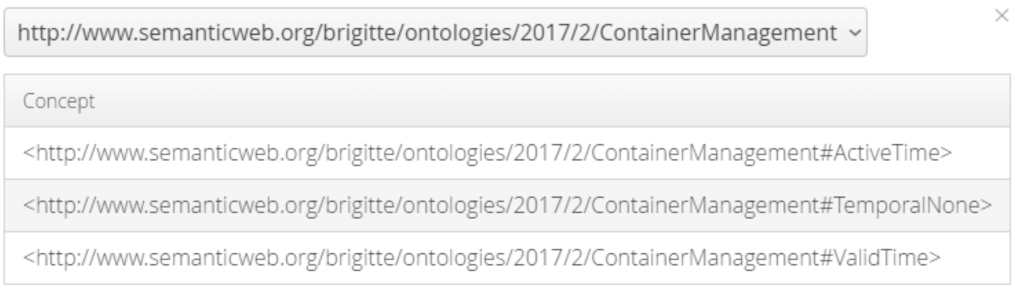


Figure 6.12: Container Management Application – Create a new container

In the part with the descriptive metadata of Figure 6.12 the name for the new data container has to be specified. For each facet a search button is available and by clicking one of these buttons the *getConceptForFacet* method of the Container Description Service is called. A separate window opens, where the user can choose one of the existing concepts for the specific facet. For example, when the user clicks on the search button of the temporal filter facet the list of available concepts is loaded as shown in Figure 6.13.

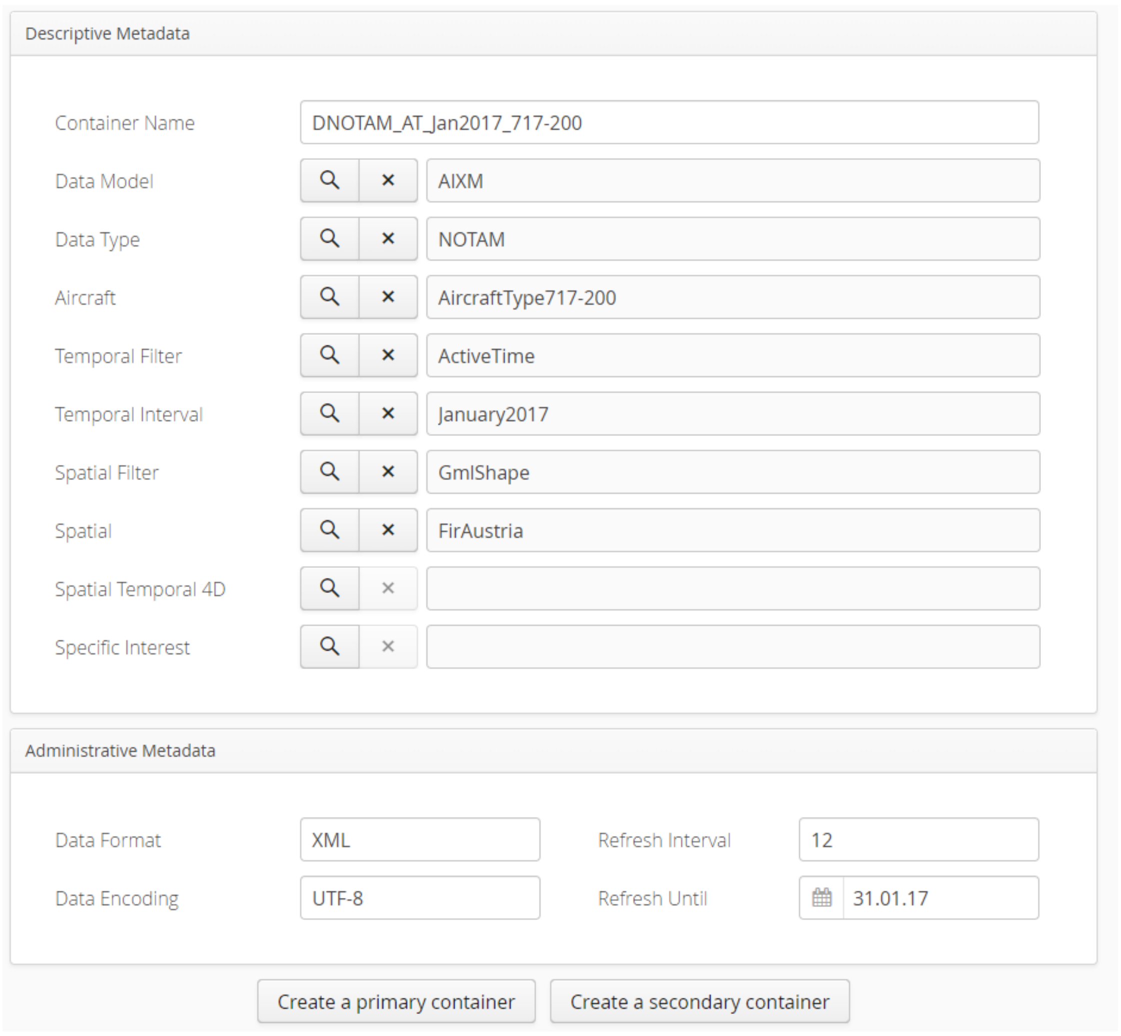


http://www.semanticweb.org/brigitte/ontologies/2017/2/ContainerManagement

Concept
<http://www.semanticweb.org/brigitte/ontologies/2017/2/ContainerManagement#ActiveTime>
<http://www.semanticweb.org/brigitte/ontologies/2017/2/ContainerManagement#TemporalNone>
<http://www.semanticweb.org/brigitte/ontologies/2017/2/ContainerManagement#ValidTime>

