FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Pluggable Design and Implementation of the XVSM Framework Core for .NET

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Andreas Grill, BSc.
Matrikelnummer 00616044

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Mitwirkung: Projektass. Dipl.-Ing. Stefan Craß

Wien, 2. Mai 2018

Andreas Grill — eva Kühn

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Pluggable Design and Implementation of the XVSM Framework Core for .NET

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Andreas Grill, BSc.
Registration Number 00616044

to the Faculty of Informatics

at the TU Wien

Advisor: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Assistance: Projektass. Dipl.-Ing. Stefan Craß

Vienna, 2nd May, 2018

Andreas Grill eva Kühn

Erklärung zur Verfassung der Arbeit

Andreas Grill, BSc.
Schumanngasse 1 / 5, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Mai 2018

Andreas Grill

Acknowledgements

I want to thank professor eva Kühn and her research assistant Stefan Craß for creating the opportunity to work in this field and for all the helpful feedback and guidance they have provided throughout the work.

Finally, I want to say a big thank you to my girlfriend, and my parents. You should know that this wouldn't have been possible without your patience and encouragement.

Kurzfassung

Im Zuge der Ausbreitung von internetfähigen Geräten gewinnt dezentrale Kommunikation an Bedeutung. Space-based Computing mittels XVSM ermöglicht intuitive Koordinationsmechanismen, welche auf dem Blackboard-Modell basieren um die Entwicklung von verteilten Systemen zu vereinfachen. Dies wird ermöglicht durch die Unterstützung von Erweiterungen und benutzerdefinierten Mechanismen zur Koordination zwischen den Systemen. Im Verlauf der Weiterentwicklung von XVSM-Implementierungen wurden diese um Features ergänzt, um XVSM in weiteren Bereichen anwenden zu können. Eine Untersuchung der XVSM-Implementierungen auf unterschiedlichen Plattformen hat allerdings ergeben, dass diverse Erweiterungen tief integriert und mit hoher Koppelung versehen sind, und benutzerdefinierte Koordinatoren sowohl ein hohes Maß an dupliziertem Code aufweisen, als auch auf implementierungsspezifische Details der XVSM-Implementierung Rücksicht genommen werden muss. Obwohl XVSM im Hinblick auf Modularität und Erweiterbarkeit entwickelt wurde, konnten gewisse Erweiterungen nicht ausreichend entkoppelt integriert werden. In dieser Arbeit wird deshalb ein Konzept für eine hoch modulare XVSM-Implementierung vorgestellt und anhand einer Referenzimplementierung für die .NET-Plattform belegt. Es wird ein Plugin-basierter Mechanismus geboten, welcher es ermöglicht, Erweiterungen durch Hinzufügen oder Ersetzen von Plugins zu verwalten. Es werden unterschiedliche XVSM- und andere Space-based Middleware-Implementierungen sowohl anhand deren Erweiterbarkeitsmechanismen, als auch im Hinblick auf deren Funktionalität und Defizite bezüglich Erweiterbarkeit verglichen. Ein besonderer Fokus wird auf die XVSM-Referenzimplementierung MozartSpaces und deren Unterstützung für benutzerdefinierte Koordinatoren gelegt. In Folge dessen wird eine modulare Architektur basierend auf dem Managed Extensibility Framework (MEF) für die .NET-Plattform entwickelt, um die XVSM-Implementierung mit zusätzlichen Erweiterungsmechanismen zu ergänzen. Mechanismen für Erweiterungen, welche auf aspektorientierten Ansätze basieren, werden erweitert, um feingranulare Integrationen zu ermöglichen. Durch eine Ergänzung der Funktionalität des Transaktionssystems wird dessen Anwendbarkeit im Hinblick auf eine Erweiterung von XVSM verbessert. Mit der vorgestellten Architektur werden nicht nur Grundlagen für neue und potentiell experimentelle Erweiterungen der XVSM-Implementierung für die .NET Plattform gelegt, sondern auch eine Referenz für andere XVSM-Implementierungen hinsichtlich Flexibilität und Erweiterbarkeit geschaffen.

Abstract

With the increasing number of smart and internet connected devices, decentralized communication between devices is getting more important. Space-based computing with XVSM provides intuitive coordination mechanisms based on a blackboard architectural model to ease development for distributed systems. Extensibility and custom coordination logic are major features of XVSM. Maturing XVSM implementations gained several extensions to widen their applicability. However, a survey of existing XVSM implementations on various platforms reveals hardwired integrations and custom coordinators with frequent code duplications and deep implementation dependencies. Even though XVSM specifies extensibility mechanisms and a modular architecture, some extensions require more fine-grained mechanisms. In order to solve this problem, we propose a highly modular plugin-based XVSM architecture along with a reference implementation for the .NET platform that supports an extension mechanism simply by adding or replacing plugins. We compared various XVSM and other space-based middlewares regarding their extensibility mechanisms, as well as their implemented features and shortcomings in regard to extensibility, but our main focus was on the XVSM reference implementation MozartSpaces and especially its custom coordinator support. Consequently we developed a modular architecture based on the *Managed Extensibility Framework* (MEF) for the .NET platform to add additional extensibility mechanisms. Further, we added aspect-oriented integration on a fine-grained level, as well as an evolutionary improvement of XVSM's transaction system to widen the applicability of XVSM's specified extensibility mechanism. The new architecture not only lays the foundation for new, even experimental, extension development for XVSM on the .NET platform, but also provides a reference for other XVSM implementations that could benefit from incorporating its extensible and flexible features.

Contents

CHAPTER 1

Introduction

With the increasing number of smart devices and the consolidation of the Internet of Things, challenges arise regarding communication and synchronization between these devices [Cis17]. Centralized architectures might not be feasible for all situations. The peer-to-peer (P2P) architecture provides an alternative with great scalability characteristics [AS04]. In contrast to a client-server architecture every host can act at the same time as a client or server. This decentralized approach releases the bottleneck of a central server host, and allows the utilization of computing resources such as CPU cycles, storage, and bandwidth of the peers while allowing to share content.

However, when it comes to collaboration between peers, coordination challenges arise. The *Tuple Space* (TS) with *Linda* as a coordination language [Gel85], which was introduced by Gelernter in 1985, provides an approach that utilizes the advantages of the P2P architecture with the simplicity of a client-server architecture. TS is a blackboard approach that decouples communication through space and time, by using tuples as its primary data structure. The basic statements of TS are *out*, *in*, and *rd* which correspond to writing a tuple to TS, taking a tuple out of TS, and reading a tuple from TS without withdrawing it. To retrieve tuples through the *in* and *rd* statements, the *Linda* coordination language is used to apply a template matching mechanism to the tuples.

The *eXtensible Virtual Shared Memory*[1] (XVSM) [KRJ05; Cra10; CKS09] is the reference architecture of the space-based computing architectural style (SBC) [Mor10], developed at the Institute of Computer Languages of TU Wien. It serves as the foundation of several middlewares in the context of distributed systems. A middleware is a system that may be characterized as an abstraction layer between the operating system and the application. For this thesis, we focus on middlewares that abstract communication services between applications to form distributed systems [Ber96; EAS07]. In XVSM

[1]http://www.xvsm.org

a space is a set of containers, which store entries and their corresponding coordination data. XVSM provides a similar basic functionality as the original TS with *Linda*, by providing methods for XVSM peers to write, take and read entries to and from a container. However, it was also developed with a focus on flexibility, extensibility and usability in mind. In contrast to TS and *Linda*, XVSM abstracts the coordination model and provides additional coordination laws that are enforced by separate *Coordinators* on each container in order to retrieve data. In addition to the predefined basic coordination laws, it is even possible in XVSM to define coordination laws that are optimized for the usage context by providing custom coordinators. The concrete implementation of this extensibility mechanism depends on the particular XVSM framework and is not exactly specified in XVSM. Integrated aspects form another mechanism that can be used to extend the XVSM framework, by providing callbacks and interceptor points to space or container operations. Even though it is possible with aspects to change the behavior of a space or a container in an easy and flexible way, this mechanism has proven to be not sufficient for all possible extensions.

Several XVSM implementations have been developed in various programming languages such as Java, C#, Haskell, JavaScript, and Objective-C. The current reference implementation MozartSpaces 2.0 [Bar10; Dön11] was developed in Java and provides the most features of all XVSM implementations. The latest C# implementations XCOSpaces [Kar09; Sch08] and TinySpaces [Mar10], although providing basic XVSM functionality, both have not gained the same amount of extensions as MozartSpaces. As TinySpaces is a lightweight XVSM implementation targeting embedded systems and XCOSpaces is older than MozartSpaces 2.0, new features from the current MozartSpaces implementation could not get easily integrated in these frameworks. Even though XVSM supports aspects as a native extensibility mechanism, many extensions such as persistence [Zar12], security [CK12], replication [Hir12], and lifecycle extensions [Wat15; Cra+17] had to be tightly integrated with the MozartSpaces framework. The extension implementations take advantage of the known behavior of the framework, which leads to tightly coupled extensions that cannot easily be transferred to other XVSM implementations. With every new extension for MozartSpaces that takes advantage of the particular framework implementation, the amount of required work to migrate the extension to another XVSM framework, with a different internal architecture than MozartSpaces, increases gradually.

This thesis aims at designing the core of a new XVSM framework that addresses the stated extensibility issues. It should lay the foundation for new development that uses XVSM as a space-based computing platform. The framework consists of a micro kernel and separate modular and encapsulated components that provide the XVSM functionality. Development of new and experimental extensions will benefit from this architecture. The integration of such extensions should be simple without the need to refactor big parts of the XVSM implementation. This thesis puts a focus on the simple integration of custom coordinators.

To achieve the stated objective, various challenges have to be resolved. In order to provide the required tools for the development of the new XVSM framework, a custom plugin framework based on existing extensibility frameworks has to be created. Even though the extensibility frameworks provide various mechanisms to create adaptable software, they must be integrated and customized to be used for all development requirements of the XVSM framework. Different types of plugins with separate extension and performance characteristics have to be developed to address the framework's extensibility requirements, while still taking the possible performance impacts into account. The challenge is to orchestrate and customize the extensibility frameworks to provide a toolset that can be used to implement the plugin-based XVSM architecture.

The XVSM architecture and its implementations are analyzed to define the plugin-based architecture. Tightly coupled code parts, as well as deeply integrated extensions of previous XVSM implementations influence the creation of the plugin architecture. The former are split into multiple plugins. Inversion of control mechanisms from the plugin framework are used to further decouple the plugins. Method signatures can be simplified, as context objects can be utilized instead of rarely used arguments.

The XVSM framework implementation of the plugin-based architecture is focused on XVSM's coordination and transaction layers, as well as providing an overall flexible framework, which was designed to be easily extended with features from other XVSM implementations. The transaction implementation supports XVSM's requirements, but also provides features that benefit the overall extensibility of the framework, as it improves the separate usage of transactions in extensions. Developing nested transactions with a plugin-based architecture that are compatible with XVSM's formal model is one of the major challenges of this thesis. Code involved in the coordination process is separated into a general part and individual coordinator semantics. Coordinators are classified with various properties that are used in the general part to control the coordination process. This leads to simpler and more streamlined coordinator implementations.

1.1 Methodology

This thesis uses the guidelines for *Design Science Research* from [Hev+04]. They provide a structural approach for effective information system development and research.

The *Problem Relevance* and *Research Contributions* guidelines have already been established in this chapter. Since XVSM is an established space-based computing model with several extensions for its concrete implementations, the new implementation, which simplifies the integration of new and existing extensions, should follow the design decisions presented in XVSM's formal model [Cra10]. To follow the *Design as a Search Process* and *Research Rigor* guidelines, concrete XVSM middlewares have been compared with other space-based middlewares regarding their extensibility characteristics. This led to the development of architectural propositions, which have been presented at a Modularity Workshop and shown to the XVSM Technical Board. Feedback is incorporated into the design and implementation decisions that are presented in this thesis. This thesis provides an introduction into the problem domain and space-based computing in general. It is approachable both for developers with already deep knowledge in space-based computing that are planning on creating an extension, as well as audiences that seek an introduction to XVSM. Thus it follows the *Communication of Research* guideline. The guideline *Design as an Artifact* is followed by not only designing the modular architecture but also aiming to provide a reference implementation. It should provide basic XVSM functionality, to allow its usage for developing custom applications. In order to reason about the new XVSM implementation, the *Design Evaluation* guideline is followed by deriving a descriptive evaluation concerning extensibility scenarios for the new architecture. We focus on the feasibility of the developed extension mechanism to implement the extensions. In addition, performance analysis is conducted to show the feasibility of the developed XVSM implementation.

1.2 Structure of Thesis

The following structure is used for this thesis:

Chapter 2 gives an introduction to different extensibility mechanisms and categorizes their properties into adaption types. The adaption types are used to compare XVSM and other space-based implementations.

Chapter 3 describes concepts and technologies used for the development of the XVSM implementation.

In Chapter 4 the results of the XVSM implementation's requirement analysis are shown and elaborated. The requirements are structured into functional and non-functional ones.

The Chapter 5 introduces the created plugin framework that lays the foundation of the new extension mechanisms for the XVSM framework and explains its concept and implementation.

Chapter 6 describes the proposed XVSM implementation and how the concepts from the previous Chapter 5 were adopted.

Chapter 7 compares the new XVSM implementation with existing implementations. Extensibility as well as performance characteristics are compared. Finally, implementation suggestions for extensions that are available for other XVSM implementations are provided.

The final Chapter 8 summarizes the results of the thesis and provides suggestions towards future research and development.

CHAPTER 2

State of the Art & Related Work

Since the tuple space architecture was introduced, several middlewares with different feature sets have been created over time. In one way or the other, extensibility has been an essential part of several of these middlewares, especially XVSM-based ones. However, the understanding of an extensible middleware can differ. For instance, some middleware implementation is considered extensible if it has a modular architecture that allows late binding of modules at startup, while for other middlewares, a contract-based implementation is seen as extensible. Thus, we are now taking a deeper look at existing space-based middlewares and related technologies, and analyze how they have incorporated extensibility and modularity concepts.

2.1 Concepts and Classification

Before we analyze the different middlewares and technologies, we give a short overview on typical programming models and mechanisms that can be used to adapt software. We also present the classification that is used to compare the space-based middlewares regarding their extensibility properties.

2.1.1 Dependency Injection

Dependency injection (DI) [Pra09] is an inversion of control pattern [1] that is used to separate the usage and the construction, assembling, and wiring of a component. The implementation of the component is hidden behind a contract. Typically an inversion of control container instantiates the component and wires it to its dependent object.

Figure 2.1 shows a simple DI example. In this example the `CarRental` class depends on instances of the `ICar` interface. However, it has no direct dependencies to the concrete `ICar` implementations. The `Injector` applies the inversion of control pattern by instantiating the concrete classes and wiring them in `CarRental`.

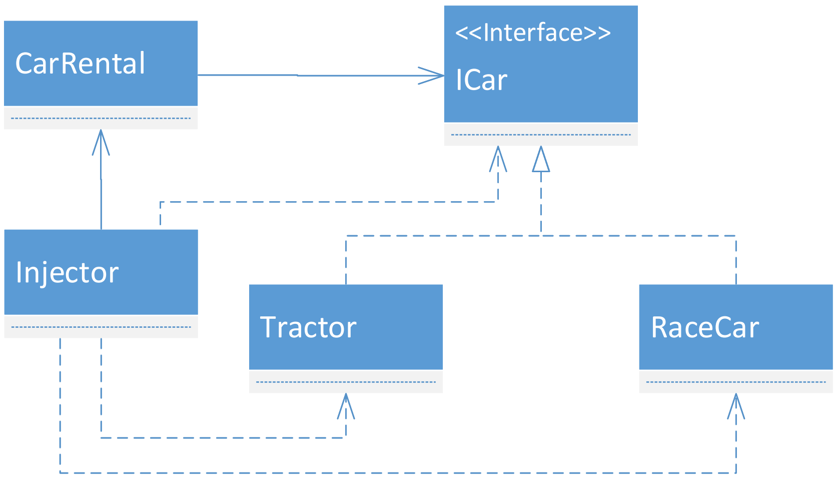


Figure 2.1: Example of a Dependency Injection.

