Diplomarbeit

Communication Protocols in XVSM-Design and Implementation

Ausgeführt am
Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
Ao. Univ. Prof. Dipl.-Ing. Dr. eva Kühn

durch
Severin Ecker
Tigergasse 11/16
A-1080 Wien
Matr.Nr.: 9925546

Wien, August 2007 ----------

Kurzfassung

In vernetzten Umgebungen, im Speziellen sind hier space basierte Anwendungen zu verstehen, existiert eine Vielzahl an unterschiedlichen interagierenden Geräten. Beispiele hierfür wären der weit verbreitete Desktop-PC, große Server aber auch mobile Geräte wie Handhelds und Vertreter der neuen Generationen von Mobiltelefonen. Diese sind oft auf verschiedensten Technologien aufgebaut und stellen unterschiedliche Ansprüche sowohl an die jeweiligen Programmierparadigmen als auch an die verwendeten Werkzeuge. Einfach eine Middleware zu verwenden, gibt insofern keine ausreichende Hilfestellung da sie die Probleme der Internetfähigkeit und der Heterogenität der Endverbrauchergeräte nicht oder nur teilweise löst. Üblicherweise müssen spezielle Programmiersprachen benutzt und eigene Hardware Schnittstellengeräte des jeweiligen Endsystems berücksichtigt werden.

Die Kommunikation in XVSM (erweiterbares virtuelles shared Memory) wurde also offenes Protokoll und sprachunabhängig spezifiziert unter Verwendung des öffentlichen und freien Standards für XML Schema.

Dies ermöglicht eine Entkopplung der XVSM Core (XCore) Implementierung von der jeweiligen Anwendung und den Restriktionen des Endgeräts. Dadurch resultiert das XVSM System in einer leichter zu verwendenden und besser integrierbaren space basierten Middleware.

Diese Diplomarbeit befasst sich sowohl mit dem Design und der Implementierung der MozartSpaces (eine open source Java Implementierung von XVSM) als auch mit der Entwicklung und der Integration des XML Protokolls.

Abstract

In a distributed environment especially in space-based computing, a large number of heterogeneous devices interact together. Examples for such devices are the widespread and well known desktop PC, servers but also mobile devices such as handhelds or smart phones. These are based on different technologies and usually pose different requirements on the programming paradigms and tools. Simply picking a middleware only provides limited support for overcoming the problems of needed network support and the heterogeneity of end user devices. Usually a specific kind of programming language, hardware interfacing device, or software technology must be available on the end user device.

The communication in XVSM (extensible virtual shared memory) is defined as open and language independent protocol. This protocol is specified by using the publicly available and well-established standard XML Schema.

This allows the decoupling of the XVSM core (XCore) implementation from the client application and the end user device characteristics, which results in an easier to use, and more adaptable space-based middleware.

This thesis discusses the design and implementation of the MozartSpaces (an open-source Java based implementation of the XVSM) and the development of the XML Communication Protocol.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. eva Kühn who made this diploma thesis possible. She is an everlasting source of interesting and ingenious ideas and an inspiration to my own work.

I also want to express my gratitude to my colleagues at the complang institute, most of all Richard Mordinyi whose constructive criticism helped me shape and structure the project documents and in the end this thesis itself. Also he never complained when I was in a bad mood or things didn't work out the way I intended them.

Additionally my thanks go out to my friends who have helped me to stay focused and continue my work. Especially I want to thank Peter Vogl who sparked my interest in computer science and Benjamin Roch who never grew tired in helping me with bureaucracy issues.

Last but not least I want to thank the Tigerbande members. They have endured my moods without complaints and wouldn't stop trying to drag me away from my work in order to party.

Table of Content

Figure List

Listings

I. Introduction

I.1. Motivation

More than a decade after the invention of the 'Internet', as we commonly know it, the number of networked devices nowadays is tremendously high. To most owners of personal computers these devices are almost useless if it were not for the worldwide connection and interaction with others. Applications that exploit these networks range from simple communication among endpoints to highly sophisticated distributed processing and resource sharing. Usually this works fine in the world of traditional computers and servers where distributed programming has been studied and established for many years. Additionally the small amount of resources on mobile computing devices is still a problem and requires careful and complex resource management. Today even the average desktop computer has a huge amount of memory and computational power which makes it a lot easier to write useful and competitive applications in a short timeframe.

Still these aren't the only problems that arise in a distributed environment. Advanced and complicated programming patterns are needed in order to ensure secure and safe applications. Not only to avoid intrusion and data manipulation from third parties but also simple protection against data corruption and data loss due to networking problems. Enhancement of distributed applications in these aspects is often hard to implement due to the complexity of the technology on one hand and the entanglement of network and application code.

Clearly an improvement of this situation is desired to speed up development time and produce more robust programs that at the same time make use of cutting edge technology achievements.

I.2. Current Situation

Programming in and for networked environments has been around for quite some time. Distributed applications also are not some new unknown ground. It is clear that support for the development of such programs has improved over the last years. Networking, package and message handling often (if not always,

not accounting some special or exotic corner cases) is only visible through some high level abstracting libraries to the average programmer. Still, since breaking up application data into separate messages that can be sent by means of such libraries is tedious enough an even higher level concept has been introduced. And developers want to exchange parts of programs and data and not messages that are unnatural to the application structure.

Developers usually have the 'don't reinvent the wheel'-concept learned from day one. Therefore the best way to achieve an improvement would be to first look at what's already there. Different program parts need to communicate with each other. Most often it's not really noteworthy since they communicate on the same machine. But here lies the key to a new concept for distributed programming.
Inter process communication has come a long way and many different solutions have been invented to support it. Similar problems also occur in multi-threaded[1] applications, that is different processes (or threads) have to communicate. A natural idea is to let multiple processes *share* a common *memory* called a *shared memory*. Each process (or thread) can work with the memory in such a way as if it would be just a normal chunk of memory[2]. Since multiple processed (or threads) can access the same memory without special hacks or intruding private memory blocks of another process, they can share data and thus communicate.

This idea of a shared memory in a non-networked environment has been adopted for distributed applications as well. In order to clearly distinct between the different environments such a shared memory is called *space* in the networked environment. But the principle is still the same; multiple (not necessarily) distributed applications share some virtual common memory which they can use to exchange data. The advantage of this concept is clearly visible to the application programmer since the burden of writing tedious network code,

[1] Multi threading is a concept that allows a program to virtually run multiple tasks simultaneously. Each different task would be implemented in a separate thread which can run on its own. In many cases this can result in tremendous performance boosts especially when slow operations (such as I/O) are involved that would otherwise block the rest of the program during their wait cycles. Additionally the multi threaded architecture is natural to the existing and spreading multi-core CPUs.
[2] Of course there usually are differences to 'normal' memory since the shared memory has to be requested from the operating system. It might also be possible (depending on the used library and OS) that the process has to do some locking and releasing when working with the memory.

breaking data into messages and similar tasks are hidden in the abstraction of a space middleware.

Quite a number of different space based middleware is already on the market. A short and not exhaustively overview will be given in chapter I.4. It will compare the currently available systems to the XVSM (eXtensible Virtual Shared Memory) which tries to solve or nullify different limitations and problems of distributed computing by providing a new methodology for space based computing.

I.3. The Goal

The XVSM system was designed from scratch with three main aspects in mind.

1. It must be possible to distribute the memory pool (space) itself throughout a networked environment. This means that the space must not be limited to a single machine but can be accessed by multiple devices.
2. It must be extensible in such a way that needed aspects can be injected into the system without a system redesign. Such aspects may concern security, encryption or user management (Note: these are only examples and not an exhaustive list.)
3. The XVSM must have an interface that is independent of a programming language so that different applications can use the same space implementation no matter which language they are written in.

Chapter III explains the different XVSM coordination patterns; chapter IV the overall system architecture; chapter V the concrete architecture of the MozartSpaces and the chapters thereafter contain detailed discussion about the MozartSpaces implementation.

Basically XSVM's objective is to be a robust, feature-rich yet easy to use space based computing middleware. It is my feeling that we indeed have accomplished our goals thus far and think that XVSM could and should be the next step to be taken in space based computing.

I.4. Basic Concepts

The XVSM (eXtensible Virtual Shared Memory) system is briefly described in this chapter. For more detailed explanations see the respective chapters.
The main concept used in XVSM is a container (see chapter VI.1). A container has a number of properties (see chapter VI.1.1). The most important properties in XVSM are the coordination types (see chapter VI.1.2). These are used to coordinate multiple clients through specific read/write/notify design patterns. Another prominent example of a container property is its size. A container can be unbounded, meaning that it does not have a size and can hold an unlimited number of entries, or bounded. The size property of a bounded container gives the number of entries that the container can hold.
The chunks of data that can be written into or read from a container are called entries (see chapter VI.2.1). These entries have a VALUE_TYPE property which specifies the type of the data this entry represents. Along with the value's type the value itself is stored in the entry.
Entries are written to and read from the container with so called selectors (see chapter VI.2.2). These can be thought of as some kind of filter in the case of reading entries and additional meta-information when writing entries. Some operations that are defined for entries (read, take, write) can block. This means that if the operation cannot successfully execute (e.g.: the desired entry that must be written does not exist) the operation is said to block. This means that the operation will only return to the caller after the specified timeout has expired or the operation can successfully execute (e.g.: someone might have written the desired entry into the container).
For each operation that is concerned with entries a so called notification can be created (see chapter VI.3.4). This notification fires whenever the hooked operation is executed successfully. Among other things this mechanism can be used to implement asynchronous behaviour.

I.5. Comparison existing Space-based Middleware

This chapter gives some examples of currently available and used middleware implementations along with a short description of each. For each of these implementations a short list of shortcomings is given and a summarizing table

of these limitations along with a relation to XVSM is presented at the end of the chapter.

Figure 1, taken [Kühn 2007], gives a general idea about the space based systems and implementations that are currently available.

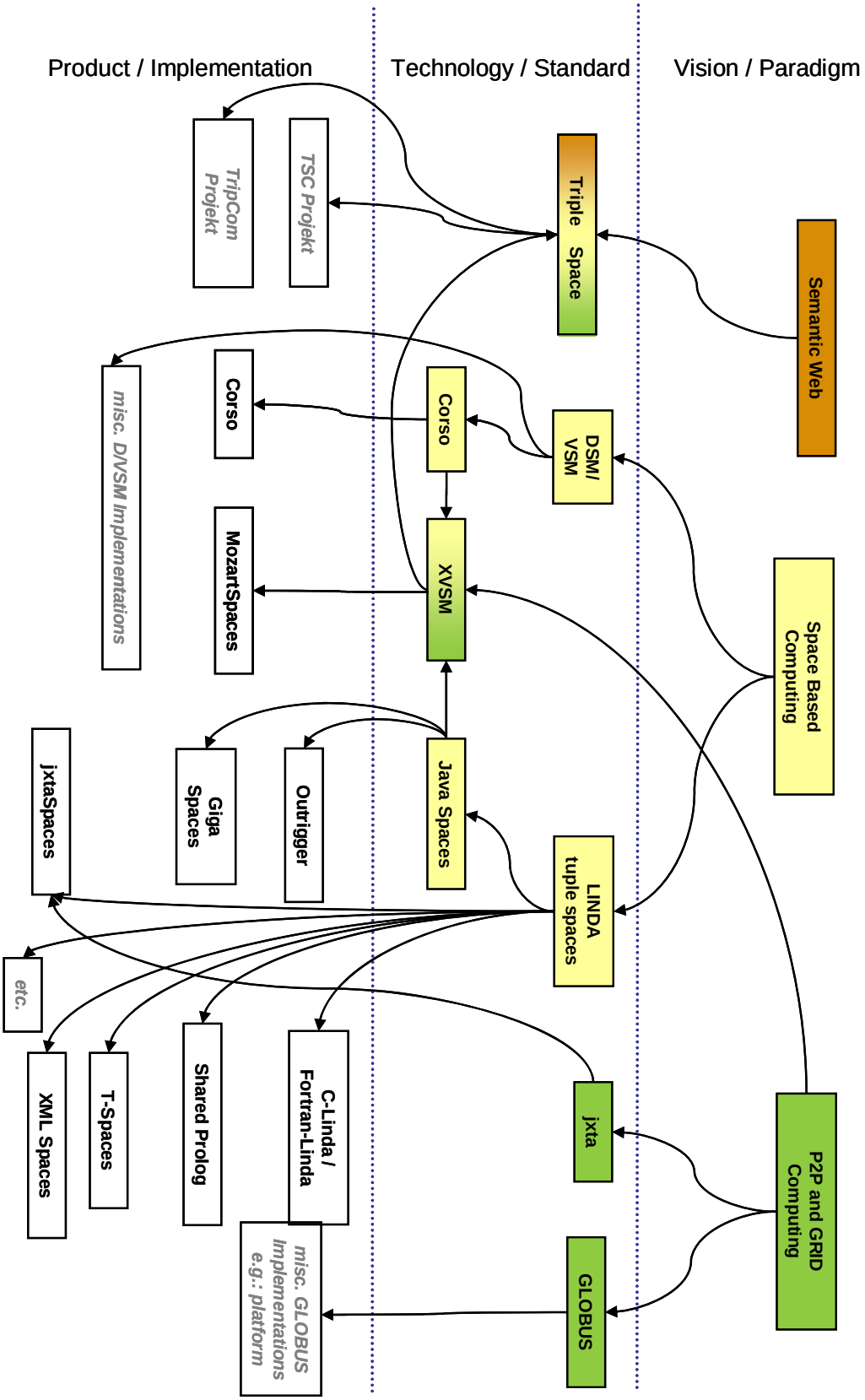

Figure 1: Space based computing overview

In the rightmost column the vision or architectural paradigm is given. Such paradigms generally give concepts and ideas about the general functionalities that such a system must have.
The middle column then sums up which standards/specifications and or technologies follow and specify the visions given in the paradigm column. Also the relationship among different technologies is visible. E.g.: the XVSM can be seen as an evolution of the Corso and Java Spaces technologies that also includes ideas and concepts of the GRID computing world.
Finally in the leftmost column available implementations of the mid-column specifications are given. E.g.: the MozartSpaces are one implementation of XVSM.

The following list gives a short overview of currently available and important players for the three general paradigms.

- **JavaSpaces**[3]

JavaSpaces is a specification developed and maintained by Sun Microsystems. It is based on the Jini technology (also introduced by Sun). Data objects that can be written into and retrieved from the space are called entries. These entries will be serialized upon writing to and de-serialized upon reading from the space. Among JavaSpace clients (so called peers) all communication is handled through the entries written to and read from the space. The space itself will not take an active and direct role in peer interaction (other than providing mechanisms for entry storage).

In JavaSpaces, as it is the case with any space based middleware, the heart lies in the space itself. The space is the storage which provides mechanisms to write and read data. In the case of JavaSpaces this memory is not distributed among several hosts but located on a single machine. Some of the disadvantages with this architecture become immediately clear. Obviously it more or less boils down to a client-server architecture including many of its limitations, most notably the problem of the single point of failure. If the service for any reason does not

[3] The currently most recent specification, version 2.1, can be found online.
http://java.sun.com/products/jini/2.1/doc/specs/html/js-spec.html

work as intended, the peers are left without any possibility of fixing the problem. The space server might become overstressed, taken down for maintenance or even worse, lose all data due to an unexpected crash. Some of these problems can be tackled by sophisticated server backup strategies. Due to the very nature of the JavaSpace architecture itself this puts a heavy burden on system administrators though.

Since JavaSpaces is an implementation specification of the TupleSpaces the single coordination model available is that of the so called LINDA tuples. The LINDA tuples provide a template tuple matching mechanism. Simply spoken this means that a template for the requested type of entry is created, filled with the desired matching fields and sent to the space. The space will then match this template against the available entries (unset fields will match any value of that given field) and return the matching entries to the client.

JavaSpace, as the name quite obviously suggests, are available only for the Java programming language therefore ruling out a rather big number of possible users beforehand.

Additionally to simple read and write operations the JavaSpaces offer the possibility of notifications which can be used to track changes in the space without the need of implementing a pull based model.

- **GigaSpaces[4]**

The GigaSpaces are a concrete implementation of the JavaSpaces specification. They provide high performance and transaction safety. Coming in different flavours (e.g.: free, caching, enterprise editions) everyone can find an edition suitable to his/her needs and budget. Unlike JavaSpaces the GigaSpaces also provide language bindings for other programming languages than Java (e.g.: .NET).

- **XMLSpaces[5]**

[4] http://www.gigaspaces.com/
[5] http://www.ag-nbi.de/research/xmlspaces.net

Just as it has been the case with the previously mentioned space implementations only the LINDA coordination model is available for XMLSpaces. XMLSpaces is an extension of the Linda coordination language which is used for web based applications. The advantages of XMLSpaces lie in the more sophisticated matching facilities that are available. The matching is performed according to an XML document, hence the name XMLSpaces. Here not only simple logical compositions are possible but also complex ones such as XPath expressions.
XMLSpaces are available for Java as well as the .NET platform.

- **Corso[6]**

The Coordinated Shared Objects space based system was designed and implemented at the Vienna University of Technology. Once again only one coordination model is supported. Every item that is written into the space has a unique ID attached (the object ID). This identifier must be used to retrieve objects from the space. An object therefore has an identity such as it is commonly known from object oriented programming languages. Depending on the situation this can be an advantage or a burden. Another disadvantage of Corso is that it doesn't offer any other GRID related features and extensible interceptor technologies.
Unlike many other space-based middleware the programmer can choose between different language bindings such as .NET, Java and C++. These of course are available on different operating systems as well (e.g.: Linux, UNIX, Window, even some mobile platforms).
Objects in Corso can be read from and written into the space, registered with names, read asynchronously and there is also a notification system available.

- **Grid Computing (Globus)[7]**

A GRID is an infrastructure that allows sharing of distributed resources. Probably two of the most prominent examples would be processing time and memory. The main advantage of this architecture is that even resource limited end points virtually have unlimited resources at their hands such as handhelds or smart phones. Additionally GRID architectures don't face the same problems

[6] http://stud3.tuwien.ac.at/~e9825311/SBC/corso/docs
[7] http://www.globus.org

as traditional client-server architectures do. Since the workload is distributed among several endpoints there is no single point of failure i.e.: the one server. One representative for GRID computing systems is Globus.

Feature	Space-based system	Available
Coordination types	JavaSpaces	One - LINDA tuple
	GigaSpaces	One - LINDA tuple
	XMLSpaces	One - LINDA tuple
	Corso	One - LINDA tuple
	Grid Computing (Globus)	GridFTP and RFT
	MozartSpaces	Order (lru, fifo, lifo, set), key, vector and template
Programming language	JavaSpaces	Java
	GigaSpaces	Java, .Net and C++
	XMLSpaces	Java and .NET
	Corso	.NET, Java and C++
	Grid Computing (Globus)	Phyton, C and Java
	MozartSpaces	Java (additional languages such as the .NET framework with its supported programming languages are already planned)
Basic operations	JavaSpaces	Read, write, take, notify
	GigaSpaces	Read, write, take, notify
	XMLSpaces	Read, write, take, notify
	Corso	Read, write, take, notify
	Grid Computing (Globus)	Read, write, take, notify
	MozartSpaces	Read, write, take, destroy, shift, notify
Distribution of the space	JavaSpaces	No, representative of Tuple Space model
	GigaSpaces	Yes, in a special version
	XMLSpaces	No, representative of

		Tuple Space model
	Corso	Yes, representative of VSM
	Grid Computing (Globus)	Yes
	MozartSpaces	Yes, via higher level profiles, representative of XVSM
Transactions	JavaSpaces	No distributed transactions
	GigaSpaces	No distributed transactions
	XMLSpaces	No distributed transactions
	Corso	Distributed transactions
	Grid Computing (Globus)	Distributed transactions
	MozartSpaces	Not yet

Table 1: Space-based computing middleware comparison

II. Evolution of Space based middleware

Four major evolutionary steps can be identified in the evolution of space based computing. This chapter will explain these steps and also give examples of the main and most important technologies for each step. These four steps are:

1. Direct communication
2. Client-Server based communication
3. Central Space Server
4. Virtual Space Server

1. In the direct communication 'age' every peer that wants to interact with another peer directly sends messages to a target. This usually puts a noticeable burden on the application designers and developers since, among other things, they have to implement the messaging code, come up with a suitable communication protocol[8] that fulfils the needs of the overall application. This directly implies that each peer must know every other peer and its communication protocol. Figure 2 shows only a small network with a few peers that interact with each other. Clearly a lot of knowledge about the other peers must be shared (e.g.: host endpoints, ports) and a lot of network traffic is involved.
   Related technologies for this kind of communication are sockets, RPC[9] and RMI[10].
   Extensions of this model are very complex and there is no caching strategy.

[8] This does not mean a transfer protocol such as TCP or UDP. Here the content of the messages which are sent from one peer to another are spoken of. Of course transfer protocols usually are managed by networking libraries and not implemented by the application developer.
[9] Remote Procedure Calls allow the invocation of methods on a different host.
[10] Remote Method Invocation can be seen of SUN's version of RPC for the Java Programming Language.

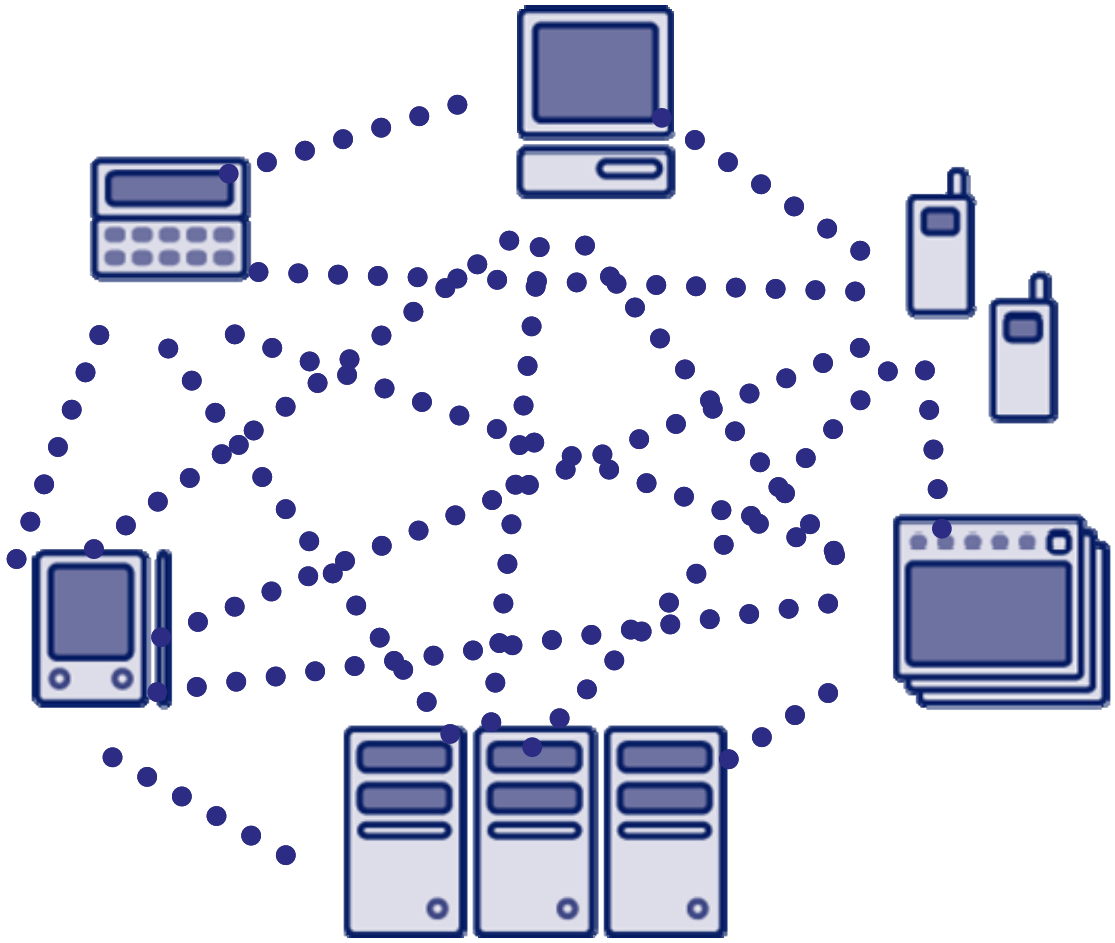

Figure 2: Direct communication

2. The client server architecture (see Figure 3) removes the necessity of every peer knowing every other peer that it wants to communicate with. Instead, all interaction is done between a client and a centralized host, the server. This server is contacted whenever a peer needs to interact with the server itself or another client in the network. Depending on the application and the network infrastructure it might be possible that the service can only be contacted by clients but may not contact clients itself. Such limitations must be taken into consideration when designing applications. Usually the servers are well equipped considering their bandwidth, computational power and memory and can do their tasks a lot quicker than the average client would be able to. The reduction of communication interfaces is an advantage of this architecture. The disadvantages must not be overlooked. Client-server architectures[11] offer

[11] In this case assume that there is no backup server available, which in reality basically is a must for important services.

a single point of failure for the application. Whenever the server is not running the application is not functional. Also the client-server architecture is still based on the message passing paradigm. Very prominent examples of client-server architectures would be HTTP servers (web browsing) or web services that are used for certain computation tasks.

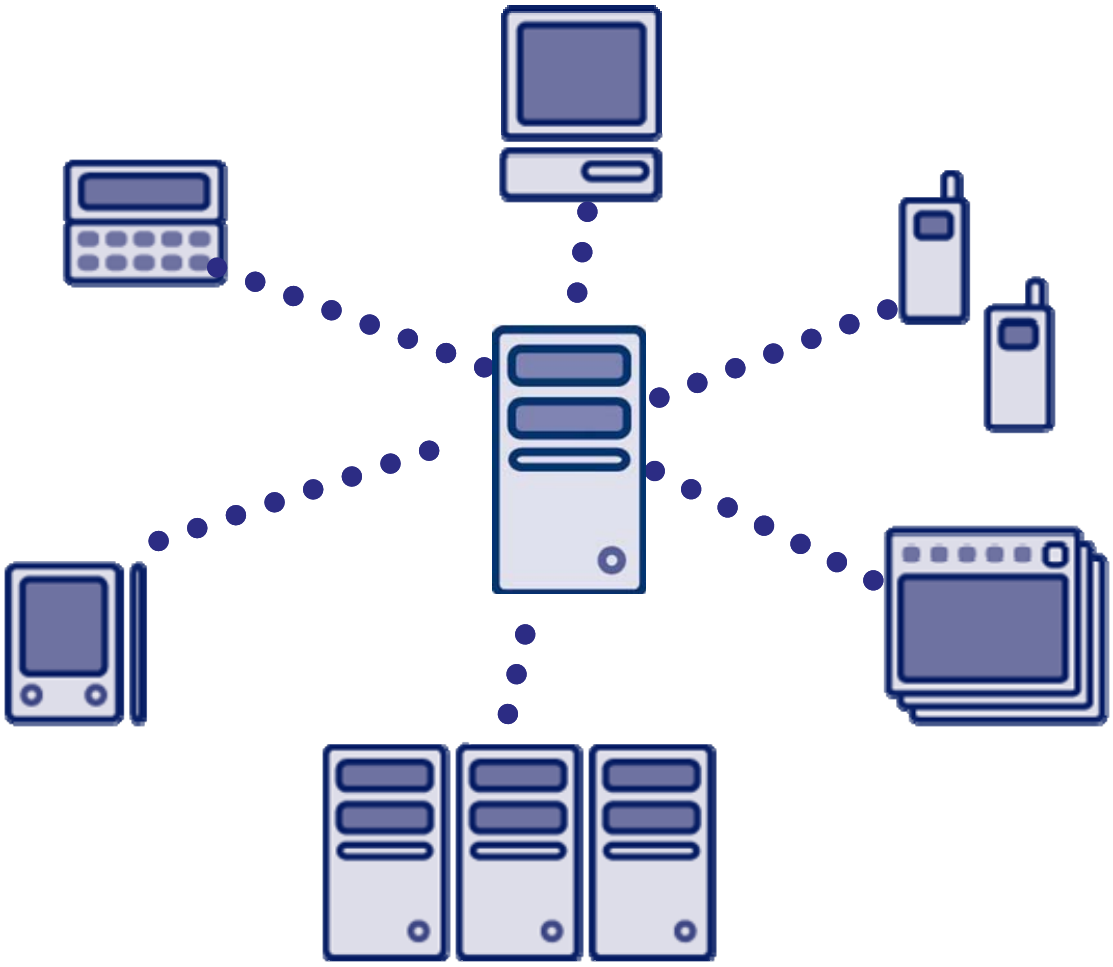

Figure 3: Client-Server based communication

3. The next step of abstraction is space based computing as shown in Figure 4. Two things have changed compared to the client-server architecture. First, the standard message passing has been changed into a space based API. This means that all network handling, message splitting and protocol issues are handled by the space based library in the background and transparent to the user. All peers can access a common storage, however, on a server and most importantly do so independently of the application. The application developers therefore don't need to worry about data to

message splitting and comply with application specific communication protocols since all this is handled in the space based library. The most serious flaw of the client-server architecture still applies to this approach: the single point of failure. Once the main server is down, the space and therefore all storage and communication facilities are not available. Important and related technologies concerning this architecture are the Java Spaces, VSM and XVSM.
This architecture is only a conceptual improvement since the underlying techniques are still the ones that have been discussed before.