Figure 6.13: Container Management Application – Concepts for temporal filter facet

When the user has selected concepts for the relevant facets of the new data container, the creation view shows all the selected concepts. Figure 6.14 shows the filled form for the creation of the data container *DNOTAM_AT_Jan2017_717-200*. For the relevant facets, a specific concept is chosen, and the administrative metadata is filled.



Descriptive Metadata

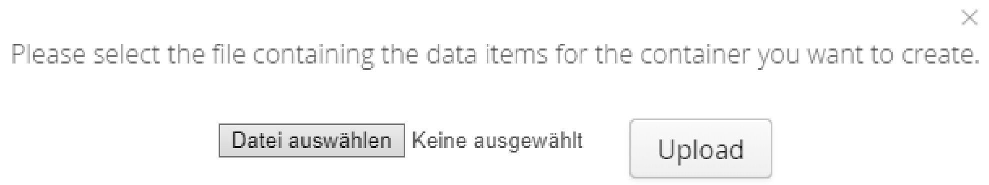
Container Name	DNOTAM_AT_Jan2017_717-200
Data Model	AIXM
Data Type	NOTAM
Aircraft	AircraftType717-200
Temporal Filter	ActiveTime
Temporal Interval	January2017
Spatial Filter	GmlShape
Spatial	FirAustria
Spatial Temporal 4D	
Specific Interest	

Administrative Metadata

Data Format	XML	Refresh Interval	12
Data Encoding	UTF-8	Refresh Until	31.01.17

Figure 6.14: Container Management Application – DNOTAM_AT_Jan2017_717-200

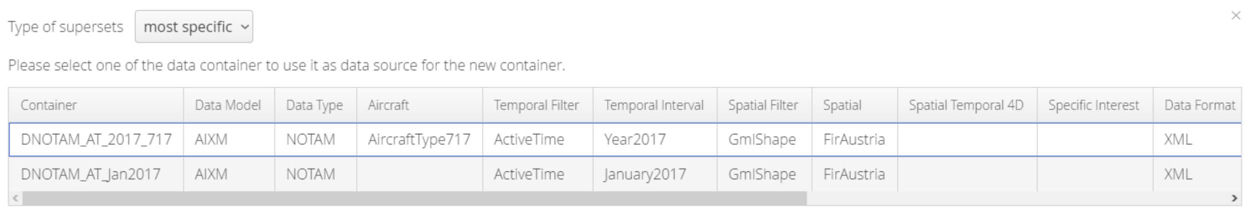
Figure 6.14 shows at the bottom, that the user can decide, whether the new data container is a primary data container or a secondary data container. When clicking on the button for creating a primary data container, a window for uploading the data set opens, shown in Figure 6.15.



Please select the file containing the data items for the container you want to create.

Figure 6.15: Container Management Application – Create a primary container

Whereas, when the user wants to create a secondary data container, a list of possible data source containers is provided to the user. In the top left corner of Figure 6.16 the user can switch between the most specific supersets and all possible supersets. The table shows all information of the data containers, the descriptive and administrative metadata, as the refresh interval and freshness timestamps are most relevant for the decision of the data source. In this case, two data containers are shown in the list of most specific data containers. For the data container *DNOTAM_AT_2017_717* only the temporal interval facet differs from the new data container, whereas for the data container *DNOTAM_AT_Jan2017* only the aircraft facet differs from the new data container. Therefore, the user can decide which of these data containers shall be used as data source.



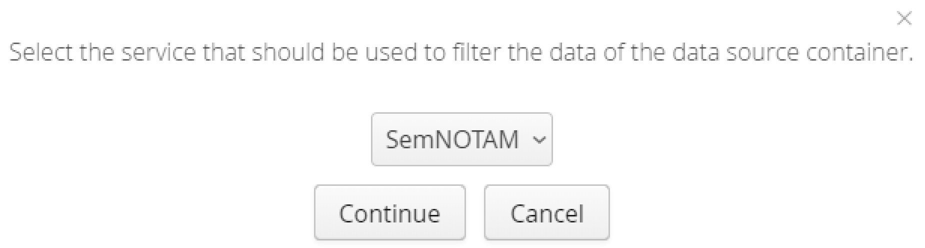
Type of supersets: most specific

Please select one of the data container to use it as data source for the new container.