This separation of using and instantiating a component provides multiple advantages. It simplifies testing, since components that are not relevant for testing can be easily replaced by mocks. Development might be guided to a modular and loosely coupled architecture. In regard to extensibility, DI is useful since it allows to easily change component implementations. Depending on the used DI framework, swapping an implementation can be a matter of changing a configuration file.

2.1.2 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [Kic+97] is a programming model that can be used to extend or change the behavior of a software application in a cross-cutting manner. Features that require an integration into multiple, scattered parts of an application are typically dealt by introducing an additional layer of abstraction in object-oriented languages. However, adding too many abstraction layers can lead to a probably unnecessarily complex architecture. It might also require to refactor existing code parts to introduce new abstraction layers. AOP, on the hand, allows to integrate individual concerns into different parts of the application by defining *join points* that specify the concrete places where the aspect-specific code may modify the application [EFB01].

In regard to extensibility, aspects can be useful as they provide a flexible framework to change or extend the behavior of an application. Depending on the used AOP implementation, this can be a powerful extensibility mechanism. For instance, if an AOP tool is used that dynamically weaves the aspects at compilation time, this practically allows to integrate custom aspects into almost every possible place in the application. Unfortunately such a mechanism has the disadvantage that it increases the coupling of the aspect and the rest of the application, since small changes of the core application could break the aspect. Thus, using AOP alone is not enough to solve the extensibility issues.

2.1.3 Adaption Type Classification

Comparison of extensibility is not trivial, since it is a general property that can have various meanings in different scenarios. In our case, we are more interested at which stage in the development cycle an application has to be adjusted to allow extension or change of existing behavior. For this, we use adaption types presented in [SM03] and [McK+04] for classification.

The adaption types of middleware implementations are divided into *static* and *dynamic* composition mechanisms. Software that needs to be rebuild to change its functionality uses either *hardwired* or *customizable* composition mechanisms. Both adaption types are *static* but the *customizable* adaption type still provides more flexibility, as it uses adaption mechanisms at compile or linking time without the demand to change the source code. Although *configurable* composition mechanisms are still considered *static*, they provide a higher degree on dynamism, as they might use algorithms and environment information to select the component to be loaded at startup. While *static* composition encompasses *hardwired*, *customizable* and *configurable* composition mechanisms, *dynamic* composition can be realized through *tunable* and *mutable* composition mechanisms. *Static* adaption types have in common that once an implementation has been selected, it cannot be changed. *Dynamic* adaption types, on the other hand, provide mechanisms to act and react according to changing environment information. *Tunable* and *mutable* composition both describe such dynamic software with the ability to change and react at runtime without halting. They differentiate only in the degree of dynamism. *Mutable* software, in contrast to *tunable*, is even able to change its functionality and purpose into something completely different.

2.2 Technologies

In this section, we are analyzing frameworks and technologies with a focus on plugin-based extensibility mechanisms. The frameworks support an application platform that allows extensions in a plugin-based manner. Even though the frameworks do not address space-based middlewares, their extension mechanisms still influenced the architecture and design decisions of the new XVSM framework.

2.2.1 OSGI

OSGI [OSG03] is a specification for Java frameworks that enables dynamic loading and interaction of plugins, called *bundles*. With OSGI, it is possible to dynamically bind *bundles* by specifying the provided and required services in order to manage and automatically resolve the dependencies. As OSGI provides mechanisms to dynamically react on changes, it is possible for *bundles* to be added or removed at runtime and allow the application to adapt dynamically. Using OSGI *bundles* forces the developer to create a modular architecture and therefore assists a decoupled application design.

2.2.2 Joram

Message-oriented middleware (MOM) was created to reduce complexity and to decouple communication in distributed systems. To further reduce the coupling to individual MOM implementations, the Java community has established a specification for MOMs called *Java Message Service* (JMS) [MC00]. JMS implementations provide a similar functionality as XVSM middleware although the focus of XVSM lies more on coordination, as XVSM provides a richer set of coordination mechanisms [MK11]. Joram [2] is a message-oriented middleware (MOM) that is programmed in Java and uses the OSGI-based plugin framework Apache Felix [3] to extend and change functionality at runtime. Joram provides interceptor extension points which can be used to filter incoming messages or change incoming and outgoing messages. The concrete interceptor implementation is located in a separate *bundle* and will be called through OSGI at runtime. Another extensibility mechanisms Joram provides are extensions for *acquisition* and *distribution* of messages to extend its compatibility with other systems. While *acquisition* extensions are used to create messages from external sources, *distribution* extensions transform and send messages to external sources. This extensibility mechanism can be used, for example, to convert emails to messages and messages to web service requests, but it can also be used to provide interoperability between incompatible platforms. This extensibility mechanism can be used to provide interoperability between incompatible platforms. Although the core of Joram is a single *bundle*, it consists of many separate and decoupled services that could be individually replaced by future bundles, which makes Joram a *tunable* software.

2.2.3 DataNucleus Access Platform

DataNucleus Access Platform [4] is a plugin-based persistence layer that is programmed in Java and uses multiple plugin mechanisms. If DataNucleus Access Platform is running on an OSGI container, the OSGI plugin mechanism is used, but it can also run in an embedded mode and use its own implementation. The core functionality of the persistence layer, although located in one single plugin, is internally implemented in separate plugins, which also serve as extension points to additional external plugins. The binding of plugins to extension points is realized through an XML protocol located as a file in the plugin root directory. Because of its extensive use of plugins, DataNucleus Access Platform is a *tunable* software when it runs on an OSGI container. If it is running in embedded mode, it is only a *customizable* software, since it is not supporting dynamic adding of plugins at runtime.

2.3 XVSM Overview & Implementations

XVSM was developed as the reference architecture of the space-based computing platform that acts as an abstraction of distributed systems with the advantage of supporting multiple architectural styles [MKS10].

2.3.1 Concept

The basic functionality of XVSM is to provide facilities that allow storing and retrieving of data in a shared memory, which is called *space*. Data is stored in the form of so-called *entries*, which act as a wrapper for the underlying user data and space-specific metadata. Entries are stored in *containers*, which are organizational structures within a space and have their own coordination mechanism [Cra10]. The space provides methods to create, remove, or lock containers. A main feature of XVSM is the support of multiple coordinators that can be used to retrieve entries in different and flexible manners. When a container is created, coordinators are associated with it, in order to prepare structures to allow coordinated access for entries to the container. The coordinators are used to select entries of the correlating container in a coordinator-specific way. With the *FifoCoordinator* entries may be selected in the same order as they were initially stored in the container. For instance, when entries “A”, “B”, and “C” are sequentially inserted into the container, a retrieval through the FifoCoordinator yields the entries in the same order “A”, “B”, and “C”. The same container may also contain additional coordinators, for instance a *KeyCoordinator* which allows to select the same entries through their individually unique identifiers. These identifiers must be included as metadata in the entry objects when entries are written to the space and are called *coordination data*. Coordination data is not only used to provide coordinator-relevant metadata for entry objects, but also to associate entries with coordinators. In this example, the entries are associated with both coordinators in order to make them accessible through both of them. When coordinators are bound to a container, they can be either marked as obligatory or optional. While the former ensures that all entries must be associated with the coordinator, the latter only makes it possible for entries to be associated with the coordinator. Entries that are not associated with a specific coordinator may not be retrieved through it. However, the association can be made automatically by *implicit* coordinators. Hence, for a FifoCoordinator, no additional metadata is required to create the coordination data of an entry.

The basic entry operations that XVSM provides are *write*, *read*, and *take*. The write operation is used to store entries in containers and, as already mentioned, make them accessible through coordinators. Containers may be bound to a maximal number of entries. This could lead to blocking or failing of the write operation depending on the current number of entries in the container and provided transaction parameters. The read and take operations select entries in a single container by making use of provided *selectors*. Selectors contain coordinator-specific data that are used as parameters for the selection operation of the coordinator. KeyCoordinators, for instance, require keys to select entries, which must be provided by KeyCoordinator-specific selectors. In addition to coordinator-specific data, selectors contain the target entry count of the selection as well. The selection operation might block, similar to the write operation, or fail if the requested number of entries is not available for the selection. The concrete behavior in such a situation depends on the provided timeout and transaction parameter. XVSM supports concrete timeouts in milliseconds, but also `INFINITE`, `TRY-ONCE`, and `ZERO` timeouts. With the `INFINITE` timeout, requests are rescheduled without expiration. `TRY-ONCE`, which is also the default timeout of XVSM requests, allows the rescheduling of requests if the required resources are currently locked. If for instance the entry count of a read operation with a `TRY-ONCE` timeout is greater than the amount of entries stored in the container, the operation is not rescheduled because the entries are not locked. Finally, the `ZERO` timeout prevents the rescheduling of requests additionally for locked resources. If an entry is locked through a take operation in a running transaction, a further take operation through a different transaction will not be rescheduled and fails immediately.

The selection operations allow the composition of multiple coordinators in a pipe-and-filter style [Mon+97]. When multiple selectors are specified, the space consecutively executes the selection and passes the results on to the next coordinator. This process is repeated for all specified selectors. Figure 2.2 shows an example with two coordinators filtering the entries in a series. The FifoCoordinator's selector only selects the first three stored entries and the KeyCoordinator checks if one of the three entries is associated with the provided key. In summary, this composition makes it possible to select an entry by its key if and only if it is among the first three stored entries remaining in the container.

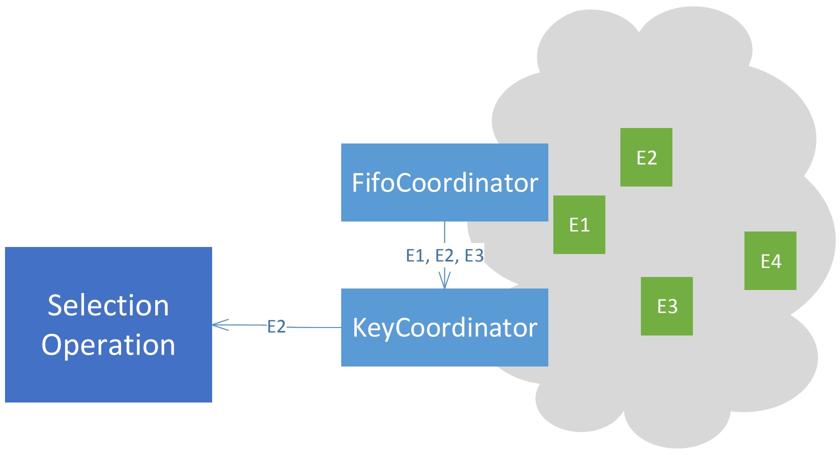


Figure 2.2: An exemplary Key- and FifoCoordinator visualization with a pipe-and-filter style.

The space provides even more entry operations such as *delete* or *test*. However, these two operations are based on read and take operations. They are optimizations that use a different return type. Read and take both return the stored entry objects, while test and delete simply return an integer representing the number of affected entries.

To ensure consistency, the operations are executed with a transactional context. XVSM supports a two-layered transactional model with a pessimistic locking implementation. The outer layer is used for user-controlled, possibly long-running transactions, whose lifecycles are manually managed. The inner layer is used for the internal operation execution, which allows to reschedule or rollback failed operations. With the introduction of transactions, XVSM may handle concurrent operation execution in a consistent matter. For instance, in a scenario where multiple workers attempt to take the same entry from the same container, transactions can ensure that an entry will be only retrieved once. If a worker crashes midst execution, the previous state of the space can be recovered and the entry may be available for other workers again.

2.3.2 Architecture

The formal model defines a multi-layered reference architecture for XVSM. The layers are called CAPI, which stands for "Core Application Programming Interface" and are numbered from one to four. XVSM is designed such that the lower level CAPI layers provide their functionality to the upper layers. Thus, with the increasing layer number the abstraction and functionality increases as well.

CAPI-1 is focused on the basic storage of entries in the space. It only provides methods to add, remove, or retrieve entries from the space. All operations are non-blocking and are not transactionally aware. Transactions are introduced in the CAPI-2 layer by augmenting the entry operations with transactions and providing separate methods to control the transactions. For instance, lifecycle operations of transactions are introduced with this CAPI layer to create, commit, or rollback transactions. Through the transaction support the space implementation gains more functionality, however the space-based coordination mechanisms are missing, since they are introduced in the CAPI-3 layer. The flexible coordination mechanism, one of the major features of XVSM, is added with the CAPI-3 layer to the XVSM architecture. Hence, it is possible to retrieve entries in a coordinated manner. Below CAPI-3 only a single, given container is supported. With CAPI-3, the container management is introduced. It is possible to create, remove, lock, and lookup containers. The final CAPI-4 layer adds the runtime and reschedule support to XVSM. The lower CAPI layers are non-blocking and return with the following four status values: `OK`, `NOTOK`, `LOCKED`, and `DELAYABLE`. The first two values `OK` and `NOTOK` lead to a direct response to the user code, since the result of the dispatched operation is either successful or unsuccessful. However, if `LOCKED` or `DELAYABLE` values are returned, the operation may be rescheduled depending on the provided operation timeout.

2.3.3 Extensibility

As its name already implies, XVSM was designed with a focus on extensibility in mind. The following three design decisions are frequently listed in the context of XVSM's extensibility possibilities [Dön11; Bar10]: Custom coordinators, the modular CAPI architecture and aspects.

With custom coordinator support, domain-specific coordinators can be used in the space next to built-in coordinators without functional limitations. The space interface was designed to seamlessly support new coordinators. However, it should be noted that the implementation complexity depends on the concrete XVSM middleware and may require deep knowledge of the middleware's implementation.

Aspects are an AOP feature that is used in XVSM to modify the behavior of the space in a crosscutting manner [Küh+09]. This allows the modification of particular parts in the middleware without exchanging whole CAPI layers. XVSM provides fixed *interception points* that correspond to join points from AOP and represent the events that can be augmented by aspects. Aspects may intercept an operation before or after its execution. Interception before execution allows the aspect to access and modify the arguments used for the operations. A custom interface allows aspects to even issue own XVSM operations. Hence, it is possible for an aspect to write the operation's arguments in a separate container, for a possible future operation analysis. When operations are intercepted after execution, their result may be modified.

XVSM provides two different types of aspects: Global and local aspects. Global aspects can be seen as space aspects that use interception points that affect the space by making no distinction between concrete containers. Local aspects, in contrast, are bound to a specific container. They only intercept operations if they are executed on the corresponding container.

Since aspects play a major role for the extensibility features of XVSM.NET as well, the term "aspect" could be confused. In order to distinguish the aspects from other aspect-oriented features described in this thesis, we use the term *XVSM aspects* in ambiguous cases.

2.3.4 MozartSpaces

MozartSpaces in its second iteration [Bar10; Dön11] is the reference implementation of XVSM for Java and the most feature complete XVSM implementation up to date. In this thesis, we will refer to the second version by MozartSpaces if not explicitly denoted otherwise. Without getting too technical, we will analyze the architecture on a high level in regard to extensibility.

The MozartSpaces implementation applies the fundamentals of the formal model [Cra10] and uses many of its concepts. It provides Java interfaces and classes that correspond to the CAPI-4 and CAPI-3 layers of the formal model. Aspects are supported by providing several interception points. MozartSpaces comes with a default implementation for a notification mechanism based on aspects. It uses aspects to write notifications to a separate notification container after space operations. This container is then used to notify registered observers.

CAPI-1 and CAPI-2 are integrated into the CAPI-3 layer of the native Java implementation. This was a decision based on performance considerations. The hypothesis was made that since the first three CAPI layers are strongly coupled, significant semantical behavior may be established between CAPI-3 and CAPI-4 [Bar10]. Hence, the layers CAPI-1 and CAPI-2, defined in the formal model, were not separately implemented. However, `Isolation`, `Operation`, and `Coordination` modules were created to modularize CAPI-3 instead. The modules used to be instantiated by the dependency injection framework GUICE [5], but in a later revision this was replaced by native Java instantiations.