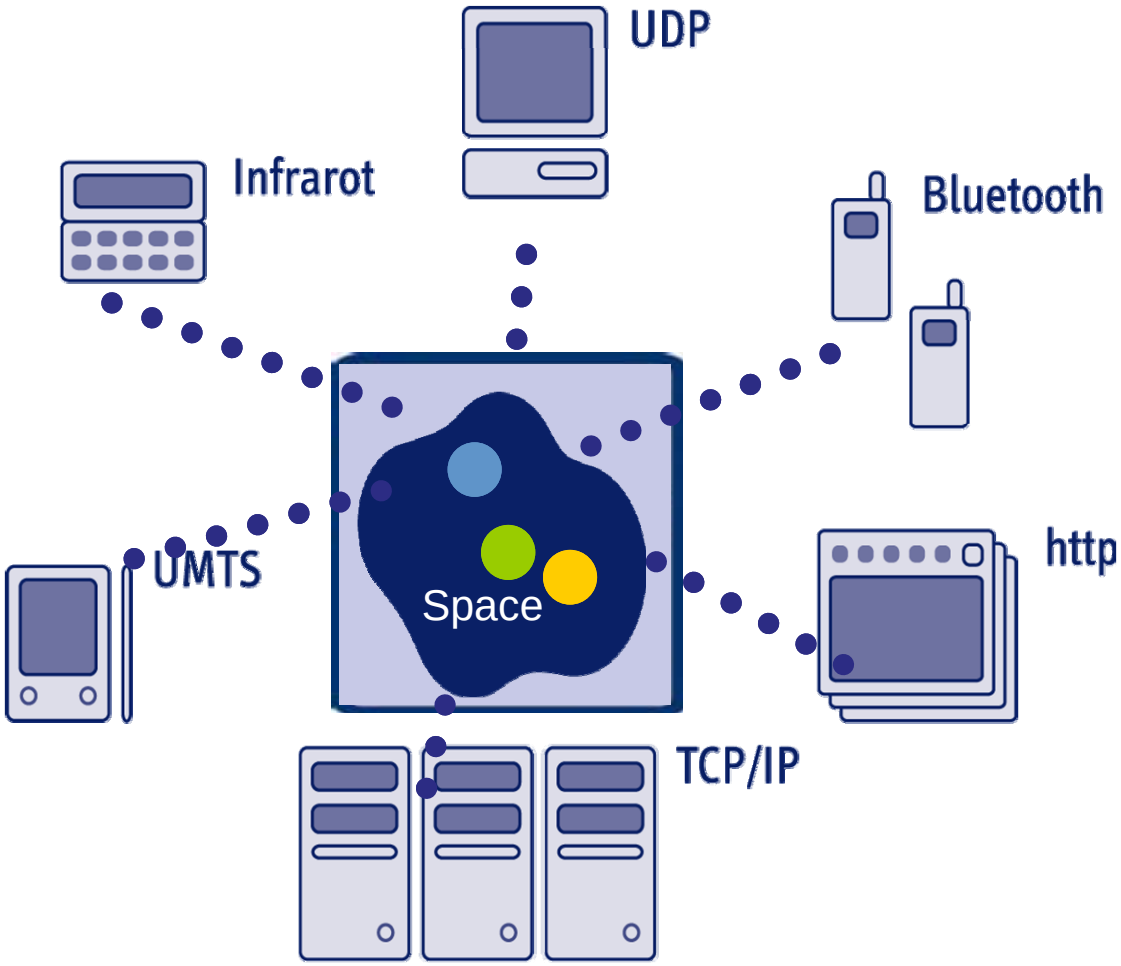

Figure 4: Central Space Server

4. The last step of evolution towards space based architectures is shown in Figure 5. The central space hosting server and thus the single point of failure has been overcome while keeping the space based advantages depicted in the previous architectural step. The space (the common storage area) now is spanned by the participating peers, the virtual space

server. This means that failure of a single device does not have as big an impact (if any) on the overall application and other peers as in comparison to a central space service. In combination with this architecture usually replication mechanisms are incorporated to keep all data available and intact even in cases where peers are not available. This also helps with the overall performance of an application since usually much shorter routes have to be taken to access certain pieces of data in this case than when only a central server is hosting the data. Related technologies for this architecture are Peer-to-Peer, GRIDs, VSM and XVSM.

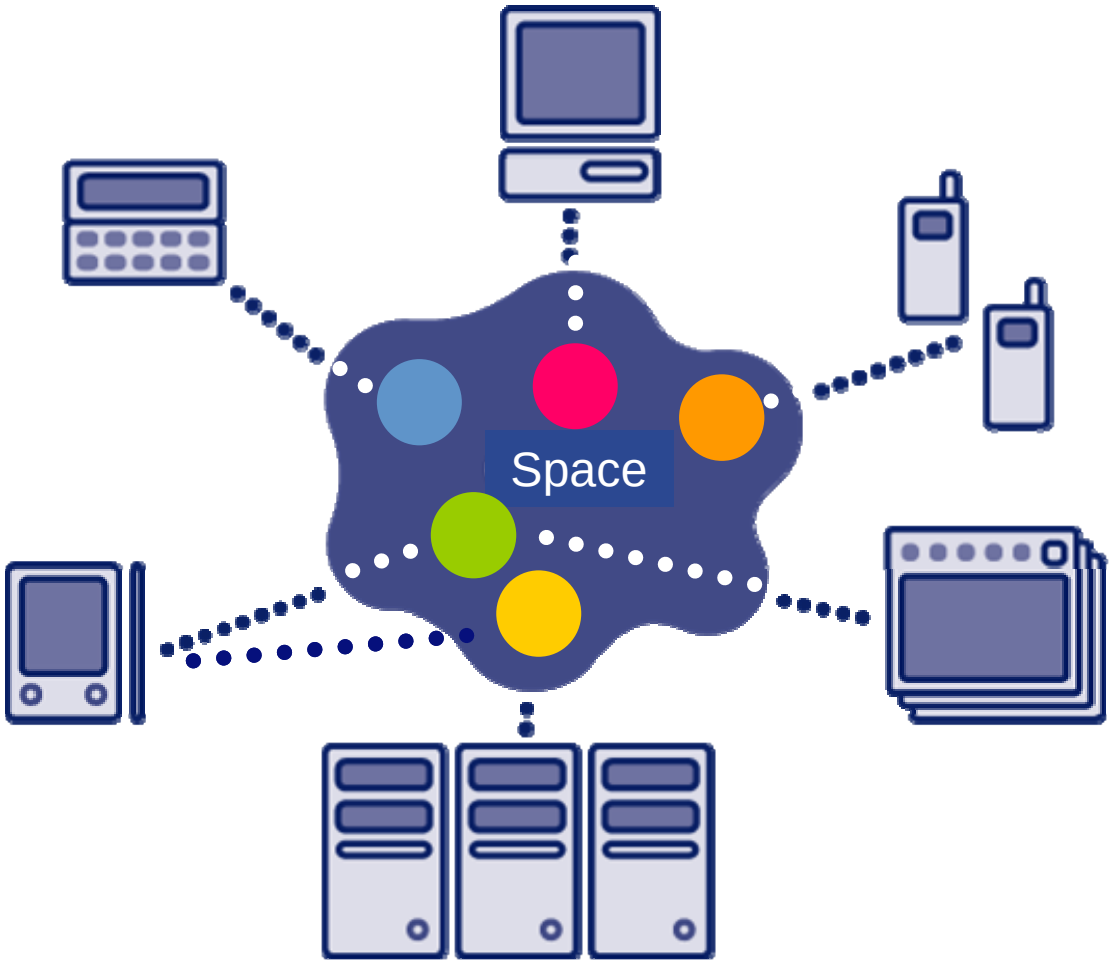

Figure 5: Virtual Space Server

III. XVSM Coordination Patterns

In chapter I.4 the LINDA coordination principle has already been mentioned. This is the most widespread coordination pattern in space based middleware. In this chapter it was also mentioned that XVSM supports more coordination patterns.
These will be described in this chapter.
Coordination patterns define the way how entries are written to and read from the space. Many different patterns can be imagined but usually many of them are very specific to a certain problem and others can be built using more basic patterns. Before introducing the coordination patterns that are available in XVSM the two important properties are introduced.

- Implicit/explicit: an implicit coordination pattern does the bookkeeping for every entry transparently to the user. This means that the user simply specifies the entry s/he wants to write into the space and the middleware itself adds any needed additional information for the designated coordination pattern. Similarly the client only needs to ask the middleware for the next entry (or more) and will get these according to the coordination pattern.
  Explicit coordination patterns on the other hand require the user to explicitly provide additional bookkeeping information that is needed for the coordination pattern.
- Complete/incomplete: a complete coordination pattern allows the client to retrieve all entries with a single method invocation and the selector that is bound to that specific coordination type. Furthermore the client does not need any information about the content of the container or the entries that are stored therein. (An analogy to this single selector get all property would be the following SQL query: SELECT * FROM container) Incomplete coordination patterns do not have this property. This means that it is not possible to retrieve all available entries with a selector that is bound to that coordination pattern in a single method invocation.

These two property types will become clearer in the remainder of this chapter.

Coordination patterns can be categorized into three coordination types.

- ORDER (implicit, complete): coordination orders are the most basic kind of coordination types so that every container[12] has one basic coordination order. There are presently four different coordination orders available and supported in XVSM
    - Random: entries are stored in the space without any coordination ("chaos"). Therefore retrieving an entry also is uncoordinated and indeterministic. The MozartSpaces implementation of the random coordination type is truly random[13] meaning that retrieval of an arbitrary entry will not always result e.g. in the first entry.
    - FIFO (first in, first out): the FIFO coordination pattern implements a behaviour commonly known as queue. The entry that has been written first will be retrieved before any other entry.
    - LIFO (last in, first out): the LIFO coordination pattern implements a behaviour commonly known as stack. The entry that has been written last will be retrieved before any other entry.
    - LRU (least recently used): the entry that has the oldest access[14] timestamp will be retrieved. The access timestamp will be added and managed by the XVSM system. While this is additional information the coordination type is still implicit since it requires no client action or information.
- VIEW (explicit, incomplete): coordination views are incomplete and explicit. This means that the client must provide additional information when writing and reading entries. This information is stored along with the entry data and used when applying a selector. Additionally the incomplete property means that it is not possible to retrieve all entries with one operation call (e.g.: one cannot provide all matching meta-information in one selector).
    - KEY: the key coordination pattern implements a behaviour that is similar to a hash map. Each entry has attached an additional information property, the key. This key is used as lookup value for the entry. Keys in XVSM can have arbitrary type i.e. they are not restricted to string values.

[12] For more details on what containers are and what they are needed for please see chapter VI.1.
[13] Please note that deterministic random number generation of programming languages and hardware architectures still apply and are not solved by this pattern.
[14] In this context access applies to both, reading and writing.

    - VECTOR: a vector provides access to its entries through indices. Each entry in the container has an integer typed index attached as meta-information. Unlike vector implementations commonly known in programming languages (e.g.: java.util.Vector in the Java programming languages) the vector does not necessarily need to be compact. It is valid to have any number and size of gaps between subsequent indices.
- TEMPLATE (implicit, incomplete): this coordination type is used for content matching[15].
    - LINDA: this is XVSM's implementation of the LINDA matching coordination which is a tuple matching paradigm.

Coordination patterns are not only used for reading and writing entries. XVSM also features another interesting operation named shift. Shifting is used to replace an existing entry with a new one and can be seen as atomic 'destroy-write' operation. Shift always operates using the base coordination order of a container. Since all containers do have a basic coordination order, this operation is always defined and operable.

[15] Other matching paradigms would be e.g. XPath or RDF. These are not implemented in the MozartSpaces and therefore not mentioned any further.

IV. XVSM System Architecture

XVSM specifies the architecture that is capable of providing the functionality that has been discussed in chapter II, the virtual space server. This chapter briefly explains the conceptual architecture of the XVSM while the next chapter discusses the architecture of the MozartSpaces, the concrete implementation of the XVSM.

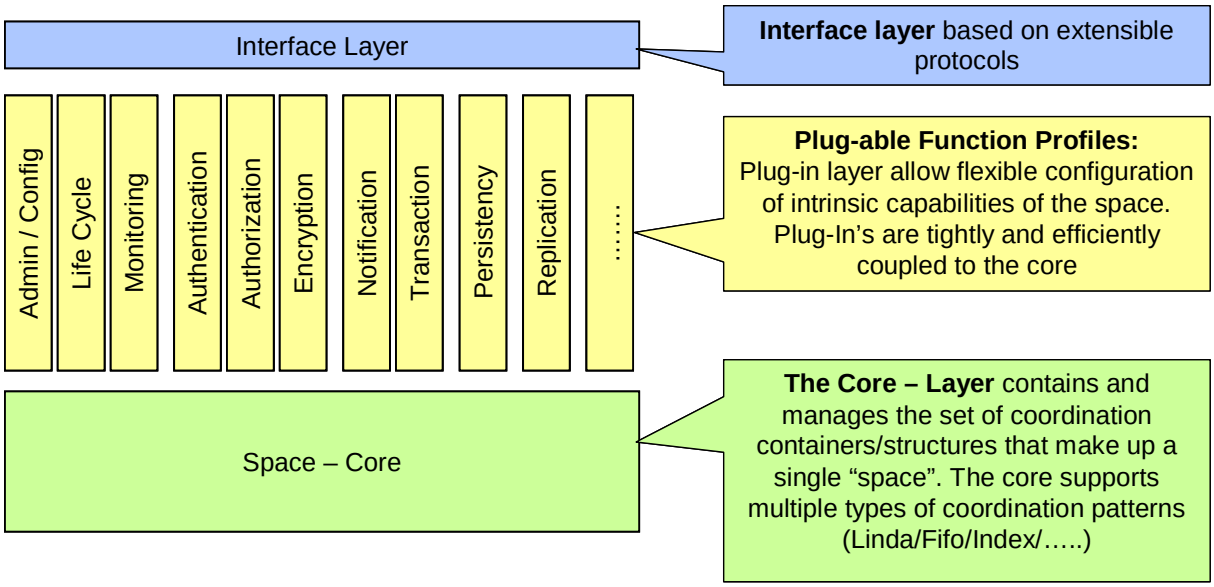

Figure 6: System Architecture

Figure 6, taken from [Kühn 2007], shows the XVSM system architecture on a very general and abstract level. The basic goals can be derived from this architecture. In principal the architecture comprises a three-layered vertical approach which, from bottom to top is:

- The Core Layer (green): The space or XVSM core implements the basic functionality of the system, namely container handling, coordination patterns and notifications. On its own, this core setup only provides a 'space implementation' for a single host and would apply to the third architectural step of the previous chapter.
- The Profile Layer (yellow): This layer contains the so called pluggable function profiles. The basic XCore functionality is extended here with additional properties such as monitoring, authentication & authorization as well as replication which are needed to implement the virtualization of

the XCore and extend the architecture towards the Virtual Space Server architecture that has been described in the previous chapter. Therefore this layer relates to the extensibility goal mentioned in chapter I.3.

- The Interface Layer (blue): Here a programming language neutral interface is provided to the XVSM clients. This decouples the client programming language requirements from the language used to implement the XCore itself.

The following chapters describe the architecture in detail especially the parts that have been implemented in version 1.0 of the MozartSpaces.
MozartSpaces is the name of the implementation of the XVSM system developed at the Vienna University of Technology.

V. Architectural Overview

The XVSM system consists of numerous parts which are explained in detail in the following chapters.
There are two possible architecture configurations of the XVSM system; the embedded and the standalone installation. In the functionality that they provide both are equal. Mainly the choice for either one of the two depends on the needs of the user. For a single site and possible multi-threaded application the embedded version of XVSM is fully sufficient. It provides all feature of the MozartSpaces 1.0 and does not require the installation of a servlet container. A real distributed application which has multiple processes running on multiple hosts must use the stand-alone version of XVSM. The stand-alone version requires a servlet container (e.g.: Apache Tomcat) installation and network access to this server. The setup is slightly more complicated but the advantage of distributed coordination should outweigh this minor inconvenience.

V.1. Embedded XVSM

The embedded version of XVSM can be seen as an application library. Like any other library the embedded XVSM runs in the process space of the application and therefore can only be used by this application[16]. Figure 7 shows the setup for an embedded XVSM System. Details about the embedded version of XVSM will be given in chapters VI (the XCore API), VII (workflow discussion) VIII (Container Access backend) and IX (collaboration of these distinct parts).

[16] The possibility of injecting applications into the process space of the host application or modifying the JavaVM in order to allow across VM library sharing is not considered here. Nonetheless if in such cases the library is used in a normal way it should work as intended and correctly even then.

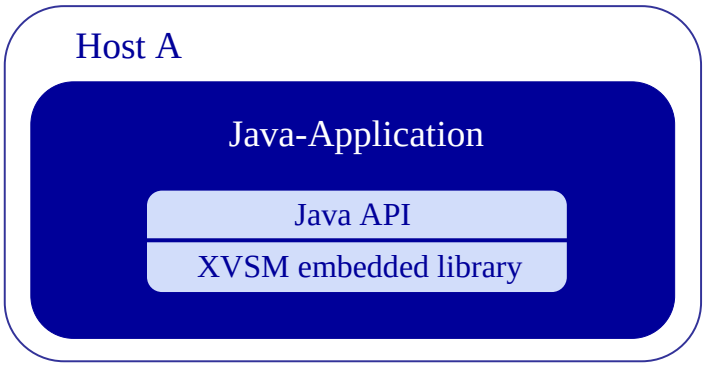

Figure 7: XVSM System embedded

The embedded XVSM System (also called XVSM Core or simply Core), comprises a three layered architecture. These layers are as follows (from outmost to inner most):

a) Capi: The Core API (Capi) provides the interface that is used by applications to make use of the XVSM system.
b) Workflow: The workflows are entities that carry out single specific tasks of the XCore. Examples for these are reading or writing of entries
c) CADAO (ContainerAccess Data Access Object): The container access entity that handles container and entry manipulation. In this version of the XVSM core the CADAO is the front end to a database.

Each of the parts listed above makes use of the part(s) below. This means that the Capi uses the Workflows and the Workflows use the CADAO.
In order to keep the system as extensible as possible this 3-part architecture was used to decouple the application from the XCore implementation on the one hand and to decouple the data holding backend (database) from the XCore.

Figure 8 shows the general architecture of the XVSM Core and the interaction with an application. The XCore is loaded as dependent library and therefore runs in the same process space as the application. The interaction between the XCore and the application is carried out using the Capi part of the XCore. This Capi is a lean Java API that provides the functionality of the XVSM system. This is the only visible entry point to the XCore that an application can use. Within the XCore an in process space database (in our case: Derby) is used as repository. Since the data handling part is decoupled from the rest of the XCore

it shouldn't be hard to use another relational database or even some completely different kind of data system such as a file based storage system.
In the current version of the MozartSpaces it is not possible for multiple applications to share one XCore (Note: there might be ways for two applications to start within the same JVM and share a loaded library but this should be considered a "hack" and not the intended way of usage). It is possible though to safely have multiple threads connected to the XCore and their operations will correctly be synchronized.

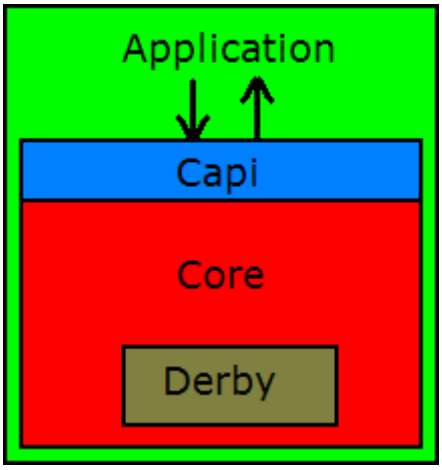

Figure 8: XVSM Core architecture

The XCore itself comprises three distinct parts which will be described in detail in chapters VI, VII and VIII.

V.2. Standalone XVSM

The standalone version of XVSM is closely related to the embedded version. In Figure 9 one can see that the client application that makes use of the XVSM System only links a small runtime library which provides an interface that is identical to the embedded version interface. This library hides network communication and therefore transparently provides the XVSM functionality that actually is carried out on another host, to the client. As communication and data protocol between the network nodes a specifically developed XML protocol has been developed that reflects the capabilities of the XVSM System. This also renders the whole standalone system independent of any specific transport protocol or network topology. Even the concrete incarnation of the

standalone 'server' (client-server, peer-to-peer…) is not relevant to the XVSM System.

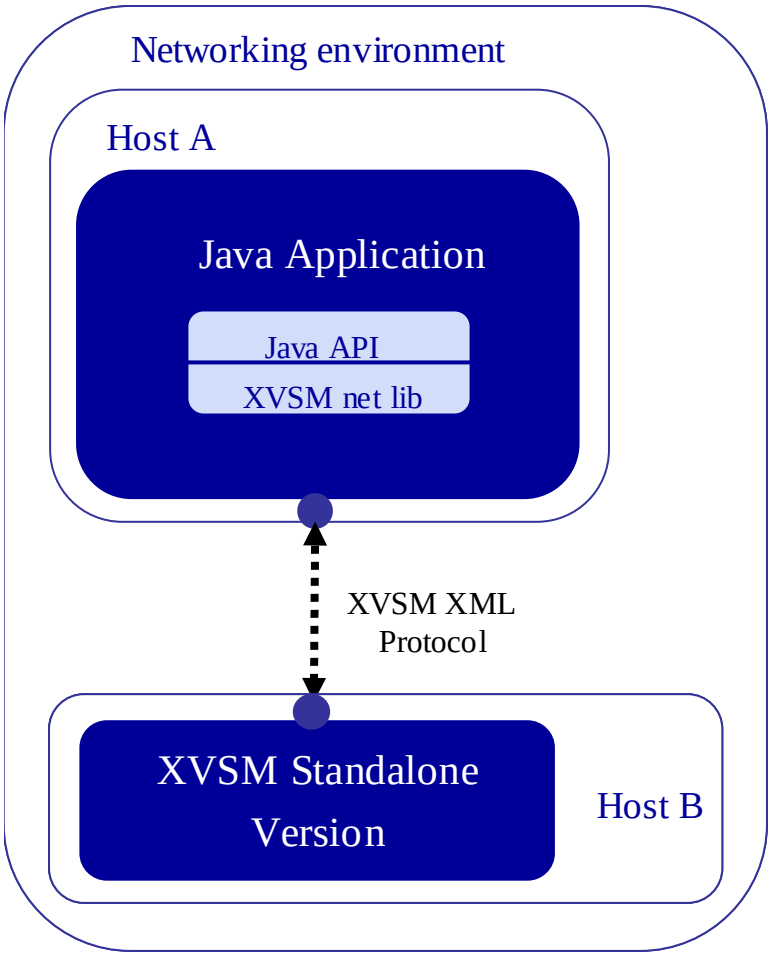

Figure 9: XVSM System standalone

VI. Core API

The Capi provides the Java interface that the applications must use to interact with the XCore. Figure 18 shows the UML class diagram of the Capi interface. It should be noted that the Capi only specifies a java interface, hiding the concrete implementation. For the application to get a usable Capi object it must request such an object from the CapiFactory (Figure 10). This allows for easy exchange of concrete API implementations. The Spring Framework[17] mechanisms are used to create the Capi implementation[18] object as Singleton[19].
The code for using Spring in this context look somewhat like this:

```
ClassPathResource res =
    new ClassPathResource("CapiImplementation.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
Capi capi = (Capi)factory.getBean("CapiImplementation");
```

Listing 1: Capi creation through factory

Setting the singleton property is done in the configuration file 'CapiImplementation.xml' and can be changed when the need arises.

```
<bean id="CapiImplementation"
    class="org.xvsm.api.core.implementation.WFCapi" singleton="true"/>
```

Listing 2: Bean configuration for Capi

The bean id specifies the name which can be used in the Java code to request the object form the factory.

The following figures show the UML diagrams of the most important classes used in the Capi part of the Core architecture. The contained methods will be shortly described but detailed information especially about parameters and return values can be found in the source code and its accompanying JavaDoc.

---

[17] Spring Framework: http://www.springframework.org/

[18] The Spring Framework provides a unified mechanism for usage of the singleton design pattern. A so called Bean-Factory can be used to retrieve an object which, if configured correctly, is guaranteed to have the singleton property.

[19] The Singleton patter is a well known design pattern. Details can be found in [GoF].

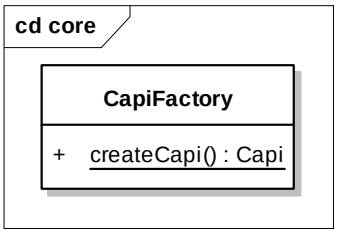

Figure 10: CapiFactory Class diagram

- createCapi: this method is used to get an object that implements the Capi interface. In the current version the Capi object is implemented as singleton therefore the overhead of object creation only matters the first time of invocation.

The Capi and its operations will be discussed in chapter VI.3.

VI.1. Definition of a Container

In XVSM a container is among the most important concepts. A container primarily is the basic storage entity in the space. That is every entry (for more details about entries see chapter VI.2.1) that is written into the space belongs to a container. Given this rule there are no "freely *floating* entries" in the space[20]. The implementation of the MozartSpaces shows this very clearly since each entry always have a Container Reference (cref) attached to it (see chapter VIII.3.1 for details). Also a container reference itself specifies a valid entry type allowing these references to be written into another container. This feature enables interesting programming patterns such as lookup tables, lists or even more complex ones.

Containers basically come in two flavours, anonymous containers and named containers. Anonymous containers are the standard type of containers since they require both less storage (an additional container is stored in the space which maps the container names to the actual container references) and a slightly less performance overhead (The creation/destruction of a container requires the name to be added/removed from the lookup container. When requesting the container reference for a name an additional read operation must

[20] The Capi description later in this chapter makes this property even clearer. There is always a so called container reference involved whenever the client wants to manipulate entries.

be carried out in the lookup container). These containers are accessibly only through their container reference. Named containers on the other hand are given a name by the user and their references are stored in a special lookup container[21] within the space. Named containers too can be accessed through their container reference. This container reference can be retrieved from the space through the container's name. The obvious advantage of such containers is that the application has access to its container even after a restart without the requirement of storing it before the program exits. Named containers also provide the means for distributed applications to have access to the same container even though these applications are located on different hosts.

Containers are the XVSM entity which provides the coordination facilities to the clients. Every container has at least a base coordination order and can have in addition multiple other coordination types. The containers and their coordination types provide a powerful yet simple to use coordination mechanism for application programmers..

VI.1.1. Properties

Each container is characterized by its properties. These ca be re/set and/or queried. The currently available properties can be seen in Figure 11.

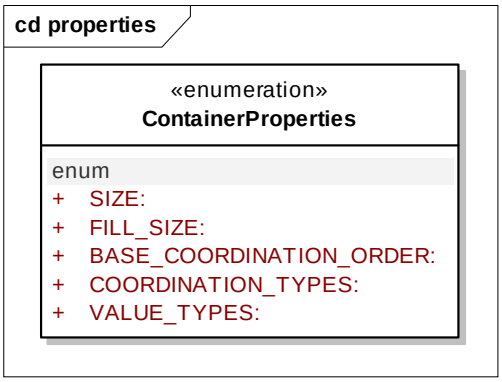

Figure 11: ContainerProperties class diagram

[21] This lookup container is strictly for internal use and not accessible or visible to the XVSM clients. Note that this special container is implemented with the basic XVSM operations and coordination types that are available to the clients as well. The lookup container is a normal KEY coordinated container which uses the container name as KEY and the container's reference as indexed entry.

- SIZE: this property specifies the size of the container. This can be a positive number or -1 to specify an unbounded container. If the size is not unbounded at most 'size' entries can be written into the container. A bounded container can only hold as many entries as have been specified with the SIZE property. When the container is bounded and filled with entries every WRITE operation blocks.
- FILL_SIZE: this property is a read only property meaning that it can't be changed by an application. It specifies the number of entries that are currently stored in the container.
- BASE_COORDINATION_ORDER: this property may only be changed if the container is empty. More details about coordination types are given in chapter VI.1.2.
- COORDINATION_TYPES: new coordination types may be added to a container at any given time. Some coordination types can only be added when the container is empty. An example is the FIFO coordination order which needs to keep track of all entries written into the container. Removal of coordination types is only permitted for empty containers.
- VALUE_TYPES: optional; this property is used to narrow the allowed value types which entries may have. Only entries whose value types are set in the VALUE_TYPES container property can be written to the container. More details about value types are given in chapter VI.2.1.

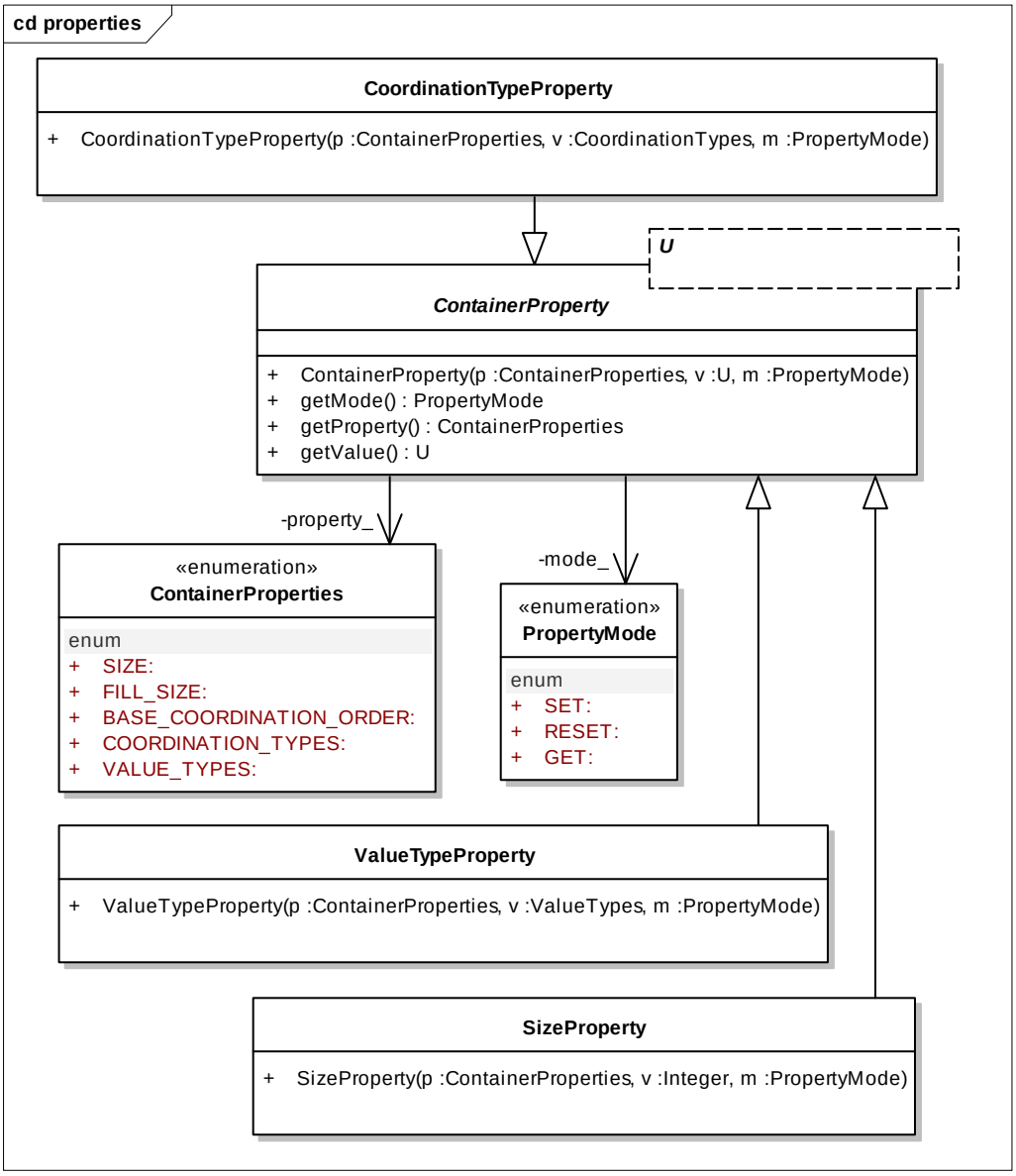

Figure 12: ContainerProperty class diagram

Figure 12 shows the class diagram of the ContainerProperty class. This class is the base class for all available container properties. This class is a generic class and its type parameter is set by its subclasses. The parameter is set the java type that represents the property's value type.