Container	Data Model	Data Type	Aircraft	Temporal Filter	Temporal Interval	Spatial Filter	Spatial	Spatial Temporal 4D	Specific Interest	Data Format
DNOTAM_AT_2017_717	AIXM	NOTAM	AircraftType717	ActiveTime	Year2017	GmlShape	FirAustria			XML
DNOTAM_AT_Jan2017	AIXM	NOTAM		ActiveTime	January2017	GmlShape	FirAustria			XML

Figure 6.16: Container Management Application – Create a secondary container

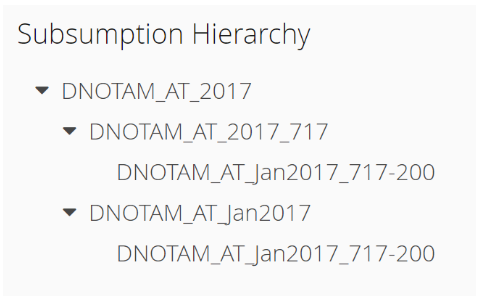
After the user has chosen one of the data containers by clicking on it, a new window opens where the user can select the service that is used for filtering. As exemplified shown in Figure 6.17, the user selects SemNOTAM as a service. By clicking the button *Continue* the service is called and the resulting data set is stored in the repository.



Select the service that should be used to filter the data of the data source container.

Figure 6.17: Container Management Application – Select service

When the creation of the new data container was successful, a success message is shown to the user and the subsumption hierarchy is refreshed. As mentioned before, the subsumption hierarchy does not show the data source relationships, but the superclass/subclass relationship based on the descriptive metadata. After creating the data container of Figure 6.14 the new subsumption hierarchy has two new entries for this data container. The new hierarchy is shown in Figure 6.18.


Subsumption Hierarchy

- DNOTAM_AT_2017
  - DNOTAM_AT_2017_717
    - DNOTAM_AT_Jan2017_717-200
  - DNOTAM_AT_Jan2017
    - DNOTAM_AT_Jan2017_717-200

Figure 6.18: Container Management Application – Updated subsumption hierarchy

To recap, the Container Management Application is an application for producers that want to share their information, combined in data containers in the SWIM environment. These data containers are then available for each consumer that wants to fulfill a specific task. The process of how the consumer gets the relevant information and the integration with a briefing application, i.e., a task-based retrieval service, is demonstrated in the next section.

6.3 Task-Based Retrieval Service

The prototype developed in this thesis can be integrated with a briefing application, such as the SemNOTAM system, to provide the consumers the possibility to retrieve all the relevant information for a specific task. Therefore, a use case flight from Vienna to Frankfurt is introduced and the process from describing the task of the consumer to the retrieving of the set of relevant DNOTAMs is demonstrated.

As described before, the consumer can use the SemNOTAM Briefing Application to describe his/her interest, respectively the task. In this use case a pilot needs to fly from Vienna to Frankfurt on June 17th, 2019 with an A380. For the pilot briefing, he/she wants to retrieve all relevant DNOTAMs concerning this task.

In the briefing application the user has to define a general interest based on the available vocabulary, i.e., the Container Ontology defined in Section 6.1. Additionally, the specific flight plan can be described in a specific interest. Therefore, the time frame for the flight, each segment of the flight path, and the used aircraft can be specified. This part needs to be implemented in the briefing application and is not part of this thesis.

The first communication between the briefing application and the Container Description Service happens, when the user wants to choose a concept for one of the facets. Therefore, the application calls the *selectConceptForFacet* service, which returns the list of all possible concepts. By using the same ontological concepts as producers do in the Container Management Application (Section 6.2), the creation of the subsumption hierarchy and hence, retrieving the most specific superset of DNOTAMs is possible.

For this use case flight from Vienna to Frankfurt the temporal concept Year2019, which was introduced in Section 6.1, can be used. As the flight is done with an A380 this Aircraft type shall be used in the aircraft facet. The concept for the spatial facet should at least include Austria, Czech, and Germany, as the flight crosses each of these flight information regions. With the filter facets the consumer needs to define on which level the temporal and spatial filtering shall be performed. In this case, we want to have the most precise filtering based on the active time and the actual shape. For the facets data model and data type the concepts AIXM and DNOTAM have to be used. The concrete flight plan is stored in the specific interest facet. This only has effect on the subsumption result if there already exists a data container for exactly this flight.

When the user has specified his/her task with the ontological concepts the *getInterestForConcept* method of the Container Description Service is called, to build the interest specification, i.e. the task description. Afterwards the interest specification is forwarded to the evaluation method of the SemNOTAM Web Service, which returns the evaluated interest specification with all relevant DNOTAMs. In this evaluation method the *getMostSpecificSuperset* method of the Container Description Service is called which evaluates the combination of the concepts to find the most specific data container. This data container is used as input in the filtering method of the SemNOTAM Web Service to evaluate the user's task. Figure 6.19 shows the process of the task-based retrieval with the SemNOTAM Briefing Application.