Coordinators in MOZARTSPACES have both a `Coordinator` and a `Selector` implementation that are required to store and retrieve entries. The `Container` implementation from the `Operation` module acquires entry locks and delegates entry operations to the coordinators. It links relevant coordinator selectors together for selection operations and bootstraps the selection by retrieving the entries through the last selector. The formal model defined the coordinators as a filter mechanism where every selector represents individual filtering stages. The filtering always starts with all entries and on every stage a subset of the previous stage will be selected and may be reordered. [Bar10] compared this stage-based filtering mechanism in terms of efficiency to a new streaming-based approach. This alternative approach may significantly reduce the number of checked entries in some situations. The original approach used a top-down evaluation starting with all entries in the container and evaluating them stage by stage, comparable to a pipe-and-filter [Mon+97] architecture. The stream-based approach is rather a bottom-up evaluation with two selection methods available in Selector implementations: `getAll` and `getNext`. The first method is used to retrieve all entries in regard to the selection and with respect to the maximum count. The latter method must also respect the selection and the maximum count but continuously returns entries when it is called multiple times. The stream-based approach was implemented in MOZARTSPACES. As the selectors are responsible to handle the special case if they have no so-called *predecessor* selector to fetch entries from and must also ensure the count constraint is not violated, the implementation of the two selector methods might not be trivial. Unfortunately, this also leads to code duplication among the selectors. For instance, when comparing[1] the methods containing the selection logic of the label and any coordinator selectors implementation, only `16` out of `72` lines were unique for the label coordinator and `4` lines out of `60` were unique for the any coordinator. Custom coordinators, which are supported by MOZARTSPACES, need to incorporate an equivalent implementation in order to function correctly.

For the persistency integration in MOZARTSPACES different approaches were discussed in [Zar12]. An aspect-oriented approach that would allow a loosely coupled integration was dismissed since required interception points were missing and the integration with the commit-phase of the transactional model proved to be error-prone. Consequently, an alternative approach with a tighter integration was followed instead. Figure 2.3 shows the dependencies to the persistence module. Individual coordinators had to be adjusted for the persistence integration. Hence, when a new coordinator is created, it needs to correctly interact with the components from the persistence module in order to function properly. Although the persistence integration is hardwired, the used persistence provider is configurable.

[1]For this comparison, the `getAll` and `getNext` methods were extracted from the selectors of the `DefaultLabelCoordinator` and `DefaultAnyCoordinator` implementations. To detect changes, the POSIX tool DIFF was used.

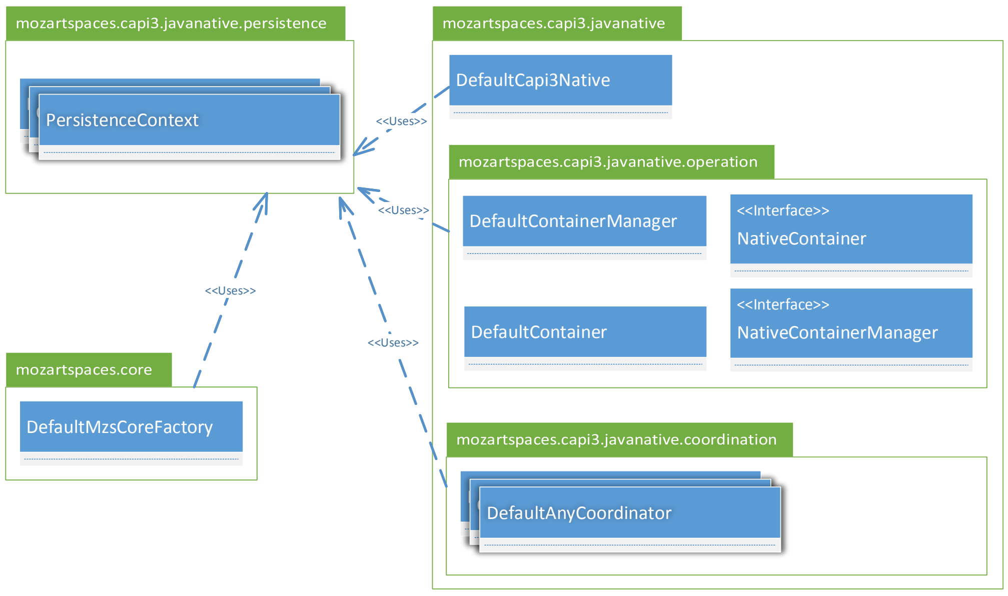


Figure 2.3: Overview of persistence dependencies in MozartSpaces.

In the course of the development of MozartSpaces further extensions have been integrated, such as access control support [CK12], a replication mechanism [Hir12], and a life cycle management [Wat15]. The replication extension has been implemented through a facade approach that introduces an additional abstraction layer. Due to an architecture decision, a possible aspect-based implementation was rejected in favor of the facade approach. The access control and lifecycle management extensions, on the other hand, had to be tightly integrated into the core implementation.

2.3.5 XCOSpaces

XCOSpaces [Sch08; Kar09] is a .NET implementation of XVSM. It supports aspects for predefined extension points to extend its functionality. Similar to MozartSpaces it supports the execution of custom code before or after space or container operations. It distinguishes between space and container aspects and supports multiple aspects per interception point. Aspects represent the main extensibility mechanism of XCOSpaces. An extension supporting notifications was solely integrated with aspects.

XCOSpaces uses a *microkernel* that supports contract first design with dynamic binding. It has a component-oriented architecture that splits the implementation from its contracts. This allows to make use of a component type by importing its contract without having knowledge of the concrete implementation. It is even possible to dynamically load the implementation based on the provided configuration. Thus, the implementation for a specific contract can be replaced without recompiling the application. XCOSpaces uses a custom mechanism to load and bind implementation classes to contracts, it uses no pre-existing dependency injection framework. It is possible to change the implementation for the following contracts extending the `IXCOService` interface:

- `IXCOCommunicationService`: Contract that is used as an abstraction for the communication between clients and the space. XCOSpaces supports a low-level TCP and a windows communication foundation implementation.
- `IThreadDispatcher`: This contract is used for the thread dispatcher service that is responsible to manage and execute tasks concurrently.
- `ISerializationHelper`: The serialization helper service contract is used for marshalling and unmarshalling of communication messages.
- `ILoggerFactory`: The logger factory service contract provides an abstraction to the used logging factory.

The previously listed service contracts are not the only used contracts in XCOSpaces, but the only service contracts. For instance, the transaction contract `ITransaction` is implemented by a single class `Transaction` and is instantiated directly. Since XCOSpaces was implemented before the formal XVSM model was developed, it uses a slightly different architectural approach than the CAPI layers. Hence, there are no explicit service contracts for the separate CAPI layers.

The coordinators in XCOSpaces use a stage-based filtering approach. When entries are retrieved from the space, the container iterates over the selectors and continuously invokes the filtering through the corresponding coordinator. Similar to MozartSpaces, the coordinator implementation is responsible to enforce the maximum count. Because of its stage-based filtering approach, it uses a single method to filter entries instead. After comparing the read method of the `LindaCoordinator` and `LabelCoordinator`, `27` out of `86` lines were unique to the *LabelCoordinator* and `19` out of `78` lines to the *LindaCoordinator*. In comparison with MozartSpaces, the stage-based approach appeared simpler, since it only needs a single method to filter entries, however, the amount of code duplication of the selection method is comparable to the MozartSpaces implementations. Other extensions to coordinators, such as the persistency module in MozartSpaces, are not available to XCOSpaces at the moment, hence spaces may only utilize in-memory storage.

2.3.6 TinySpaces

TinySpaces is an implementation of XVSM for the .NET Micro Framework that focuses on embedded devices and limited resource usage [Mar10]. It uses a contract first approach similar to XCOSpaces but uses CAPI layers from the XVSM formal model. Components that make up the middleware are interacting through their interface and not directly. There are contracts for transactions, coordinators, and the runtime, for instance. Instead of providing a dynamic binding of the implementation after compilation, the contract implementations are manually, statically bound through *facades*. This decision was made due to performance considerations. When TinySpaces is instantiated, the contract implementations may be passed to the space's main class. The facades included in TinySpaces are nested together in order to be supplied as configuration to the space. Figure 2.4 shows the structure of the facades for the current TinySpaces implementation. The implementation of the `IRuntimeFacade` is the outer-most facade that encapsulates all other facades used to start the space.

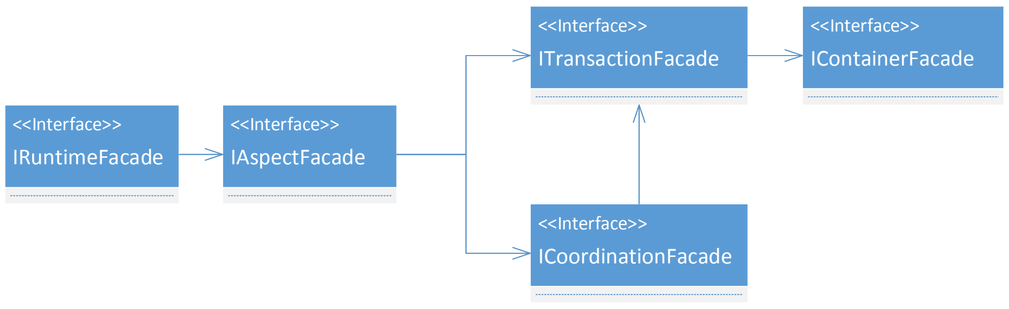


Figure 2.4: Overview of facade nesting in TinySpaces.

Coordinators in TinySpaces use a similar stage-based filtering approach as XCOSpaces. It includes implementations for `FifoCoordinator` and `KeyCoordinator`. Both coordinators include logic to handle count constraints from selections. Due to performance reasons, the duplication check in the `KeyCoordinator` does not incorporate transactional visibility.

2.4 Apache River

River is the current implementation of the Jini [Wal99] architecture by the Apache Software Foundation. Jini provides conventions that aim to help create a distributed system with a service-oriented architecture that is dynamic and may easily change over time. When a client requests access to a Jini provided service, it first needs to find the lookup service. Depending on the infrastructure this may be either accomplished by sending a broadcast on the network for dynamic detection, or in case the host running the lookup service is already known by direct connection. The lookup service specified in Jini may be used to gain access to arbitrary registered services. Services can be queried by a pattern matching mechanism equal to the pattern matching mechanism used in JavaSpaces [Arn99; FAH99], which is similar to tuple matching in the Linda Tuple Space. Hence, by providing a service template that may contain a unique service identifier, service types, and attribute sets, services are filtered and matched. Services are matched with the template if their unique identifier matches, they are compatible with all provided service types, and all attribute set entries can be matched with attributes defined in the service. Omitting parts of the template, such as the unique service identifier, as an example, marks the identifier as a wild card and therefore matches every service's identifier. The attribute set can be used to distinguish between equivalent or very similar services. For instance, if you want to make use of a Clock-Service for a specific time zone, you could specify the time zone in the attribute set. When a service is successfully queried and requested, a proxy object implementing the service interfaces is downloaded and may then be used to consume the service's offered functionality. The proxy object may communicate with the service by using an arbitrary protocol. This process is transparent to the client since it only interacts with the interfaces of the service proxy [New07].

The River implementation is designed around its services since Jini is a platform for service-oriented computing [New07]. It includes a tuple space implementation, called outrigger, which follows the JavaSpaces specification. Outrigger makes use of a service for transaction management. Instead of simply including the required code by itself, it uses the service lookup to gain access to a provided transaction manager. With this it is possible to replace the service for the transaction management without the need to update and redistribute the outrigger JavaSpaces implementation. Since services are leased for a specific time, it is even possible to exchange a service dynamically, for instance, if a lease is expired and a new service request provides a different service.

2.5 GigaSpaces XAP

XAP (eXtreme Application Platform) is, inter alia, a space-based middleware by GigaSpaces Technologies Ltd [6]. Previously only commercial, GigaSpaces offers now an open source edition of XAP that was used in the following to examine its core architecture with regard to extensibility and flexibility in version 12.2. Furthermore, we will focus in this thesis on the space-based middleware aspects of XAP and leave out other features of XAP that are focused on an application platform. XAP is a Java application that uses the Spring framework [7] for dependency injection. Hence, it is possible to predetermine the behavior of the middleware by configuring the used beans. In fact XAP's documentation [8] explains how to configure different fundamental parts of the middleware by specifying different bean implementations in configuration files.

Even though the coordination mechanisms of XAP are not as flexible as those of XVSM frameworks, it provides different types. It is possible to access entries if their unique identifiers are known through Id Queries. Template matching is also supported in a similar fashion to River. The most flexible coordination mechanism of XAP provides an SQL-like selection language. It allows to specify constraints for the entry properties. For instance, it is possible to select entries by comparing numeric values of their properties. It is even possible to use SQL-like `GROUP BY` clauses and apply aggregation functions such as `SUM` or `AVG`. However, custom composable coordinators, such as XVSM provides, are not supported.

XAP supports multiple transaction managers that use Spring's abstraction `Platform-TransactionManager`. The transaction manager may be chosen and configured by using Spring's configuration system. XAP supports declarative and programmatic usage of transactions through Spring's transaction management. This allows the usage of optimistic as well as pessimistic transaction management in XAP. However, to use optimistic transaction management, in addition to configure the space, the entry types need to be extended to include a version ID. The version ID is required to detect conflicting changes from different transactions. Hence, changing from optimistic to pessimistic or vice-versa is more work than simply updating the configuration, as the entry model needs to be updated too.

Persistency in XAP is realized through two components, one that is responsible for initial loading of data and the other for persisting changes in the space. XAP comes with multiple implementations for the components to support the Hibernate ORM [9] or NoSQL databases such as Cassandra [10], or Mongo [11] out of the box. To allow custom persistency endpoints, it is intended in XAP to extend the provided classes `SpaceDataSource` and `SpaceSynchronizationEndpoint` to override their methods. The subtypes supporting the custom endpoint can then be configured using Spring's configuration system.

XAP supports interceptors called *Space Filters* that allow the execution of custom logic before or after space operations. Space Filters are configured via Spring's configuration system and may therefore be used only for embedded spaces. When multiple Space Filters are configured, it is possible to specify the order in which they are executed with an integer defining the priority. It is possible to intercept operations such as `read`, `update`, or `write`, among others[2]. When a Space Filter is used to intercept before a `write` operation, it may modify the entry before it gets written to the space. The manipulation of entries before they are written or after they are retrieved from the space, based on business logic, gives more flexibility to the developer. XAP even allows to access its space interface from a Space Filter which makes it possible, for instance, to write additional entries for every `write` operation. Space Filters provide similar applicability as XVSM's aspects, even though they cannot be dynamically registered as aspects in MozartSpaces.

2.6 Comparison

In Table 2.1 we make a comparison of the modularity and extensibility properties of the space-based middlewares that we previously analyzed: XAP, River, MozartSpaces, XCOSpaces, and TinySpaces. The classification of extensibility is not a simple "yes" or "no" question due to numerous degrees of extensibility. Thus, we use the adaption types from Section 2.1.3 for classification. The static adaption types *hardwired*, *customizable*, and *configurable* are abbreviated with S1, S2, and S3, while the dynamic adaption types *tunable* and *mutable* are D1 and D2. The following modules and properties were used for the classification:

- **Storage**: The storage layer responsible to retain written entries. An extensible layer, for instance, is changeable from an in-memory to a persistent storage.

[2]The full list of possible interception points can be found in the official documentation [8].

- **Transaction**: An extensible transaction layer supports the change of different transaction implementations, such as distributed transaction, optimistic or pessimistic transactions.
- **Coordination**: The coordination layer is responsible to retrieve entries with different coordination mechanisms, such as pattern matching or queues.
- **Runtime**: The runtime is responsible to handle execution and rescheduling of dispatched operations. It is also responsible for transmitting operations over the network and the serialization of entries.
- **Space Interception**: This property is no specific layer, but describes whether it is possible to modify space operations to write or retrieve entries.

	XAP	**River**	**MozartSpaces**	**XCOSpaces**	**TinySpaces**
Storage	S3	D1	S3	S1	S2
Transaction	S3	D1	S1	S1	S2
Coordination	S1	S1	S3	S2	S2
Runtime	S1	S1	S2	S3	S2
Interception	S3	N/A	D2	D2	D2

Table 2.1: Comparison of space-based middleware regarding extensibility properties.

Requirements for the extensibility property are not identical for all frameworks and modules. For some frameworks, it might be sufficient to provide extensions solely through hardwired integrations, since the development could be streamlined and the application could provide a single version with all extensions. For those, S1 could suffice, though, this is not always the case. Extensions exist for XVSM-based frameworks that are optional or experimental and might not be sensible to include in the main framework version. For such cases, the S3 adaption type appears to be the sweet-spot since it allows to exchange a module but does not require recompilation and access to the code.