VI.1.2. Coordination Types

Figure 13 shows the class diagram of the available coordination types. Coordination types specify how entries are read from and stored in a container.

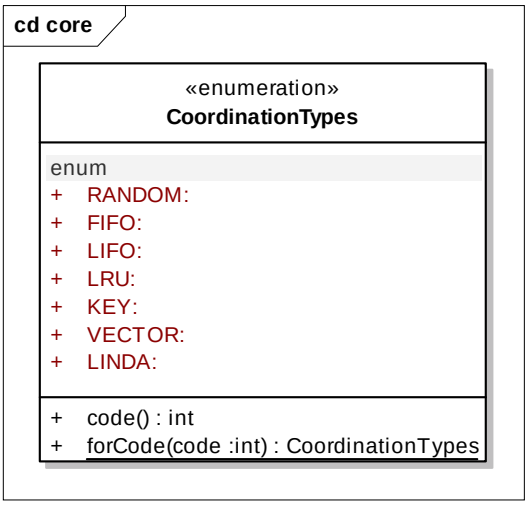

Figure 13: CoordinationTypes class diagram

More details about coordination types their usage and properties can be found in chapter III.

VI.2. Definition of API Arguments

The following subchapters give details about the most important entities that are used as parameter and return values in the XCore API.

VI.2.1. Entry

A central type of the Core is the Entry type (Figure 14). An entry[22] is an object that can be written to or read from a container. An entry basically consists of a ValueType (Figure 16) and an Object field that holds the value.

In the first version of MozartSpaces the Entry class was designed as generic class taking a type parameter. This type parameter was the Java equivalent to the ValueType (e.g.: java.lang.String was used for STRING_UTF8). This is a clean way of injecting type information into a class because the client would get a correctly Java-typed object that s/he could use in her application, independent of the way the entries were stored in the backend. The doubled information of ValueType and type parameter was needed since the entries are stored as strings in the database backend. Doing this allowed for an efficient and compact

[22] Entry type: the Java type that is defined by a java class. Entry: an instance of the entry class (entry type)

database schema design but the need arose for an additional type flag so that the correct typed entry could be restored when read from the database. Severe problems would arise when users misused this design and entered different types for the type parameter and the ValueType. This led to bugs and crashes of the MozartSpaces library. Additionally when creating new entries the user was forced to write the same lines of Java code for each entry which was rather tedious. Therefore the XCore design was completely changed in this respect.

Entries now are not generics anymore. The value holding field in the entry type is of type java.lang.Object. A field of type ValueType still keeps the information which concrete type is stored in the entry. In order to be able to check whether the type of the value complies with the ValueType, entries can't be created directly anymore. Rather an entry factory is used to accomplish that (see Figure 15). One could argue that the same can be done by using the constructor of the Entry class but the Factory has one further advantage. Whenever entries are read from the database backend the CADAO has to create a new entry and store this information. Since the type of the entry value will (for now) always be java.lang.String, there is a mismatch between the value's type and the ValueType field. This is needed and therefore the Factory needs to provide a method that allows for unchecked entry creation.
The following code segment shows the type checking used in the Entry class:

```
private Entry(Object val, ValueTypes type, List<Selector> sel)
{
    if (!isSubType(val.getClass(), type.getTypeClass()))
        throw new IllegalValueTypeSpecifiedException(type, val.getClass().getName());
          //…
}
```

Listing 3: Subtype checking of entry and value type

The ValueTypes class provides a method (getTypeClass) that returns the Class object of the class that is mapped to a specific ValueType (e.g.: java.lang.String for STRING_UTF8).
This class is then checked against the value's type. If the value class has a subtype relationship to the ValueType class then the entry creation is legal, otherwise an IllegalValueTypeSpecifiedException is thrown.

The isSubType method recursively checks all super classes and interfaces. If the requested one is found, the super/subtype relation is satisfied.[23]

```
protected boolean isSubType(Class a, Class b)
{
    if (a == null || b == null)
        return false;

    if (a.equals(b))
        return true;

    if (isSubType(a.getSuperclass(), b))
        return true;

    Class[] ilist = a.getInterfaces();
    for (Class i: ilist) {
        if (isSubType(i, b))
            return true;
    }

    return false;
}
```

Listing 4: isSubType implementation

The Entry::Factory's newTypelessInstance omits this type checking and simply creates an entry whose type flag and the concrete type of the entry's value don't match. This entry will then be typed in the Capi implementation before it's returned to the client.

One problem still resides though. Since Java does not provide the concept of 'friend' classes, this method must be public even though only one class will ever be legally allowed to use it. Formally though, everyone can use (and in this sense misuse) this method to create un-typed entries. There still needs to be done some research on how to resolve this problem.

[23] There still needs to be looked through the Java Platform API whether there is a simple standardized method that allows for recursive super/subtype checking.

When entries are read from the XCore the list of selectors (see chapter VI.2.2 for details) will always be **null**. These are solely used when writing entries where additional information such as keys needs to be assigned to this entry.

Entries of simple types that are strings, integers, crefs (Container Reference), are represented with the entry type. Since the XVSM Core also supports tuples[24] there was the need for an additional type. The factory returns an object of type Tuple whenever an entry with ValueType TUPLE is created. The tuple class is similar to the entry class; it simply adds a few convenience methods for adding single fields to the tuple instead of just allowing the setting of the whole tuple.

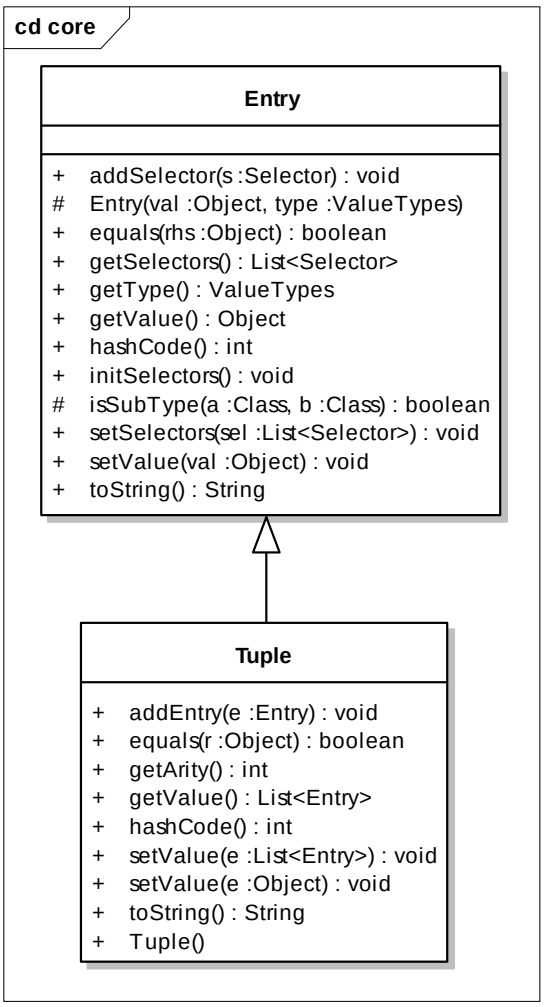

Figure 14: Entry and Tuple class diagram

The (non trivial) operations of the Entry class are as follows:

[24] A tuple is an ordered -list of values, where each single value is called field.

- addSelector: add a selector to the list of selectors (only needed when writing entries)
- initSelectors: constructs a new list object for the selectors. This must be called before adding selectors to the entry; otherwise a NullPointerException will be thrown.

The additional (non trivial) operations of the Tuple class are:

- addEntry: this adds a new field at the end of the tuple list.
- getArity: retrieves the number of fields of this tuple.

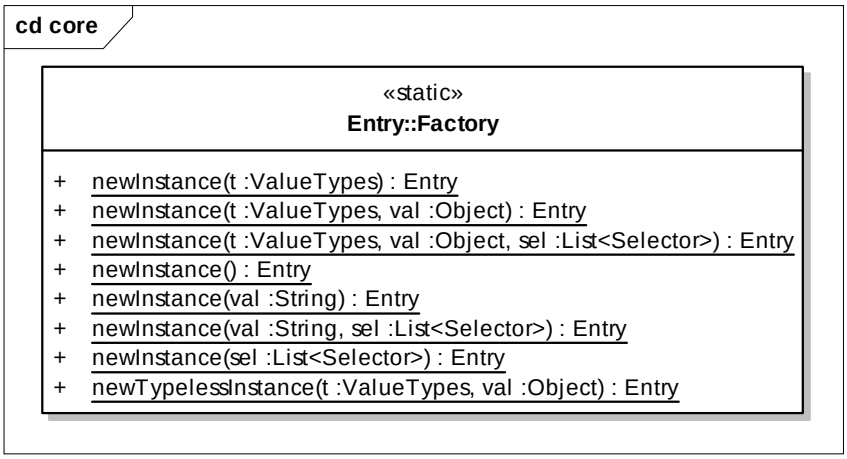

Figure 15: Entry::Factory class diagram

The Entry::Factory methods are rather simple, these are:

- newInstance: creates a new object of type Entry which is typed correctly according to the ValueType
- newTypelessInstance: creates a new instance of type Entry which has not set its type information according to the value's type. It rather contains a string value and the ValueType information. This method must not be used by anyone but the CADAO.

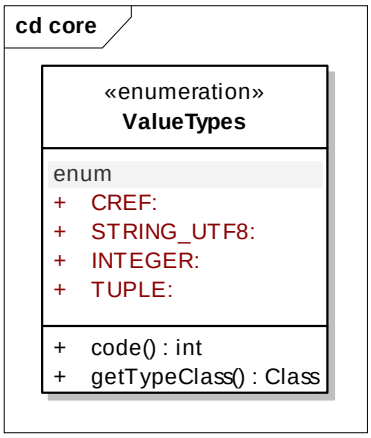

Figure 16: ValueTypes class diagram

The enumeration values of the ValueType class are rather self-explanatory. Each of the values is mapped to a Java type, which can be retrieved using the getTypeClass method. The value-type mapping is as follows:

- CREF org.xvsm.api.core.ContainerRef
- STRING_UTF8 java.lang.String
- INTEGER java.lang.Integer
- TUPLE org.xvsm.api.core.Tuple

VI.2.2. Selector

Just as the name suggests selectors in XVSM are used to select entries within a container. Additionally these selectors are used to specify the needed book-keeping information for explicit coordination types when writing entries. Figure 17 gives an overview about the class structure of XVSM's selectors. The following selectors are available, one for each coordination type. Most of the selectors are self explanatory still they're briefly described in this chapter.

- SetSelector: this selector randomly selects the given amount of entries from the container. The related coordination order is RANDOM.
- FifoSelector: this selector retrieves entries in a first in first out (queue) fashion. The related coordination order is FIFO.
- LifoSelector: this selector retrieves entries in a last in first out (stack) fashion. The related coordination order is LIFO.
- LruSelector: this selector retrieves the entries in a least recently used fashion. The related coordination order is LRU.

- KeySelector: this selector retrieves the entries based on the provided key. The key type and key value must be given. While an entry can have multiple keys which are distinct through the key's name, the value of a key must be unique. Due to this unique nature of keys, only one entry can be selected at a time. The related coordination view is KEY.
- VectorSelector: this selector retrieves the entries based on the integer index. If more than one entry is selected the position depicts the starting index within the container along with the number of successive entries in the container. The related coordination view is VECTOR.
- TemplateSelector: this selector is used to match the provided template against the tuples in a container. Only entries of type tuple can be retrieved with this selector. The related coordination template is LINDA.

Figure 17 also shows that the Vector-, Template- and KeySelectors implement an interface called NotificationSelector. This interface is a so called tagging interface. It is needed to use the implementing classes polymorphically for parameters which would otherwise be unrelated. While in this case the three selectors would share a common super class, namely the Selector, there are other classes that have this super class as well. The NotificationSelectors are special selectors which can be used when creating notifications on entries. While this has not yet been implemented in the current version of the MozartSpaces it certainly is in the concept. Before providing full support for entry level notifications there are still a few other problems that need to be solved.

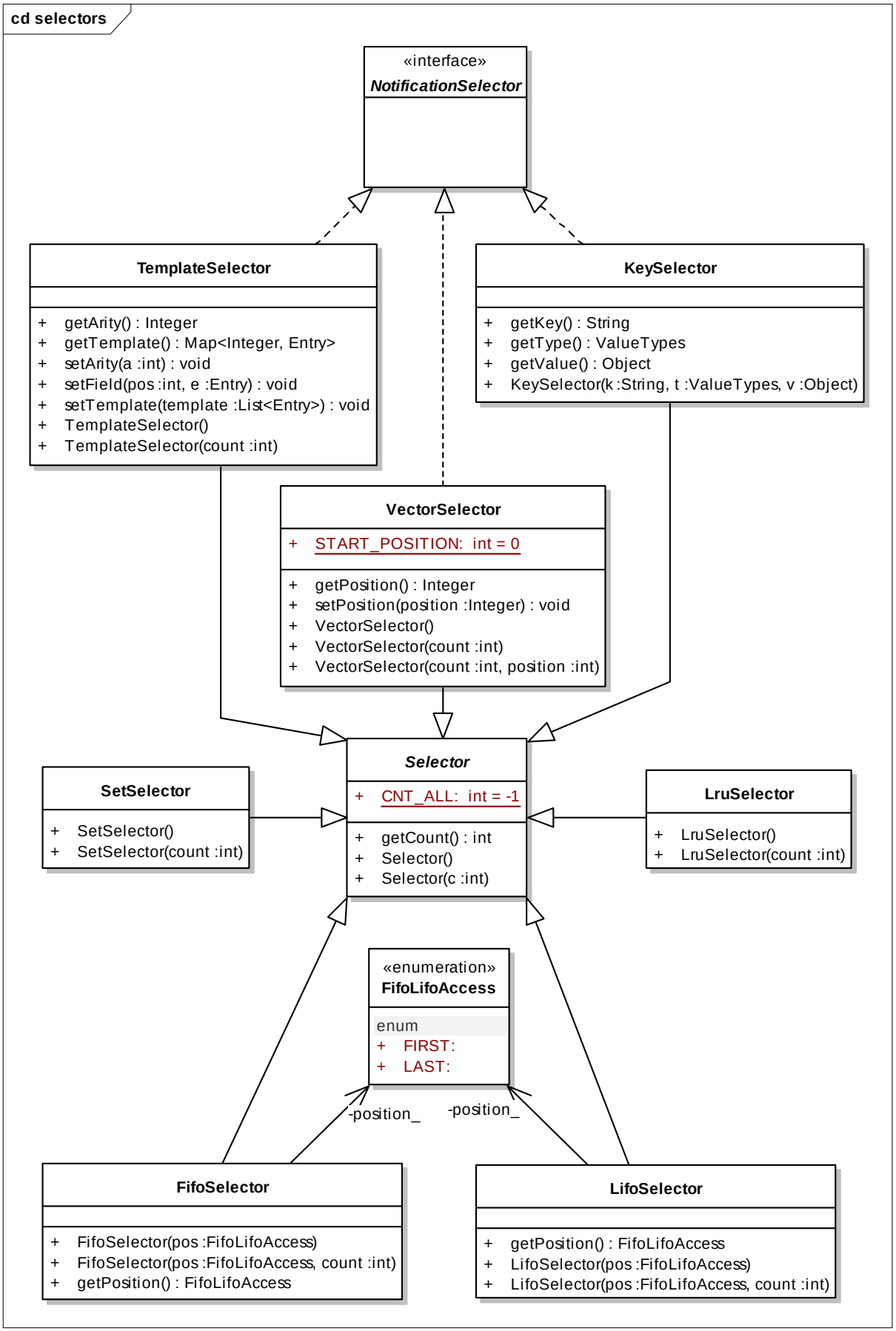

Figure 17: Selector class diagram

Whenever a selector is chosen it should be checked whether the selector is compatible with the coordination types of the container, otherwise the XCore will respond with an error.

VI.3. Core API Operations

The Capi operations can be grouped into several distinct functionalities. These will be described in more detail in the following subchapters. Figure 18 shows the available methods in the Capi's class diagram.

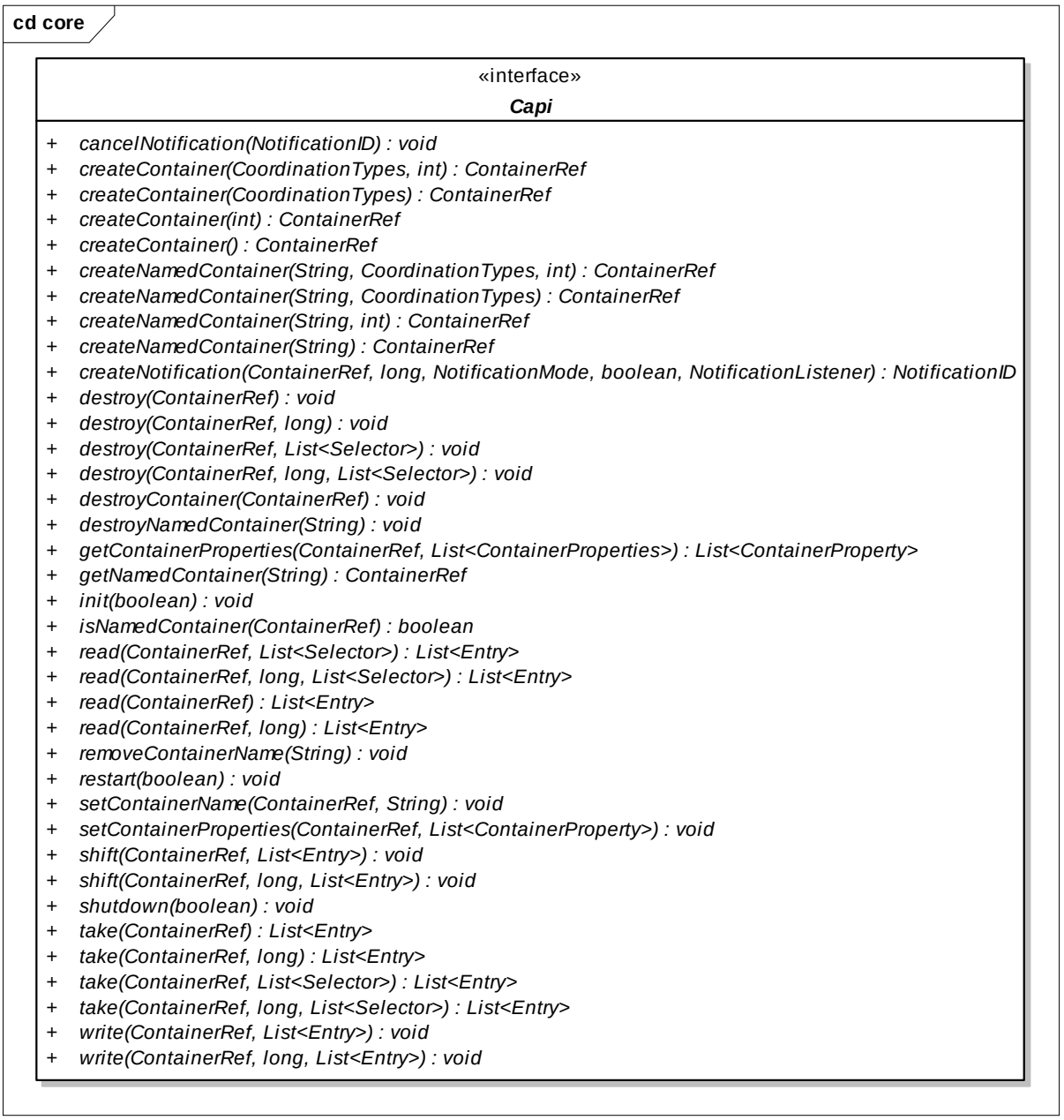

Figure 18: Capi Interface diagram

The Core API's operation categories are

- Core management operations: these usually are only needed once and not during the production life of a program. They correctly initialize and shutdown the XCore.
- Container operations: methods that manipulate containers and not their contents.
- Entry operations: these methods manipulate entries, either store them in a container retrieve or delete them.
- Notifications. Notifications can be created that monitor changes on containers.

VI.3.1. Core management operations

- init: this method is used to correctly initialize the XCore and all its systems. Most importantly it initializes the backend data store and starts up the connection between the data store and the CADAO.
- shutdown: frees the used system resources and correctly shuts down the backend data store.
- restart: this basically is a shortcut method for shutting down the XCore and starting it up again.

VI.3.2. Container operations

A container is an entry holding entity. As container operations we understand operations that directly manipulate containers in contrast to entry operations which manipulate the contents of a container. The functionality for these operations is implemented by the CADAO and exposed through its interface. These methods are used directly by the Capi to provide the container operations to the client. These operations are:

- createContainer: creates a new container in the space which can then be used by the client.
- createNamedContainer: creates a new container just as createContainer does. With this operation though the client can bind a name to the newly

created container. This name can be used for container lookup at a later point in time.

- destroyContainer: destroys an existing container.
- destroyNameContainer: this method destroys an existing named container, rendering the name invalid.
- getContainerProperties: retrieves the specified properties of a container. Such might be the size, the current fill size or the supported coordination types.
- getNamedContainer: this retrieves the container handle (a so called Container Reference) for a container name.
- isNamedContainer: checks whether a certain Cref depicts a named or just an ordinary container.
- removeContainerName: this method removes the binding of a name and a container so the name is free to be used for another container.
- setContainerName: binds a container name to a Cref.
- setContainerProperties: changes the specified properties of a container. Not all property changes are allowed at all times. E.g.: trying to remove a supported coordination type while there are entries stored within the container results in an error.

VI.3.3. Entry operations

An entry is an entity which may be stored in a container. (More details about entries can be found in chapter VI.2.1) For each of the five possible operations that can be carried out for entries the Capi creates a new corresponding workflow (see chapter VII) object which then fulfils the task. Workflow objects are not created directly but retrieved from a factory[25]. For the loading of the workflow classes and the creation of these workflow objects, again the spring framework's methods and implementations are used. The following code segment shows the usage in principal.

[25] Check computer science literature on design patterns, specifically the ‚Factory' Design Pattern. A good reference for design patterns is [Gamma, Helm, Johnson, Vlissides, 1995].

```
ClassPathResource res = new ClassPathResource("Workflows.xml");
XmlBeanFactory factory_ = new XmlBeanFactory(res);
    Workflow wf = (Workflow)factory_.getBean(wfname);
```

Listing 5: Workflow creation with the factory

The advantage of loading workflows through the spring framework and creating them with the factory pattern lies in the easiness of exchange. Loading another workflow implementation is simply a change of just one line change in the configuration file. This comes handy especially for testing when one needs to test correct behaviour when working with workflows and their possible states and conditions.
The following section is taken from the implementation's 'Workflows.xml' file. Using a different workflow implementation can be injected by changing the 'class' attribute.

```
<bean id="ReadWorkflow"
    class="org.xvsm.ms.workflows.ReadWorkflow" singleton="false"/>
```

Listing 6: ReadWorkflow bean configuration

- read: entries stored in a container will be retrieved and returned to the client if they match the list of selectors provided.
- take: this operation has the same semantics as the read operation in the sense of retrieving entries. Though, other than read the retrieved entries are removed from the container.
- destroy: again entries are chosen using the selector matching and are then removed from the container without returning the matching entries to the client.
- write: new entries are written to the container if there is still enough free storage in the container.
- shift: again new entries are written to the container, though if there is not enough free storage, existing entries are 'shifted' out of the container using its base coordination order.[26]

[26] Details about coordination orders can be found in chapter III.

VI.3.4. Notifications

Notifications are used to inform about a certain event that occurred.[27] In XVSM notifications can be reduced to simple call-back functions.[28] In XVSM notifications can be created on the level of containers. This means that whenever an entry is written to a container that a notification has been registered with, the notification target will be notified. The Capi operations for notifications are:

- createNotification: creates a new notification for container
- cancelNotification: stops an existing notification

Notifications are managed in the Capi implementation. This means that the Capi needs to keep track of all active notifications. This is done using a hash map and since the Capi is used as singleton and multiple threads may need access to this hash map concurrently, it must be a thread safe implementation.

```
ConcurrentHashMap<ContainerRef, Map<NotificationID, Notification> > notifications_
= null;
```

Since multiple notifications might be active for a single container at the same time, the value type of the hash map must be some kind of container type. A map from NotificationIDs to Notifications was chosen for a simple reason. It supports fast removal of a specific NotificationID whenever the cancelNotification operation is invoked.

Creating a new notification is shown by the following code segment (operation parameters for the Notification constructor have been omitted for readability reasons):

[27] Notifications can be found in many aspects of life and in various forms: when a fire is discovered, the fire fighters are *notified* through a pager or phone call; when a registered letter has been delivered the sender is notified with a note in his/her post box..

[28] In the embedded version this is correct, but in the standalone version notifications are more complex. Details will be given in section XI.4.

```
Notification not = new Notification ();
Map<NotificationID, Notification> tempmap = notifications_.get(cref);
if (tempmap == null) {
    tempmap = new ConcurrentHashMap<NotificationID, Notification>();
}
tempmap.put(not.getId(), not);
notifications_.put(cref, tempmap);
```

Listing 7: Creation of Notifications in Capi

Notifications are created with a specific mode (Figure 20), for a specific container and a NotificationListener.

Figure 19 shows the class diagram of a notification. It comprises a given timeout, the notification mode, the notification's identifier and of course the call-back object. The remaining fields are used internally for bookkeeping and implementation support.

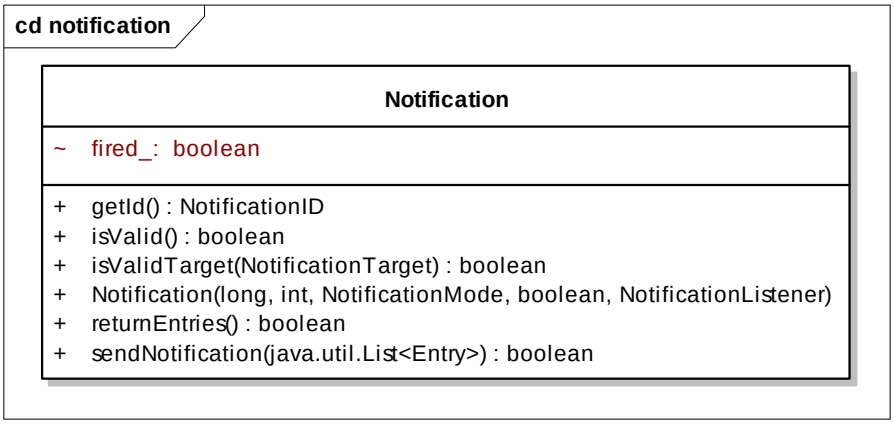

Figure 19: Notification class diagram

The only non trivial or self explanatory method is sendNotification. This method is used to invoke the NotificationListener's call-back operations. Since this method still runs in the same thread as the XCore, problems arise when the call-back operation does other tasks than a trivial flag setting. In the worst case the client sends the call-back thread to sleep which would then stall the whole XCore. This situation clearly is not desirable. That is why the sendNotification operation in the Notification class had been added and implemented in the following way:

```
timeout_.update();
if (!isValid())
    return false;
```

Listing 8: checking notification validity

As mentioned before, the first thing is to check whether the notification is still valid. Then there is just a simple conditional statement which checks whether the entries that were written, and therefore cause the notification, must be returned to the client.

In any case, the key functionality here is that a new Thread is created for each firing notification which will then be started and invokes the call-back method. This way only the newly (and other than that unneeded) thread will be stalled or delayed if the call-back method is fairly complex. The XCore itself will continue to work and accept client requests.

```
final java.util.List<Entry> lentries = entries;
    if (returnEntries()) {
        new Thread(new Runnable() {
            public void run() {
                listener_.sendNotification(getId(),
lentries);
            }
        }).start();
    }
    else {
        new Thread(new Runnable() {
            public void run() {
                listener_.sendNotification(getId());
            }
        }).start();
    }

    return true;
```

Listing 9: Notification Callback

The check whether a notification is still valid[29] the following two steps are performed; first, the validity of the timeout is checked and then the correct mode behaviour.

The notification modes can be used to specify the firing behaviour of a notification in the following way

- ONCE: the notification is invalid (and will be removed) after it has fired once
- PROLONG: whenever the notification fires, the timeout is reset to the executing thread's current time.
- RESTRICTED: the notifications fires every time while the timeout is valid
- INFINITE the notification fires every time and there is no timeout

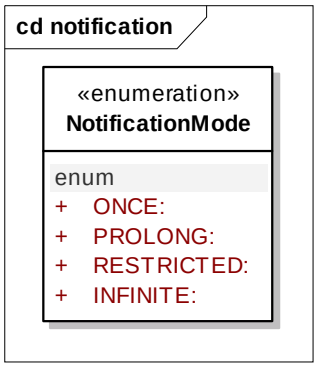

Figure 20 NotificationMode class diagram

Whenever an entry has successfully been written to a container and this container has an associated notification which is active then this notification fires. If a notification does not fire within the given timeout period it will be removed from the system (unless the notification mode is INFINITE).

This listener is the object that provides the call-back operation that will be invoked when a notification fires. Figure 21 shows the listener interface that the application requesting a notification must implement. If the notification was created in the sense of returning the entries, the second operation is invoked when the notification fires. Any entry that caused this firing will be returned to the application. This is useful since usually a fired notification is followed by a read operation requesting those entries.

[29] A notification is valid if the notification mode is INFINITE or the timeout has not expired.