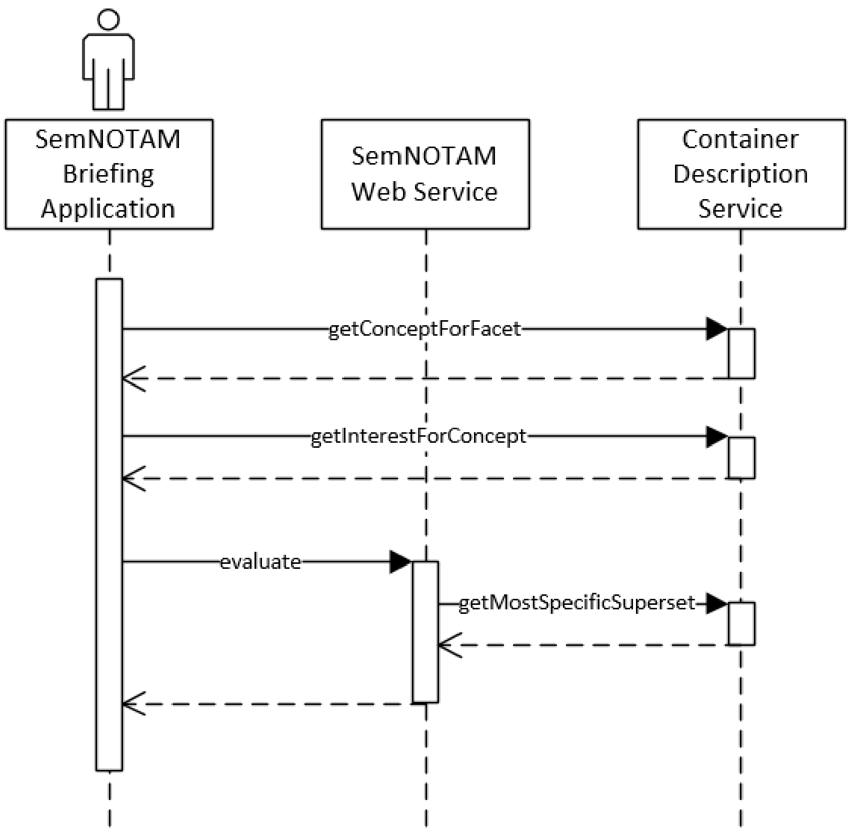


Figure 6.19: Task-Based Retrieval

The consumer is interacting with the SemNOTAM Briefing Application to describe the task for which he/she wants to retrieve the relevant information. In Figure 6.19 the process of this task-based retrieval is shown, where the briefing application first communicates with the Container Description Service. This ensures that the consumer uses the same concepts for the description of his/her task. Afterwards the evaluation method of the SemNOTAM Web Service is called, which returns the relevant information to the user. The retrieval of the most specific superset data container is only relevant for the filtering process to minimize the data set. In the end, the consumer retrieves the evaluated interest specification with all relevant DNOTAMs and their annotations.

7 Conclusion

Due to the increasing number of DNOTAMs, which results in an information overload, an intelligent and fine-grained filtering of DNOTAMs is an important aspect of air traffic safety. The SemNOTAM system addresses that need. This thesis supplements the system by the specification of an ontology, that is needed to ensure a common understanding in the SWIM environment. Furthermore, a web application for managing the data containers of this environment is implemented, which allows producers to provide data containers, that are described by a semantic label referring to concepts of the shared ontology. For the integration of a briefing application and the SemNOTAM system, three services, that enable linking to the ontology and find the most specific superset based on the metadata, are provided. One service allows the user to define his/her task (*selectConceptForFacet*), the second service is needed to build the corresponding interest specification (*getInterestFromConcept*), and the last service allows to find the most specific superset for the given task (*getMostSpecificSuperset*).

Besides giving a theoretical background, this thesis analyzes and defines requirements for the three implementation parts based on the task description. For the Container Ontology the requirements for the concept of data containers, descriptive metadata and administrative metadata are extracted. The requirements for the Container Management Application are structured based on the functionality it provides to the user. The requirements for the Container Description Service concern information about the inputs and outputs of the three services. The identified requirements are used to develop an approach for building the ontology and the applications.

This thesis covers the implementation of an ontology for managing a vocabulary to describe data containers by semantic labels, a web application for managing these data containers, and services that serve this application and enable task-based retrieval of DNOTAMs by briefing application. Important aspects of the implementation, such as the refreshing of data containers, or the selection of a superset data container, are described in more detail. Furthermore, the usage of the Container Ontology and the Container Management Application is demonstrated. The management of concepts in the facets and the definition of new data properties are covered by the ontology. All the functionality of the Container Management Application is demonstrated by using simple concepts and small data containers concerning Europe, and especially Austria.

Concluding, all identified requirements, except the already mentioned restrictions, are implemented. The implemented prototype covers an important aspect for the ATM communication in combination with the SWIM vision and helps to overcome the information overload in pilot briefings. Future work can be conducted to implement the distinction between public and private repositories as defined by Requirement 2.2.8. Moreover, other filtering services can be included in the Container Management Application, to allow the user to filter the superset's data set with additional services.