Extensibility for the storage layer should be flexible since the requirements to persist entries might differ among usage scenarios. Providing separate framework versions with different storage layers might be cumbersome and requires maintenance. Hence, in this case S3 or better appears to be a sensible choice. River is the most flexible framework for this module, however, XAP and MozartSpaces also provide flexible persistence provider integration.

An extensible transaction layer might make sense for situations where transaction implementations with different performance characteristics could be used. For instance, in some scenarios an optimistic transaction handling could improve the system's performance. In another scenario it could make sense to integrate the transaction system with an existing one, such as a distributed transaction system, for instance. Consequently, S3 provides sufficient flexibility. River is again the most flexible in this case, but XAP also provides a flexible enough transaction system.

Custom and flexible coordinator support is a major advantage of XVSM-based frameworks. In this case, MozartSpaces is the framework with the most flexible extensibility by providing S3. When XVSM is used and a custom coordinator with optimizations for the domain logic is created, S3 should be the minimum adaption type, as otherwise, the framework must be recompiled to integrate a custom coordinator.

The only framework with a configurable runtime layer is XCOSpaces. It could make sense in some situations to have such a flexible runtime layer that could be adjusted for the use case. For instance it could make use of a different rescheduling algorithm depending on the concrete scenario. However, since almost all of the analyzed frameworks do not have such a high extensibility level for this module, a rather lower extensibility level of S2 or even S1 should suffice.

The interception support through aspects is another feature XVSM-based frameworks. All analyzed XVSM frameworks provide the maximum flexibility by supporting D2. However, for many scenarios S3 could suffice as well.

XVSM.net targets at least the S3 property for all the mentioned modules. However, it should also provide more fine-grained modules with the same extensibility property as well to support easier integration of extensions.

CHAPTER 3

Background

After comparing the existing implementations, we describe in this chapter the concepts and technologies that were used for the development of the new XVSM implementation. We also explore the reasons for using the mentioned technologies.

3.1 .NET Platform

The .NET Framework is a software development platform from Microsoft. It uses a virtual machine called *Common Language Runtime* (CLR) to execute code. The .NET platform supports multiple programming languages that may even follow different paradigms such as object-oriented or functional programming. These languages tend to be high level and are required to be compiled into an intermediate form, which is interpreted and just-in-time (JIT) compiled by the CLR. The most popular programming language on the .NET platform according to the TIOBE Index [12] is C#. It is a mainly object-oriented programming language with several functional programming features, such as lambda expressions and higher-order functions.

C# and the .NET Platform were chosen for the new XVSM implementation for several reasons. The existing XVSM implementations for the .NET platform do not support the extensions that were introduced in the current version of MozartSpaces. However, the .NET platform remains relevant. Since the .NET platform supports a variety of programming languages, a .NET implementation of XVSM is desired as it allows a better integration in other .NET applications. C# was chosen due to its popularity and flexible programming model.

For the development of XVSM, the standard IDE for C# on Windows VisualStudio with Resharper [13] was used. Resharper extends VisualStudio with many useful refactoring and static code analysis tools that helped to speed-up the development. For the development with Resharper an academic license was provided by its developer.

For testing the popular unit testing framework NUnit [14] was used along with the mocking framework NSubstitute [15]. Both are open-source and simplify the test development. NLog [16] is a modern and fast logging framework that is frequently used on the .NET platform and therefore was also used for this project.

3.2 Peer Model

The *Peer Model* is an abstract programming model, introduced in [Küh+13], with a focus on coordination patterns for concurrency and distributed systems. It separates the coordination from the application logic and uses a component-based approach to improve scalability of an evolutionary application. Space containers from space-based computing are used to define internal stages of the model. The Peer Model is relevant for this thesis since as a programming model for distributed systems, it contains promising coordination and synchronization mechanisms that are used for the runtime implementation of the new XVSM middleware implementation. Thus, we will summarize the concepts relevant for the runtime implementation from [Küh+13] in the following.

3.2.1 Concepts

Peers are the core elements of the Peer Model. They can be seen as uniquely addressable (URI) components that can be used for coordination and service execution. Peers provide two spaces, the *peer-in-container* (PIC) and *peer-out-container* (POC). Since the PIC and POC are spaces, they provide mechanism to store and access entries in a coordinated way. For a typical peer, entries that are sent to the peer are stored in the PIC, and processed entries are eventually stored in the POC.

When entries are stored in the PIC, *wirings* are used to act on them. Wirings are active parts that can trigger the entry movement between the containers and invoke service executions. Wirings contain so-called *guards* that can be seen as pre-condition checks. For instance, a guard could require at least two entries of a specific type "A". Thus, the wiring will only activate if two or more entries of that type are available in the PIC. It should be noted that this wiring will process the maximum amount of entries that satisfy its guard. When all conditions of the guard are satisfied, the entries are taken from the container and the next execution stage is entered. *Service calls* can be registered on the wiring that could execute arbitrary business logic on the entries and may even transform, filter, or add new entries. Finally the *action* of the wiring is executed to transfer entries either to a local or remote container. The disposal of entries is also possible, when no action is provided.

Figure 3.1 shows a peer with a type conversion semantic through its graphical notation. The peer has one specified wiring (W1) with one guard and one action link. The guard takes a single entry of type `X` from the PIC to activate the wiring. The entry is moved to the `ConversionService` in order to be converted to a resulting entry of type `Y`. Finally, the W1's action link moves the resulting entry to the peer's POC.

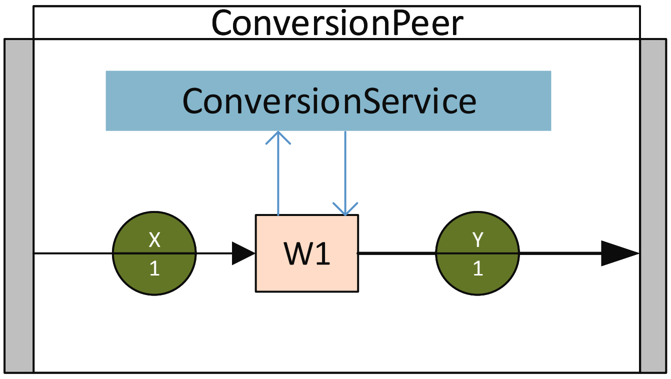


Figure 3.1: An exemplary peer showing a single wiring.

Both guard and action links contain preconditions and may either use read or take operations. In the graphical notation, take operations are depicted as filled circles, and read operations as white circles. However, in this thesis only take operations are used. Peers also may contain *sub-peers*, as an additional abstraction layer that allow to further organize the modeled logic. Actions may move entries to the PIC of a sub-peer, where the next wirings are waiting to process entries.

When entries are written to a peer's container, it is possible to provide *time-to-live* TTL and *time-to-start* TTS properties. The former property marks an expiration date on the entry that automatically removes the entry, and the latter hides the entry from the wirings until the start date is passed. These properties are especially useful for the basic scheduling implementation for the runtime, since some XVSM operations such as read or write support timeouts.

3.2.2 Peer Model Implementation

PeerSpace.NET [Rau14] is an implementation of the Peer Model for the .NET platform. It internally uses the framework XCoordination Application Space [17], short AppSpace, as the foundation for the communication between peers. The AppSpace is based on the space-based computing principles and may be described as an abstraction layer that leverages Microsoft's asynchronous programming model Concurrency and Coordination Runtime (CCR) [18] to distributed systems.

The PeerSpace.NET implementation was developed to provide the Peer Model concepts with an easy to use API. Since it is also based on the .NET platform and provides advanced coordination mechanisms, it is suitable for XVSM.net's preliminary runtime implementation. Due to features such as TTL and the possibility to transparently move entries to remote containers, it simplifies the implementation of basic runtime functionality.

3.3 Extensibility Platform

The focus of the new XVSM implementation lies on extensibility and flexibility, thus the choice of the extension platform is important and has far-reaching consequences, since it would be difficult to switch the underlying extensibility platform. Consequently, we evaluated several major extensibility and dependency injection frameworks. Popular dependency injection frameworks for .NET, such as Castle Windsor [19], Ninject [20], or Spring.net [21] were evaluated at first. Even though their injection mechanisms provide useful features to decouple components for better maintainability, they lack extensibility properties in comparison with the Managed Extensibility Framework (MEF) [TBS10]. Dependency injection frameworks are not designed to provide extension mechanisms for yet unknown functionality [See12]. Thus, this makes them not an ideal choice in our case, since we want to create a foundation that allows future extensions that are even unknown at the present.

MEF takes a different approach than dependency injection frameworks, even though it provides inversion of control mechanisms. MEF was introduced along with version 4 of the .NET Framework. It is an extensibility framework that allows to extend an existing application by loading and wiring of separate .NET assemblies (DLLs) at run time. Its component model [TBS10] is based on the following primitives: `Export`, `ExportDefinition`, `ComposablePart`, `ComposablePartDefinition`, and `ImportDefinition`.

An `ExportDefinition` provides meta information of an offered service. When services are evaluated to be consumed, their `ExportDefinitions` are used for decision making. For instance, if multiple services are available for a required serialization contract type, their `ExportDefinitions` may be used to decide by non-functional means which concrete service should be chosen. An `Export` is instantiated by MEF if the corresponding `ExportDefinition` object is consumed. In MEF, `Exports` wrap the concrete service objects along with the originating `ExportDefinition`. Analog to `ExportDefinitions`, `ImportDefinitions` provide meta information to consume a service. The constraints on `ExportDefinitions` may then be used to decide which offered service should be consumed. `ExportDefinitions` and `ImportDefinitions` may be grouped together as a reusable part in the form of a `ComposablePartDefinition`. When all `ImportDefinitions` of the `ComposablePartDefinition` are satisfied, it may be instantiated as a `ComposablePart`, which consequently may be used to retrieve the exports. Figure 3.2 gives an overview of the component model.

An advantage of this approach used by MEF is that the concrete loading of the code may be deferred until the `Exports` are instantiated, which according to [TBS10] has significant performance advantages in contrast to eager loading.

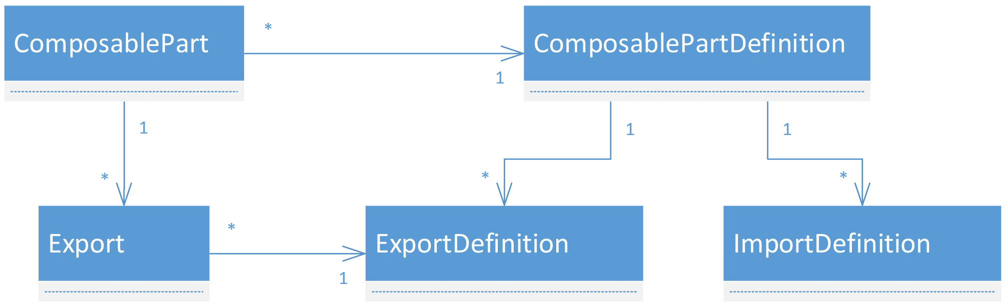


Figure 3.2: Overview of MEF primitives.

3.4 Proxy Framework

For the development of the plugin framework, which is described in Chapter 5, a proxy framework was used. Proxy frameworks may be used as a mechanism for aspect-oriented programming [Kic+97; SP02]. They allow to wrap a regular object with a proxy object to support interception and modification of its method invocations.

The open-source DynamicProxy [22] library was used, since it provides a simple proxy implementation that was easily integrated in the plugin framework. The advantage of the library is that it is able to create the proxy objects while the application is running and does not require a separate compilation step. According to [22], DynamicProxy is used by several other .NET libraries, such as Castle Windsor [19], NSubstitute [15], or previous NHibernate [23] versions.

3.5 Configuration

The .NET framework has an integrated XML-based configuration support, which was the first choice for the configuration system because of the direct support in VisualStudio. Unfortunately, it was not possible to use the integrated configuration capabilities, as they had limitations with the use of config files in library assemblies. The .NET system does not natively support settings in class libraries, which are the obvious choice for plugin projects. Additionally, the handling of multiple configuration files, which can be useful for plugin implementations, proved to be cumbersome with the integrated configuration system. Therefore, it was decided to use a different configuration solution. In the process of researching an adequate configuration or serialization library the YAML [BEI09] file format was found to be an excellent choice. YAML – a recursive acronym for "YAML Ain't Markup Language" – is a data serialization language that was designed with the goals to be human-friendly in regard to reading and writing, but also to work well with modern programming languages and support a variety of advanced features. Because of the lightweight format, the integrated datatype support for lists and dictionaries, and the possibility to use comments, it was decide to use YAML instead of XML or JSON for the configuration file format. YamlDotNet [24], which is a popular and open source

C# serialization library with support for YAML, is used for the deserialization of the configuration files.

CHAPTER 4

Requirements

The previous chapters pointed out the motivation for creating a new space-based middleware. Existing XVSM and other space-based middlewares were compared regarding their extensibility characteristics. In this chapter, we analyze the requirements that serve as the foundation for the new XVSM implementation.

4.1 Functional Requirements

The formal model [Cra10] describes the requirements for an XVSM middleware. In this thesis we build a prototypal XVSM implementation with the functionality focus on the lower CAPI layers 1 to 3.

CAPI layers

The formal model describes four separate CAPI layers. Analogous to MozartSpaces and TinySpaces, although the modularization of the CAPI layers are followed, the concrete method signatures and response types are not implemented. Hence, the *Hypothesis 1* presented in [Bar10] is followed in the XVSM.NET implementation as well. However, the CAPI-3 layer should be explicitly implemented.

Container

The concept of containers are part of the CAPI-1 layer, as described in the formal model. It must be possible to create containers with optional or obligatory coordinators, and of course to destroy them. Containers must provide basic mechanisms to manage the entry storage, such as add, remove, and retrieve entries. Containers may be bound to a maximum size of allowed entries.

Transactions

In the formal model, transactions are the fundamental part of the CAPI-2 layer. The transaction model uses a pessimistic locking mode that eagerly acquires locks on entries to prevent violating access of concurrently executing transactions. The distinction between *user* and *sub-transactions* may be generalized as nested transactions. The *repeatable read* isolation level is the default XVSM isolation level and should therefore be implemented as such in XVSM.NET. In addition, *read committed* and *read uncommitted* should be implemented as well. Hence, XVSM.NET's architectural design should take different isolation levels into account.

Coordinator

Coordinator support is essential for XVSM middlewares. Basic coordinators defined in the formal model should be implemented in XVSM.NET. The System coordinator is omitted, since it is omitted in XCOSpaces and TinySpaces as well, and MozartSpaces provides the Any coordinator as a replacement. As MozartSpaces is the XVSM reference implementation, the Any coordinator will be implemented instead. The Query coordinator is omitted as well, since it would exceed the scope of this thesis.

Runtime

Even though the runtime is not the focus of this thesis, a simple runtime implementation should be provided in order to support basic XVSM functionality. The rescheduling functionality of the runtime need not be as sophisticated as the implementation presented in [Dön11] for MozartSpaces. It is sufficient to ensure failed operations are retried eventually. The runtime should support blocking operations and timeouts. The interface of the runtime should be based on the MozartSpaces implementation to increase familiarity among the different platforms.

4.2 Non-functional Requirements

Some of the non-functional requirements emerged from deficiencies of other XVSM implementations. The biggest focus lies here on the modularity and extensibility properties of the middleware.

Modularity & Extensibility

Modularity and extensibility are important requirements for the XVSM.NET architecture. The middleware should be designed with these requirements in mind to make the architecture ready for future changes. The requirements of the core XVSM functionality should be satisfied by an orchestration of components. Advanced and possible future features should be accomplished by replacing and adding such components. This should be possible without the need to recompile the middleware from the source code, but instead provide the component orchestration before startup or at runtime. Thus, the targeted adaption type is either the static configurable adaption type or one of the dynamic adaption types.

The transaction implementation should be extended to include arbitrarily nested transactions instead of the two-layered transaction system proposed in the formal XVSM model. This should improve the flexibility of XVSM aspects, even though aspect implementations are not the focus of this thesis, since they are closely related to the runtime implementation, which only needs to provide basic XVSM functionality for this thesis. However, other aspect-based extension mechanisms presented in this thesis also gain more flexibility by this approach. An advantage of supporting arbitrarily nested transactions is that extensions can be easier decoupled from implementation details. For instance, an extension may use its own nested transaction which can be safely rollbacked, which is not easily possible if it uses a transaction that is shared among other parts of the application.