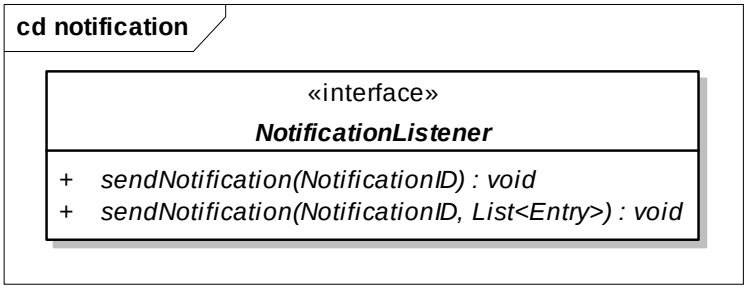

Figure 21: NotificationListener class diagram

Firing a notification works as follows:

```
Map<NotificationID, Notification> tempmap = notifications_.get(cref);
if (tempmap == null)
    return;

Iterator<Notification> it = tempmap.values().iterator();
while (it.hasNext()) {
    it.next().sendNotification(entries);
}
```

Listing 10: Notification firing

After checking whether the specified cref has any notification assigned their notification operations are invoked.

VI.4. Exceptions

The XCore can throw various exceptions depending on the operation that has been invoked and what the exact error reason was. There are two flavours of exceptions, RuntimeExceptions and 'normal' Exceptions. In the Java programming language exceptions must be declared in the exception list of an operation and therefore the caller must either add exception handling to their method (using a try/catch block) or declare the exception in the exception list to further promote it. But this is not the case with RuntimeExceptions. Neither do they have to be declared nor does the caller have to handle them (usually this

situation results in a program abortion with a stack trace). One of the most prominent examples of a RuntimeException is the NullPointerException.
In XVSM RuntimeExceptions are used for the following types of erroneous conditions:

- A fatal internal XCore condition occurred. The XVSM user has no chance of performing any actions that would 'repair' the XCore; therefore it doesn't make sense to force the client to handle the exception.
- Operation parameter might under some circumstances be used in a wrong way.[30]
- Again an internal XCore condition occurred but some resource that is used by the XCore imposes a problem (e.g.: no entries can't be written to the database)

The exception classes themselves do not have any special functionality therefore a simple list of the available XCore exception will be given without class diagrams.

VI.4.1. Runtime Exceptions

- FatalCoreException: a fatal internal error has occurred.
- InvalidParameterValueException: the parameter value of a method has an invalid value (e.g.: the list of selectors for the Capi.read() operation is **null**)
- IllegalValueTypeSpecifiedException: this exception is thrown when a ValueType (see Figure 16) is specified that does not match the generic template parameter of the entry.

VI.4.2. Operational Exceptions

- ContainerAlreadyLockedException: whenever entries are written to a container this container must be locked in the database. If the container is already locked this exception is thrown.

[30] According to the method's specification certain parameter values are illegal. Usually they would not be used in a wrong way but it's still possible to do so. Since this method is also used in the core itself and in that case it is guaranteed to be used in the right way, adding a normal exception would result in unnecessary code bloat within the core.

- ContainerAlreadyUnlockedException: whenever an unlocked container is requested to unlock this exception is thrown.
- ContainerFullException: bounded containers only have limited space for entries. If the write operation is invoked but there is not enough space to write all entries this exception is thrown.
- ContainerNameOccupiedException: one tries to either create a new named container or bind a name to a container which is already occupied by another named container.
- ContainerNotEmptyException: if a container coordination type should be removed and the container is not empty.
- CoordinationTypeNotSupportedException: whenever a container is accessed using a coordination type which is not supported by this container, this exception is thrown.
- CountNotMetException: whenever a read/take/destroy operation is invoked and the count parameter is greater than the number of matching entries.
- DoubleCrefException: a container is requested to be created with a specific cref that already exists.
- FatalException: Whenever an unexpected or exception or a condition that is not repairable occurs.
- SelectorNotCompatibleException: at least one of the provided selectors requires an unsupported coordination type.
- TimeoutExpiredException: the timeout for a given operation has expired.
- UnknownContainerException: an entry is read from or written to a container reference which is not valid.
- UnknownContainerNameException: a named container must be retrieved though the name does not depict any.
- WriteSelectorNeededException: thrown when a write selector is missing for explicit coordination types or invalid.

VII. Entry Workflow

For all entry operations (see chapter VI.3.3) a dedicated workflow exists. All specific workflow classes are derived from the abstract Workflow class (see Figure 22). Each XVSM workflow implements a certain list of tasks that are needed to fulfil the requested operation. The available workflows are

- ReadWorkflow
- TakeWorkflow
- DestroyWorkflow
- WriteWorkflow
- ShiftWorkflow

The most important purpose of the workflows is to trigger the correct CADAO methods in the correct order.

VII.1. Workflow principles

Most operations of the workflow class are protected. These are needed for internal use while processing each specific workflow. The public interface that the workflow clients use consists of three methods and the constructor.
A workflow can be used by two different entities in the XVSM system, the Capi on the one hand and the CADAO on the other hand. Whenever the Capi receives any workflow related method invocation (see chapter VI.1.1 for details) a new workflow is created and initialized. Immediately after that the workflow processing is invoked through its *start* method. The *note* method is not used by the Capi implementation but the CADAO. Whenever a workflow is in a blocked wait status the CADAO notifies, and therefore awakes it, if certain conditions are met.
Details about processing internals of the workflows are given later in this chapter. The remainder of this subchapter will explain the class diagram and shed some light on the purpose and function of blocking workflows.
The abstract Workflow class provides the following methods:

- block: this method is used to switch the currently executing workflow into blocked stated. All necessary bookkeeping and registration with the CADAO is handled by this method.

- init: initialization of the workflow with the container reference, the entry which must be written and/or contains the selector list for reading, a timeout depicting the maximum wait time of the workflow in case of blocking and the CADAO object that is used for data backend handling.
- lockContainer: the container specified by the container reference is locked when a workflow starts its processing. This method in combination with the block method is used for handling blocking workflows. Their interaction will be explained in detail later in this subchapter.
- notif[31]: the CADAO uses this method to notify a blocked workflow that the desired container has been unlocked and is available for further processing.
- process: this method probably is the most interesting one for all workflows. It implements the workflow specific behaviour and must be overridden by the concrete workflow subclasses.
- unlockContainer: removes the workflow's lock form the container.
- unlockContainerIfNecessary: only unlocks the container if it really has been locked by this workflow before.

The protected fields in the Workflow class are used for internal bookkeeping and processing. The source code documentation should be checked for a more detailed and thorough explanation.

VII.1.1. Blocking Workflows

There are two situations in which a workflow can block. Blocking in this sense means that the processing of the workflow's tasks stops due to certain conditions and stalls until it is instructed to continue its work. The definition of each workflow's functionality gives a clue about which workflows can block and which cannot. Details about each respective workflow are given in the following chapters but in principle all workflows but the ShiftWorkflow may block.

- Read/Take/DestroyWorkflow blocking: if the entry count of a read request cannot be satisfied the workflow thread registers itself at the CADAO as blocking and sleeps for the specified timeout period. If a new

[31] Note that the name ‚notif' was chosen since every class in Java inherits a 'notify' method from the Object base class which shouldn't be overwritten with application specific behaviour.

entry is written into the container while the workflow is still blocking the CADAO notifies this workflow. The notification cancels the thread's sleeping state and the workflow once again tries to fulfil the request. If there are still not enough matching entries available in the container the workflow blocks again. This block-notify cycle continues until the workflow's timeout has expired. It then cancels and returns an error to the client.

- WriteWorkflow blocking: whenever a number of entries are about to be written into a container but there is not enough room to store them, the WriteWorkflow will block. The notify-block cycle works in principle just as it does in the ReadWorkflow case. The only difference is that a WriteWorkflow is notified whenever an entry is removed form the container, therefore leaving more room than when the blocking occurred.

One important fact about blocking workflow must be noted here. Workflow instances are always created and invoked in the thread context of the invoking thread. Therefore, if the workflow blocks, this whole thread context stalls. This is important for application designers and implementers. By default workflow have a synchronous fashion that is if an application invokes a read request which happens to block, the whole application is stalled until the ReadWorkflow finishes. If such behaviour is not desired the application developer must create a separate thread and invoke the read operation within this new thread's context. Another important detail is that while a workflow might block the whole application, the MozartSpaces XCore itself will not be blocked. Therefore if the developer invokes a read request in a dedicated thread which happens to block it is still possible to invoke other requests from the main or different dedicated threads. If this would not be the case, there would be no possibility to break the workflow's blocking behaviour other than by an expiring timeout which clearly would be a highly inappropriate characteristic.

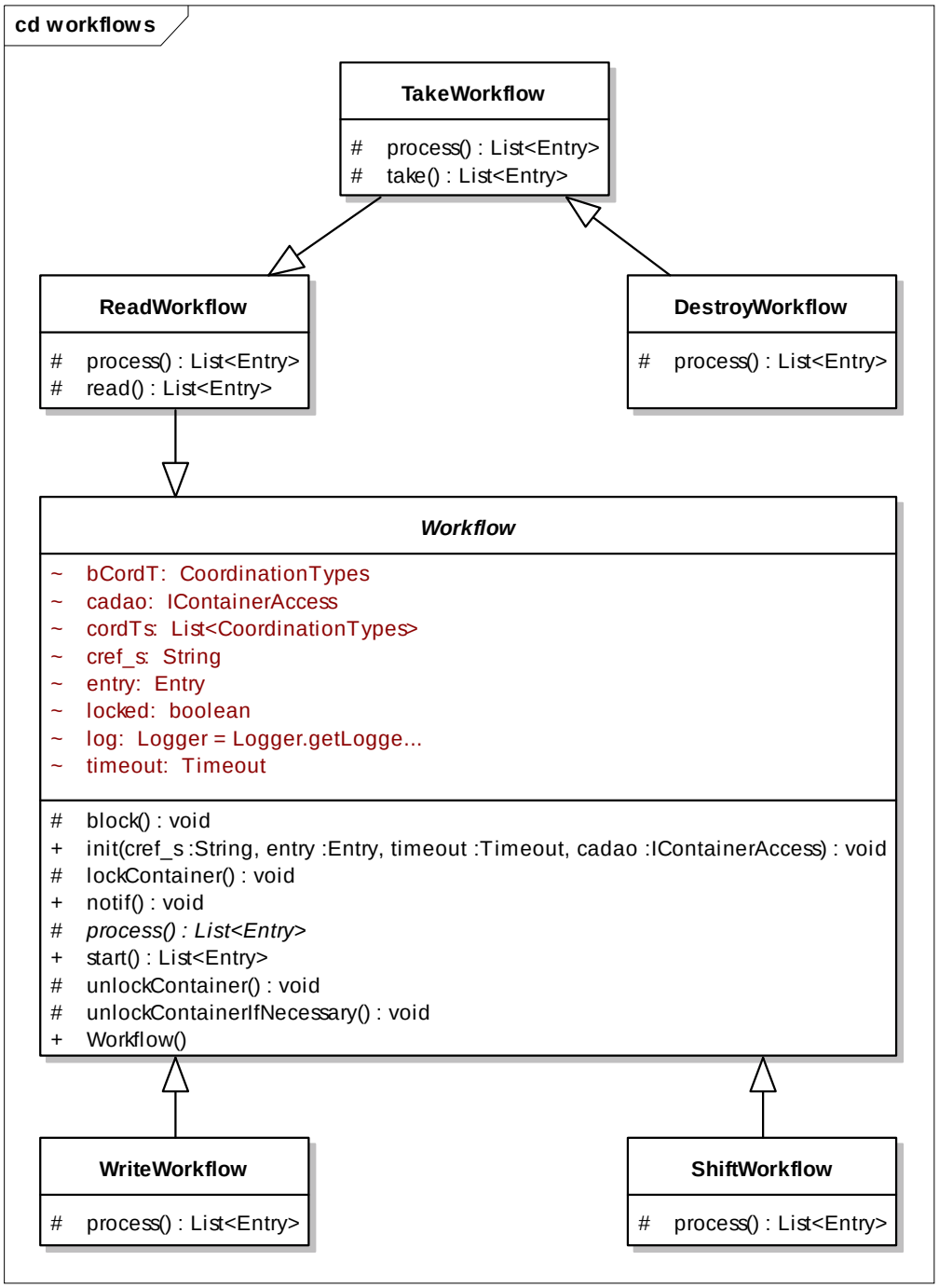

Figure 22: Workflow class diagram

VII.1.2. Workflow

The Workflow class is the abstract base class for all concrete workflow implementations. By implementing the template method design pattern all workflow subclasses only need to override the process method. This method contains the workflow specific part of the algorithm; all common functionality is already implemented in the Workflow class itself.

Invoking the workflow's start method will initiate the process of reading entries from or writing entries to a container. The algorithm itself works as follows:

1. Check the validity of the given selectors with the SelectorOptimizer helper class.
2. Lock the target container.
3. Run the workflow specific part of the algorithm (process)
4. Unlock the container
5. Return the results in case of successful processing or an error otherwise.

Figure 22 shows all available workflow classes that are available in the MozartSpaces. In principle they can be divided into two categories: READ and WRITE. The READ and WRITE specific workflows will be discussed in chapter VII.3 and chapter VII.4 respectively.

VII.2. Creation of a Workflow

The creating of the concrete workflows is done by exploiting another well known design pattern, the factory method pattern. While using this pattern doesn't provide any obvious advantages for the production code it is very helpful to inject stripped down and configurable workflow instances during test runs.
A quick look at the entry operations of the Capi (see chapter VI.3.3) shows that each has its own concrete workflow implementation. The respective mapping is as follows:

- Capi.read() ReadWorkflow
- Capi.take() TakeWorkflow
- Capi.destroy() DestroyWorkflow
- Capi.write() WriteWorkflow
- Capi.shift() ShiftWorkflow

Since the initialization and the binding to the target container and the desired entries/selectors is already implemented and handled by the abstract Workflow class (see chapter VII.1) no additional information is needed for the process operation.
Figure 23 shows the principle of the workflow creation. This process is the same for all available workflow classes with the exception of the factory method

parameter. Therefore only the sequence diagram for the ReadWorkflow is given as an example. (Note: parameters that are unnecessary for the understanding of the algorithm have been replaced with '…' for brevity and clarity reasons).

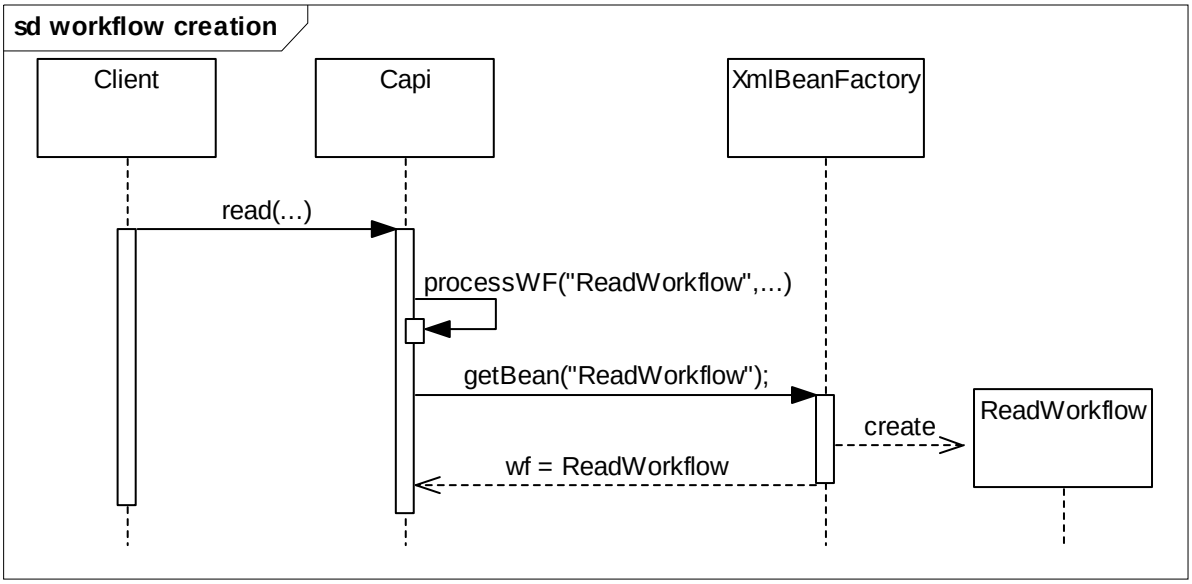

Figure 23: Workflow creation sequence diagram

The XmlBeanFactory in Figure 23 is a factory class implementation that is part of the Spring Framework[32]. This factory class is configured with special XML files where the 'bean name', the parameter to the factory method (in this case the getBean method) is linked with the class that implements the desired workflow. Exchanging one workflow implementation with another one is simply a matter of changing the configuration file and no compilation is required.

VII.3. READ Workflows

There are three workflows that handle the different READ requests. These are read, take and destroy. The concrete functionality will be discussed in the following subchapters.

VII.3.1. Capi.read/ReadWorkflow

Figure 24 show a sequence diagram of a read request.

[32] http://www.springframework.org/

Note: Figure 24 shows only a simple *create* message for creating a new workflow instance. In reality this is slightly more complex involving a factory method which depicts and creates the instance. Details can be found in chapter VII.2.

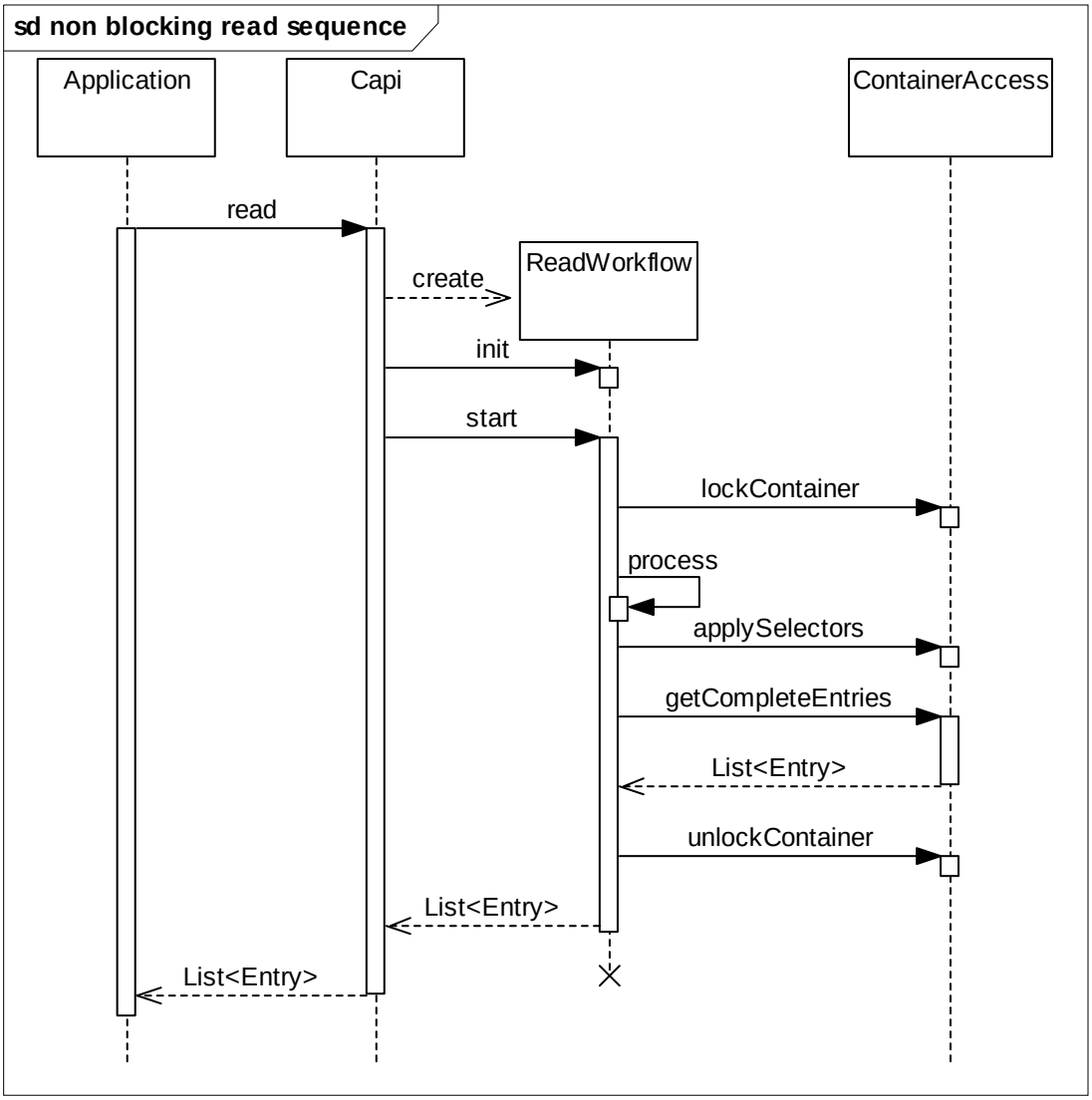

Figure 24: Sequence diagram for the read workflow

The application invokes the read method of the Capi. A new ReadWorkflow instance is created and initialized with the request parameters of the application. These request parameters include the target container reference, a list of selectors that depict the desired entries and optionally a timeout for the operation. After this initialization process the workflow sequence as it has been described in chapter VII.1.2 is started. The workflow in Figure 24 shows a non-blocking version i.e. the read request can be satisfied immediately. After the Capi has received the desired entries from the workflow this workflow instance

is no longer needed and can be destroyed. In the case of the MozartSpaces implementation the reference to the instance is dropped and the Java GC takes care of removing the object.

ReadWorkflow's implementation of the process method invokes an internally private read method. This is needed since ReadWorkflow's subclasses Take/DestroyWorkflow need a slightly different implementation. The read method tries to optimize the selectors that were given in the request and then prepares the CADAO for a read request. Applying all available selectors will either work properly or result in an exception if these selectors can't be satisfied. In case of an exception the workflow then blocks (see Figure 25; only the blocking part between the ReadWorkflow and the CADAO are shown, the surrounding method invocations are equal to those that are shown in Figure 24). If the selectors can be satisfied the workflow retrieves the entries from the CADAO are and returns them to the Capi.

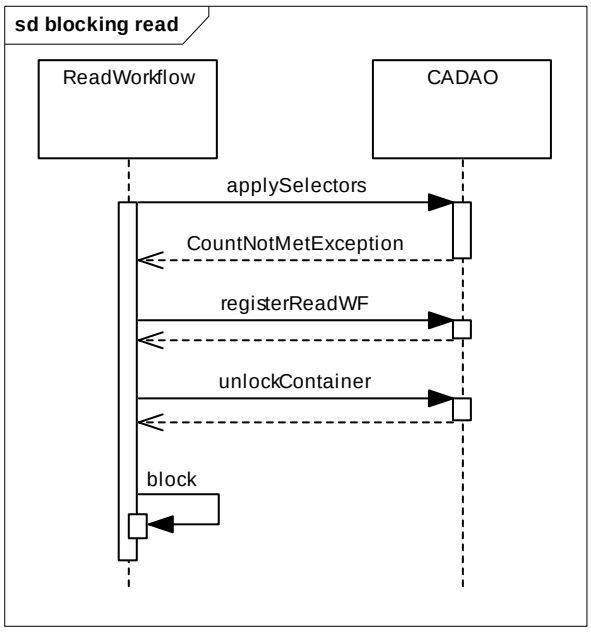

Figure 25: Sequence diagram for the blocking read workflow

To better understand the actions going in during a read request Figure 26 shows an activity diagram of the ReadWorkflow's process method.

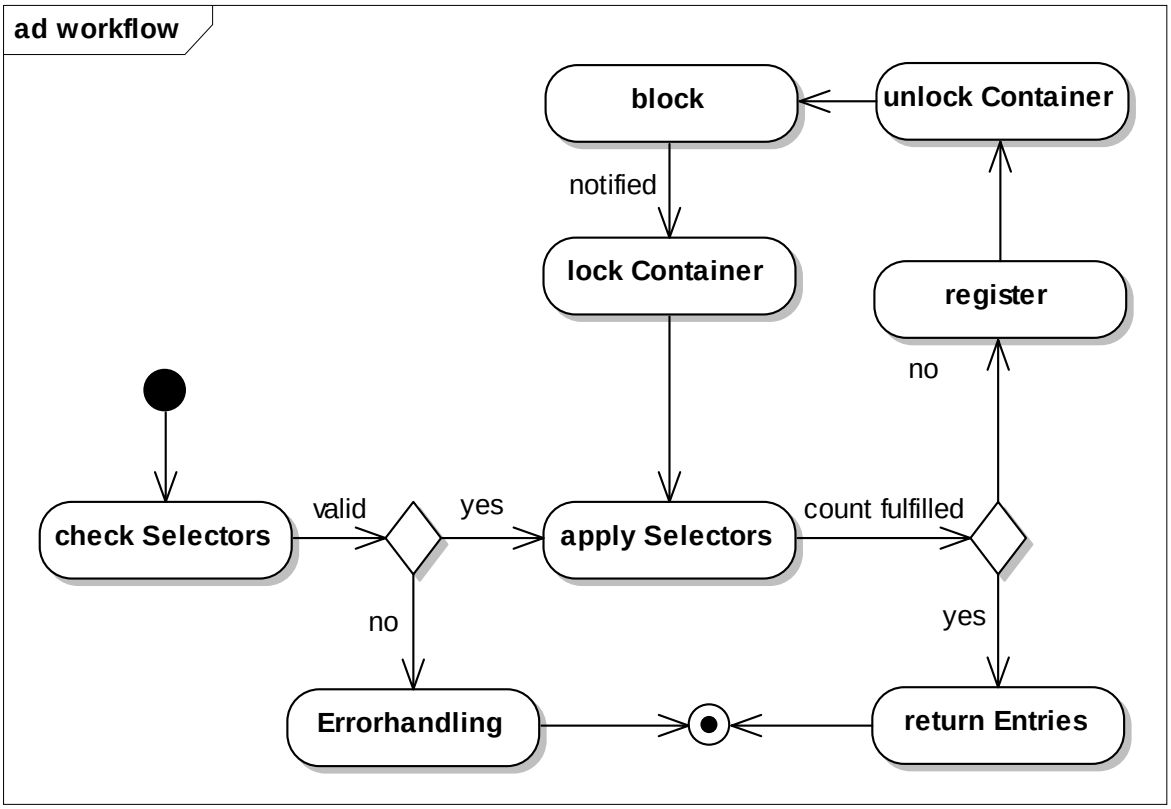

Figure 26: Main activities of read operation

- Purpose

  A ReadWorkflow can be seen as a query for entries against a container. Entries are selected through the application of a number of selectors. While ReadWorkflows are, or more precisely can be, blocking operations this behaviour can be suppressed by specifying a timeout value of 0. That way, whenever no entries can be found the workflow immediately returns to the calling application.

VII.3.2. Capi.take/TakeWorkflow

Figure 27 shows the sequence diagram for the take operation. In principle this is the same as the read operation. This property is even further emphasized by the fact that the take workflow executes its super class' read method. The read method handles the selection of the correct entries with the CADAO. In the same sense the blocking is also handled by this operation. The only difference between the take and the read operation in the workflow implementation is the invocation of the CADAO's destroy method. This removes the previously selected

entries from the target container. Details about this operation will be discussed in chapter VIII.1.2.

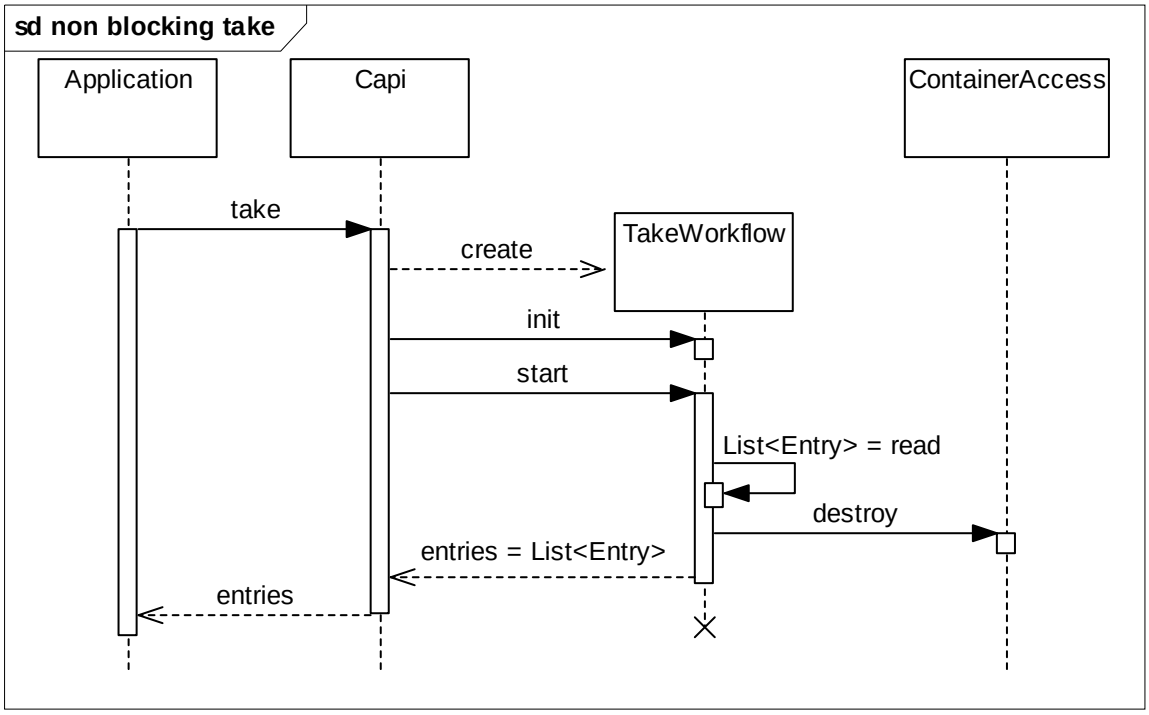

Figure 27: Sequence diagram for the take workflow

- Purpose

  A TakeWorkflow can be seen as a ReadWorkflow that is immediately followed by the removal of the selected entries from the target container. This is one operation; therefore either the entries are read and destroyed or nothing happens at all. It is not possible to read the entries but not to remove them from the container.

VII.3.3. Capi.destroy/DestroyWorkflow

Figure 28 shows the sequence diagram of the destroy operation. Taking a closer look at the diagram reveals that this in fact is exactly the same as the take operation with one minor exception. The entries that are retrieved with the workflow's take operation are not returned to the Capi. This results in effectively removing the selected entries from the container.