Bibliography

[1] EUROCONTROL, “EUROCONTROL Seven-Year Forecast February 2017: Flight Movements and Service Units 2017-2023,” Feb. 2017. http://www.eurocontrol.int/sites/default/files/content/documents/official-documents/forecasts/seven-year-flights-service-units-forecast-2017-2023-Feb2017-1.pdf (accessed Jan. 02, 2018).
[2] EUROCONTROL, “EUROCONTROL Seven-Year Forecast September 2017: Flight Movements and Service Units 2017-2023,” Sep. 2017. http://www.eurocontrol.int/sites/default/files/content/documents/official-documents/forecasts/seven-year-flights-service-units-forecast-2017-2023-Sep2017.pdf (accessed Jan. 02, 2018).
[3] “What is air traffic management? | Eurocontrol.” https://www.eurocontrol.int/articles/what-air-traffic-management (accessed Nov. 26, 2017).
[4] “Digital NOTAM (Phase 3 P-21) | Eurocontrol.” http://www.eurocontrol.int/articles/digital-notam-phase-3-p-21 (accessed Nov. 26, 2017).
[5] EUROCONTROL, “Digital Notam - The Digital Age,” Jun. 2010. http://www.eurocontrol.int/sites/default/files/publication/files/20100601-digitalnotam-brochure.pdf (accessed Jan. 07, 2018).
[6] EUROCONTROL and Federal Aviation Administration, “Digital Notam Event Specification,” Oct. 2010. http://www.eurocontrol.int/sites/default/files/content/documents/information-management/20101010-digitalnotam-event-specification-increment1.pdf (accessed Jan. 07, 2018).
[7] J. Meserole and J. Moore, “What is System Wide Information Management (SWIM)?,” *IEEE Aerosp. Electron. Syst. Mag.*, vol. 22, no. 5, pp. 13–19, 2007, doi: 10.1109/MAES.2007.365329.
[8] “System Wide Information Management (SWIM) | Eurocontrol.” https://www.eurocontrol.int/swim (accessed Nov. 26, 2017).
[9] I. Kovacic *et al.*, “Ontology-based data description and discovery in a SWIM environment,” Apr. 2017, p. 5A4-1-5A4-13, doi: 10.1109/ICNSURV.2017.8011928.
[10] “SemNOTAM: Ontology-based representation and semantic querying of Digital Notices to Airman.” http://dke.jku.at/research/projects/details.xq?name=SemNotam (accessed Mar. 25, 2018).
[11] Federal Aviation Administration, “Web Service Description Document - Federal NOTAM Service (FNS) - NOTAM Distribution Service (NDS),” 2003. http://notamdemo.aim.nas.faa.gov/fnshelp/AIMMS1_FNS_NDS_WSDD.pdf.
[12] F. Burgstaller, D. Steiner, M. Schrefl, E. Gringinger, S. Wilson, and S. van der Stricht, “AIRM-based, fine-grained semantic filtering of notices to airmen,” Apr. 2015, pp. D3-1-D3-13, doi: 10.1109/ICNSURV.2015.7121222.
[13] D. Steiner, I. Kovacic, F. Burgstaller, M. Schrefl, T. Friesacher, and E. Gringinger, “Semantic enrichment of DNOTAMs to reduce information overload in pilot briefings,” Apr. 2016, p. 6B2-1-6B2-13, doi: 10.1109/ICNSURV.2016.7486359.
[14] “Extensible Markup Language (XML).” https://www.w3.org/XML/ (accessed Jul. 26, 2018).
[15] “BEST - SESAR.” http://www.project-best.eu/ (accessed Mar. 25, 2018).
[16] E. Gringinger *et al.*, “The Semantic Container Approach: Techniques for ontology-based data description and discovery in a decentralized SWIM knowledge base,” presented at the Proceedings of the SESAR Innovation Days 2018 (SID 2018), Salzburg, Austria, 2018, Accessed: May 18, 2019. [Online]. Available: https://www.sesarju.eu/sites/default/files/documents/sid/2018/papers/SIDs_2018_paper_78.pdf.
[17] B. Neumayr, E. Gringinger, C. G. Schuetz, M. Schrefl, S. Wilson, and A. Vennesland, “Semantic data containers for realizing the full potential of system wide information management,” Sep. 2017, pp. 1–10, doi: 10.1109/DASC.2017.8102002.
[18] BEST Consortium, “BEST - Final Project Results.” 2016, [Online]. Available: http://project-best.eu/downloads/D6.3%20Final%20Report.pdf.
[19] R. Studer, V. R. Benjamins, and D. Fensel, “Knowledge engineering: Principles and methods,” *Data Knowl. Eng.*, vol. 25, no. 1, pp. 161–197, Mar. 1998, doi: 10.1016/S0169-023X(97)00056-6.
[20] T. R. Gruber, “Toward principles for the design of ontologies used for knowledge sharing?,” *Int. J. Hum.-Comput. Stud.*, vol. 43, no. 5, pp. 907–928, 1995.