Simplified Coordinators

Coordinators should be decoupled from the transaction implementation. It should be possible to create custom coordinators without internal knowledge of the transaction implementation or other framework internals. Redundancy should be avoided and extracted where possible. The goal of this requirement is to make the development of custom coordinators easier and more focused on the coordination logic and less on the integration with the middleware.

Stability

Similar to other XVSM implementation, XVSM.NET should provide a stable implementation that gracefully communicates failed operations. The framework should stay stable when the operation execution leads to conflicts that make rescheduling necessary.

Scalability

The CAPI-3 implementation of XVSM.NET should provide comparable performance characteristics as MozartSpaces but should lay its focus on concurrent execution of space operations. For instance, it should be possible to execute multiple CAPI operations on the same container at once, which is not possible with MozartSpaces at the moment.

Usability & Documentation

The APIs and contracts designed for the framework should be focused on usability and should be well documented on the code level. This is important since this framework should be usable by third party developers that might not have read this thesis. Modern features of the development platform should be incorporated when designing the contracts. However, the runtime CAPI layer should be recognizable for developers familiar with other XVSM implementations.

Testability

Testing is part of the software development process. It is important to provide sufficient tests that guarantee the functionality of the middleware framework. The architectural design should respect this requirement and provide an architecture that supports testing with different abstraction granularity.

CHAPTER 5

Plugin Framework Design & Implementation

In the previous chapter, we have gathered requirements and examined existing technologies. Built around MEF, we propose a plugin framework, internally named *Plugicity*, that will serve as the foundation for XVSM.NET. In this chapter the new framework will be presented with the fundamental concepts that were used and their concrete architectural design. The chapter concludes with a look at the recomposition support of the plugin framework.

5.1 Conceptional Core Elements

The plugin framework incorporates different technologies to provide a modular, flexible, and maintainable programming model. In this section we give an overview of the elemental structures that were chosen for the plugin framework to meet the requirements. Implementation-specific details are kept at minimum to provide an informational overview of the supported features.

Plugins

The primary element of the plugin framework are *plugins* that fulfill so-called *plugin contracts* and can be located in arbitrary assemblies. A plugin may use other plugins through their respective plugin contracts without having dependencies of the concrete implementations. Plugins are never directly instantiated from other plugins, but new instances can be created either by injection or the usage of the service locator. The plugin framework supports dynamic service locators that may utilize metadata, applied to plugins, in order to choose from multiple plugins that implement the same plugin contract. Depending on several factors, which are covered in Section 5.3, it is possible that the plugin framework creates a separate *proxy* object after a plugin's instantiation and deploys it instead of the plugin. A proxy is an object that can be seen as a stand-in replacement for an actual object that wraps its functionality. The usage of proxy objects instead of the desired objects is transparent. Since plugins make use of contracts, proxy objects can be used to wrap method invocations of a plugin without the need to change code in the caller. With a proxy it is possible for every plugin to support *plugin aspects* (see Section 5.7) and *recomposition* (see Section 5.8) functionality.

Plugin Aspects
Plugin aspects are special plugins that are able to intercept and modify calls of plugins or plugin parts to manipulate their behavior, without the need to access and change the original code. Plugin aspects extend the plugin framework with AOP functionality to execute code before or after method invocations of plugins or plugin parts. Further, it is possible to manipulate the arguments of the original method invocation, change the method's result and even handle possible thrown exceptions. The rather powerful mechanism was introduced as a tool for developers to extend or change the behavior of existing plugins.

Plugin Parts
Plugin parts are lightweight sub-components. Due to the performance implications of instantiating regular plugins, which will be discussed in Section 5.3, the support of plugin parts was established as a fast and lightweight alternative that still can be modified and manipulated by plugin aspects. However, plugin parts are provided through plugin part factories, which are a special kind of plugin. Thus, plugin parts are no strict replacement for plugins but rather lightweight plugins with different semantics. Situations that require frequent instantiations benefit from the use of plugin parts.

5.2 Plugins

The basic elements in the plugin framework are called *plugins*, which implement plugin contracts and provide the functionality that the contract guarantees. In this section we are covering implementation-specific details of the plugins, as well as their concepts and how they might be used.

Figure 5.1 provides a UML class diagram of the relevant programming structures that are used for plugins in this plugin framework.

5.2.1 Plugin Contracts

In the plugin framework, a plugin must implement exactly one plugin contract, which is a C# interface that inherits from the `IPlugin` marker interface. Signatures defined in the plugin contracts can be used in other plugins that have no access to the concrete implementation of the plugin contract. Hence, it is important to design the contracts with care, as changes in the contracts must be propagated to all plugins that make use of

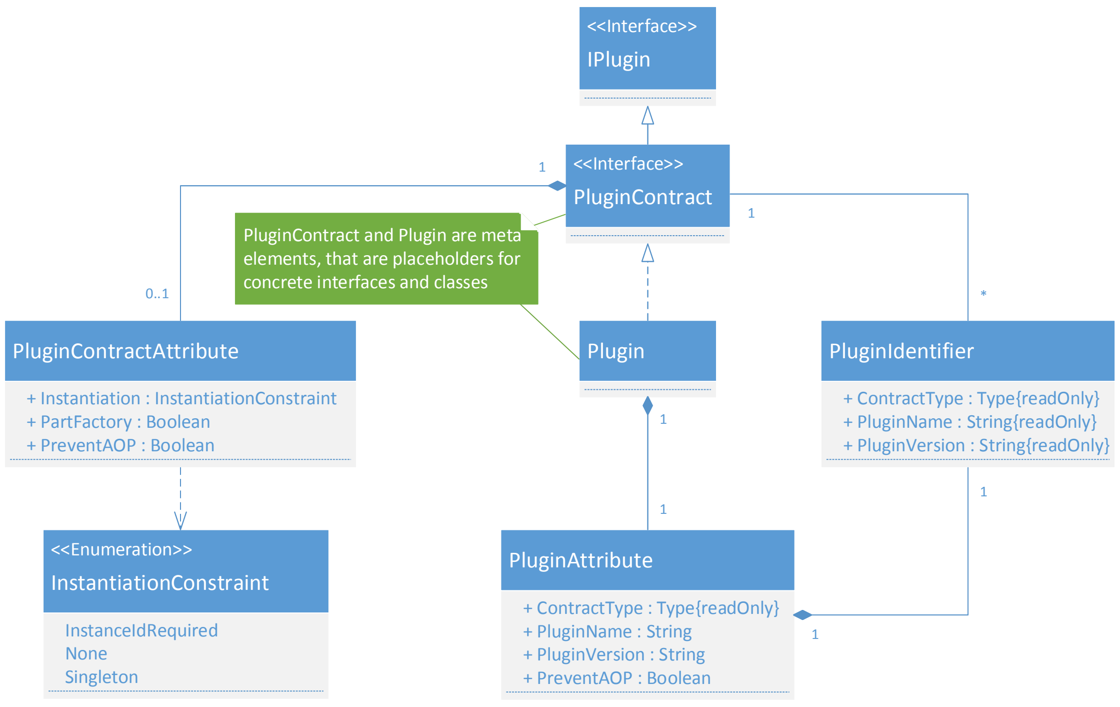


Figure 5.1: UML class diagram of the plugin meta model.

the contracts. Because of the partition of plugins and plugin contracts, they are usually deployed in separate assemblies. In addition to the method and property signatures, it is optionally possible to apply a `PluginContractAttribute` on the plugin contract interface to influence the behavior of the plugins. This attribute and its possible settings are presented in more detail in the relevant sections 5.3, 5.6, and 5.7.

5.2.2 Plugin Specification

A plugin is defined in the plugin framework as a C# class that implements a plugin contract and has a `PluginAttribute` applied. In Listing 5.1 a simple hello world example of a plugin `HelloWorld` and its plugin contract `IHelloWorldContract` are given.

This minimalistic plugin definition with the use of a plugin contract and the application of the `PluginAttribute` was designed to be sufficient for default usage scenarios, but it is also possible to customize the plugin definition by providing custom options in the attribute that are interpreted by the plugin framework. The `PluginAttribute` defines the following changeable properties:

- `PluginName` defines the name of the concrete plugin. By default the simple name of the implementing class of the plugin is used, but it is also possible to define a custom name by providing a `string`.

- `PluginVersion` specifies the version of the concrete plugin. If no custom version is specified as a `string`, the version "1.0" is used.

- `PreventAOP` is a boolean property that specifies whether aspect-oriented programming mechanisms are enabled for this concrete plugin or not. It is also possible to generally disable AOP through the plugin configuration system by setting the `PreventAOP` property of the `PlugicityConfigContract`. The general configuration system is presented in Section 5.5. There is even a third option to disable AOP by specifying it in the plugin contract through its `PluginContractAttribute`. If any of this three `PreventAOP` options are set to `true`, AOP is disabled for the corresponding plugins.

```
[Plugin]
internal class HelloWorld : IHelloWorldContract
{
    public void Hello()
    {
        Console.WriteLine("Hello World!");
    }
}

[PluginContract]
public interface IHelloWorldContract : IPlugin
{
    void Hello();
}
```

Listing 5.1: Example code of a plugin and its plugin contract

The readonly property `ContractType` stores the most specialized plugin contract the plugin implements. This property is automatically set.

The properties `ContractType`, `PluginName` and `PluginVersion` are additionally stored and aggregated in the `PluginIdentifier` class and are accessible as read-only properties. The class is required to distinguish multiple concrete plugin implementations that implement the same plugin contract and is used throughout the plugin framework. When the default `PluginAttribute` is used without customization, its application might seem redundant but is still required by the plugin framework. This was designed to make plugin implementations more easily identifiable for developers.

Even though the only attribute that is a necessity for plugins is the `PluginAttribute`, it is recommended to provide custom attributes for plugins if there are multiple plugin implementations for the same plugin contract, since it simplifies the selection mechanism. The custom attributes can be simple C# attributes that do not need to extend from a special attribute and may be used to distinguish between multiple plugin implementations for the same contract. Consequently, they are not necessarily connected to the plugin contract, even though this is a useful pattern that can also be applied.

5.3 Plugin Instantiation and Lookup

The plugin framework provides dependency injection and service locator patterns to instantiate and lookup plugins. The dependency injection mechanisms are the recommended option to instantiate plugins, but because of C# language limitations requiring constant values in attributes, the service locator approach is also supported to provide a dynamic alternative.

The default behavior of the plugin framework is to find the concrete plugin matching a plugin contract. To find all available concrete plugins, it automatically loads all assemblies from the application directory. However, this mechanism is adjustable through the configuration system presented in Section 5.5 by customizing the plugin framework's general configuration `PlugicityConfigContract`. It contains the following properties to adjust the plugin injection mechanism:

- `AssemblyPath` is the base path the plugin framework uses to detect and load assemblies. If this property is not set, the plugin framework tries to automatically detect the base path.
- `PluginSelection` maps plugin contracts to a list of concrete plugins. The default behavior of the plugin framework is to automatically map plugin contracts to all available concrete plugins. With this property it is possible to override this behavior for specific plugin contracts.
- `PluginBlacklist` can be used to specify concrete plugins that should be ignored by the plugin framework.

The dependency injection implementation in the plugin framework is called *plugin injection* and provides two different ways to inject plugins: *constructor injection* and a form of setter injection here called *property injection*, which only supports injection in public C# properties. While both ways provide the same injection functionality, constructor injection is the more flexible approach, as it allows to directly use the plugins after injection. If property injection is used instead, it is not possible to access the injected plugins from within the constructor as the plugins are set right after the constructor was called.

5.3.1 Property Injection

The property injection implementation allows the injection of plugins in properties of other plugins, as can be seen in Listing 5.2. This example illustrates the instantiation and usage of other plugins and therefore uses the hello world example plugin from Listing 5.1.

For the property injection it is required that the property is readable and writable, has *public* visibility, has the `InjectPluginAttribute` applied and its property type is a plugin contract, a plugin contract array or `IEnumerable<>` of plugin contracts. If any of the previous requirements are violated the injection will fail. Some restrictions, such as the property visibility are a consequence of the internal usage of MEF, which will be discussed in Section 5.4, but these restrictions have turned out to be of no significant relevance for the plugin framework or XVSM implementation. The properties' *public* visibility, for instance, is usually only relevant for internal classes within the plugin assembly, as other plugins, which typically reside in different assemblies, have no access to the class anyway and therefore keep the *Information Hiding* [Boo86] principle among plugins intact.

```
[Plugin]
internal class HelloWorldRunner : IHelloWorldRunner {
    [InjectPlugin]
    public IHelloWorldContract SimpleHelloWorld { get; set; }

    public HelloWorldRunner()
    {
        // SimpleHelloWorld == null
    }

    public void RunHelloWorld()
    {
        SimpleHelloWorld.Hello();
    }
}
```

Listing 5.2: Property injection example

5.3.2 Constructor Injection

The mechanism behind the constructor injection is similar to the mechanism of the property injection in Section 5.3.1. Both mechanisms only differ among each other in terms of notation and instantiation point of time. We will now have a look at the differences of constructor injection and property injection:

Instead of properties the constructor parameters are injected and therefore multiple `InjectPluginAttribute` may be applied to the parameters. However, the application of this attribute is optional for constructor injection, as all constructor parameters must be injectable anyway. It is still possible to apply the attribute to customize the injection behavior. The concrete customization possibilities are presented in Section 5.3.3.

With constructor injection, plugins are injected before the constructor gets called with the plugins as arguments, whereas the property injection mechanism injects the plugins only after the constructor was called. Situations where the injected plugins are used in the constructor may therefore require constructor injection. In contrast, there are no situations where property injection may not be replaced by constructor injection. Even though property injection is less powerful than constructor injection, it was included in the plugin framework because it leads to simpler code, on some occasions. Listing 5.3 is an implementation of the `IHelloWorldRunner` interface with constructor injection and shows a typical constructor injection usage.

```
[Plugin]
internal class HelloWorldRunnerCtor : IHelloWorldRunner
{
    private readonly IHelloWorldContract _simpleHelloWorld;

    public HelloWorldRunnerCtor(IHelloWorldContract helloWorld)
    {
        // helloWorld != null
        _simpleHelloWorld = helloWorld;
    }
    public void RunHelloWorld()
    {
        _simpleHelloWorld.Hello();
    }
}
```

Listing 5.3: Constructor injection example

5.3.3 Injection Customization

Both injection mechanisms, property injection and constructor injection, provide the same specific injection customization through the `InjectPluginAttribute`.

If the injection type is a single plugin contract, the plugin framework tries to find exactly one matching plugin implementation on default. If more plugins are found, the injection fails and produces an error, as the plugin framework cannot decide by its own which plugin it should instantiate. If no plugin was found, it fails too unless the injection is marked as *optional* with the `InjectPluginAttribute`'s `Optional` property. When this property is set to `true`, the injection process will not fail if no fitting plugin was found.

The `Id` property of the attribute is used to associate a plugin instance with an identifier. Hence, the identifier must be unique for all plugin instances with the same plugin contract. When an identifier is already associated with a plugin instance and used again with the same plugin contract, the plugin framework will not instantiate a new plugin instance but instead inject the already associated plugin instance.

The association of plugin instances to identifiers may be further affected by the `PluginContractAttribute` of the target plugin contract. With its `Instantiation` property, additional instantiation constraints may be specified either to force the usage of identifiers with the enum constant `InstanceIdRequired` or limit the instantiations to a single plugin instance with `Singleton`. When the `InstanceIdRequired` constraint is applied to the target plugin contract, injections of this plugin contract will use a default identifier if no custom identifier was specified. This constraint is often used in this thesis, to provide an uncomplicated way to inject a default plugin instance, but still keep the plugin flexible enough to support multiple instances. `Singleton` sets all injection identifiers that use the plugin contract to the same value. In contrast to the `InstanceIdRequired` constraint, the plugin framework produces an error if an identifier is manually specified for the injection.

If the property type is a collection of plugin contracts, the `Optional` and `Id` properties should be omitted and left on default, as they do not apply to a collection of plugin contracts. In this version, the plugin framework only supports injecting of multiple plugins that implement the same plugin contract without further customized semantics. If a more sophisticated algorithm is required, it is recommended to make use of the *plugin service locator*.

5.3.4 Plugin Service Locator

The service locator pattern [KJ04] is a programmatic alternative to the already presented dependency injection mechanisms. A service locator can be seen as a kind of registry which returns services. In this plugin framework, plugins correspond to services, hence the service locator returns plugin instances and thus is called *plugin service locator*.