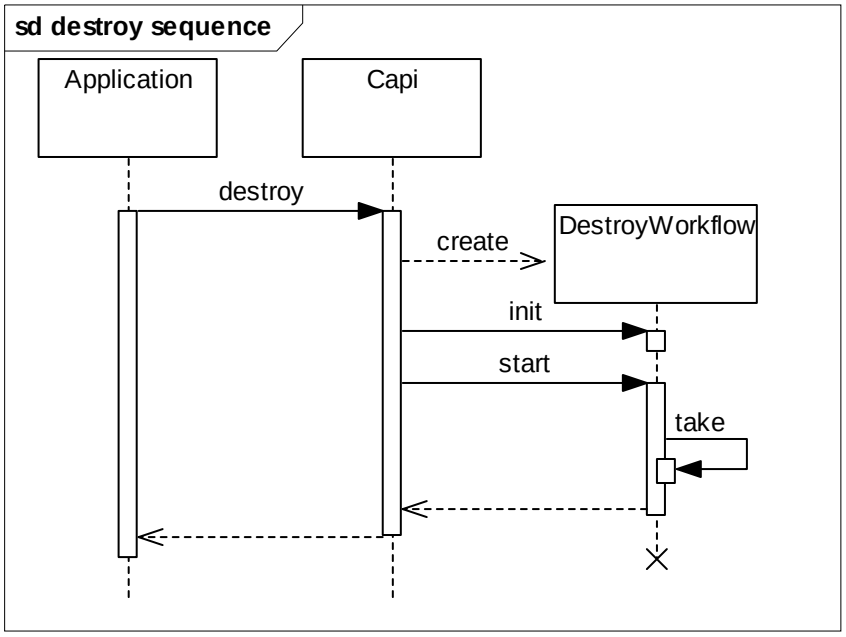

Figure 28: Sequence diagram for the destroy workflow

- Purpose

  A DestroyWorkflow can be seen as a TakeWorkflow without returning the selected entries. In other words, this workflow removes the selected entries from a container.

VII.4. WRITE Workflows

In XVSM there are two different kinds of WRITE request. These are write and shift. The concrete functionality will be discussed in the following subchapters. Entries are always written into the container using the base coordination order. The explicit selector values are merely meta-information that can be used to read entries in a more sophisticated manner.

VII.4.1. Capi.write/WriteWorkflow

The WriteWorkflow is the one WRITE request that may block. A write operation inserts the specified entries into the target container if and only if there is enough room to add them. Enough room in this sense means two different things equally. If the target container does not contain other coordination types other than coordination orders then this property only applies to the container's

size. If on the other hand there are explicit (see chapter III) coordination types involved then the selectors' values must not yet be occupied. An example would be a container that has the KEY coordination view enabled and an entry with the key value of 'key1'. Let's further assume that the container is unbounded and therefore there is enough room for any additional entries. If the client tries to write an entry whose key specifies the values 'key1' it will not be possible and block the operation since the key is already occupied.
Figure 29 shows a sequence diagram of the non-blocking write operation. Again the workflow locks the target container, writes the entry into the container and then unlocks the container again.

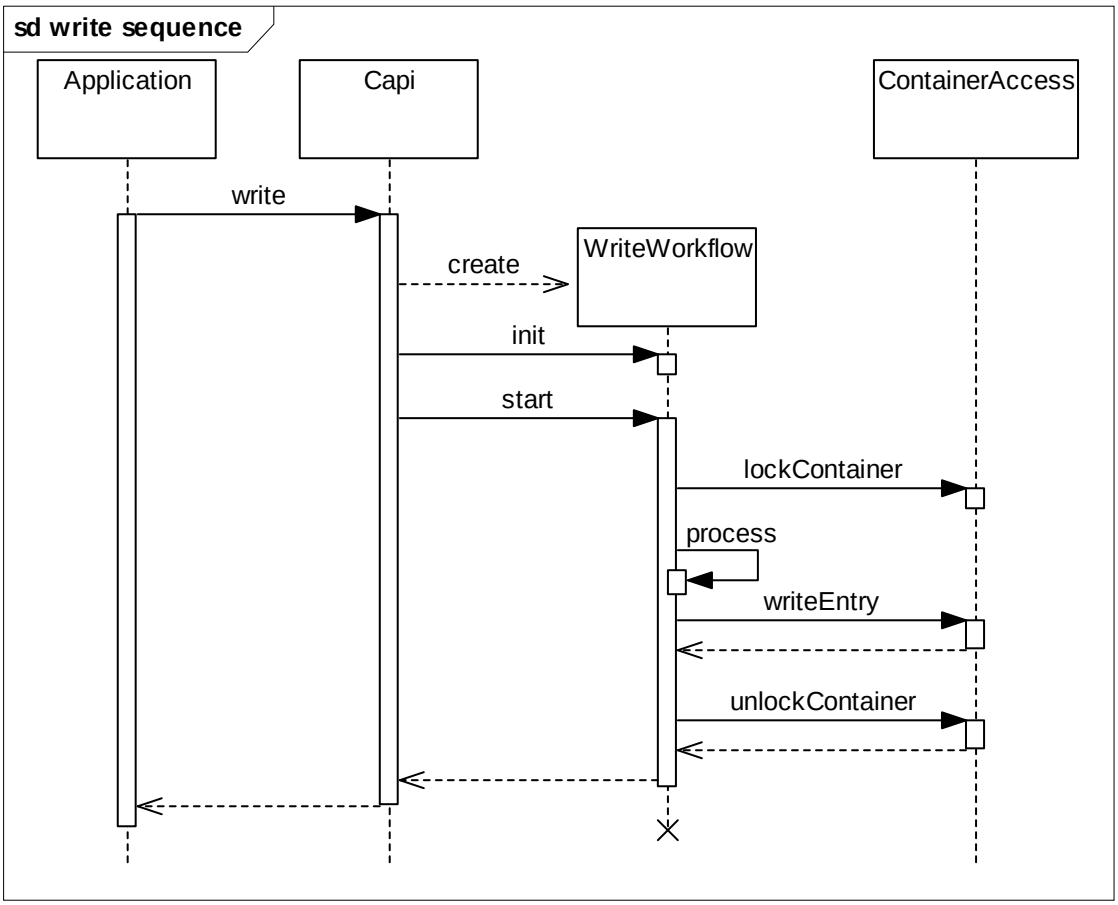

Figure 29: Sequence diagram for the write operation

Figure 30 shows the workflow details of the blocking case. If the container does not provide enough unoccupied room, or if the value of the explicit selector contains an already available value the operation must block. The workflow therefore registers itself at the CADAO and unlocks the container. The CADAO

will notify the registered workflow whenever entries are removed from the container or if the container's size is changed.

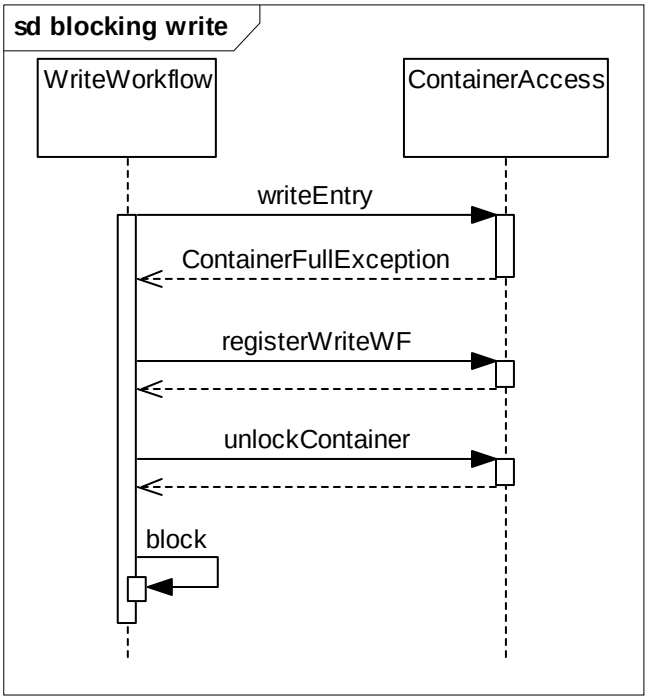

Figure 30: Sequence diagram for the blocking write operation

- Purpose

  The WriteWorkflow is used to add new entries to the specified target container. If there is not enough room or if the explicit selector values are occupied the operation blocks. The same coding principles for blocking operations apply here just as they have been explained in the previous chapter.

VII.4.2. Capi.shift/ShiftWorkflow

Figure 31 shows the sequence diagram of the shift operation. The ShiftWorkflow is XVSM's only workflow that cannot block. In principle it is a non blocking write operation. This behaviour is accomplished by modifying the write operation properties as follows. If the container does not provide enough unoccupied room for the entries they are 'shifted out' of the container by applying the base coordination order. In principle this means that prior to writing the entries into the container a base order destroy request takes place.

The number of entries that must be destroyed can be calculated by subtracting the number of free places in the container from the number of entries that must be written. Also if an explicit selector value is already occupied this entry is replaced by the new one. Note: this does not effectively remove the entry from the container if merely removes the link from the meta-information to the entry. Taking a KEY enabled container as an example and informally speaking the key's target is switched to the new entry.

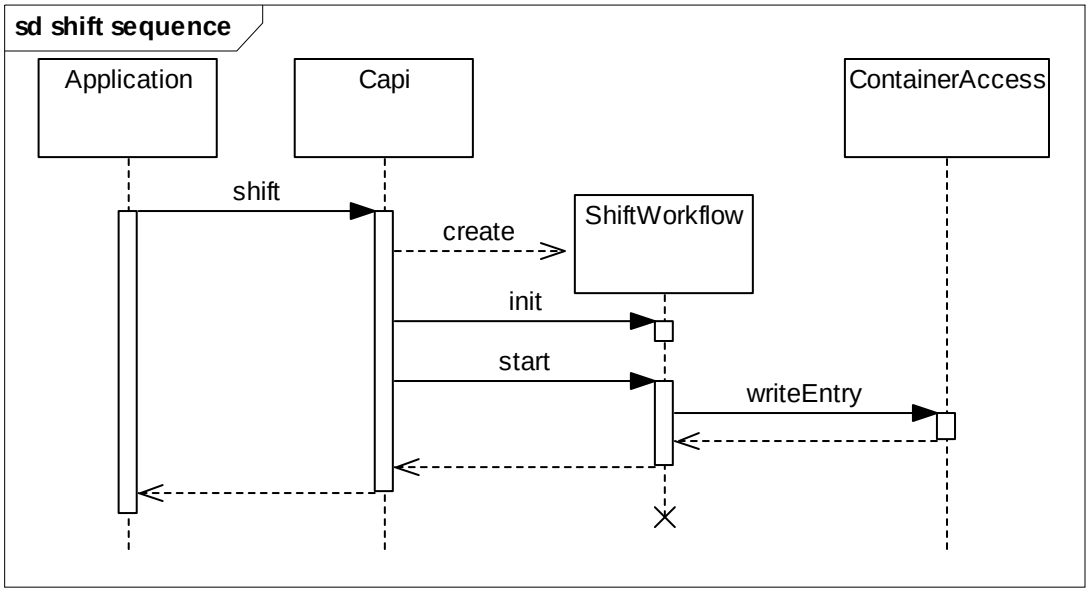

Figure 31: Sequence diagram for the shift operation

- Purpose

  The ShiftWorkflow can be seen as non blocking WriteWorkflow with eventual entry overriding.

VII.5. Optimizing Selectors

While the name of the class that is shown in Figure 32 partly misleading it still covers a big part of its purpose. The most important task of the SelectorOptimizer is to check the validity of the given selectors for the specified container. Only then an optimization trying to minimize the work that must be done in the CADAO makes sense.

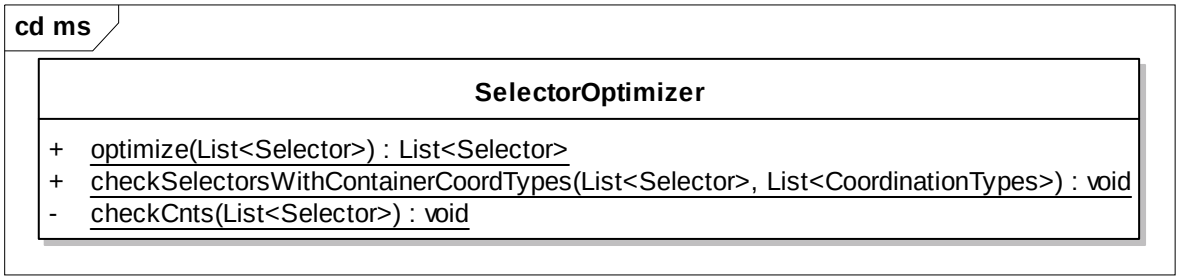

Figure 32: SelectorOptimizer class diagram

The interface of the SelectorOptimizer class only contains three methods each of which are static. Since there is no internal state that must be managed there is no need to use a normal class and create instances of this class.
The methods of the interface are the following:

- optimize: This method receives the list of selectors that has been provided to the workflow class through the entries. It is used to optimize and therefore minimize the work for the CADAO.
- checkSelectorsWithContainerCoordTypes: Since there is a predefined and specified connection between the existing coordination types and selectors only matching selector-coordination type pairs are valid. This method checks this validity.
- checkCnts: Each selector can specify the number of entries that must be selected. This method is used to check whether the count values of subsequent selectors are sensible and valid.

The details about selector matching will be given in chapter VIII but the most important fact about selectors is that they are sequentially applied to the result set of the previous selector application. The selectors therefore form a logical AND kind of conjunction.

VII.5.1. Checking Validity

There are two checks that must be performed to ensure the validity of the provided selectors. If the selectors are not found to be valid a SelectorNotCompatibleException is thrown.
The first check that must be done is the coordination type matching. This means that only selectors can be used, which are supported by the container's

coordination types. Table 2 gives a small and intuitive example of both a valid and an invalid selector list. In the valid case the container has the FIFO, LRU and KEY coordination types enabled. The request contains both a Fifo- and a KeySelector. The FifoSelector is covered by the FIFO coordination type and the KeySelector by the KEY coordination type. Therefore the selectors of this request are valid. In the invalid case the container only has enabled the LRU and RANDOM coordination types. The request selector on the other hand contains a KeySelector which is not covered by any of the available coordination types. This request therefore is invalid and results in a SelectorNotCompatibleException being thrown.

validity	***Selectors***	***Containers Coordination Types***
Valid	FIFO, KEY	FIFO, LRU, KEY
Invalid	KEY	LRU, RANDOM

Table 2: Selector/Coordination Type matching

The second type of validity check involves the count values of the selectors. Since the selectors are applied in a subsequent manner by definition it is clear that a selector's count value can only be equal or less than the previous selector's count value.

A simple valid example would be two FifoSelectors where the first selector requests 5 entries and the second one requests 2. Using the same value and exchanging the count values will result in an invalid request. Clearly 5 entries cannot be selected from a set that only contains 2 entries.

VII.5.2. Optimization

While optimization has not yet been implemented in the MozartSpaces a few ideas can already be given and are subject to be available in a future version of the MozartSpaces.

These optimizations are only a few examples and there are quite a few other optimization possibilities. In any case caution must be taken when optimizing the selectors. Before specifying and implementing new optimizations it must be proven that the end result is correct and equal to the not optimized case. This can be a difficult task and during the course of the specification and

implementation of the MozartSpaces quite a few cases have been found that intuitively looked like a good optimization possibility. On a second more thorough look though, it turned out that these would in fact change the semantic behaviour of the operation.
In some cases extensive profiling must be done prior to moving these optimizations into production code. It might be possible that simply running through selector processing in the CADAO is faster than performing complex optimizations beforehand.

The simplest optimization contains subsequent selectors for the same base coordination order (namely the SetSelector, FifoSelector, LifoSelector and LruSelector). Any number of equal selectors from this list can be replaced by a single selector that has the count value of the last (and therefore the smallest) one.

TemplateSelectors are already more difficult to optimize correctly.
One optimization regards the arity of the tuples. If multiple template selectors request different arity tuples this request cannot be fulfilled correctly. Clearly a tuple with arity of 3 cannot be selected among a set of tuples with arity 2 or 4.
The second optimization involves the types of a tuple's fields. Whenever subsequent TemplateSelectors request different types for the same field position the TemplateSelectors are incompatible. Once again an intuitive example would be the following: The first TemplateSelector requests a string in the field at position zero while the second TemplateSelector requests an integer for the same field. Of course this optimization can be extended to concrete values not only types (See Table 3 for an example).
A third optimization includes type/value narrowing. In this case the selectors can be merged. Assume a TemplateSelector that requests a field to have a specific type and a subsequent TemplateSelector requests the same type or a concrete value (of the correct type) for the same field. Additionally mutual 'don't care' conditions for any fields do not invalidate this optimization. If these properties hold for all specified fields, then these TemplateSelectors can be merged.
Table 3 sums up the previously mentioned optimizations. Cases where a SelectorNotCompatibleException would be thrown are marked as INVALID in

the third column. Cases where subsequent selectors can be merged are marked as MERGED, followed by the merged TemplateSelector.

[ ] arity = 4	[ ] arity = 5	INVALID
[string, ?, int]	[int, ?, int]	INVALID
[string, ?, int]	[1234, ?, int]	INVALID
[‘hello’, ?, int]	[int, ?, int]	INVALID
[string, ?, int]	[‘hello’, ?, int]	MERGED, [‘hello’, ?, int]
[string, ?, 3]	[‘hello’, ?, int]	MERGED, [‘hello’, ?, 3]
[string, ?, ?]	[‘hello’, ?, int]	MERGED, [‘hello’, ?, int]
[string, ?, 3]	[?, ?, int]	MERGED, [string, ?, 3]

Table 3: TemplateSelector optimizations

Compared to the TemplateSelector the VectorSelector optimization is simple. The request is valid if and only if all VectorSelectors select the same position.

VIII. Container Access Data Access Object

The MozartSpaces implementation of the XVSM uses the Derby[33] database as space backend. The name Container Access Data Access Object is rather long but reflects its origin. Data Access Object (abbr. DAO) is the name of a design pattern. This pattern allows the abstraction of access to certain resources, usually data storing and data providing resources. Examples for such resources are databases, streams or files. Using the DAO pattern enables the developer to simply exchange the DAO implementation without the need to adapt or perform changes in the application itself. This way it is rather simple to exchange the Derby database with another database a RAM disk implementation or even direct file access[34]. The implementation class of the DAO in the MozartSpaces is called ContainerAccess[35].

VIII.1. ContainerAccess API

Figure 33.shows the class diagram of the ContainerAccess class. The interface of this class might look rather complex but most operations have already been informally introduced. The list of methods along with explanations of their purpose will be given in the following subchapters. An important note is that container references are simply strings. The ContainerRef class is only a wrapper class for a unique string but this string value has already been extracted at time of invocation of the CADAO methods.

[33] Derby is an OSS database implementation written in Java. As of version 1.6.0 of the Java programming language this database is included in the JDK as JavaDB.
[34] Note: these are only examples that emphasize the advantages of the DAO design pattern and not intended as concrete proposals for the change of the MozartSpaces implementation.
[35] Even though the class itself is just named ContainerAccess during the process of specifying and implementing the MozartSpaces the name CADAO has been used informally as name for writing documents and discussion.

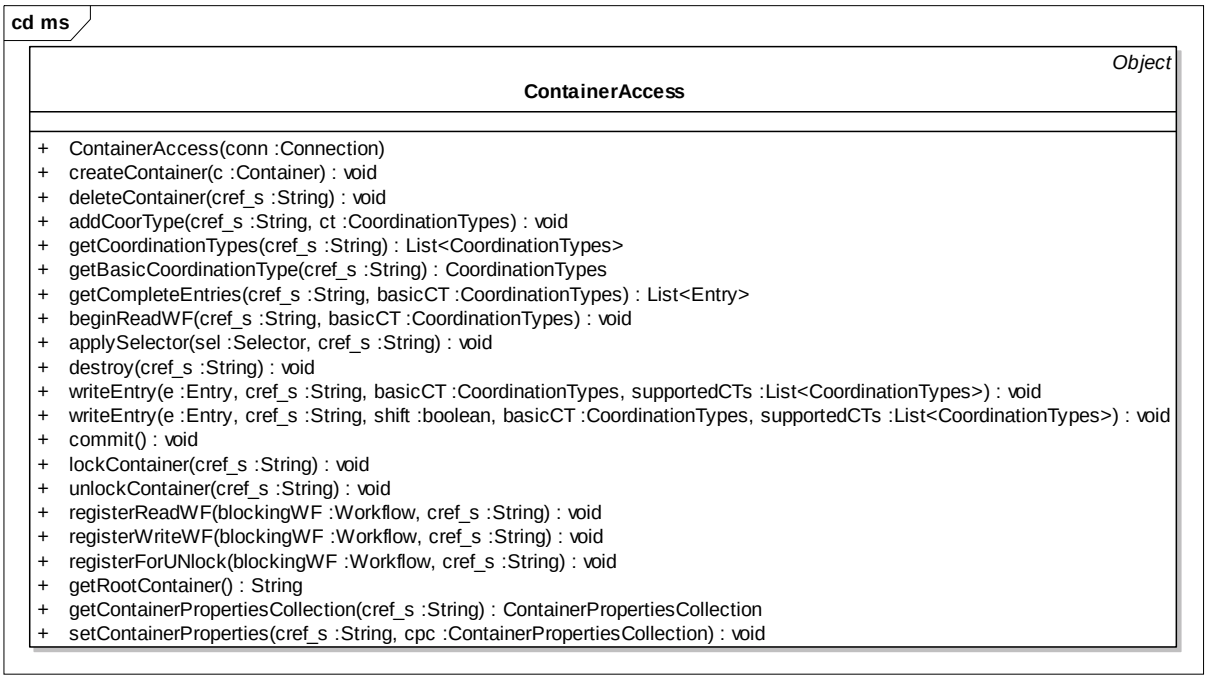

Figure 33: ContainerAccess class diagram

VIII.1.1. Container Management

Container Management operations can be seen as the ContainerAccess counterpart to the container operations that are described in chapter VI.3.2. This set of methods is used to create, destroy and manage container properties. These operations are used by both, the Capi implementation and the workflow classes.

- createContainer: Creates a new container in the database with the specified container reference.
- deleteContainer: removes the specified container from the database.
- addCoorType: This method is used to add a new coordination type to the specified container. The parameters are the container reference and the new coordination type.
- getCoordinationTypes: retrieves a list of all coordination types that are enabled at the specified container.
- getBasicCoordinationTypes: retrieves the base coordination order of the specified container.
- lockContainer: locks the specified container. This prevents other threads from making modifications at this container.

- unlockContainer: removes the lock from a container.
- getContainerPropertiesCollection: retrieves a list of all properties of the specified container.
- setContainerProperties: sets the specified container properties at the container depicted by the container reference.

VIII.1.2. Entry and Selector Methods

The entry and selector methods contain all necessary algorithms to read and write entries from a container. These are only used by workflow classes.

- getCompleteEntries: retrieves all entries that have been selected by the previous application of selectors.
- beginReadWF: prepares the temporary table for a read request.
- applySelector: This method applies the specified selector
- destroy: deletes the entries that have previously been selected by a read request.
- writeEntry: adds a new entry to the specified container. Depending on the shift flag either a write or a shift operation is performed.

VIII.1.3. Miscellaneous Methods

The Miscellaneous methods contain bookkeeping and management operations. These are not directly related to containers or entries but rather the functionality of those. Again these methods are used by the Capi implementation and the workflow classes.

- ContainerAccess: The class' constructor takes a connection as parameter. In the case of MozartSpaces this parameter is the connection to the Derby database.
- commit: commits all changes to the database.
- registerReadWF: registers a read workflow for notification. This essentially adds the workflow instance in a list which is then added into a container which maps the container reference to this list.
- registerWriteWF: registers a write workflow for notification.

- registerForUnlock: registers a workflow for notification when a container becomes unlocked.
- getRootContainer: retrieves the MozartSpaces root container. The root container is used to store the name container reference mapping of the named containers.

VIII.1.4. Exceptions

Most of the CADAO exceptions have already been mentioned in the context of the Capi and workflows. While the list of exceptions that can be thrown by the CADAO is rather long, all are related to container and/or entry operations and specify wrong usage or unexpected states. Only one exception cannot be categorized like the others and is used to flag internal and severe errors. All exceptions are rather simple and basically only exist to give certain conditions useful names rather than complex functionality.
The following list contains all exceptions that the CADAO might throw.

- ContainerAlreadyLockedException: this exception is thrown when a workflow tries to lock a container, which already has been locked by another workflow.
- ContainerAlreadyUnlockedException: this exception is thrown when a workflow tries to unlock a container, which already has been locked by this or another workflow.
- ContainerFullException: this exception is thrown when a workflow tries to write more entries than there are free places in the container.
- ContainerNotEmptyException: this exception is thrown when an attempt is made to change a container's coordination types but the container still holds at least one entry.
- CoordinationTypeNotSupportedException: this exception is thrown when an attempt is made to apply a selector to a container whose coordination types do not support that selector type.
- CountNotMetException: this exception is thrown when there are not enough entries in the container that match the specified selectors.
- DoubleCrefException: this exception is thrown when an attempt is made to create a new container with an already existing container reference.

- UnknownContainerException: this exception is thrown when an attempt is made to access a container with a non existing container reference.
- WriteSelectorNeededException: this exception is thrown when a write request is made, the target container has explicit coordination types but the entries do not container selector values that satisfy these coordination types.
- FatalException: this exception is thrown when an unexpected or otherwise internal erroneous condition occurs.

VIII.2. Selector application

It has already been mentioned that selectors are applied in a sequential manner. The index of the selector list therefore also specifies the order which the selectors are evaluated in. Selector rejection and merging has already been discussed in chapter VII.5. No optimization takes place in the CADAO. Each selector is applied one after another similar to successive filtering of the previous result set..

One of the most important steps that the CADAO has to perform prior to applying the selectors to a container, is a process called 'selector translation'. Since the data is stored in a database it would be highly inefficient to read all entries that are stored in the target container into entry instances in main memory. There are virtually no limits on the number of entries in a container therefore this could take a long time and eat tremendous amounts of memory. Therefore each selector is expressed as SQL query and directly performed in the database. The CADAO simply chooses the appropriate SQL query, fills it with the selector's current values and executes it in the database. All SQL queries of course are executed as prepared statements for reasons of both, speed and security.

Compiling a new statement each time a selector is executed would cost huge amounts of CPU cycles in the CADAO code as well as in the database itself. Additionally SQL injection and other security issues are inherently more difficult to exploit when using prepared statements.

In the current implementation there exist a number of database tables (details will be discussed in chapter VIII.3) for each container. The most important one for the process of selector application is the WorkingSet table. At the beginning

of a read operation the target container's WorkingSet table is cleared and all entry identifiers of this container are added to the WorkingSet table. A selector's SQL query then removes all entry identifiers from the WorkingSet table that do not match this particular selector. Then the next selector's SQL query is executed. This process description makes it immediately obvious that selectors form a logical AND relationship as it has been mentioned before. After all selectors have been executed the WorkingSet table contains only entry identifiers of those entries that match all selectors. Depending on the intended workflow the result of this process is handled differently.

- ReadWorkflow: The entries that are identified through the entry ids in the WorkingSet table are read from the Entry table and returned to the workflow.
- TakeWorkflow: The entries are returned to the workflow in the same way that has been described for the ReadWorkflow. Immediately thereafter all entries that are identified through the entry ids in the WorkingSet table are deleted from the Entry table.
- DestroyWorkflow: All entries that are identified through the entry ids in the WorkingSet table are deleted from the Entry table.

One interesting detail should be mentioned regarding selector counts. If more entries match the selector than the specified count value of the selector, the desired number of entries is chosen using the base coordination order of the target container.

Writing or shifting entries into a container involves selector processing only when the target container has an explicit coordination type enabled. In that case the selector's meta-information simply is added to the appropriate database table.

VIII.3. Database

A database was chosen as the data handling backend for the MozartSpaces implementation since it provides a vast number of features that are needed by the XVSM system. Developing these features would have required many resources both in manpower and time for implementation and extensive testing afterwards. Databases provide transaction handling, table locks and most

importantly a simple and thoroughly tested querying mechanism, an SQL processor.
A vast number of databases are available but some constraints had to be taken into account before making a concrete decision. MozartSpaces was intended to become an Open Source Software (OSS) project but should not include any viral licence such as the GPL. Java was the preferred programming language for the MozartSpaces therefore an easy integration/usage with Java was also needed.
In the end the decision fell on Derby for reasons of simple handling and integration, a reasonably small footprint and the possibility to run the whole database within a process' memory space. Additionally it was already known that Derby would be included in Java 6 and therefore no additional database library would be needed once Java 6 was available[36].

In the MozartSpaces implementation the database is loaded into the process space and is kept totally transparent to the XCore user (and even to all the XCore entities other than the CADAO).

VIII.3.1. EER Diagram

In its current state the database model for the XCore looks as shown in Figure 34

[36] Java 6 has already been available for some months as of the writing of this thesis.