[21] "Digital NOTAM Specification - Digital NOTAM - AIXM Confluence." https://ext.eurocontrol.int/aixm_confluence/display/DNOTAM/Digital+NOTAM+Specification (accessed Nov. 26, 2017).
[22] "AIXM." http://www.aixm.aero/ (accessed Nov. 26, 2017).
[23] EUROCONTROL and Federal Aviation Administration, "AIXM 5 - Temporality Model," 2010. http://aixm.aero/sites/aixm.aero/files/imce/AIXM51/aixm_temporality_1.0.pdf.
[24] "Geography Markup Language | OGC." http://www.opengeospatial.org/standards/gml (accessed Jul. 26, 2018).
[25] M. Endsley, "Endsley, M.R.: Toward a Theory of Situation Awareness in Dynamic Systems. Human Factors Journal 37(1), 32-64," *Hum. Factors J. Hum. Factors Ergon. Soc.*, vol. 37, pp. 32–64, Mar. 1995, doi: 10.1518/001872095779049543.
[26] EUROCONTROL and Federal Aviation Administration, "Aeronautical Information Exchange Model (AIXM) - Exchange model goals, requirements and design," 2006. http://aixm.aero/sites/aixm.aero/files/imce/AIXM50/aixm_5_proposal_20060620_whitepaper_.pdf.
[27] S. Kendal and M. Creen, *An introduction to knowledge engineering*. Springer-Verlag London, 2007.
[28] C. Beierle and G. Kern-Isberner, *Methoden wissensbasierter Systeme: Grundlagen Algorithmen Anwendungen*, 3rd ed. Friedr. Vieweg & Sohn Verlagsgesellschaft, 2006.
[29] "Ontologien — Enzyklopaedie der Wirtschaftsinformatik." http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/daten-wissen/Wissensmanagement/Wissensmodellierung/Wissensreprasentation/Semantisches-Netz/Ontologien (accessed Mar. 30, 2017).
[30] "Ontology," *Philosophy Terms*, Oct. 30, 2016. https://philosophyterms.com/ontology/ (accessed May 03, 2019).
[31] S. Staab and R. Studer, Eds., *Handbook on Ontologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.
[32] J. Sequeda, "Introduction to: Open World Assumption vs Closed World Assumption," *DATAVERSITY*, Nov. 30, 2012. https://www.dataversity.net/introduction-to-open-world-assumption-vs-closed-world-assumption/ (accessed May 18, 2019).
[33] Stefano Ceri, Georg Gottlob, and Letizia Tanca, "What you Always Wanted to Know About Datalog (And Never Dared to Ask).," *ResearchGate*. https://www.researchgate.net/publication/3296132_What_you_Always_Wanted_to_Know_About_Datalog_And_Never_Dared_to_Ask (accessed Oct. 18, 2020).
[34] C. Hartshorne, P. Weiss, and A. W. Burks, *Collected Papers of Charles Sanders Peirce*, vol. 1. Harvard University Press Cambridge, 1931.
[35] "Definition: nicht monotones Schließen," *Gabler Wirtschaftslexikon*. https://wirtschaftslexikon.gabler.de/definition/nicht-monotones-schliessen-39223/version-262637 (accessed May 16, 2019).
[36] "Definition: monotones Schließen," *Gabler Wirtschaftslexikon*. https://wirtschaftslexikon.gabler.de/definition/monotones-schliessen-39447/version-262855 (accessed May 16, 2019).
[37] "WIN DKE." http://www.dke.uni-linz.ac.at/index.html (accessed Jun. 01, 2019).
[38] "Frequentis AG," *Frequentis.com*. https://www.frequentis.com/de (accessed Jun. 01, 2019).
[39] E. Gringinger, D. Eier, and D. Merkl, "NextGen and SESAR moving towards ontology-based software development," in *2011 Integrated Communications, Navigation, and Surveillance Conference Proceedings*, May 2011, pp. H3-1-H3-10, doi: 10.1109/ICNSURV.2011.5935286.
[40] R. Bobrow, "Intelligent Semantic Query of Notices to Airmen (NOTAMs)," BBN TECHNOLOGIES CAMBRIDGE MA, 2006.
[41] N. Zimmer, J. Schiefele, K. Bayram, T. Hankers, S. Frank, and T. Feuerle, "Rule-based NOTAM Weather notification," in *2011 Integrated Communications, Navigation, and Surveillance Conference Proceedings*, May 2011, pp. O1-1-O1-9, doi: 10.1109/ICNSURV.2011.5935352.
[42] BEST Consortium, "BEST - Project Summary." 2016, [Online]. Available: http://project-best.eu/downloads/The_BEST_project_summary.pdf.
[43] M. Fernández-López, A. Gómez-Pérez, and N. Juristo, "Methontology: from ontological art towards ontological engineering," 1997, Accessed: Jan. 09, 2017. [Online]. Available: http://oa.upm.es/5484/1/METHONTOLOGY_.pdf.
[44] "Punning - OWL." https://www.w3.org/2007/OWL/wiki/Punning (accessed Mar. 30, 2017).