The plugin service locator is implemented itself by a plugin through the plugin contract `IPluginServiceLocator` and is injected before being used. The plugin contract contains multiple method signatures to provide a multitude of instantiation options. The plugin retrieval may be tailored to specific needs by the use of methods with different selection possibilities. In the following enumeration, an overview of all available retrieval methods and their selection capabilities for the plugin service locator is given:

M.1 The parameterless generic method to retrieve a single plugin by specifying its plugin contract as a generic parameter.

M.2 The generic method to retrieve multiple plugins by specifying their plugin contract as a generic parameter and providing the required plugin cardinality.

M.3 The non-generic method to retrieve multiple plugins by specifying their plugin contract type as parameter and provide the required plugin cardinality.

M.4 The generic method to retrieve multiple plugins by specifying their plugin contract with an attribute type as generic parameters, a selection predicate, and the required plugin cardinality.

M.5 The generic method to retrieve multiple plugins by specifying their plugin contract as generic parameter and providing a selection predicate and the required plugin cardinality.

M.6 The non-generic method to retrieve multiple plugins by specifying their plugin contract type as parameter and providing a selection predicate and the required plugin cardinality.

The plugin cardinality defines whether the plugin service locator should return an instance of exactly one plugin, zero or one plugins, or zero or more plugins. The plugin cardinality should not be confused with the number of plugin instances as the plugin service locator's retrieval methods always return at most one instance for every plugin per call.

The methods from M.1 to M.3 may be used for basic plugin retrieval without a custom selection predicate. Plugins are only filtered by their plugin contract and the given cardinality. Whereas, the other methods from M.4 to M.6 support a more sophisticated plugin selection mechanism through predicates.

Method M.4 intensifies the filtering of the plugins in addition to the plugin contract type filter used in the basic methods. Because of the provided attribute type, the plugin framework will only select plugins that have an attribute of the specified type applied. The selection predicate is a function of type $a \to bool$, of which the single parameter a is an instance of the generic class `MatchWithAttribute<>`. It contains an instance of the plugin's custom attribute, as well as an instance of the `PluginAttribute`. The concrete plugin can be chosen by using the metadata from these attribute instances. The framework instantiates the plugin or plugins according to the predicate results with respect to the specified cardinality. A code example of this method invocation is shown in Listing 5.4.

```
[Plugin]
internal class RunnerFactory : IRunnerFactory {
    [InjectPlugin]
    public IPluginServiceLocator ServiceLocator { get; set; }

    public IHelloWorldRunner CreateRunner(string name) {
        return ServiceLocator
         .GetPlugins<IHelloWorldRunner, RunnerInfoAttribute>(
            match => match.MetadataAttribute.RunnerName == name
            , Cardinality.ExactlyOne).First();
    }
}
```

Listing 5.4: Example of advanced plugin instantiation through the plugin service locator.

In this code example an `IRunnerFactory` plugin is implemented to return instances of plugins with the `IHelloWorldRunner` plugin contract that have a specific name.

Through the predicate function, the concrete plugin with the matching name of its `RunnerInfoAttribute` is picked. The lambda argument `match` is a generic `MatchWithAttribute<>` instance with two properties: `MetadataAttribute` and `PluginAttribute`. The former returns the instance of the custom attribute and the latter the plugin attribute. Since the used cardinality guarantees that the resulting list of the `GetPlugins` method will contain exactly one element, or otherwise an exception is thrown, the `First` method is used to pick and return the element. Listing 5.5 shows an exemplary concrete plugin with the custom `RunnerInfoAttribute` applied that is chosen for instantiation by the name "simple".

```
[Plugin, RunnerInfoAttribute("simple")]
internal class SimpleHelloWorldRunner : IHelloWorldRunner {
    private readonly IHelloWorldContract _simpleHelloWorld;

    public SimpleHelloWorldRunner(IHelloWorldContract
       helloWorld) {
        _simpleHelloWorld = helloWorld;
    }
    public void RunHelloWorld() {
        _simpleHelloWorld.Hello();
    }
}
```

Listing 5.5: Example of a plugin with a custom attribute used for custom selection.

The methods from M.5 to M.6 are similar to method M.4 as they also use a selection predicate but they do not prefilter the plugins on the basis of a specific attribute. They evaluate the predicate with an instance of the non-generic `MatchWithAttributes` class that stores beside the plugin attribute all attributes that are applied to the plugin. The method M.6 is a non-generic version of method M.5 with an additional parameter of the contract type instead of a generic parameter.

The methods M.3 and M.6 are non-generic variants of the methods M.2 and M.5. Instead of a generic parameter that defines the concrete plugin contract, the methods provide an additional regular parameter that is used to dynamically specify the plugin contract as a C# `Type`. Thus, the return value of the methods is also non-generic and uses the super interface `IPlugin` instead of the concrete plugin contracts. These non-generic variants are the most flexible selection methods of the plugin service locator, since they can be used for plugin contracts that were unknown at compilation time.

The internal instantiation mechanism is delegated by the plugin service locator plugin to the plugin loader plugin through the plugin contract `IPluginLoader`, which is discussed in Section 5.4.

Even though the instantiation of a plugin through the plugin service locator requires more glue code than constructor or property injection, it is a more flexible mechanism, since it allows to programmatically instantiate plugins and use selection predicates to choose the correct plugin. However, it is recommended to prefer the injection mechanisms over the plugin service locator if they are a viable solution.

5.4 Plugin Framework Implementation

In this section we give an overview of the architectural design of the plugin framework and show how the basic functionality of the plugin mechanism is realized. We show how we used MEF to detect and instantiate plugins.

The plugin framework supports a bootstrapping mechanism that loads the minimal required plugins to initialize the plugin framework. The goal of the bootstrapping is to create the plugin framework container. With this container it is possible to instantiate the first plugin and, therefore, provide an entry point for the custom application.

5.4.1 Overview

Figure 5.2 depicts internal plugins and their dependencies, which are used in the plugin framework to provide the presented functionality.

The main plugin of the plugin framework is the `IPlugicityContainer`, named after the plugin framework's internal name *plugicity*. This plugin is responsible to instantiate and wire plugins required for the plugin framework. However, the plugin requires working instances of `IConfig` and `IPluginLoader` plugins. The bootstrapping mechanism used to provide the plugin instances is explained with more detail in Section 5.4.2.

The `IConfig` plugin, as explained in Section 5.5, is used to access the configuration system of the plugin framework and is required for the `IPluginLoader` plugin. In addition, the `IPluginLoader` plugin requires the `INamedPluginsRegistry`, `IProxyManager`, and `IPluginAspectRegistry` plugins to provide basic functionality. Plugins that are injected with an identifier, or that have a constraint on their plugin contracts that forces them to use an identifier, are registered at the `INamedPluginsRegistry` plugin. This plugin is used as a storage for already instantiated plugins that are accessible through their identifier. The `IProxyManager` plugin is used to create and manage proxy objects that wrap concrete plugins in order to allow aspect and recomposition mechanisms. The `IPluginAspectRegistry` plugin contains a list of all plugin aspects and the corresponding plugin contracts they target. The `IProxyManager` depends on it, to wire plugin aspects to the created proxy objects.

The `IRecompositionHandler`, which is described in Section 5.8, is instantiated by the `IPlugicityContainer` plugin to react on changes in the filesystem to start the recomposition process. It has a dependency on the `IPluginAspectRegistry` plugin to notify the registry of changed plugin aspects because of a recomposition. It also has further dependencies to the `IPluginLoader` and `IProxyManager` to instantiate new plugins and replace old plugins in the existing proxies.

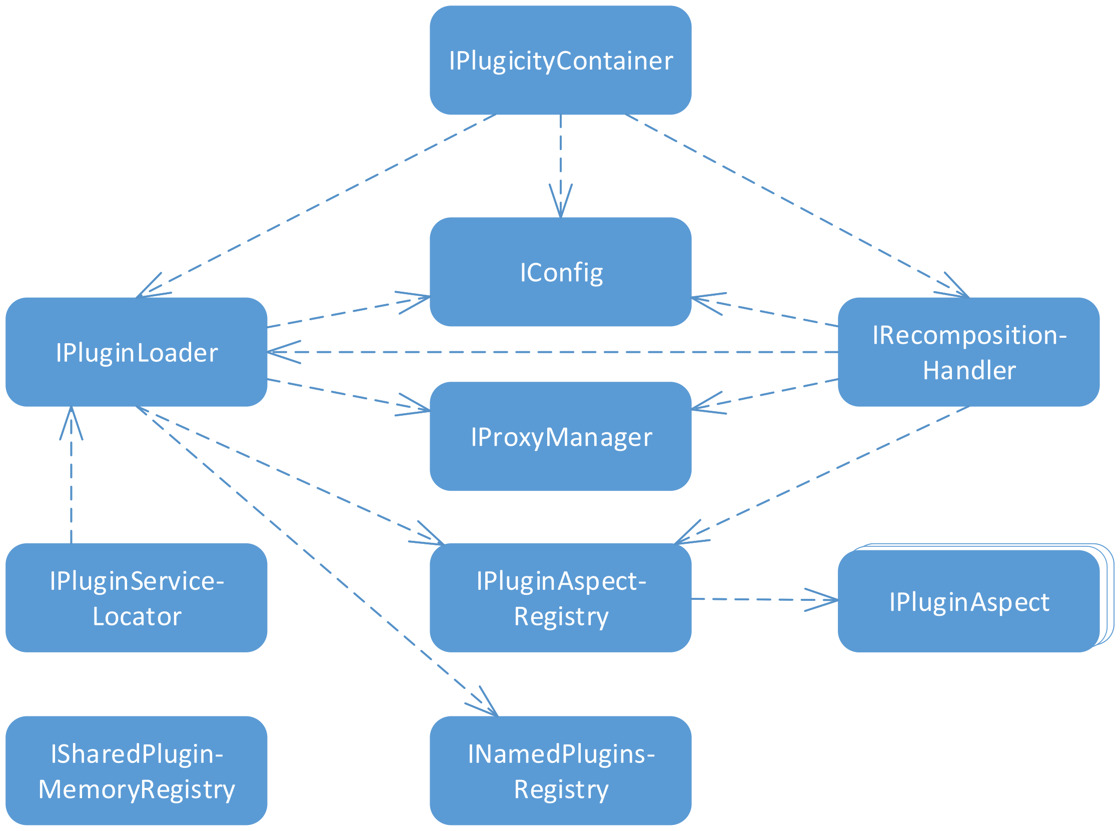


Figure 5.2: Architectural overview of the plugin framework.

The `IPluginServiceLocator` depends on the `IPluginLoader` to select and instantiate concrete plugins with the specified semantics. This plugin is not necessarily required for the plugin framework to work, however it increases its functionality and is included in the core assemblies. The `ISharedPluginMemoryRegistry` allows shared state between plugins and is also included in the core assemblies. The plugin and its mechanisms are explained with more detail in Section 5.7.5.

5.4.2 Bootstrapping

The `PlugicityBootstrapper` class provides a single method to instantiate a `PlugicityContainerConfig` object, which can be used to customize the plugin framework configuration by overriding configurations or providing a custom configuration file path. The classes involved for the bootstrapping are meant to be used with a fluent style, hence the `PlugicityContainerConfig` object provides a method to instantiate the plugin framework container right after the configuration, as can be seen in the code extract in Listing 5.6. In this example, the `PluginBlacklist` setting of the plugin framework's main configuration contract `PlugicityConfigContract` is overwritten.

```
var container =
    PlugicityBootstrapper.InstantiatePlugicityContainer()
        .ManualConfig(c =>
           c.Overwrite<PlugicityConfigContract>(new
        {
            PluginBlacklist = new[] {"MyXYZPlugin"}
        })).InstantiatePlugicityContainer();
var app = container.GetPlugin<MyApplicationPlugin>();
```

Listing 5.6: Exemplary plugin framework container bootstrapping and instantiation of a custom plugin.

The plugin framework bootstrapping can be split into three phases. In the first phase the plugin framework configuration is loaded and interpreted to gather the required assemblies. In the second phase a separate MEF container is created and provided with instances of `AssemblyCatalog` to load the plugin framework core plugins, such as `IConfig`, `IPluginLoader`, and `IPlugicityContainer`, which are used in the final bootstrapping phase three. The `IPlugicityContainer` plugin is initialized with the other two plugins and the originally provided configuration. It configures the plugin loader, loads the remaining required plugin framework plugins and binds them to the naming registry. The plugin loader uses its own MEF container with a custom `ExportProvider` that contains more functionality than the bootstrapping container used earlier. Then, the `IProxyManager` plugin for creating proxy objects that are required for plugin aspects, as well as the registry for the plugin aspects `IPluginAspectRegistry` and all available plugin aspects are loaded. After the initialization procedure, the `IPlugicityContainer` plugin provides a delegation of `GetPlugin<>()` invocations to the `IPluginLoader` plugin, to instantiate custom plugins.

5.4.3 Plugin Loader

The `IPluginLoader` plugin is responsible to load and manage plugins through MEF. During initialization, a MEF directory catalog is used to scan for assemblies that contain plugins. Since we do not use MEF's attributes to import or export plugins but use custom interfaces and attributes, we make use of MEF's custom convention system with its `RegistrationBuilder` class. It allows the plugin framework to select the plugins through their `PluginAttribute` and wire the corresponding imports. In addition we use a custom `ExportProvider` that extends MEF's default *export provider* and overrides its `GetExportsCore()` method. MEF supports custom export providers since they allow a more refined control of the exports. A major part of the export provider's responsibilities is the plugin selection. For instance, if a plugin is blacklisted through the configuration system, the export provider filters the plugin from possible instantiation candidates.

5.5 Plugin Configuration

Configuration is still needed for plugins to quickly make adjustments without the need for recompilation or the creation of own plugins with an almost identical code base. As already mentioned in Section 3.5, it was decided to create a custom configuration system and not use the integrated .NET system to provide a more flexible solution. The configuration system is implemented by the built-in `Config` plugin in the `Plugicity.Plugin.Config` namespace. The `Config` plugin implements the `IConfig` interface and is solely responsible for accessing the configuration.

5.5.1 Configuration Concept

The configuration is structured with *config contracts*, which are classes that are used as a container for coherent settings. Such a config contract may be filled by data from configuration files or manually through code. The config contract must be a specialization of the `AbstractConfigContract` class and only contain public properties that represent the settings. Even though the properties may be of any type, not all types can be deserialized from a config file. YAML is used as the file format of the configuration file and it supports primitive datatypes, enumerations and several collections, such as lists and dictionaries. Config contracts with YAML-incompatible data types are not illegal as long as the illegal property is not included in the YAML config file. As the configuration can also be provided programmatically, settings that are incompatible with YAML can be helpful in certain scenarios. For instance, this allows to programmatically provide a custom predicate function through the configuration system.

Config contracts were designed for two different usage scenarios: as a settings implementation that can be accessed from multiple plugins that have access to the config contract, but also for private plugin configurations that other plugins do not necessarily need access to.

5.5.2 Configuration Access

The `Config` plugin is the central point of the configuration system and provides access to the configuration through config contracts. The first time a specific config contract gets requested, the `Config` plugin instantiates an instance of the specific config contract and looks in the search path for a compatible YAML configuration file that has compatible elements. If a compatible YAML configuration was found, it gets deserialized and mapped to the requested config contract. The plugin framework provides a facility to specify manual, volatile configuration overwrites for the config contracts. When the plugin framework container gets initialized, it is possible to specify an `IManualConfig` instance that contains *manual configuration* elements to overwrite specific configurations on a per-setting granularity. Details regarding the creation process of `IManualConfig` are covered in Section 5.5.3.

The relevant configuration elements of `IManualConfig` are applied to the config contract after the YAML deserialization and mapping have been finished, in order to ensure the possibility to overwrite the settings.

Finally a clone of the prepared config contract is created by calling the `Clone()` method of the `AbstractConfigContract` to prevent an external modification of the settings.

A visual illustration in form of a UML sequence diagram of the described access mechanism is given in Figure 5.3.