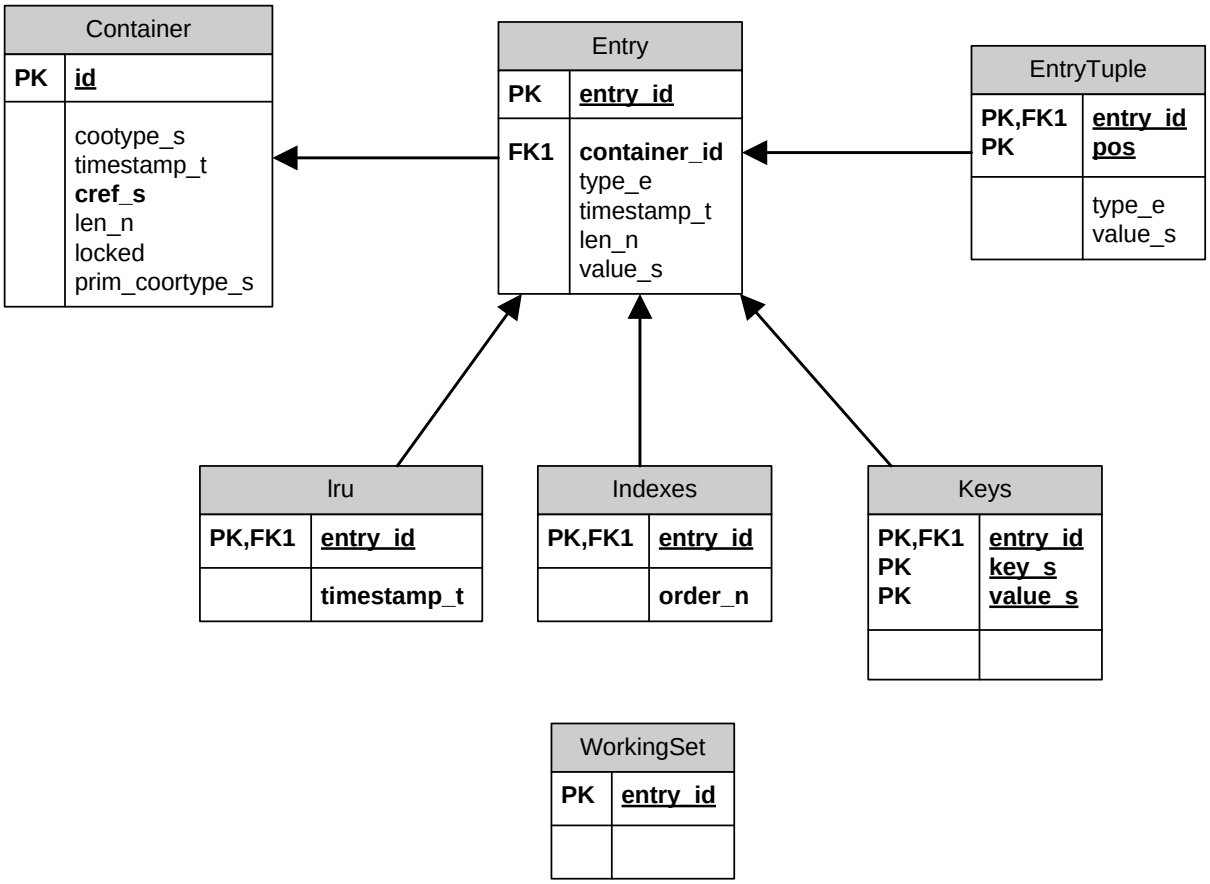

Figure 34: XVSM Core database model

One important detail about the current implementation of the MozartSpaces and probably one of the biggest performance bottlenecks is the fact that one set of these tables[37] is created and managed for every container in the space. Additionally the Derby database itself is rather slow.

The tables are as follows and will be explained in detail in the next subchapters.

- Container:

  The container table is probably the most important table of the database. It stores information about the containers that are currently available in the XCore and can therefore be seen as a table or container catalogue. This table is the only one that exists only once per XVSM core. The table fields are:

  - id: declares an internal primary key.

[37] An exception is the container table, which can be seen as table/container catalogue.

- prim_coortype_s: stores the base coordination order.
- coortype_s: stores the coordination types of a container. This is a bit field of the coordination types that are enabled at the container.
- timestamp_t: specifies the container's creation timestamp.
- cref_s: stores the unique Container-Reference of a container.
- len_n: is set to the number of elements a container may contain if it is bounded.
- locked: is a field used for transaction-management. Specifies whether the container is locked by a workflow.

- Entry:

  Entries represent the entities that may be stored in a container. Even though the application might store entries of different types, they are converted by the CADAO and stored as string representations in the database. Every container has its own Entry table; therefore the name of the concrete instance of the entry table in the database is the following: '<Cref>_Entry'. The Entry-table fields are:

  - entry_id: declares the primary key for an entry. This id is also used for the WorkingSet table entries.
  - container_id: is the foreign key to the container's id wherein an entry is kept.
  - type_e: specifies the type of the entry's value.
  - timestamp_t: is entered by a trigger on creation.
  - len_n: is the length of the entry.
  - value_s: stores the string representation of the entry's value.

- EntryTuple:

  Even though a tuple is basically nothing more than another type of entry (see ValueType in chapter VI.2.1), more information is needed internally for book-keeping and template matching. For this reason tuples have their own table, again one for every container with a similar naming pattern: '<Cref>_EntryTuple'.

  This internal distinction between entries and tuples is totally transparent to the application. To the client a tuple in fact is nothing more than an entry

and reading/writing of a tuple works with the entry handling methods. The EntryTuple-table fields are:

- entry_id: specifies the entry id if this tuple. Since a tuple contains one or more entries as its fields more than one tuple table entries can have the same entry_id.
- pos: specifies the field position within the tuple.
- type_e: specifies the type of the tuple entry's value.
- value_s: stores the string representation of the tuple entry's value.

An important detail that must be considered when using tuples is the fact that tuples will be flattened. This means that while any degree of nested tuples can be stored into the database only the topmost level will be reconstructed when reading the tuple. This behaviour results from the fact that information about the correct type is lost due to the conversion of tuple field's values to strings.

- Keys:

  The Keys table contains the meta-information needed for the KEY coordination type. Once again the naming scheme of this table is '<Cref>_Keys' and exists once per container.
  The Key-table fields are:
  - entry_id: specifies the entry that this key is bound to.
  - key_s: specifies the type of the key value.
  - value_s: specifies the string representation of the key's value.

- Indexes:

  The Indexes table contains the meta-information needed for the VECTOR coordination type. The naming scheme of this table is '<Cref>_Indexes' and it exists once per container.
  The Indexes-table fields are:
  - entry_id: specifies the entry that this index is bound.
  - order_n: specifies the index that this entry has in the container

- Lru[38]:

  The Lru table contains the meta-information needed for the LRU coordination type. The naming scheme of this table is '<Cref>_Lru' and it exists once per container.
  The Lru-table fields are:

  - entry_id: specifies the entry that this timestamp is bound to.
  - timestamp_t: specifies the timestamp of the last access to the referenced entry.

- WorkingSet:

  The WorkingSet table is a temporary table used whenever selectors are applied to a container. Details about the selector application can be found in chapter VIII.2. The entry ids of the requested container are initially stored in the table when a new list of selectors must be applied. Then every selector (possibly) filters out entry ids from this list until the final table setup contains the entry-ids of those entries that match the list of selectors. These can then simply be read from the Entry table.
  The WorkingSet-table fields are:

  - entry_id: specifies the entry_id of the entry that matches the selectors that have been applied thus far.

VIII.3.2. Database Processing

This chapter is intended to give some examples of entry operations and how they change the database tables. Additionally the method invocations of the CADAO are shown to clarify the context and timeframe of the database changes. The following examples assume a container that has the FIFO base coordination order and is bounded to five entries.

[38] Note: the LRU coordination type has been removed in a more recent version of the XVSM specification. This is due to the reason that the LRU coordination type changes an entry upon reading which clearly should not alter the state of an entry.

- Read

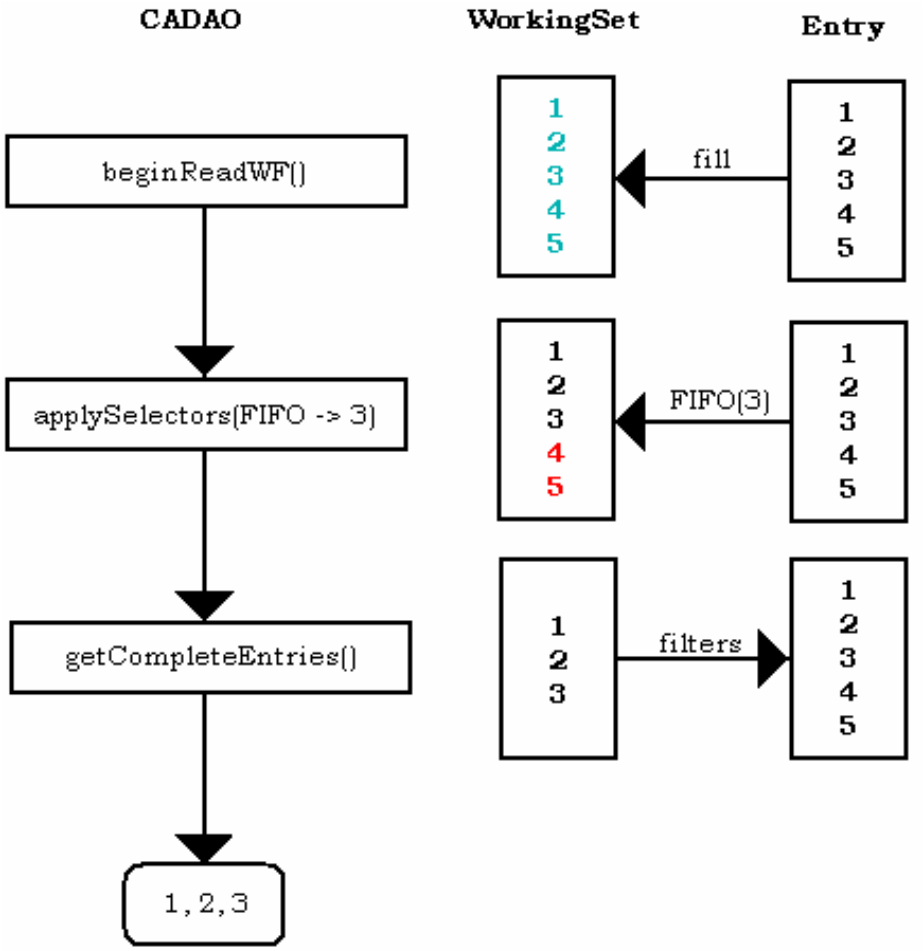

Figure 35: Database read sequence

When the beginReadWF method is invoked all entry identifiers that are available in the container are copied from the Entry table to the WorkingSet table. (Note: the WorkingSet table is cleared beforehand if it's not empty).

Then the selectors of the read request are applied. In this example there is a single FifoSelector that reads three entries from the space. All non matching entry identifiers in the WorkingSet table are removed; in this case these are the entry ids 4 and 5. The getCompleteEntries operation then reads the entries from the Entry table whose entry ids are in the WorkingSet table; in this case these are the entries 1, 2 and 3. These entries are then returned to the workflow.

- Take/Destroy

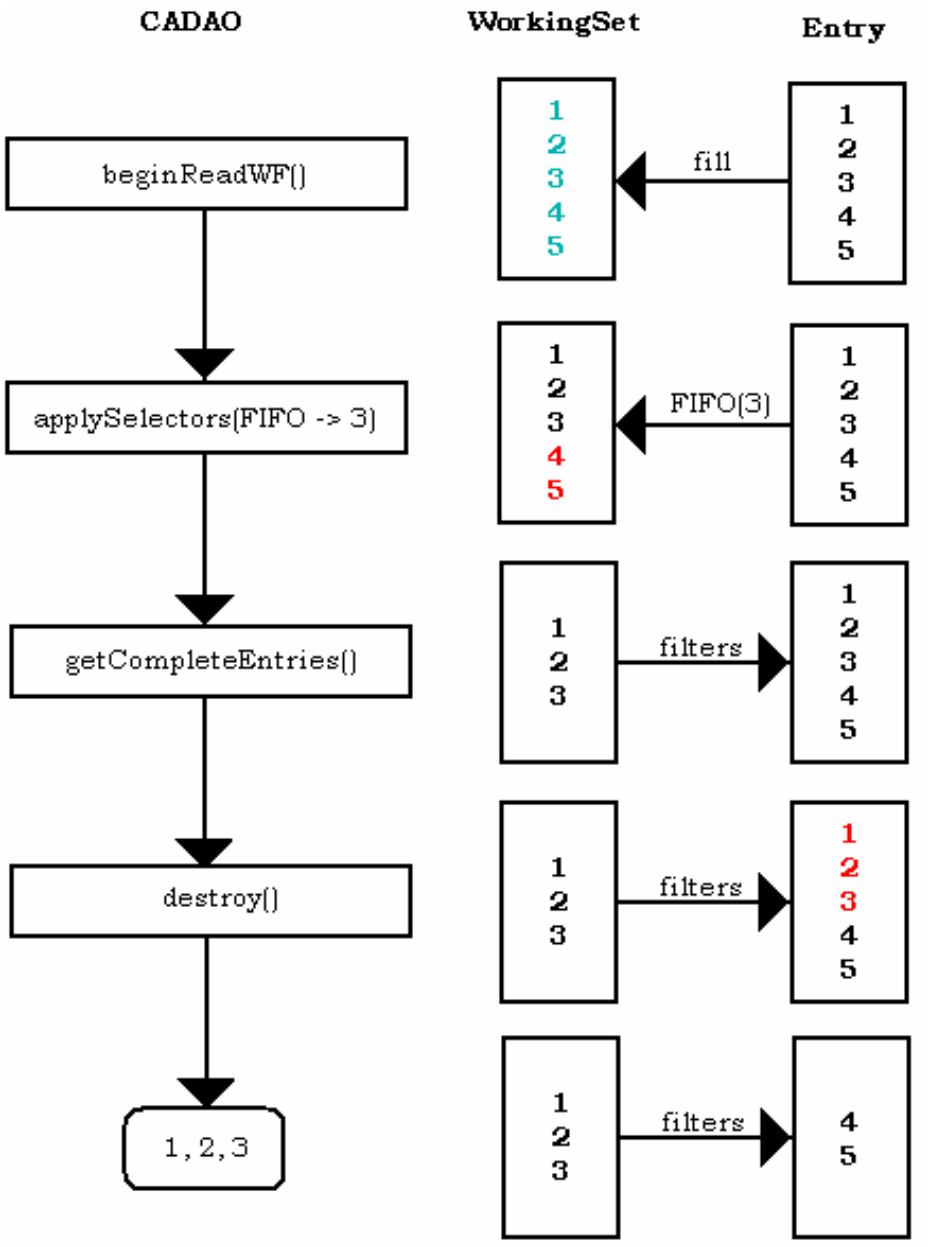

Figure 36: Database take/destroy sequence

There is one combined example for take and destroy since these two operations are equal concerning the database processing. The distinction between take and destroy and therefore whether the entries should be returned to the client or not is made in the workflow implementation.

Again the beginReadWF method invocation initiates the filling of the WorkingSet table with the available entry ids. Then the selectors, again a FifoSelector for three entries is used in this examples, are applied. The remaining entries are retrieved through the invocation of the getCompleteEntries method. Thereafter the workflow implementation calls the destroy method which checks the

WorkingSet table and removes those entries from the Entry table whose entry ids match.

- Write

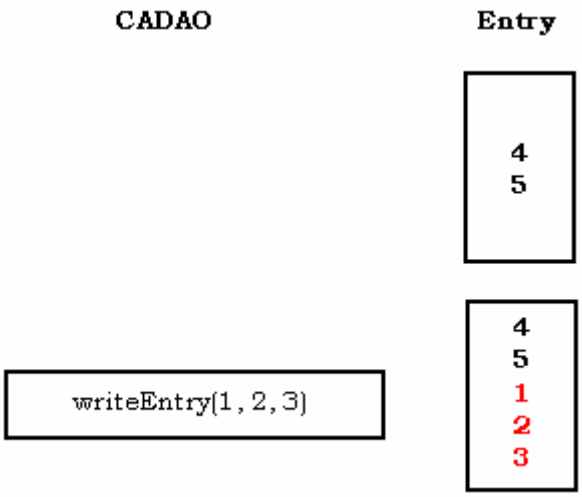

Figure 37: Database write sequence

Writing entries into the space is a fairly simple operation on the database side. All blocking is handled by the workflow and CADAO implementations therefore this example concerns only the non-blocking write case. If there is enough room, the entries are simply written into the Entry table; in this case the entries 1, 2 and 3.

- Shift

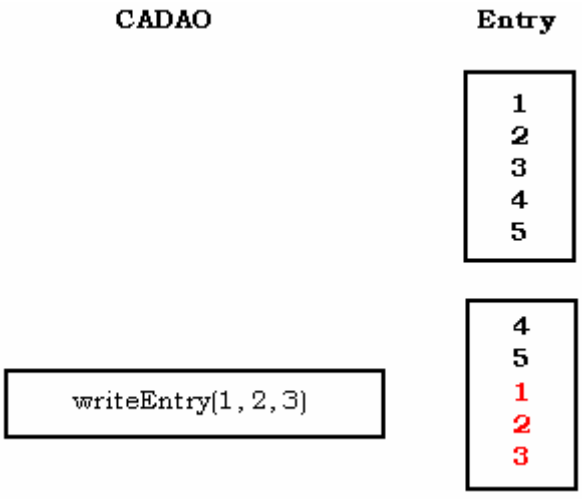

Figure 38: Database shift sequence

Assume that the bounded container contains all five possible entries. When the writeEntry method is invoked and requested to perform a shift action the number

of free places in the container is checked against the number of entries to be written. In this case there is no room in the container and three entries must be written. Therefore three entries are shifted out of the container according to the base coordination order, FIFO. Therefore the entries 1, 2 and 3 are removed from the Entry table. Entries 4 and 5 are moved upfront in their position according to FIFO and the new entries 1, 2 and 3 are written into the container.

IX. Collaborations within XVSM

This chapter presents sequence diagrams of the most important processes within XVSM. The purpose of those is to show the interaction of the three previously described parts.

IX.1. Core Management Operations

IX.1.1. Init

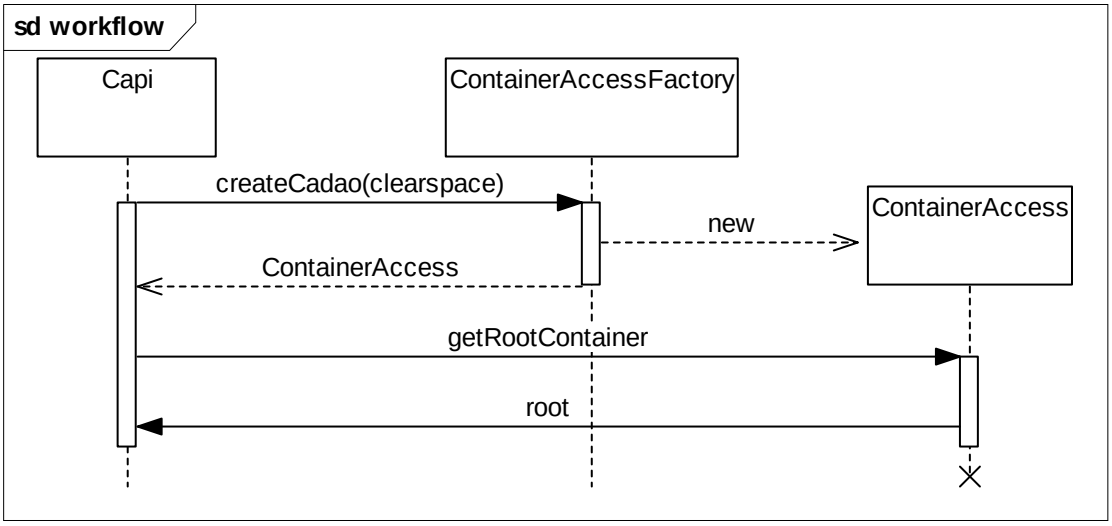

Figure 39: Capi.Init sequence diagram

The Capi requests a new ContainerAccess object from the ContainerAccessFactory. The clearSpace parameter specifies whether the backend database should be cleared (that is, all possibly available contents must be deleted) or it the previous state must be restored. After the successful creation of the ContainerAccess object its root container is retrieved.

IX.1.2. Shutdown

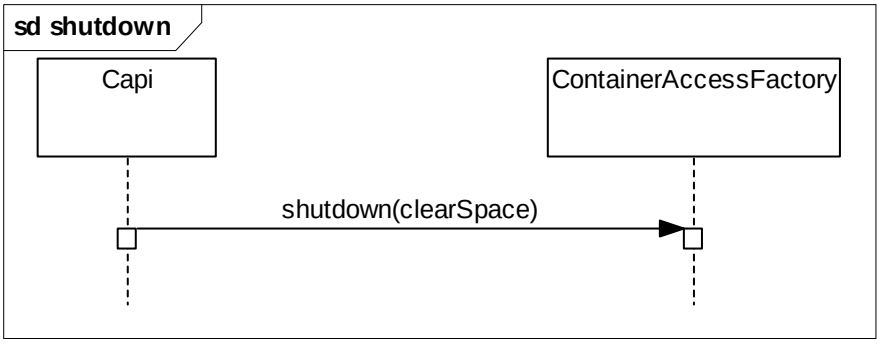

Figure 40: Capi.shutdown sequence diagram

The XCore shutdown simply is carried out by invoking the shutdown method of the ContainerAccessFactory. This will handle any needed cleanup.
Again the clearSpace parameter specifies whether the available contents must be destroyed or kept.

IX.2. Container Operations

IX.2.1. Create Container

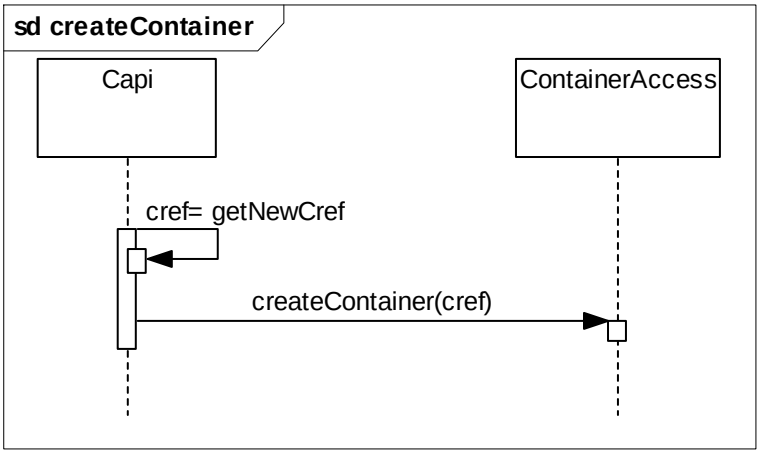

Figure 41: Capi.createContainer sequence diagram

After creating a new container reference the createContainer method of the ContainerAccess is invoked. While the cref parameter is not entirely true (some intermediate objects are created and passed to the CADAO object), the meaning and order of the operation is correct.

IX.2.2. Destroy Container

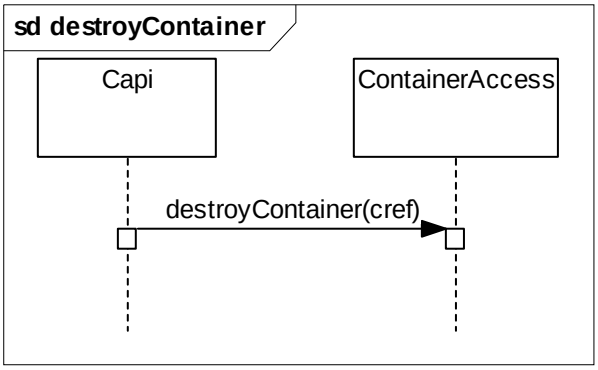

Figure 42: Capi.destroyContainer sequence diagram

IX.2.3. Read/Take/Destroy/Write/Shift

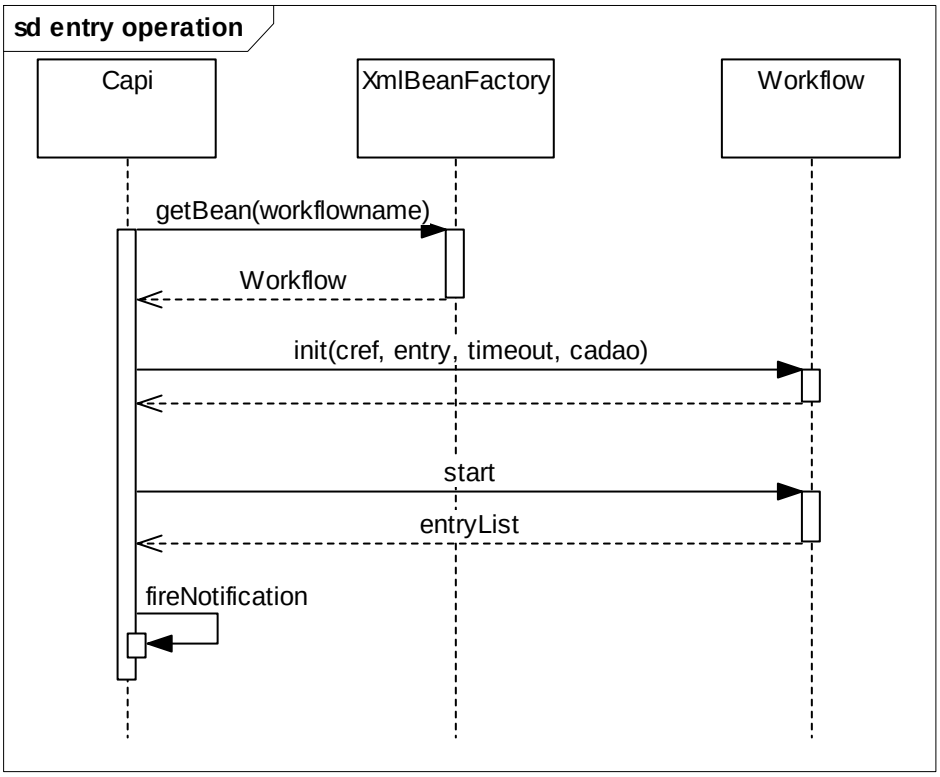

Figure 43: entry operation sequence diagram

The entry operations are exactly the same for all 5 methods. The only difference is the workflowname which depends on the operation that must be invoked. Also whether the entryList is returned to the client depends on the operation (e.g.: the destroy method does not have a return value).

X. XVSM Protocol

The embedded version of the XVSM is good as a starting point for simple space based applications. Multithreaded applications and multiple processes that run in the same java virtual machine can be coordinated. But the real strength of XVSM lies in coordination of peers distributed in a networking environment. A short introduction to the XVSM standalone version has already been given in chapter V.2 and a more thorough discussion can be found in chapter XII.
One main focus in the process of developing the standalone version of XVSM was that is should be independent of any technology as far as possible. Therefore interfaces like remote method invocation or even CORBA where clearly not a desirable interface target. The XML protocol has been developed to specify the communication between a XVSM client and the standalone version.
All functionality of the XVSM core can be expressed in the XML protocol. Since it's mainly a payload protocol and does not depend on a specific technology (apart from XML processors) or transport protocols, the XVSM server can be implemented using the best fitting technology.

Thorough information about the XML protocol is given in chapter XIII.

XI. XVMS Client Site

Together with the XVSM Server, the XVSM Client provides access to Capi operations over the network. The transport protocol for the client-server communication is HTTP. The client sends HTTP POST requests to the server. The payload of those requests is valid XVSM Protocol Capi-requests (please see chapter XIII for details) and interprets the XVSM Server's Ipac response.

XI.1. ClientCapi: Implementation of the Capi interface

The XVSM client implements the Capi interface. Therefore it is totally transparent to the application whether the MozartSpaces are loaded as library or accessed through a network using the XVSM client and server. Every Capi call is being encoded to a XVSM XML request which is sent to the server. The XVSM XML response is received and decoded to objects of types that a Capi call of that function would return.

XI.2. Transforming Capi calls to XVSM XML

As mentioned earlier, Capi function calls must be encoded in appropriate XVSM XML and XVSM XML answers from the server must be decoded to objects specified in the Capi interface. This is done through so called Creators and Readers. For every possible request there exists a Creator and a Reader which are used to process XML requests and responses respectively.
Creators are used to map objects from the Capi to valid XVSM XML code. They construct a request step by step. The PlainCapiDocumentCreator creates a plain Capi document which is then extended by other creators. Every Creator can transform a special request to XVSM XML.
Readers are used to parse XML responses from the server step by step by processing a small part of the response returned. Every Reader can process a part of the answer (corresponding to the request).
Creators and Readers make it possible to react to further enhancements of the Capi because there can easily be written another Creator or another Reader which deals with the new functionality. Creators and Readers also can be used

to provide support for compound requests (requests that are composed of multiple requests).

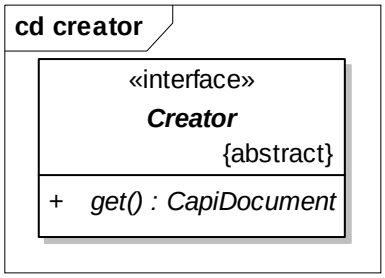

Figure 44: Creator class diagram

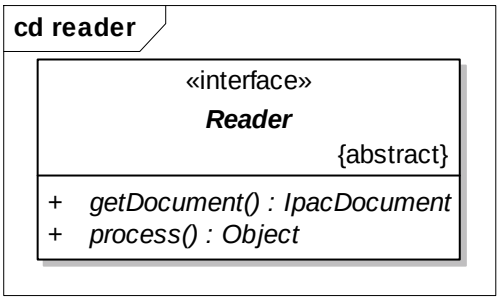

Figure 45: Reader class diagram

XI.3. Connections to the server

The communication between the client and server is performed through connections. These connections are created and managed by the ConnectionManager. The class diagram of the ConnectionManager is shown in Figure 46. The ConnectionManager provides different connections each of which implements the Connection interface shown in Figure 47.

The usefulness of this architecture becomes immediately clear, because it decouples the process of getting a connection to a server from the transformation of Capi calls to XVSM XML. In fact it would be possible to implement another ConnectionManager which provides Connection objects that could communicate to servers with any other transport protocol than HTTP. The client provides a DefaultConnectionManager which creates connections to a server (given by its URL) on the fly. The client also provides TomcatConnection, a class, which implements the Connection interface and provides Connection to a server through HTTP POST requests[39]. The DefaultConnectionManager creates

[39] In this implementation HTTP 1.0 is used. Unfortunately this version does not permit the connection to be kept alive meaning there is exactly one connection for a single request followed by a single response.

only connections of type TomcatConnection. The ClientCapi's default constructor assigns the DefaultConnectionManager to the ClientCapi, so that the ClientCapi obtains connections to the server from it.