[45] “Web Application - Design Patterns.” http://researchhubs.com/post/computing/web-application/design-pattern.html (accessed Sep. 30, 2018).
[46] “Model View Controller Pattern Definition & Erklärung | Datenbank Lexikon,” *Datenbanken - für Anfänger und Profis*. http://www.datenbanken-verstehen.de/lexikon/model-view-controller-pattern/ (accessed Sep. 30, 2018).
[47] “Vaadin - Framework,” *Vaadin*. https://vaadin.com/framework (accessed Nov. 17, 2017).
[48] “Advanced Application Architectures | Vaadin Framework 8 | Vaadin 8 Docs,” *Vaadin*. https://vaadin.com/docs/v8/framework/introduction/intro-goals.html (accessed May 02, 2019).
[49] “The ArchiMate® Enterprise Architecture Modeling Language | The Open Group.” http://www.opengroup.org/subjectareas/enterprise/archimate-overview (accessed Jul. 26, 2018).
[50] “Enterprise Architecture | The Open Group.” http://www.opengroup.org/subjectareas/enterprise (accessed Jun. 25, 2018).
[51] M. Lankhorst, *Enterprise Architecture at Work: Modelling, Communication and Analysis*, 3rd ed. Berlin Heidelberg: Springer-Verlag, 2013.
[52] The Open Group, “ArchiMate Specification.” Jun. 2016.
[53] “OWL - Semantic Web Standards.” https://www.w3.org/OWL/ (accessed Apr. 27, 2017).
[54] “protégé.” http://protege.stanford.edu/ (accessed Apr. 27, 2017).
[55] “Erfahren Sie mehr über die Java-Technologie.” https://java.com/de/about/ (accessed Apr. 30, 2019).
[56] “Overview | Vaadin Framework 8 | Vaadin 8 Docs,” *Vaadin*. https://vaadin.com/docs/v8/framework/introduction/intro-overview.html (accessed May 02, 2019).
[57] “Technological Background | Vaadin Framework 8 | Vaadin 8 Docs,” *Vaadin*. https://vaadin.com/docs/v8/framework/introduction/intro-goals.html (accessed May 02, 2019).
[58] “Themes | Vaadin Framework 8 | Vaadin 8 Docs,” *Vaadin*. https://vaadin.com/docs/v8/framework/themes/themes-overview.html (accessed May 02, 2019).
[59] “Apache CXF -- Index.” http://cxf.apache.org/ (accessed Nov. 17, 2017).
[60] “JAX-WS.” https://javaee.github.io/metro-jax-ws/ (accessed May 02, 2019).
[61] “Building Web Services with JAX-WS - The Java EE 6 Tutorial.” https://docs.oracle.com/javaee/6/tutorial/doc/bnayl.html (accessed May 02, 2019).
[62] “TabSheet | Vaadin Framework 8 | Vaadin 8 Docs,” *Vaadin*. https://vaadin.com/docs/v8/framework/layout/layout-tabsheet.html (accessed Nov. 02, 2020).