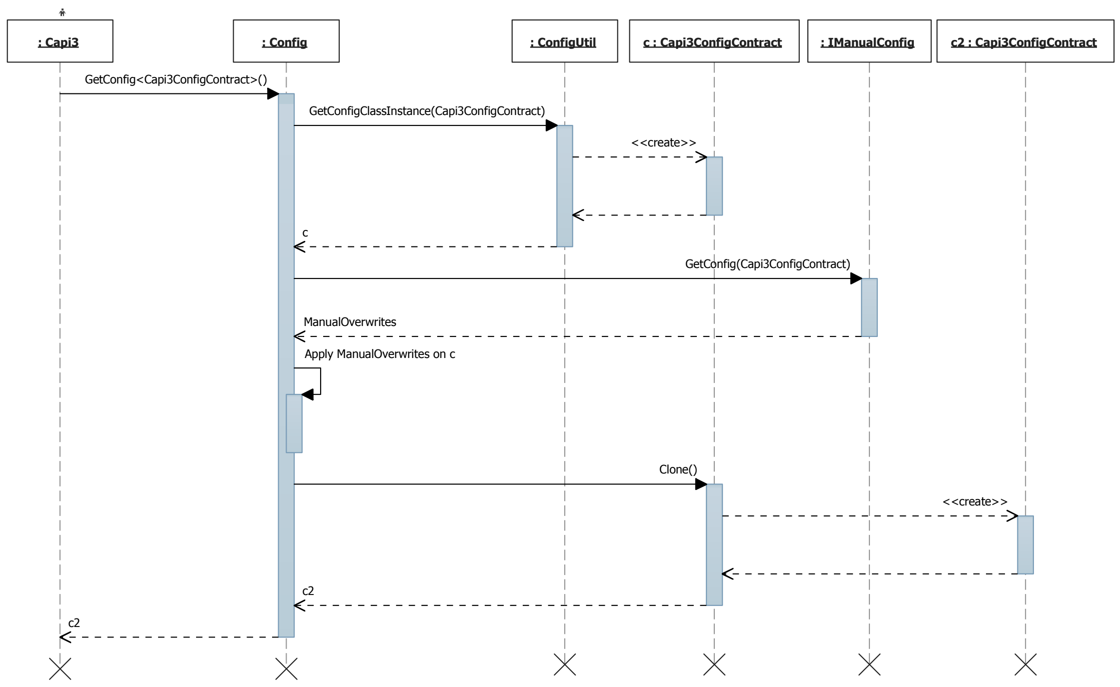


Figure 5.3: UML sequence diagram for the first configuration read access of a config contract with manual overwrite.

5.5.3 Manual Configuration

Through the `ManualConfig` class it is possible to provide custom volatile settings that overwrite default or serialized ones. `ManualConfig` provides multiple overloaded `Overwrite()` methods that can be used to specify the custom settings.

The different `Overwrite()` methods either require the config contract type as a generic parameter, or its full name as a string in order to map the manual configuration to the right config contract. The overwritten settings may be either specified in a dictionary or as an anonymous type. Listing 5.7 gives an example of all possible `Overwrite()` methods and their usages.

```
// typed with anonymous object
plugicityConfig.ManualConfig(
    c => c.Overwrite<HelloWorldConfigContract>(new {Message =
       "Hello (configured) World"}));

// typed with dictionary
plugicityConfig.ManualConfig(
    c =>
        c.Overwrite<HelloWorldConfigContract>(new
           Dictionary<string, object>
        {
            {"Message", "Hello (configured) World"}
        }));

// dynamic with dictionary
plugicityConfig.ManualConfig(
    c =>
       c.Overwrite("XVSM.CodeExamples.HelloWorldConfigContract",
       new Dictionary<string, object>
    {
        {"Message", "Hello (configured) World"}
    }));
```

Listing 5.7: Exemplary programmatical overriding of plugin configurations.

5.6 Plugin Parts

Plugin parts are lightweight components with support for plugin aspects that do not suffer from instantiation overhead such as regular plugins.

5.6.1 Plugin Parts Concept

Whenever a regular plugin is instantiated, the MEF's selection algorithm to find the correct implementation is run. This frequent reevaluation is significantly slower than the instantiation of simple classes, which internal micro-benchmarks have shown. These performance implications might make it unfeasible to use plugins for situations where quick and frequent instantiations are required. The developer then needs to decide early in development whether the performance implications might be problematic for this concrete situation or not. The problem is that the advantages of plugins, such as the support of plugin aspects, might not be very relevant at development time, but shine later when an existing application may be modified without recompilation. Hence, it was decided to create another conceptional element, called *plugin parts*, that should help developers to use the plugin framework without affecting the instantiation performance too much.

Plugin parts are lightweight sub-components of plugins, which fulfill so-called *plugin part contracts* that extend the interface `IPluginPart`. In contrast to plugins, plugin parts may not be injected or instantiated through the plugin service locator, but also do not support the injection of other plugins. Plugin parts are directly instantiated instead by plugins called *plugin part factories* that implement special plugin contracts. Plugin parts instantiated by these plugin part factories fully support plugin aspect mechanisms.

5.6.2 Plugin Part and Plugin Part Factory Specification

Plugin parts are simple objects that do not require to have specific attributes applied in order to work, in contrast to plugins, however they still must implement a contract interface that is a subtype of `IPluginPart`. Plugin part factories are regular plugins implementing plugin contracts with `PluginContractAttributes` that have the property `PartFactory` set to `true`.

Listing 5.8 gives an example definition of a plugin part and its corresponding plugin part factory.

```
public interface IActor : IPluginPart {
    void Run(object[] objects);
}

[PluginContract(PartFactory = true)]
public interface IActorFactory : IPlugin {
    IActor CreateActor();
}

[Plugin]
internal class ActorFactory : IActorFactory {
    public IActor CreateActor() {
        return new Actor();
    }
}

internal class Actor : IActor {
    public void Run(object[] objects) {
        // Concrete implementation
    }
}
```

Listing 5.8: Simple example of plugin part instantiation.

Plugin part factories were designed to be solely used to instantiate plugin parts. Therefore, methods with return values are required to return single plugin part contract interfaces. Any other return value will lead to an exception when the plugin part factory is initialized.

5.6.3 Instantiation

Plugin parts are directly instantiated like any other C# class by plugin part factories in their designated methods. As plugin parts do not support the injection of other plugins, they depend on regular plugins to satisfy their plugin dependencies. Plugin part factory plugins are well equipped to handle such situations, as they can easily instantiate or lookup the required plugins and provide them in the constructor of the plugin part class. Listing 5.9 shows the instantiation of a plugin part whose plugin dependencies are provided as arguments in the constructor. Since the factory typically keeps the required plugins in-memory, instantiations of the plugin part have little overhead.

```
[Plugin]
internal class ActorFactory : IActorFactory {
    private readonly IExecutionContext _context;

    public ActorFactory(IExecutionContext context) {
        _context = context;
    }
    public IActor CreateActor() {
        return new Actor(_context);
    }
}
```

Listing 5.9: Plugin part instantiation with plugin dependencies.

5.6.4 Implementation

The integration of plugin parts is possible solely through the provided functionality of the plugin framework. Plugin parts are implemented through a single plugin aspect (cf. Section 5.7). The plugin aspect intercepts the methods of the plugin part factories that return `IPluginPart` instances. Only the interception after the method invocation that instantiates the plugin part is relevant for the plugin aspect. The resulting `IPluginPart` instance is then wrapped by the proxy manager with a proxy object. This allows the interception of plugin parts with further plugin aspects.

5.7 Plugin Aspects

Plugin aspects provide mechanisms to transparently intercept and modify the behavior of plugins or plugin parts. With plugin aspects it is possible to change the arguments of method calls, change the return value of methods, and even to handle thrown exceptions, all while being transparent to the caller and callee.

5.7.1 Plugin Aspects Concept

Plugin aspects are bootstrapped as regular plugins that implement the generic interface `IPluginAspect<>`, which extends the `IPluginAspect` plugin contract. The generic type of the interface specifies the *target type* that should be intercepted. The target type is usually a specific plugin or plugin part contract but may be also `IPlugin` or `IPluginPart` interfaces to apply the aspect to multiple targets. In this section the term *target* refers to an intercepted instance of the target type, which might be either a plugin or a plugin part.

Plugin aspects require the application of two attributes: the `PluginAttribute`, which is a general plugin requirement, and the `PluginAspectAttribute`. The latter attribute is needed to specify a unique aspect id for the plugin aspect.

Through the generic `IPluginAspect<>` interface, plugin aspects may provide a predicate to select the methods that should be intercepted by their signatures. Since plugin contracts may have many methods but plugin aspects may only target a few of them, this selector predicate was introduced to simplify the plugin aspect development. In addition to the predicate, the interface contains two methods to provide interception points before and after the corresponding plugin method was called. With this two interception points it is possible to change the behavior of the target by modifying the arguments, return values, but also by handling exceptions that might have occurred in the plugin.

It is possible for multiple plugin aspects to be registered on the same target. The target is then intercepted in a specific order by all the corresponding plugin aspects one by one. This list of plugin aspects is called *aspect chain*.

The instantiation and registration of plugin aspects is handled by the plugin framework. Plugin aspects are only instantiated once and registered to possibly multiple targets. This leads to a reuse of plugin aspects on calls of different targets.

5.7.2 Interception with Plugin Aspects

The generic `IPluginAspect<>` interface provides two method signatures that provide interception points before and after the target's methods are called.

InterceptBeforeExecution

This method is called before the target's method is called. The interception method provides the concrete instance of the intercepted target, a `MethodInfo` object, and an array with the original call's arguments. The `MethodInfo` class is part of the .NET reflection support and contains metadata about the called method of the target. The arguments of the original call are stored in an object array. It is possible to modify the arguments by simply changing them in the array. Further executed aspects in the same chain will be called with the modified arguments and have no access to the original arguments anymore. After the code of the interception method was executed, the plugin framework will continue with calling the aspect chain and finally calling the original target, unless an exception has occurred. In the case of an exception, the exception will be escalated and the target will not be called at all. Further aspect methods aspect methods that should have been executed before the target will be skipped as well.

InterceptAfterExecution
After the method of the target was called and either returned normally or stopped prematurely because of a thrown exception, this method is called to handle the result or the thrown exception. The signature of the interception method contains the same parameters as the `InterceptBeforeExecution` method with the addition of the following two: an object which contains the result of the target, and an `Exception` object. The exception object contains the instance of the thrown exception, if one was thrown at all. In contrast to the other interception method, this method also provides a return type, which may include the resulting object and a directive that specifies how the exception should be handled if one was thrown. The directive may be set to `PostponeHandling` to leave the exception handling to other aspects in the aspect chain or escalate the exception to the caller if the last aspect in the aspect chain uses this directive. With the `DropException` directive the thrown exception will be skipped. Further executions of the aspect chain and the original caller of the target will not be notified about the exception. The `EscalateException` directive aborts the execution of the aspect chain right away and re-throws the exception to notify and leave the handling to the caller.

The `IPluginAspect<>` interface provides the signature of the readonly property `MethodSelector`, which is a C# Predicate with `MethodInfo` as parameter. This property is used to select the concrete methods of the targets that should be intercepted. Hence, the Predicate will be called for every method of the target. The `MethodInfo` parameter may be used to decide which methods should be intercepted. An example implementation of the `MethodSelector` property is shown in Listing 5.10, which excludes all methods that are not named "Hello".

```
public Predicate<MethodInfo> MethodSelector
{
    get { return m => m.Name == "Hello"; }
}
```

Listing 5.10: Method selection in a plugin aspect.

5.7.3 Execution Order

The plugin aspects that are contained in a single aspect chain can be sorted and executed in a custom order. The sorting details are specified in the `PluginAspectsConfig-`

`Contract`, which provides two properties:

OrderedAspectIds
With this property it is possible to manually specify a list with plugin aspect identifiers in the favored order for target types. Plugin aspects whose identifiers have a lower index are then called before ones with a higher index. Unknown plugin aspects are inserted in alphanumeric order after the specified plugin aspects. Aspect chains of unspecified target types are also sorted in alphanumeric order.

AspectIdsComparer
If the manual sorting is not powerful enough, it is possible with this property to specify a custom comparer object of type `IComparer<IPluginAspect>`. With a custom comparer object, an individual sorting is possible, which is depending on the available aspects and not on a static configuration.

If no custom sorting mechanism is used, an alphanumeric sorting based on the plugin aspect identifiers is applied instead. The order in which the plugin aspects are organized in the aspect chains define the execution order of the plugin aspects' interception methods. The before methods are executed in an ascending order and the after methods in descending order.

5.7.4 Instantiation

As previously mentioned, plugin aspect instances may be shared across different targets. This is the result of a performance optimization, which was integrated because of the fact that plugin parts are also possible targets besides regular plugins. As plugin parts are lightweight and possible short-lived elements, it is essential that the creation process of plugin parts should not be delayed because of plugin aspects. The use of separate plugin aspects for every instantiated plugin part would have negated the elemental concept behind plugin parts because of the instantiation overhead of plugin aspects. Plugin aspects are plugins and therefore share the required instantiation computations of regular plugins. This led to the decision to share plugin aspect instances with different targets.

However, the sharing of plugin aspect instances may affect the way individual plugin aspects are implemented when they are stateful. As every intercepted call provides the target's object, it is possible to manually manage the state by using the target's object as an identifier in a dictionary with separate objects that hold the individual states of the plugin aspect.

5.7.5 Shared Memory of Plugin Aspects

Even though plugin aspects have access to the target with the intercepted method, its calling parameters and return value, they have no direct access to previously used parameters or calculated return values of other plugins or plugin parts. Accessing such previously used elements with additional plugin aspects is trivial, however, transferring the data from one plugin aspect to another is more difficult. As plugin aspects already support the modification of method parameters, the obvious approach would have been to use the method parameters to transfer the elements, but this leads to several issues. C# is a static programming language, therefore method parameters cannot be easily added at runtime and would require byte-code manipulation. If the elements should be transferred along the execution path of multiple plugins, then all involved classes were subject to byte-code manipulation as the parameters must be passed along. If the elements should be transferred to a plugin aspect which is not in the execution chain of the former plugin aspect, transferring through the parameters is not feasible. This issue has an impact on the power of plugin aspects, as it would not allow plugin aspects on independent parts of systems to have access to a shared state with other plugin aspects. Hence, a different approach with a central registry was followed. The plugin contract `ISharedPluginMemoryRegistry` with its default implementation provides a convenient access to a data registry. The `ISharedPluginMemoryRegistry` plugin contract has the `InstanceIdRequired` instantiation constraint to guarantee that the same instance of the `ISharedPluginMemoryRegistry` plugin will be injected on default. Figure 5.4 depicts the interfaces that are used for the registry.