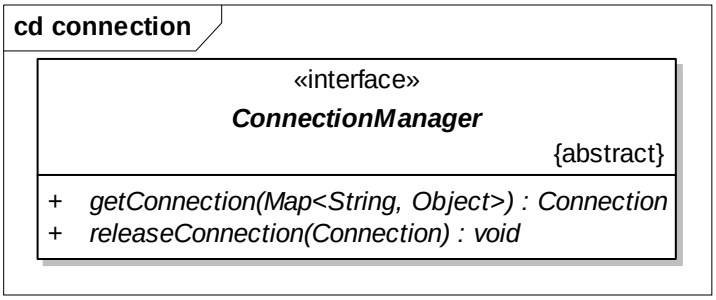

Figure 46: ConnectionManager class diagram

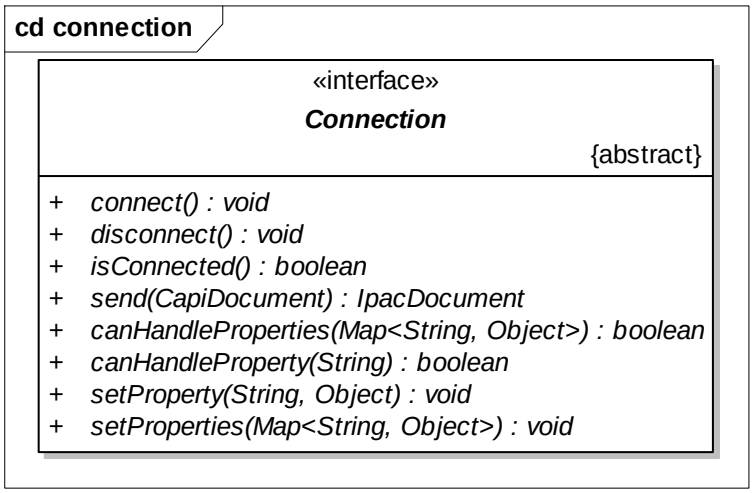

Figure 47: Connection class diagram

XI.4. Notifications

Since notifications cannot actively be sent from the server to the client (see also chapter XII.1 for more details) a different approach has been implemented. Instead of an active push model where the server notifies the client actively, the server stores notifications and the client is responsible for polling these notifications.

The ClientCapi can be given an object implementing the NotificationManager (see Figure 48 for its class diagram) interface. At the NotificationManager is used to register notifications. For every notification there will be an object of a class derived from AbstractNotificationReceiver (see Figure 49 for its class diagram). This notification receiver is a thread which is responsible for polling

the notifications. The NotificationManager manages the scheduling of the registered notification receivers. The DefaultNotificationManager which is assigned to the ClientCapi by its default constructor uses Java's ScheduledExecutorService to schedule notification polling.

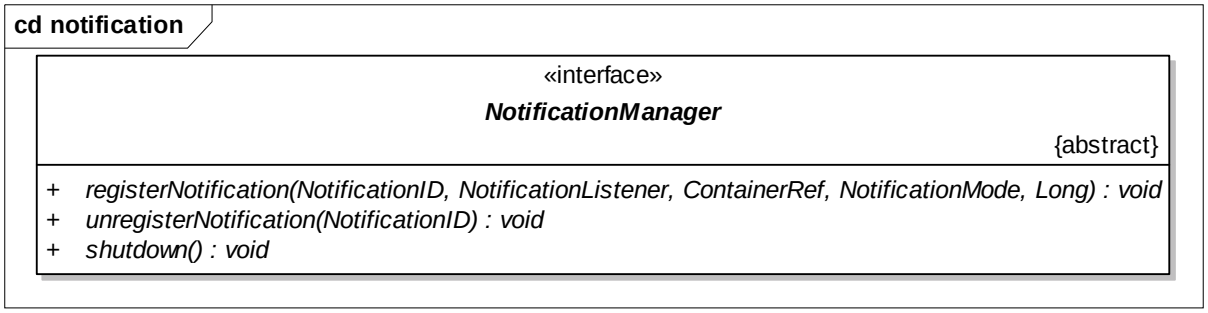

Figure 48: NotificationManager class diagram

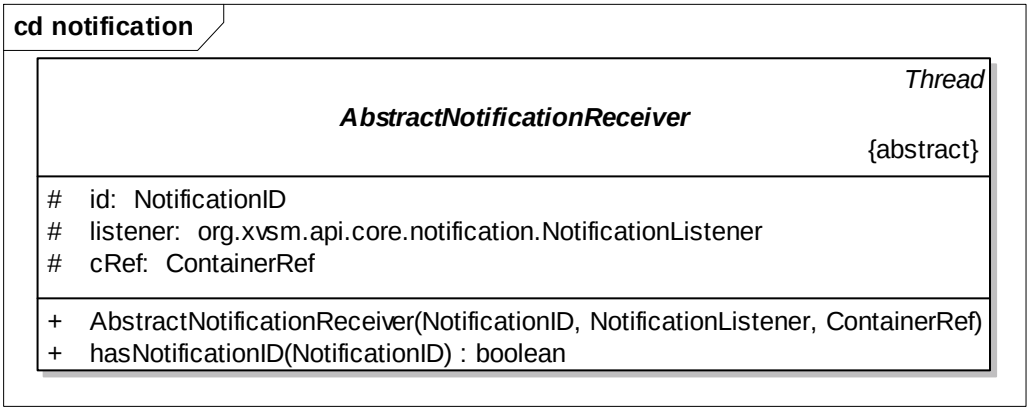

Figure 49: AbstractNotificationReceiver class diagram

XI.5. Exceptions thrown by the client

All Exceptions thrown by the client are subclasses of XVSMClientException (which is itself a subclass of FatalException defined by the XVSM Core. This has the advantage that again the standalone XVSM can be used transparently in an application even though there are more possible error sources. The client throws exceptions, if a timeout has occurred (XVSMTimeoutException) or if a connection error takes place (XVSMConnectionException).

XII. XVSM Server Site

The main and most important part of the XVSM standalone version is the server. Correctly speaking this is not a server itself but merely a web application[40]. The servlet accepts HTTP GET and HTTP POST requests. As it has already been explained in chapter X the payload of the request must be a valid XML string formulated according to the XML protocol.

Figure 50 shows the class diagram of the servlet class. Additionally there are supportive classes needed for the request processing. These class diagrams can be seen in Figure 51 through Figure 52.

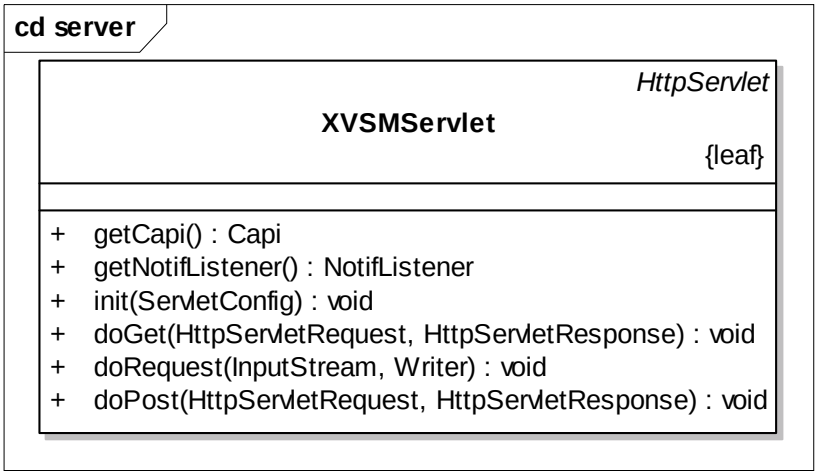

Figure 50: XVSMServlet class diagram

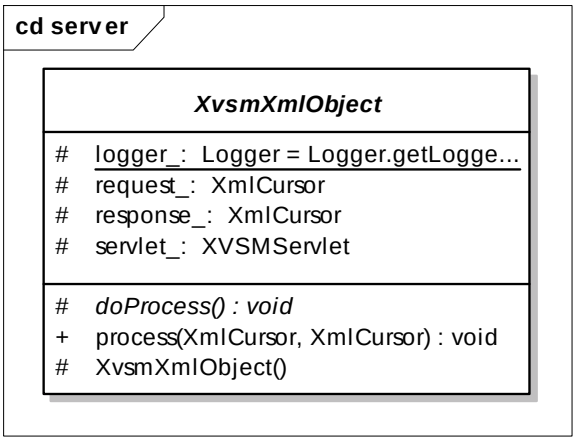

Figure 51: XVSMXmlObject class diagram

[40] The XVSM server is implemented as Java Servlet. This type of web application needs a so called servlet container (e.g.: Apache Tomcat) which is used to load and execute the servlet.

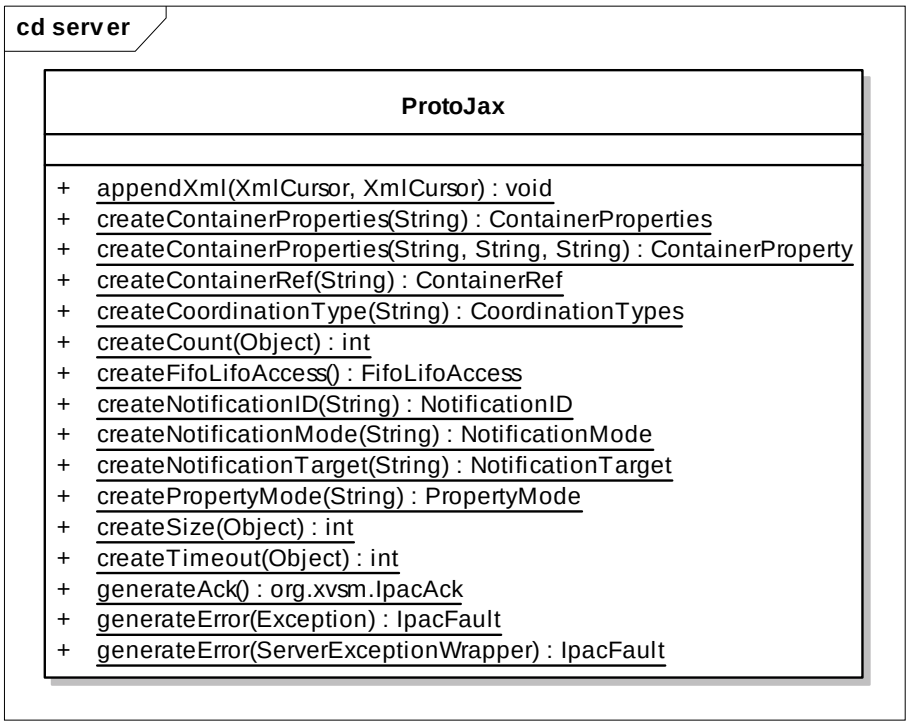

Figure 52: ProtoJax class diagram

The XVSMServlet class is the implementation class of the servlet. A servlet is a special type of java program which must be executed in a so called servlet container. This servlet container handles networking issues like connection management, request distribution and similar tasks. The XVSMServlet uses the standard servlet processing methods and request handling. The doPost method (doGet for HTTP GET request respectively) first checks the well formed- and validity of the request. XMLBeans[41] are used to accomplish this task. After the checking, the request is passed on to the correct XVSMXmlObject. This class is a base class and for each possible request exists a concrete subclass that implements the mapping from the XML request to the appropriate XVSM Core method invocation. The ProtoJax class is a simple helper class that implements the most common transformations from the XMLBeans Java types to XCore parameter types and vice versa.

[41] Apache XMLBeans provides XML to Java types data binding. This binding with the according java classes can be automatically generated from the XML Schema files.

XII.1. Notifications

Notifications in a non-RPC[42] need special treatment. Since it was a design decision to have a platform and technology independent XVSM standalone version it was also not possible to implement any of RPC or similar techniques for notifications. Notifications can't be sent through a Call-back-Object as it has been implemented for the embedded version of XVSM.

In standalone XVSM both client and server need to collaborate to provide the functionality of notifications to the user. Since the server can't open a connection to a client without implementing a full blown client tracking and a notification to client binding, the server can't actively send out notifications but the client must poll for them. Such a method has been chosen since keeping notification connections alive for all clients would be highly inefficient and kill server resources and system handles very fast.

The server creates a special FIFO container for each newly created notification. The container reference for it will be returned to the client along with the notification id[43]. The NotifListener class (see Figure 53) takes care of the container handling. It also implements the NotificationListener interface (see Figure 21) and registers itself in the internal embedded XVSM core as listener. Each time a notification fires, this information with possibly attached entries will be stored in the notification container. This container can and will periodically be checked by the client side of the XVSM standalone version. The cancellation of the notification removes the container from internal bookkeeping but does not destroy the container. This is needed for the polling and not synchronized nature of the notification checking.

[42] Remote Procedure Call; Also named RMI (Remote Method Invocation in Java). This is a means of invoking methods on a remote site through normal programmatic method invocation as it is (more or less) specified in the programming language used. Network communication and other management tasks are hidden from the user.

[43] The client will strip the container reference and keep it for the notification handling. More details on the client part of notifications can be found in section X.

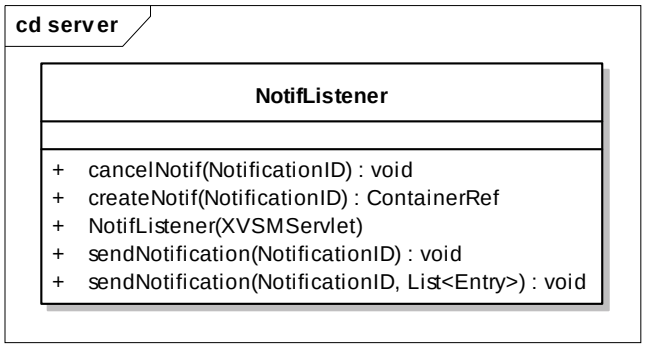

Figure 53: NotifListener class diagram

XII.2. Timeout

In the current version of the XVSM standalone implementation timeouts are solely handled by the client and timed connections. Further details on timeouts can be found in chapter X.

XIII. The XML Protocol

The XVSM protocol has already briefly been mentioned in chapter X. It is the payload protocol between the XVSM server and client implementation which in the case of the MozartSpaces is transported using HTTP.
Extensive element and attribute explanations of the XML protocol can be found in [Kühn, Ecker 2006]. This document forms the specification of the protocol. Still, in order to give the readers a clue the XML protocol schema files are appended to this thesis and can be found in Appendix A.
Care has been taken in order to keep the protocol simple and human readable while still expressive enough to encode all operations and responses of the XVSM core.
The protocol schema is divided into three different files.

- The XMLSchema_capi_request.xsd, which contains all elements that are needed to encode the operation requests to the code. XML data according to this schema is written by the XVSM client and parsed by the XVSM server.
- The XMLSchema_ipac_response.xsd, which contains all elements that are needed to encode the operation responses as well as possible exceptions. XML data according to this schema is written by the XVSM server and parsed by the XVSM client.
- The type_definitions.xsd, contains all common XML Schema type definitions that are needed in both the capi and the ipac protocol parts.

During the development and specification of the XML protocol several tricky aspects have been discovered. These usually are related to the mapping of special values for types[44] within programming languages to appropriate encoding in the XML protocol. Naiive mappings can result in unexpected errors or, in more severe cases, in silent production of wrong results. Such unexpected errors usually are related to special values, such as 'null', that have been mentioned before. A more subtle bug would result in simply using a string type with additional encoding information. Problems in encoding and decoding could take place especially in multi-programming language environments since Java

[44] A prominent example of such a special value is the 'null' value for references in the Java programming language.

does not use the unicode encoding for strings as it is defined in the ISO standard.
Another interesting problem is related to exception propagation. In single programming language environments this would be fairly easy to solve. Also, in the case of binary and language bound facilities such as RPC or RMI these cases have a well defined solution. For XVSM we again face the problem of multiple involved programming languages on the one hand and binary compliance of the embedded Capi and the client's Capi implementation. These two aspects obviously require the encoding of exceptions in such a way, that they can correctly be reconstructed. The XML protocol therefore specifies exceptions in a way that the full name of the exception class is specified in the payload. Additionally the contents of the exception[45] is encoded. The XVSM client side can then use this information to construct an exception of the correct type and the client application code is not broken. In the Java programming language the process of constructing an instance of a named type is fairly easy due to the dynamic nature of Java and it's class loading abilities during runtime.
An extensive list of samples for the correct usage of the XML protocol can be found in [Kühn, Ecker 2006].

The following chapter will now give an extensive example of the XVSM method invocations and return values. Additionally the according XML Protocol strings will be given.

[45] The exception content in XVSM is a string message for all possible exceptions.

XIV. Capi and Sample Code

This chapter will show the Capi invocations and XML protocol data of a small example. Complete details about how to correctly use the MozartSpaces can be found in the JavaDoc and the accompanying source code documentation. The example assumes that two programs are coordinated through the standalone version of the MozartSpaces. The first program will be named 'A' while the second one has the name 'B'. Boiling down the example to the basic functionality it can be described as remote dictionary lookup where client 'A' asks client 'B' for the German translation of English words.

Both clients know a named container called 'Dictionary' which uses the key coordination type. The key of the container is the English word while the value is the German translation. A second container named 'Word-queue' has the FIFO coordination type enabled. Client A writes each word it needs to be translated into this container. Client B takes those words and stores the translation in the dictionary. Both clients use notifications, client A on the dictionary container and client B on the word-queue container. For simplicity reasons client A creates both containers and we assume that both named containers exist when client B is started.

The following steps will be shown in the example:

1. A: Create a named container called 'Dictionary' with default values for base coordination order and size.
2. A: add the key coordination type
3. A: Create a named container called 'Word-queue' with FIFO coordination order and default size.
4. A: Register a WRITE notification for the 'Dictionary' container with automatic return of the written entries.
5. B: Retrieve named containers
6. B: Register a WRITE notification for the 'Word-queue' container.
7. A: Write a word into the 'Word-queue' container.
8. B: After being notified, take the available words from the 'Word-queue' container
9. B: Shift the translation of all words into the 'Dictionary' container. Shifting avoids the need for error handling when words are requested multiple times.

10.A: After being notified, read the translation from the 'Dictionary' container.

A)

```
//1) create the 'Dictionary' container
ContainerRef dictA = Capi.createNamedContainer("Dictionary");

//2) add the KEY coordination type
List<ContainerProperty> request = new LinkedList<ContainerProperty>();
request.add(new CoordinationTypeProperty(ContainerProperties.COORDINATION_TYPES,
    CoordinationTypes.KEY, PropertyMode.SET));
Capi. setContainerProperties(cref, request);

//3) create the 'Word-queue' container
ContainerRef queueA = Capi.createNamedContainer("Word-queue", CoordinationTypes.FIFO);

//4) register a WRITE notification for the 'Dictionary' container
NotificationID nid = Capi.createNotification(dictA, -1, NotificationTarget.WRITE,
        NotificationMode.INFINITE, true, clientANotifListener);
```

B)

```
//5) retrieve 'Dictionary' container
ContainerRef dictB = Capi.getNamedContainer("Dictionary");
ContainerRef queueB = Capi.getNamedContainer("Word-queue");

//6) register a WRITE notification for the 'Word-queue' container
NotificationID nid = Capi.createNotification(queueB, -1, NotificationTarget.WRITE,
        NotificationMode.INFINITE, true, clientBNotifListener);
```

A)

```
//7) write words into the queue
List<Entry> entries = new LinkedList<Entry>();
entries.add(Entry.Factory.newInstance(ValueTypes.STRING_UTF8, "hello"));
entries.add(Entry.Factory.newInstance(ValueTypes.STRING_UTF8, "big"));
entries.add(Entry.Factory.newInstance(ValueTypes.STRING_UTF8, "world"));
Capi.write(queueA, entries);
```

B)

```
//B's notification fires here!
```

```
//8) read entries from queue
List<Selector> sellist = new LinkedList<Selector>();
sellist.add(new FifoSelector(FifoLifoAccess.FIRST, Selector.CNT_ALL));
List<Entry> words = Capi.read(queueB, sellist);

//9) store the translation
KeySelector hello = new KeySelector("Key",ValueTypes.STRING_UTF8, “hello”);
KeySelector big  = new KeySelector("Key",ValueTypes.STRING_UTF8, “big”);
KeySelector world = new KeySelector("Key",ValueTypes.STRING_UTF8, “world”);
List<Selector> sellist = new LinkedList<Selector>();
List<Entry> translation = new LinkedList<Entry>();
Entry temp = Entry.Factory.newInstance(ValueTypes.STRING_UTF8, “hallo”);
sellist.clear()
sellist.add(hello);
temp.setSelectors(sellist);
translation.add(temp);
Entry temp = Entry.Factory.newInstance(ValueTypes.STRING_UTF8, “große”);
sellist.clear()
sellist.add(big);
temp.setSelectors(sellist);
translation.add(temp);
Entry temp = Entry.Factory.newInstance(ValueTypes.STRING_UTF8, “welt”);
sellist.clear()
sellist.add(world);
temp.setSelectors(sellist);
translation.add(temp);
Capi.write(dictB, translation);
```

A)

```
//A’s notification fires here and delivers the new entries
```

These steps are now shown in XML protocol notation.

A)

```
//1)
<Capi>
  <ContainerNamedCreate ContainerName='Dictionary'/>
</Capi>

<Ipac>
```

```
  <ContainerNamedCreate>
    <ContainerRef>dict</ContainerRef>
  </ContainerNamedCreate>
</Ipac>
```

//2)

```
<Capi>
  <ContainerProperties>
    <ContainerRef>dict</ContainerRef>
    <Property mode='SET' name="COORDINATION_TYPE" value='KEY'/>
  </ContainerProperties>
</Capi>
```

```
<Ipac>
  <ContainerProperties>
    <ACK/>
  </ ContainerProperties >
</Ipac>
```

//3)

```
<Capi>
  <ContainerNamedCreate ContainerName='Word-queue' BaseCoordination='FIFO'/>
</Capi>
```

```
<Ipac>
  <ContainerNamedCreate>
    <ContainerRef>queue</ContainerRef>
  </ContainerNamedCreate>
</Ipac>
```

//4)

```
<Capi>
  <CreateNotification mode='INFINITE'>
    <ContainerRef>dict</ContainerRef>
    <NotificationTarget>WRITE</NotificationTarget>
  </CreateNotification>
</Capi>
```

```
<Ipac>
  <CreateNotification>
```

```
    <ContainerRef>dict</ContainerRef>
    <NotificationID>notA_id</NotificationID>
  </CreateNotification>
</Ipac>
```

B)

//5)

```
<Capi>
  <GetNamedContainer>
    <ContainerName>Dictionary</ContainerName>
  </GetNamedContainer>
</Capi>

<Ipac>
  <GetNamedContainer>
    <ContainerRef>dict</ContainerRef>
  </GetNamedContainer>
</Ipac>

<Capi>
  <GetNamedContainer>
    <ContainerName>Word-queue</ContainerName>
  </GetNamedContainer>
</Capi>

<Ipac>
  <GetNamedContainer>
    <ContainerRef>queue</ContainerRef>
  </GetNamedContainer>
</Ipac>
```

//6)

```
<Capi>
  <CreateNotification mode='INFINITE'>
    <ContainerRef>queue</ContainerRef>
    <NotificationTarget>WRITE</NotificationTarget>
  </CreateNotification>
</Capi>

<Ipac>
  <CreateNotification>
    <ContainerRef>queue</ContainerRef>
    <NotificationID>notB_id</NotificationID>
```

```
  </CreateNotification>
</Ipac>
```

A)

//7)

```
<Capi>
  <Write operation='WRITE'>
    <ContainerRef>queue</ContainerRef>
      <Entry type='string/utf8'>
        <Value type='string/utf8'>hello</Value>
      </Entry>
      <Entry type='string/utf8'>
        <Value type='string/utf8'>big</Value>
      </Entry>
      <Entry type='string/utf8'>
        <Value type='string/utf8'>world</Value>
      </Entry>
  </Write>
</Capi>

<Ipac>
  <Write operation='WRITE'>
    <ACK/>
  </Write>
</Ipac>
```

B)

//8)

```
<Capi>
  <Read operation='READ'>
    <ContainerRef>queue</ContainerRef>
    <Selector>
      <Fifo position='FIRST' count='ALL'/>
    </Selector>
  </Read>
</Capi>

<Ipac>
  <Read operation='READ'>
    <Value type='string/utf8'>hello</Value>
    <Value type='string/utf8'>big</Value>
    <Value type='string/utf8'>world</Value>
  </Read>
```

```
</Ipac>
```

//9)

```
<Capi>
  <Write operation='WRITE'>
    <ContainerRef>queue</ContainerRef>
    <Entry type='string/utf8'>
      <Value type='string/utf8'>hallo</Value>
      <Selector>
        <Key>
          <Value type='string/utf8'>hello</Value>
        </Key>
      </Selector>
    </Entry>
    <Entry type='string/utf8'>
      <Value type='string/utf8'>große</Value>
      <Selector>
        <Key>
          <Value type='string/utf8'>big</Value>
        </Key>
      </Selector>
    </Entry>
    <Entry type='string/utf8'>
      <Value type='string/utf8'>welt</Value>
      <Selector>
        <Key>
          <Value type='string/utf8'>world</Value>
        </Key>
      </Selector>
    </Entry>
  </Write>
</Capi>

  <Ipac>
      <Write operation='WRITE'>
         <ACK/>
      </Write>
  </Ipac>
```

XV. Evaluation and Benchmarking

A famous saying goes like this: “Pre-emptive optimization is the root of all evil.” This is true and very much so in the domain of software development. Pre-emptive optimization often leads to unexpected behaviour and errors that can be hard to find and are more problematic and a slightly slower running piece of code. And still, sometimes a developer just knows the bits and pieces that need work. This would be the case that applies here.

While the design and architecture of the XVSM itself is already quite mature and solid, the MozartSpaces implementation on the other hand still is very young and some parts are still in the state of a prototype implementation. In this case it’s not yet too useful to run performance benchmarks.

First and foremost the MozartSpaces had to be implemented from scratch as fast as possible. We needed a proof-of-concept for the new API and programming models. It was clear upfront that this version of the MozartSpaces would most likely not survive the prototype phase and be replaced by more sophisticated versions. Of course the experience that we collected with the MozartSpaces implementation was invaluable to further design decisions and provides a solid bases to start design, architecture and implementation considerations for future versions.

XVI. Future Work

It has already been mentioned that the current MozartSpaces has multiple aspects that need attention. Improvements and future work that can and should be done will be listed here. This list is a starting point and does not claim to be complete.

- Changing the database structure to a more sensible and performance-friendly approach. This means that there shouldn't be a set of six tables for every container. While this might be an easy to handle approach, and it certainly helped in getting the MozartSpaces up and running, it is rather hostile towards performance. Every time a new container is created all those tables must be created as well and destroyed upon container removal.
- Implement tuples as real entries in the database. Currently, tuples are treated in a special way which permits easy template matching. The downside of this approach is the information loss about field types of nested tuples. One possible solution would be to treat tuples exactly the same way as entries and store links to other 'real' entries for every field of the tuple. This would prevent information loss and allow correct recovery of deeply nested tuples. The downside of this approach would probably be more sophisticated and complex SQL queries in order to read and write tuples into the space.
- The MozartSpaces have been implemented in a thread safe manner so that multiple threads can access the space concurrently without the danger of data corruption. The main goal in the first revision was to implement a feature complete XVSM system which most likely does not reflect optimal performance. Synchronized blocks that stall all threads but one should be revised and kept as local as possible.
- Currently the CADAO uses a single database connection. Again this approach works as a proof of concept but should be replaced with the usage of a connection pool. Resource pools contain and manage a (usually configurable) number of resources which are used by the client. In this case the pool would contain multiple database connection which

can be used by the CADAO to perform different space operations concurrently.

- The XML schema specification allows that syntactic description of valid XML data. This is an easy and intuitive approach to define structured data such as the XVSM protocol. The biggest problem here is that no semantic relations can be expressed. With XML schema such properties must be explained in the document section of each XML element. As it is the case with any contract that is not supported and enforced by tools errors might and most definitely will slip in and cause usually hard to track bugs. A semantic annotation extension for XML schema should be used to enforce these relations through tools as a supportive mechanism for developers.
- Once these rather basic optimizations have been implemented some real world performance and stress tests should be run against the MozartSpaces to further optimize and stabilize this XVSM system.

XVII. Conclusion

In this diploma thesis the architecture and implementation of the XVSM system has been presented.
It has been shown that rather simple concepts can be complex to implement and some complex concepts are fairly simple. Great care has been taken to provide a solid, stable and easy to use API to application programmers while still keeping the extensibility requirement in mind.
Extensibility is supported through multiple mechanisms such as the structural architecture of the XVSM core itself and the exploitation of the embedded XVSM to provide a full blown XVSM implementation over the network with the standalone version.

Also the simplicity of usage has been shown in the case of named containers. These are thoroughly implemented and supported through basic XVSM features. It is clear that sophisticated features can be created quite easily. This helps the spreading and adoption of the XVSM system in general and the MozartSpaces in particular. Application developers desire complex features that are packaged and accessible through a simple interface. Therein lies the need and usage for a middleware; it must support the developers in accomplishing their tasks faster and more reliable.

While all of this sounds great, this is still not the end of the road. Having a network aware XVSM implementation is a start but the next big step that must be taken is the distribution of the space itself through sophisticated replication mechanisms.

The XVSM itself as well as the MozartSpaces implementation are under an ongoing development and improvement. Up to date information can always be found at http://www.xvsm.org. Additionally information about space based computing in general can be found at http://www.spacebasedcomputing.org.

References

1. AG Netzbasierte Informationssysteme, *XMLSpaces*, http://www.ag-nbi.de/research/xmlspaces.net

2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995, *Design Patterns*, Addison-Wesley Professional Computing Series

3. eva Kühn, 2007, *Verteiltes Programmieren mit Space Based Computing Middleware*, http://www.complang.tuwien.ac.at/eva/Teaching/SBC/sbcIndex.html

4. eva Kühn, Johannes Riemer, Gerson Joskowicz, *XVSM (eXtensible Virtual Shared Memory) Architecture and Application*, Technical Report TU-Vienna, E185/1, SBC-Group, 2005

5. eva Kühn, Johannes Riemer, Lukas Lechner, *XVSMP/Bayeux: A Protocol for Scalable Space Based Computing in the Web*, Workshop on Interdisciplinary Aspects of Coordination Applied to Pervasive Environments: Models and Applications (CoMA), At the 16th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), Paris 2007.

6. eva Kühn, Severin Ecker, 2006, *XVSM – Protocol XML View Version 4.0.2*

7. eva Kühn, Severin Ecker, 2006, *XVSM – Protocol Samples Version 4.0.2*

8. Gerson Joskowicz, eva Kühn, Martin Murth, *The XD Model: Extending XML and DOM to Standards Based Coordination*, Proceedings of the 10th IASTED Int. Conf. on Software Eng. and Appl. (SEA), pp.146-152, Nov. 13-15, 2006, Dallas, USA, 2006.