List of Figures

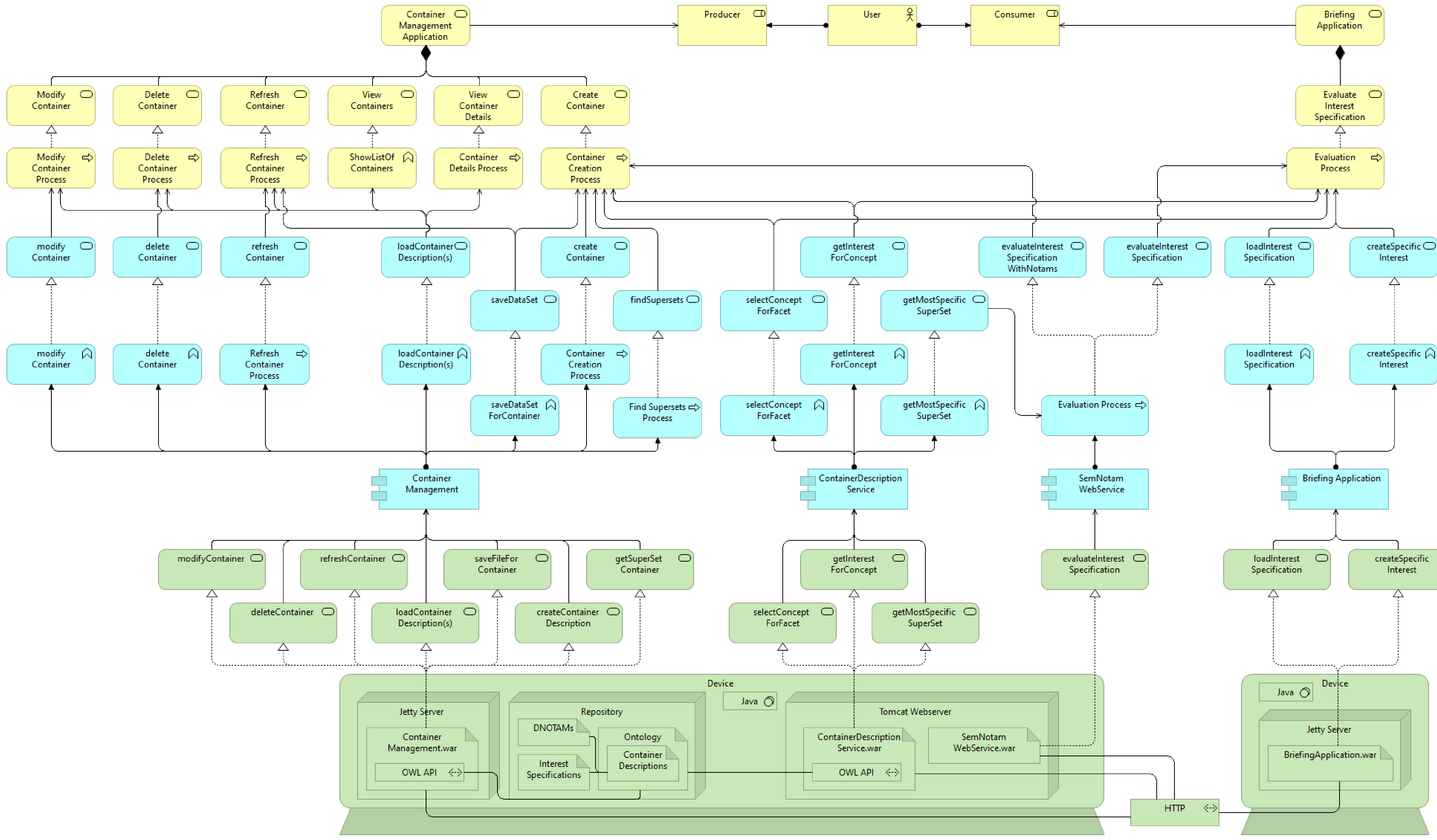
List of Tables

List of Listings

Acronyms

Acronym	Meaning
ATM	Air Traffic Management
NOTAM	Notice to Airmen
SWIM	System Wide Information Management
BEST	Achieving the BEnefits of SWIM by making smart use of Semantic Technologies
EUROCONTROL	European Organization for the Safety of Air Navigation
ATC	Air Traffic Control
ATFM	Air Traffic Flow Management
AIS	Aeronautical Information Services
AIM	Aeronautical Information Management
DNOTAM	Digital NOTAM
AIXM	Aeronautical Information Exchange Model
FAA	Federal Aviation Administration
GML	Geography Markup Language
FNS-NDS	Federal NOTAM Service and NOTAM Distribution Service
XML	eXtensible Markup Language
OWL	Web Ontology Language
CWA	Closed World Assumption
OWA	Open World Assumption
SESAR	Single European Sky ATM Research Program
NextGen	Next Generation Air Transportation System
AIRM	ATM Information Reference Model
METAR	METeorological Aerodrome Report
IRI	Internationalized Resource Identifier
API	Application Programming Interface
MVC	Model-View-Controller
MVP	Model-View-Presenter
HTTP	Hypertext Transfer Protocol
W3C	World Wide Web Consortium
RDF	Resource Description Framework
HTML	Hypertext Markup Language
CSS	Cascading Style Sheet
JAX-WS	Java API for XML Web Services
SOAP	Simple Object Access Protocol
WXXM	Weather Information Exchange Models and Schema
UI	User Interface
URI	Unified Resource Identifier

Appendix


Container Management Application
Producer
User
Consumer
Briefing Application
Modify Container
Delete Container
Refresh Container
View Containers
View Container Details
Create Container
Evaluate Interest Specification
Modify Container Process
Delete Container Process
Refresh Container Process
ShowListOf Containers
Container Details Process
Container Creation Process
Evaluation Process
modify Container
delete Container
refresh Container
loadContainer Description(s)
create Container
getInterest ForConcept
evaluateInterest Specification WithNotams
evaluateInterest Specification
loadInterest Specification
createSpecific Interest
saveDataSet
findSupersets
selectConcept ForFacet
getMostSpecific SuperSet
modify Container
delete Container
Refresh Container Process
loadContainer Description(s)
Container Creation Process
getInterest ForConcept
loadInterest Specification
createSpecific Interest
saveDataSet ForContainer
Find Supersets Process
selectConcept ForFacet
getMostSpecific SuperSet
Evaluation Process
Container Management
ContainerDescription Service
SemNotam WebService
Briefing Application
modifyContainer
refreshContainer
saveFileFor Container
getSuperSet Container
deleteContainer
loadContainer Description(s)
createContainer Description
getInterest ForConcept
selectConcept ForFacet
getMostSpecific SuperSet
evaluateInterest Specification
loadInterest Specification
createSpecific Interest
Device
Java
Jetty Server
Container Management.war
OWL API
Repository
DNOTAMs
Ontology
Interest Specifications
Container Descriptions
Tomcat Webserver
ContainerDescription Service.war
SemNotam WebService.war
OWL API
HTTP
Device
Java
Jetty Server
BriefingApplication.war