9. gigaspaces, *GigaSpaces*, http://www.gigaspaces.com/

10. SUN, *Java Spaces Service Specification Version 2.2*, http://java.sun.com/products/jini/2.1/doc/specs/html/js-spec.html

11. The globus alliance, *GLOBUS*, http://www.globus.org

12.TUWien complang institute, *CORSO*, http://stud3.tuwien.ac.at/~e9825311/SBC/corso/docs

Abbreviations

API	Application Programming Interface
CADAO	Container Access Data Access Object
CAPI	Core API
CORSO	Coordinated Shared Objects
CREF	Container Reference
HTTP	Hypertext Transport Protocol
Java GC	Java Garbage Collector
OSS	Open Source Software
VSM	Virtual Shared Memory
W3C	World Wide Web Consortium
XCore	XVSM Core
XML	Extensible Markup Language
XSD	XML Schema Definition
XVSM	Extensible Virtual Shared Memory

Appendix A – XML Protocol XML Schema

The XML Schema Definition of the XML Protocol has been divided into three parts for clarity and maintainability reasons.

- XMLSchema_capi_request.xsd: Defines the request side of the protocol.
- XMLSchema_ipac_response.xsd: Defines the response side of the protocol.
- type_definitions.xsd: Defines the data types that are used within the previous two schema definitions.

XMLSchema_capi_request.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xvsmns="http://www.xvsm.org"
   targetNamespace="http://www.xvsm.org"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

  <xsd:include schemaLocation="type_definitions.xsd"/>

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
        The core api - request structures
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="Capi">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">

        <xsd:element name="ContainerNamedCreate">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">
          The ContainerNamedCreate command creates a new named container. The command
          must be given the desired container name as well as a size and a base coordination
          order. As in the CreateContainer command these are optional. Not considering the
          container name this command behaves exactly as the CreateContainer command.
            </xsd:documentation>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="ContainerName" type="xsd:string" use="required"/>
            <xsd:attribute name="ContainerSize" type="xvsmns:CONTAINER_SIZE"
use="optional" default="UNBOUNDED"/>
            <xsd:attribute name="BaseCoordination"
type="xvsmns:BASE_COORDINATION_ORDER" use="optional" default="RANDOM"/>
          </xsd:complexType>
        </xsd:element>

        <xsd:element name="DestroyNamedContainer">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">
          The DestroyNamedContainer is used to completely destroy a named container. This
          removes the container name from the space rendering the name inusable to any clients
          as well as destroying the container denoted by the container name.
            </xsd:documentation>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
              <xsd:element name="ContainerName" type="xsd:string"/>
            </xsd:sequence>
```

```
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="GetNamedContainer">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
      The GetNamedContainer command is used to retrieved the container reference of a
      container, that's denoted by the specified container name.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
          <xsd:element name="ContainerName" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="SetContainerName">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
      Given a container name and the reference to an already existing container, the
      SetContainerName command can be used to tie a name to a container. If successful,
      this container reference will then be returned by the GetNamedContainer command.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
          <xsd:element name="ContainerName" type="xsd:string"/>
          <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="RemoveContainerName">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
      In cases where it's needed to 'unname' a container without completely destroying it the
      RemoveContainerName command must be used. This unties the name from the
      container, removing it from the list of valid conatiner names but leaves the container as
      such untouched. All existing entries stay intact.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
          <xsd:element name="ContainerName" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="IsNamedContainer">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
```

```
        The IsNamedContainer command can be used to query the core whether the given
        container reference is named. It is not possible though to retrieve the name of a
        container reference.
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
            <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ContainerCreate">
        <xsd:annotation>
          <xsd:documentation xml:lang="en">
        The ContainerCreate command creates a new container. It can be embedded in a
        transaction.
        Only the most basic properties of a container can be set with this command; the size of
        the container and it's coordination type.
        All other properties have default values and can be changed using the
        ContainerSetProperties command.
        The advantage is that the ContainerCreate command stays simple.

        If not specified otherwise the new container will be created as unbounded container with
        no base ordering (= coordination type is RANDOM)
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="ContainerSize" type="xvsmns:CONTAINER_SIZE"
use="optional" default="UNBOUNDED"/>
          <xsd:attribute name="BaseCoordination"
type="xvsmns:BASE_COORDINATION_ORDER" use="optional" default="RANDOM"/>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ContainerDestroy">
        <xsd:annotation>
          <xsd:documentation xml:lang="en">
        Q: how exactly should this command work:
        we have a GC for removing unused containers and entries.
        - unpublish it, remove all references to it so the GC can delete it
        - explicitly delete it (hopefully with removal of all references to it and unpublishing it)
        - ...
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
            <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
          </xsd:sequence>
        </xsd:complexType>
```

```
    </xsd:element>

    <xsd:element name="ContainerProperties">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
      The properties of a container can be changed with this command.
      One command that can be used to set and 'reset' properties of a container results in a
      simpler API, yet as expressive and potent as if it would be with separate
      create/delete/add/remove/change properties command.
      Another side effect is a symmetry in the API between ContainerSet- and
      ContainerGetProperties.

      The command takes a container reference an optional transaction reference and a list of
      properties that must be set.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
          <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
          <xsd:element name="Property" type="xvsmns:property_type"
maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="ContainerPublishReference">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
      Publishing a container makes it accessible to clients outside the space.
      After a container has been published the GC is forbidden to remove this container, since
      it is still visible and accessible to (all) clients.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="ContainerUnPublishReference">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
      The ContainerUnPublishReference command removes the container reference from the
      publically accessible list of containers.
      If this container is not accessible (indirectly) from the outside anymore, the GC is free to
      delete this container.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
```

```
<xsd:element name="StartTransaction">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
  The StartTransaction command opens and starts a client initiated transaction. The
  transaction reference that is returned by this command stays valid until either event
  happens:
  - The specified timeout has been reached
  - The transaction is committed using the CommitTransaction command
  - The transaction is rolled back using the RollbackTransaction command
    </xsd:documentation>
  </xsd:annotation>
</xsd:element>

<xsd:element name="CommitTransaction">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
  The CommitTransaction command tries to commit the specified transaction.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="RollbackTransaction">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
  The RollbackTransaction command rolls back the specified transaction.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="CreateNotification">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
  The CreateNotification command creates a new notification for the specified container.
  Whenever a Write command is successfully carried out at this particular container the
  notification fires according to the timeout and the notification mode.

  - ContainerRef: the container that should be monitored for notifications
  - Timeout: the timespan during which the notification is valid
  - mode: the particular notification mode
  - returnEntries: specifies whether the entries that caused the notification to fire should
  be returned/saved or not.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
      <xsd:element name="Timeout" type="xvsmns:TIMEOUT" minOccurs="0"/>
```

```
          <xsd:element name="NotificationTarget" type="xvsmns:NOTIFICATION_TARGET"
maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="mode" type="xvsmns:NOTIFICATION_MODE" use="required"/>
        <xsd:attribute name="returnEntries" type="xsd:boolean" use="optional"
default="false"/>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="CancelNotification">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
        The CancelNotification command is used to end an existing notification. As soon as this
        command has been carried out not more notifications fire for that particular notification
        ID.
        </xsd:documentation>
      </xsd:annotation>

      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="NotificationID" type="xvsmns:TRANSACTION_ID"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="Read">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
        The Read command reads entries stored in a container. It has one attribute of type
        READ_OPERATION which specified the mode of the read action.
        The default read mode is READ.
        The Read command has a container reference where the read action must be carried
        out, an optional transaction reference and an optional timeout.
        It also has a list of Selectors that must be used to specify the entries that must be
        retrieved.
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
          <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
          <xsd:element name="Timeout" type="xvsmns:TIMEOUT" minOccurs="0"/>
          <xsd:element name="Selector" type="xvsmns:read_selector_type"
minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="operation" type="xvsmns:READ_OPERATION" use="optional"
default="READ"/>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="Write">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
        The Write command stores entries in the specified container.
        It has one attribute of type WRITE_OPERATION which specified the mode of the read
        action. The default write mode is WRITE.
```

The Write command has a container reference where the write action must be carried out, an optional transaction reference and an optional timeout.
It has a list of entries that must be written and for each entry a list of selectors that specify which 'meta-information' must be written.

In WRITE mode either all entries are written or the command blocks.

```
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
          <xsd:element name="TransactionRef" type="xvsmns:TRANSACTION_REF"
minOccurs="0"/>
          <xsd:element name="Timeout" type="xvsmns:TIMEOUT" minOccurs="0"/>
          <xsd:element name="Entry" type="xvsmns:entry_type" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="operation" type="xvsmns:WRITE_OPERATION"
use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
 </xsd:complexType>
</xsd:element>
</xsd:schema>
```

XMLSchema_ipac_response.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xvsmns="http://www.xvsm.org"
   targetNamespace="http://www.xvsm.org"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

  <xsd:include schemaLocation="type_definitions.xsd"/>

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
        The core api - response structures.
        In case of internal or other errors or core failures, a Fault response is always returned at
        the level of occurance.
        E.g.: if the capi request is malformed, an ipac containing the Faule response is returned.
        On the other hand if the write command tries to write to a non existing container an
        ipac-write response containing the Fault response is returned.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="Ipac">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="ContainerNamedCreate">
            <xsd:complexType>
              <xsd:annotation>
                <xsd:documentation xml:lang="en">
        If the named container could not be created a Fault response is returned. In case of
        successful creation the new container reference is returned.
                </xsd:documentation>
              </xsd:annotation>
              <xsd:choice>
                <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
                <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
              </xsd:choice>
            </xsd:complexType>
          </xsd:element>

          <xsd:element name="DestroyNamedContainer" type="xvsmns:ipac_simple_return"/>

          <xsd:element name="GetNamedContainer">
            <xsd:complexType>
              <xsd:annotation>
                <xsd:documentation xml:lang="en">
        If the specified name does not depict a named container a Fault response is returned.
        Otherwise the container reference of the named container is returned.
                </xsd:documentation>
              </xsd:annotation>
              <xsd:choice>
                <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
                <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
              </xsd:choice>
            </xsd:complexType>
```

```
</xsd:element>

<xsd:element name="SetContainerName" type="xvsmns:ipac_simple_return"/>

<xsd:element name="RemoveContainerName" type="xvsmns:ipac_simple_return"/>

<xsd:element name="IsNamedContainer">
 <xsd:complexType>
  <xsd:annotation>
   <xsd:documentation xml:lang="en">
The response attribute specifies whether the provided container reference depicts a
named conatiner.
   </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
   <xsd:element name="Fault" type="xvsmns:ipac_fault" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="response" type="xsd:boolean" use="required"/>
 </xsd:complexType>
</xsd:element>

<xsd:element name="ContainerCreate">
 <xsd:complexType>
  <xsd:annotation>
   <xsd:documentation xml:lang="en">
Returns the container reference of the newly created container.
   </xsd:documentation>
  </xsd:annotation>
  <xsd:choice>
   <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
   <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
  </xsd:choice>
 </xsd:complexType>
</xsd:element>

<xsd:element name="ContainerDestroy" type="xvsmns:ipac_simple_return"/>

<xsd:element name="ContainerProperties">
 <xsd:complexType>
  <xsd:annotation>
   <xsd:documentation xml:lang="en">
Returns the list of Properties that must be retrieved. If only the SET/RESET property
modes were used a simple ipac response is returned.
   </xsd:documentation>
  </xsd:annotation>
  <xsd:choice>
   <xsd:element name="ACK" type="xvsmns:ipac_ack"/>
   <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
   <xsd:sequence maxOccurs="unbounded">
    <xsd:element name="Property" type="xvsmns:property_type"/>
   </xsd:sequence>
  </xsd:choice>
 </xsd:complexType>
</xsd:element>

<xsd:element name="ContainerPublishReference" type="xvsmns:ipac_simple_return"/>
```

```
        <xsd:element name="ContainerUnPublishReference"
type="xvsmns:ipac_simple_return"/>

        <xsd:element name="StartTransaction" type="xvsmns:ipac_simple_return"/>

        <xsd:element name="CommitTransaction" type="xvsmns:ipac_simple_return"/>

        <xsd:element name="RollbackTransaction" type="xvsmns:ipac_simple_return"/>

        <xsd:element name="CreateNotification">
          <xsd:complexType>
            <xsd:annotation>
              <xsd:documentation xml:lang="en">
        Returns the list of Properties that must be retrieved. If only the SET/RESET property
        modes were used a simple ipac response is returned.
              </xsd:documentation>
            </xsd:annotation>
            <xsd:choice>
              <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
              <xsd:sequence>
                <xsd:element name="ContainerRef" type="xvsmns:CONTAINER_REF"/>
                <xsd:element name="NotificationID" type="xvsmns:TRANSACTION_ID"/>
              </xsd:sequence>
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>

        <xsd:element name="CancelNotification" type="xvsmns:ipac_simple_return"/>

        <xsd:element name="Read">
          <xsd:complexType>
            <xsd:annotation>
              <xsd:documentation xml:lang="en">
        If the read operation was DESTROY, a simple ipac return value is returned. If the read
        operation was either READ or TAKE the list of retrieved entries is returned.
              </xsd:documentation>
            </xsd:annotation>
            <xsd:choice>
              <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
              <xsd:choice>
                <xsd:annotation>
                  <xsd:documentation xml:lang="en">
        The read ipac either is a simple ACK element if the operation was DESTROY or the list of
        entries in case of READ/TAKE.
                  </xsd:documentation>
                </xsd:annotation>
                <xsd:element name="ACK" type="xvsmns:ipac_ack"/>
                <xsd:sequence>
                  <xsd:element name="Value" type="xvsmns:template_value_type"
maxOccurs="unbounded"/>
                </xsd:sequence>
              </xsd:choice>
            </xsd:choice>

            <xsd:attribute name="operation" type="xvsmns:READ_OPERATION" use="optional"
default="READ"/>
          </xsd:complexType>
        </xsd:element>
```

```
        <xsd:element name="Write">
          <xsd:annotation>
            <xsd:documentation xml:lang="en">
        This is basically just a simple ipac return value, adding the write operation mode as
        attribute to the response.
            </xsd:documentation>
          </xsd:annotation>

          <xsd:complexType>
            <xsd:choice>
              <xsd:element name="ACK" type="xvsmns:ipac_ack"/>
              <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
            </xsd:choice>
            <xsd:attribute name="operation" type="xvsmns:WRITE_OPERATION"
use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:choice>
  </xsd:complexType>
 </xsd:element>

</xsd:schema>
```

type_definitions.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xvsmns="http://www.xvsm.org"
   targetNamespace="http://www.xvsm.org"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

  <!-- TYPE DEFINITIONS -->
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
        List of complex type definitions.
    </xsd:documentation>
  </xsd:annotation>
  <!-- SELECTOR TYPE DEFINITIONS -->
  <!-- SET -->
  <xsd:complexType name="set_selector_type">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        Randomly selects the specified number of entries.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="count" type="xvsmns:COUNTER" use="optional" default="1"/>
  </xsd:complexType>

  <!-- LRU -->
  <xsd:complexType name="lru_selector_type">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        Selects the least recently used entries.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="count" type="xvsmns:COUNTER" use="optional" default="1"/>
  </xsd:complexType>

  <!-- VECTOR -->
  <xsd:complexType name="vector_selector_type">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        The Vector selector can be used to select a number of subsequent items starting from a
        specified index.
        This selector can't be used for set containers since they by definition are unordered
        whereas the Vector selector requires ordering of entries.

        Two attributes specify the entries that must be selected.

        - position: the starting position of the entries
        - count: the number of entries that must be selected (a negative number specifies a
        backward direction)
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="Value" type="xvsmns:vector_value_type"/>
    </xsd:sequence>
    <xsd:attribute name="count" type="xvsmns:COUNTER" use="optional" default="1"/>
```

```
</xsd:complexType>

<!-- FIFO -->
<xsd:complexType name="fifo_selector_type">
 <xsd:annotation>
  <xsd:documentation xml:lang="en">
     The Vector selector can be used to select a number of subsequent items starting from a
     specified index.
     This selector can't be used for set containers since they by definition are unordered
     whereas the Vector selector requires ordering of entries.

     Two attributes specify the entries that must be selected.

     - position: the starting position of the entries
     - count: the number of entries that must be selected (a negative number specifies a
     backward direction)
  </xsd:documentation>
 </xsd:annotation>
 <xsd:attribute name="position" type="xvsmns:FIFO_LIFO_ACCESS" use="required"/>
 <xsd:attribute name="count" type="xvsmns:COUNTER" use="optional" default="1"/>
</xsd:complexType>

<!-- LIFO -->
<xsd:complexType name="lifo_selector_type">
 <xsd:annotation>
  <xsd:documentation xml:lang="en">
     The Vector selector can be used to select a number of subsequent items starting from a
     specified index.
     This selector can't be used for set containers since they by definition are unordered
     whereas the Vector selector requires ordering of entries.

     Two attributes specify the entries that must be selected.

     - position: the starting position of the entries
     - count: the number of entries that must be selected (a negative number specifies a
     backward direction)
  </xsd:documentation>
 </xsd:annotation>
 <xsd:attribute name="position" type="xvsmns:FIFO_LIFO_ACCESS" use="required"/>
 <xsd:attribute name="count" type="xvsmns:COUNTER" use="optional" default="1"/>
</xsd:complexType>

<!-- KEY -->
<xsd:complexType name="key_selector_type">
 <xsd:annotation>
  <xsd:documentation xml:lang="en">
     The Key selector selects entries in the container that match the specified key.
     Since the keys must be unique for a given key name, at most one entry can be retrieved
     with this selector.

     - Value: value entry that contains the value of the key selector.
     - name: the name of the key (this allows entries to have multiple keys)
     - type: the type of the key value
     - arity: if the value is a tuple, the arity must be specified
  </xsd:documentation>
 </xsd:annotation>
 <xsd:sequence maxOccurs="unbounded">
  <xsd:element name="Value" type="xvsmns:key_value_type" maxOccurs="unbounded"/>
```

```
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xvsmns:VALUE_TYPE" use="required"/>
    <xsd:attribute name="arity" type="xsd:int" use="optional"/>
  </xsd:complexType>

  <!-- TEMPLATE -->
  <xsd:complexType name="template_selector_type">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
          The templte selector is a template matcher that can be used to select tuples.
          As has been stated above several subsequent templte selectors can be merged due to
          performance reasons.

          - Value: the value of the template
          - count: the number of entries that the selector returns
          - arity: the number of fields in the tuple
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="Value" type="xvsmns:template_value_type"
maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="count" type="xvsmns:COUNTER" use="optional" default="1"/>
    <xsd:attribute name="arity" type="xsd:int" use="required"/>
  </xsd:complexType>

  <!-- VALUE ELEMENT TYPE DEFINITIONS -->
  <xsd:complexType name="template_value_type" mixed="true">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
          This type is used to specify the value of a template. The mixed type allows for recursive
          template specification.

          - Value: the value of the template field
          - type: the type of the template field
          - arity: the number of fields in the tuple, if the type is tuple
          - id: the position of the field in the template. Positions start at 0
      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="Value" type="xvsmns:template_value_type"/>
    </xsd:sequence>
    <xsd:attribute name="type" type="xvsmns:VALUE_TYPE" use="required"/>
    <xsd:attribute name="arity" type="xsd:int" use="optional"/>
    <xsd:attribute name="id" type="xsd:int" use="optional"/>
    <xsd:attribute name="isNull" type="xsd:boolean" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="key_value_type" mixed="true">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
          This type is used to specify the value of a key.

          - Value: the value of the key
          - type: the type of the key
          - arity: the number of fields in the key, if the type is tuple
```

```
    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="Value" type="xvsmns:key_value_type"/>
  </xsd:sequence>
  <xsd:attribute name="type" type="xvsmns:VALUE_TYPE" use="required"/>
  <xsd:attribute name="arity" type="xsd:int" use="optional"/>
</xsd:complexType>

<xsd:complexType name="vector_value_type" mixed="true">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      This type is used to specify the value of a vector. The value of the vector position is
      directly contained in this type.
      If a range should be selected the following sequence of elements is used to specify that.

      - From: the range starting index.
      - To: the range end index (inclusive).
    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="From" type="xsd:int"/>
    <xsd:element name="To" type="xsd:int"/>
  </xsd:sequence>
</xsd:complexType>

<!-- READ SELECTOR DEFINITION -->
<xsd:complexType name="read_selector_type">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      A Selector is a sequence consisting of any combination of different selector types.
      Currently there are 6 selector types specified:
      - Set
      - Vector
      - Fifo
      - Lifo
      - Template
      - Key

      This selector sturcture allows arbitrary complex selection queries.
      e.g.: vector matching on the entries that were previously selected by using template
      matching on tuples.
      If no selector is specified then the core automatically creates the default selector for the
      given container. This means a selector for the container's base order with a default
      count of 1.

      The selection is carried out as follows (possible optimizations like merging subsequent
      template selectors is not specified here)
      for the first element in the selector
      - find the matching entries in the target container
      for all subsequent elements in the selector
      - take the previously selected entries
      - find the matching entries for the current selector in that result set and make this the
      new current result set
```

```
        return the final list of matching entries (considering blocking read and count values) to
        the client
      </xsd:documentation>
    </xsd:annotation>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="Set" type="xvsmns:set_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="Vector" type="xvsmns:vector_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="Fifo" type="xvsmns:fifo_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="Lifo" type="xvsmns:lifo_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="Key" type="xvsmns:key_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="Template" type="xvsmns:template_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="Lru" type="xvsmns:lru_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:choice>
  </xsd:complexType>

  <!-- WRITE SELECTOR DEFINITION -->
  <xsd:complexType name="write_selector_type">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        A Selector is a sequence consisting of any combination of different selector types.
        Currently there are 3 selector types specified:
        - Vector
        - Template
        - Key

        The write operation can be/is carried out as follows:

        - the entry is written to the container using the base order coordination
        - the selectors are carried out, checking if the writing of the selector meta-information is
        possible (e.g.: if the specified key is occupied)

        if the write operation is not possible, the command blocks.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="Vector" type="xvsmns:vector_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="Template" type="xvsmns:template_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
      <xsd:element name="Key" type="xvsmns:key_selector_type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:choice>
  </xsd:complexType>

  <!-- WRITE ENTRY TYPE DEFINITION -->
  <xsd:complexType name="entry_type">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        An Entry, whose type is specified with the type attribute is a single entity that is written
        into a container.
```

```
        It contains a list of selectors that specify the meta-information that must be written for
        this entry (and of course where it must be written).

        Whether the Entry is a primitive type or a tuple, the value that must be written is a
        value element (primitive type) or a list of Arguments (tuple).
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="Selector" type="xvsmns:write_selector_type" minOccurs="0"/>
      <xsd:element name="Value" type="xvsmns:template_value_type"
maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="type" type="xvsmns:VALUE_TYPE" use="required"/>
    <xsd:attribute name="arity" type="xsd:int"/>
  </xsd:complexType>

  <!-- CONTAINER PROPERTIES -->
  <xsd:complexType name="property_type">
    <xsd:annotation>
      <xsd:documentation>
        A property consists of a name, which may be any of the names listed in the
        CONTAINER_PROPERTY type and a value that must be set for that specific property.
        The mode specifies whether the property must be set or reset to its default value.

        The keyName and keyType are mandatory if the KEY coordiantion type is set or reset.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="name" type="xvsmns:CONTAINER_PROPERTY" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="optional"/>
    <xsd:attribute name="keyName" type="xsd:string" use="optional"/>
    <xsd:attribute name="keyType" type="xvsmns:VALUE_TYPE" use="optional"/>
    <xsd:attribute name="mode" type="xvsmns:PROPERTY_MODE" use="required"/>
  </xsd:complexType>


  <xsd:annotation>
    <xsd:documentation xml:lang="en">
        List of simple type definitions that are used as attribute types in the schema.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleType name="READ_OPERATION">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        The Read mode can be either of three values:
        - READ: reads entries and leaves them in the container
        - TAKE: reads entries and removes them from the container
        - DESTROY: removes entries from the container
      </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="READ"/>
      <xsd:enumeration value="TAKE"/>
      <xsd:enumeration value="DESTROY"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="WRITE_OPERATION">
```

```
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      The Write mode can be either of two values:
      - WRITE: writes entries to a container, blocking if the position/key.. is occupied
      - SHIFT: writes entries to a container, shifting out entries using the base coordination if
      needed
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="WRITE"/>
    <xsd:enumeration value="SHIFT"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="NOTIFICATION_MODE">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      The notification mode specifies the behaviour and number of notifications fired when the
      specified container is altered.

      - ONCE: the notification fires a single time when entries are written to the container. If
      the timeout has been succeeded nothing happens.
      - PROLONG: the notification fires any time when entries are written to the container. If
      the notification fires, its timeout is reset starting at the beginning. If the timeout has
      been succeeded nothing happens.
      - RESTRICED: the notification fires any time when entries are written to the container as
      long as the timeout is still valid.
      - INFINITE: the notification fires any time when entries are written to the container. The
      notification timeout is ignored.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="ONCE"/>
    <xsd:enumeration value="PROLONG"/>
    <xsd:enumeration value="RESTRICTED"/>
    <xsd:enumeration value="INFINITE"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="NOTIFICATION_TARGET">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      The notification target is used to specify the operation(s) which trigger the notification.
      Whenever an operation has successfully been completed an a notification has been
      created with this operation as notification target, it fires.
      The targets are as follows (for more details check the detailed operation description):

      - WRITE: a write operation (writing entries to the container)
      - SHIFT: a shift operation (shifting entries to the container)
      - READ: a read operation (reading entries from the container)
      - TAKE: a take operation (taking entries from the container)
      - DESTROY: a destroy operation (removing entries from the container)
      - DESTROY_CONTAINER: a container destruction operation (destorying a container)
      - ANY: any of the above operations
    </xsd:documentation>
  </xsd:annotation>
```

```
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="WRITE"/>
    <xsd:enumeration value="SHIFT"/>
    <xsd:enumeration value="READ"/>
    <xsd:enumeration value="TAKE"/>
    <xsd:enumeration value="DESTROY"/>
    <xsd:enumeration value="DESTROY_CONTAINER"/>
    <xsd:enumeration value="ANY"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="BASE_COORDINATION_ORDER">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
       Specifies the base coordination of a container. This coordination is used for default
       selecting entries and writing entries.
       - RANDOM: no particular order
       - FIFO: first in first out/queue ordering
       - LIFO: last in first out/stack ordering
       - LRU: least recently used
    </xsd:documentation>
  </xsd:annotation>

  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="RANDOM"/>
    <xsd:enumeration value="FIFO"/>
    <xsd:enumeration value="LIFO"/>
    <xsd:enumeration value="LRU"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="COORDINATION_TYPE">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
       Coordination types are used for reading and writing entries. They can be attached and
       removed from the container (as long as certaing properties hold such as container being
       empty or one is not trying to remove the base order)
       Valid coordination types are:

       - RANDOM: no particular order
       - FIFO: first in first out/queue ordering
       - LIFO: last in first out/stack ordering
       - LRU: least recently used
       - VECTOR: numerically indexed container
       - KEY: key-indexed conatiner
       - LINDA: container providing possibility for Linda template matching
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="RANDOM"/>
    <xsd:enumeration value="FIFO"/>
    <xsd:enumeration value="LIFO"/>
    <xsd:enumeration value="LRU"/>
    <xsd:enumeration value="KEY"/>
    <xsd:enumeration value="VECTOR"/>
    <xsd:enumeration value="LINDA"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<xsd:simpleType name="CONTAINER_PROPERTY">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
       This enumeration specifies the properties of a container that can be set and retrieved.
       - SIZE: the size of the container
       - FILL_SIZE: the number of entries that are currently stored in the container
       - COORDINATION_TYPES: all coordination types of the container
       - BASE_COORDINATION_TYPE: the base coordination type of the container
       - VALUE_TYPES: the allowed types for the entries in the container
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="SIZE"/>
    <xsd:enumeration value="FILL_SIZE"/>
    <xsd:enumeration value="COORDINATION_TYPES"/>
    <xsd:enumeration value="BASE_COORDINATION_ORDER"/>
    <xsd:enumeration value="VALUE_TYPES"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="PROPERTY_MODE">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
       Specifies the status of a container property
       - SET: the property is set/exists/is active
       - GET: retrieve the specified container protperty
       - RESET: the property is reset/removed/inactive
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="SET"/>
    <xsd:enumeration value="GET"/>
    <xsd:enumeration value="RESET"/>
    <xsd:enumeration value="REPORT"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="VALUE_TYPE">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
       Specifies the type of an entry
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="string/utf8"/>
    <xsd:enumeration value="string/8859-1"/>
    <xsd:enumeration value="binary/enc64"/>
    <xsd:enumeration value="integer/ascii"/>
    <xsd:enumeration value="cref/ascii"/>
    <xsd:enumeration value="tuple"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="CONTAINER_REF">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
       The type for specifying a reference to a container.
```

```
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="TRANSACTION_ID">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Unique identification for a notification.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="TRANSACTION_REF">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      The type for specifying a reference to a transaction.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="FIFO_LIFO_ACCESS">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      This type specifies the position for a Vector selector. It can be either of three different
      values (arroding to the base ordering):
      - FIRST: the first element of the container
      - LAST: the last element of the container
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="FIRST"/>
    <xsd:enumeration value="LAST"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="CONTAINER_SIZE">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Specifies the size of a container. It can either be a positive number (bounded container)
      or 'unbounded'.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:union memberTypes="xsd:int">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="UNBOUNDED"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>

<xsd:simpleType name="TIMEOUT">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
```

```
        Specifies a timeout in seconds. Infinite specifies an infinite timeout and 0 means no
        timeout.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:union memberTypes="xsd:int">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="INFINITE"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>

  <xsd:simpleType name="COUNTER">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        Specifies either a specific number of entries that must be retrieved or all that are
        available.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:union memberTypes="xsd:int">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="ALL"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>

  <xsd:complexType name="ipac_simple_return">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        basic return type format for operations that don't have any return value.
      </xsd:documentation>
    </xsd:annotation>
    <xsd:choice>
      <xsd:element name="ACK" type="xvsmns:ipac_ack"/>
      <xsd:element name="Fault" type="xvsmns:ipac_fault"/>
    </xsd:choice>
  </xsd:complexType>

  <xsd:complexType name="ipac_fault">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        specifies an erroneous operation execution
      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="Description" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="exceptionclass" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="ipac_ack" mixed="false">
    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        specifies a simple acknowledge message, stating the successful execution of an
        operation with no other return value.
```

```
      </xsd:documentation>
    </xsd:annotation>
  </xsd:complexType>
</xsd:schema>
```