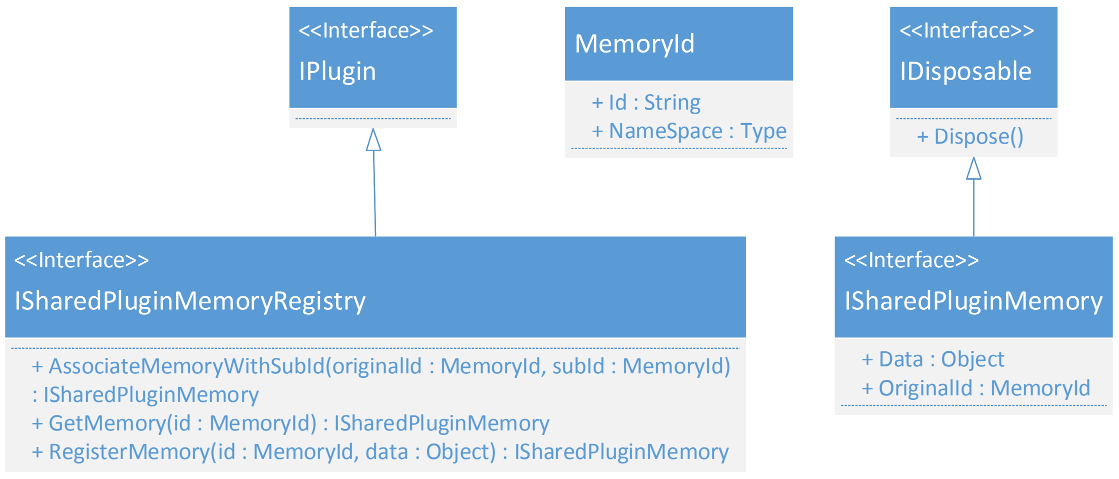

<<Interface>>
IPlugin
MemoryId
+ Id : String
+ NameSpace : Type
<<Interface>>
IDisposable
+ Dispose()
<<Interface>>
ISharedPluginMemoryRegistry
+ AssociateMemoryWithSubId(originalId : MemoryId, subId : MemoryId) : ISharedPluginMemory
+ GetMemory(id : MemoryId) : ISharedPluginMemory
+ RegisterMemory(id : MemoryId, data : Object) : ISharedPluginMemory
<<Interface>>
ISharedPluginMemory
+ Data : Object
+ OriginalId : MemoryId

Figure 5.4: UML Class Diagram of Shared Plugin Memory Interfaces

The `ISharedPluginMemoryRegistry` interface provides the method signature `Get-Memory` to retrieve an `ISharedPluginMemory` instance from the registry by providing a *memory identifier* through `MemoryIdentifier`. A memory identifier consists of a textual identifier and an optional type. The latter is used to organize memory instances in namespaces. Hence, it is recommended to use the plugin contract type of the plugin that uses the shared plugin memory. Before the data can be retrieved it must have been registered first, which is possible through the `RegisterMemory` method signature. The method stores an arbitrary object in the registry and associates it to the additionally specified memory identifier for efficient lookup functionality through the `GetMemory` method. Both method signatures return an instance of the interface `ISharedPluginMemory`, which encapsulates the data and the memory identifier that were specified in the `RegisterMemory` method. One might notice the absence of a method signature in the `ISharedPluginMemoryRegistry` interface to unregister data, however, the `ISharedPluginMemory` interface is a specialization of the C# `IDisposable` interface and therefore must provide a `Dispose` method that should be called to get rid of the `ISharedPluginMemory` instance. The remaining `AssociateMemoryWithSubId` method signature of `ISharedPluginMemoryRegistry` is used to specify a *sub-memory identifier* in addition to the previously associated original memory identifier for the same memory instance. The `GetMemory` method will return the same `ISharedPluginMemory` instance for the sub-memory identifier as it would for the original memory identifier. For every associated sub-memory identifier the corresponding `ISharedPluginMemory` instance must be separately disposed before the original instance can be disposed. This design decision might seem uncomfortable for development but it ensures a dedicated lifetime handling of the shared memory in order to help prevent memory leaks and access of obsolete memory. Without a concrete example, the mechanism to associate sub-memory identifiers might seem unmotivated, however, it was introduced due to requirements resulting from XVSM's *request context* feature implementation, which is discussed in Section 6.4.3.

5.7.6 Programmatic Target Selection

The presented mechanism to provide plugin aspects for plugins (or plugin parts) either requires to directly target the concrete contract or use the general `IPlugin` or `IPluginPart` interfaces. However, if certain plugins with different or even unknown contract should be intercepted, a more refined selection mechanism is required. To solve this, a programmatic target selection mechanism is available through the `PluginSelector` property of the generic `IPluginAspect` interface. It is a property that requires a boolean selection function that decides whether a plugin or plugin part should be used as a target type for the aspect. It provides two parameters, the type and its plugin contract attribute if available.

5.8 Recomposition

The plugin framework provides support to change the program code at runtime without requiring a restart of the application. This feature is called *recomposition* in the plugin framework and is a type of *dynamic software update* (DSU) [HN05]. It supports the dynamic adding, removal and replacement of plugins and plugin aspects.

5.8.1 Controlling Recomposition

At the moment recomposition may only be bootstrapped by adding or removing assemblies from the plugin directory. This leads to a re-evaluation of the available plugins and starts the recomposition process. The plugin framework provides hooks to control the recomposition process with the `IRecompositionHandler` plugin.

The plugin provides a registrable event `OnPluginsChanged` that is fired when plugins are added or removed. The event includes the plugin attributes of the changed plugins in separate lists for added and removed plugins. It allows to provide a mechanism to programmatically decide when the recomposition should be started. For instance, an application might first finish open requests and block or delay new requests until the recomposition is finished. When the application decides the recomposition should be started, the `RecomposePlugins()` method of the `IRecompositionHandler` plugin must be executed in order to start the recomposition. When the recomposition is finished, the plugin provides a notification through a separate event on its registered event handlers.

```
[Plugin]
class RequestHandler : IRequestHandler {
    private readonly ReaderWriterLockSlim _lock = new
       ReaderWriterLockSlim(LockRecursionPolicy.NoRecursion);

    public RequestHandler(IRecompositionHandler handler) {
        handler.OnRecompositionStarted +=
            (s, e) => _lock.EnterWriteLock();
        handler.OnRecompositionFinished +=
            (s, e) => _lock.ExitWriteLock();
        handler.OnPluginsChanged +=
            (s, e) => handler.RecomposePlugins();
    }

    public void Handle(object request) {
        _lock.EnterReadLock();
        try {
            // execute request
        } finally {
            _lock.ExitReadLock();
        }
    }
}
```

Listing 5.11: Plugin that controls the recomposition while queuing incoming requests.

Arbitrary plugins are able to inject the `IRecompositionHandler` plugin and register for the events. Listing 5.11 shows an example plugin that queues incoming requests while the recomposition is active. The plugin uses a `ReaderWriteLockSlim` to queue incoming requests until the recomposition is finished and the exclusive write lock is released.

5.8.2 Implementation

Recomposition is an optional feature of the plugin framework. It must be explicitly enabled by setting the `Recomposition` property of the plugin framework's main configuration contract to `true`. The `IRecompositionHandler` plugin is responsible to handle the recomposition lifecycle. It listens on an event handler of the `IPluginLoader` plugin to detect when the available plugins have changed. The `IPluginLoader` uses MEF's functionality via a `DirectoryCatalog` to react on changes of the plugin assemblies on the file system. Through the information passed to the `IPluginLoader` it is able to detect which plugins were added and which were removed. This information is then passed to the `IRecompositionHandler` through the previously mentioned event. After receiving this information, the `IRecompositionHandler` notifies its registered listeners through the `OnPluginsChanged` event and stores the information for a future recomposition.

When the `RecomposePlugins` method is called and the recomposition is started, the `IRecompositionHandler` executes the following steps:

1. Notify the registered listeners that the recomposition has started.
2. Instruct the plugin loader to load the new plugins and mark the removed plugins as deleted.
3. Instantiate new plugins and migrate existing instances through their proxy objects.
4. Update the registered plugin aspects through the proxy objects.
5. Notify the registered listeners that the recomposition is finished.

Unfortunately, the recomposition process has a few limitations at the moment. It is not possible to exchange plugin implementations of plugins that were injected with a cardinality higher than `1`. For instance, when an `IEnumerable<IMyPlugin>` of plugins are injected, their instances are not replaced. The plugin framework would have to exchange the `IEnumerable<>` instance, or create a proxy for it too. However, at the moment the only workaround is to programmatically change the `IEnumerable<>` instance during the recomposition.

Plugin parts suffer from a similar issue that prevents them from being automatically exchanged with newer version. Even though plugin parts could be exchanged in theory, due to their usage of proxy objects, it is unfeasible from a performance perspective. It would be required to keep a list of all plugin parts in order to change their instances, but such a list would negate the idea and benefit of plugin parts, as short-lived and cheap instances.

Another limitation of the recomposition mechanism is that the removed plugins still stay in memory. The problem is that the .NET platform does not allow to unload specific assemblies from the `AppDomain` at the moment [25]. Nevertheless, it would be possible to create a new `AppDomain`, load all required assemblies from the new `AppDomain`, and destroy the old `AppDomain`. However, this approach could be followed in a future version of the plugin framework, if it is needed.

CHAPTER 6

Plugin-Based XVSM Design & Implementation

The plugin framework that was presented in the previous chapter is the base for the XVSM.NET implementation. This chapter begins with an architectural overview and continues with an in-depth look at the main components of the space-based framework and how they were designed in accordance with the plugin framework. The chapter concludes with illustrations on how XVSM.NET can be used from a developer's point of view.

6.1 Architectural Overview

XVSM.NET is organized in over forty separate *Visual Studio projects*, which compile to individual assemblies. To get a better understanding of XVSM.NET we will take a quick glance at all relevant projects. The projects can be grouped into the following four categories:

1. **Contract projects** contain Plugin and PluginPart contracts with associated types, such as interfaces, classes, attributes, structs or enums. These projects do not contain elements with business logic code unless they are used to ensure valid usage of the contracts, or to provide helper functionality through extension methods. It is also possible to include configuration contracts and files in these projects to provide a configuration which can be used by multiple Plugins and PluginParts. Contract projects may depend on other contract projects and extend or use their contract interfaces or associated types, but they must not depend on other types of projects.

   The names of contract projects are prepended with "`XVSM.Contract`".

2. **Plugin projects** contain the business logic of XVSM.NET and therefore depend on at least one contract project to implement one or more Plugin or PluginPart contracts. These projects are not restricted, such as contract projects, in terms of functionality or behavior, but should only contain coherent elements regarding the business logic. Similar to contract projects, plugin projects may provide configuration capabilities as well, but the configuration contracts must be bound to Plugins in order to be used. In addition to implementations of contract projects, plugin projects may use other injected Plugin and PluginPart implementations through their corresponding contracts.

   The names of plugin projects are prepended with "`XVSM.Plugin`".

3. **Utility projects** are mixed projects that are required, in addition to contract and plugin projects, to run XVSM.NET. Some projects are required on startup while others may be used optionally by a client. Utility projects do not include Plugins, PluginParts or the corresponding contracts, as these elements are located in contract and plugin projects.

   The names of the projects start with "`XVSM`" but do not overlap with the naming guideline of the contract or plugin projects.

4. **Supporting projects** are additional projects that are included in the VisualStudio solution for testing and documentation but, as they do not provide crucial functionality for the space-based framework, are not covered any further in this thesis.

In the following, we are going to take a closer look at the concrete projects in the respective category.

6.1.1 Contract Project Instances

Figure 6.1 illustrates all used contract projects with their mutual dependencies. In the following, we will give a simple overview of the individual contract projects.

XVSM.Contract.Core includes essential XVSM elements that all other XVSM contract projects directly or indirectly depend on. `IXvsmCore` is a plugin contract, which acts as the framework's API endpoint, and can be used to issue XVSM requests and handle their results. With `IXvsmCore` it is possible to access an implementation of the `ICapi` interface, which provides method signatures that developers should be familiar with. *Capi* stands for *Core Application Programming Interface* and was established in the formal model [Cra10]. The `Entry` class, which contains the entry data and its corresponding coordination data, is located in the project along other XVSM-related dependencies of `ICapi`, such as `ITransactionReference` or `IContainerReference`, which are interfaces that represent references to XVSM elements but are solely used for an external representation.

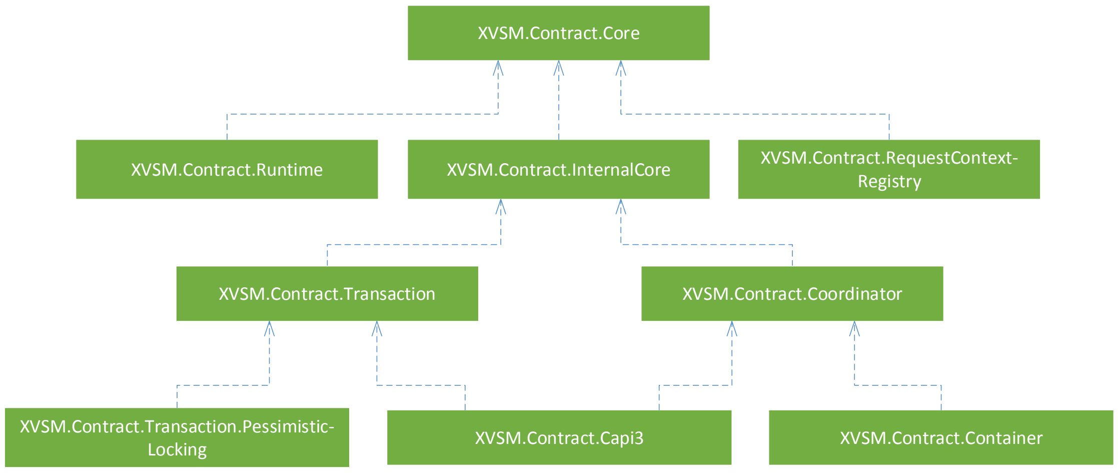

XVSM.Contract.Core
XVSM.Contract.Runtime
XVSM.Contract.InternalCore
XVSM.Contract.RequestContext-Registry
XVSM.Contract.Transaction
XVSM.Contract.Coordinator
XVSM.Contract.Transaction.Pessimistic-Locking
XVSM.Contract.Capi3
XVSM.Contract.Container

Figure 6.1: Contract projects dependencies.

Internally used references are located in the `XVSM.Contract.InternalCore` contract project.

`IRequestContext` and the other elements in the sub-namespace `RequestContext`, such as the plugin contract `IRequestContextFactory`, are used to provide a data container that is shared throughout an XVSM request.

The coordination sub-namespace is used by concrete coordination implementations such as custom coordinators or the default coordinators located in the `XVSM.StandardCoordinators` project.

XVSM.Contract.Runtime provides the contracts that are relevant for the XVSM Core Processor (XP) as described in [Cra10]. `ICapiService` is a plugin contract that is used for arbitrary Capi requests such as Read- or Write-Entries. `ICapiServiceMapper` is used to lookup the correct `ICapiService` plugin corresponding to the concrete request. `IRequestResponseHandler` manages incoming requests, invokes the lookup and execution of the CAPI services through the `ICapiServiceExecutor` and maps the service result to the correct response instance.

XVSM.Contract.RequestContextRegistry is a contract project that was designed to be used throughout several XVSM.NET plugins and therefore directly depends on `XVSM.Contract.InternalCore`. It contains the plugin contract of `IRequestContextRegistry`, which is used to register and lookup instances of `IRequestContext`. Section 6.4.3 gives more details about its mechanism.

XVSM.Contract.InternalCore provides basic internal XVSM elements that are not needed by the client and therefore not included in the `XVSM.Contract.Core` project. The only plugin contracts that are included in the project are `IXvsmReferenceFactory` and `IEntryFactory`. The former can be used to create internal references to address elements in the space. External references such as `ITransactionReference` and `IContainerReference` need to be remapped to instances of `IXvsmReference` in order to be usable for the core processor.

`CoreEntry` is the internal representation of `Entry`, which includes, additionally to the entry data and the coordination data, the internal reference of the entry stored in `IEntryReference`. The latter plugin contract `IEntryFactory` is used to create `CoreEntry` instances. The specified `IEntryReference` interface was used instead of the more general `IXvsmReference` for easier and clearer custom coordinator implementations.

XVSM.Contract.Transaction contains abstract transaction-related elements, such as the `ITransaction` plugin part contract with its supported lifecycle operations *commit* and *rollback*. Transactions are managed and can be retrieved through plugins that implement the `ITransactionRegistry` plugin contract. The `ITransactionFactory` plugin contract provides a method to create a new transaction by providing the space URI and the desired isolation level.

XVSM.Contract.Transaction.PessimisticLocking includes the contracts used to realize the pessimistic transaction implementation in XVSM.NET. Hence, plugin contracts such as `ILockListManager` to handle the locked resources, as well as `IIsolationLevelHandler`, which supports separate plugins for isolation level handling, and the `IIsolationLevelRegistry` that returns the concrete plugin for a provided isolation level, are included in this project.

XVSM.Contract.Coordinator provides all coordinator-related elements that are relevant for coordinator development. It contains the `ICoordinator` plugin part contract, which defines the basic functionality that a coordinator must support, but also provides basic attributes such as the `CoordinatorAttribute` that is used to provide meta information about the coordinator implementation. Coordinators are instantiated through plugins with the `ICoordinatorFactory` plugin contract, which wraps the concrete coordinators with plugin parts of the `IUniversalCoordinator` plugin part contract. The latter plugin parts are used to provide a non-generic coordinator interface and make use of the `ICoordinationContextFactory` during coordination operations to provide context objects to the concrete `ICoordinator` plugin parts. The last plugin contract of this project is the `IImplicitCoordinationDataFactory` plugin contract. It is used by containers to automatically create coordination data for supported coordinators.

XVSM.Contract.Container contains the contracts used to manage containers and entries. Hence contracts for the `IContainer` plugin part as well as the `IEntryStorage` are contained in this project. The `IContainerFactory` and `IContainerRegistry` plugin contracts to instantiate and register container plugin parts are also included in this project.

XVSM.Contract.Capi3 provides access to the CAPI-3 layer representation of XVSM-.NET. It most notably contains the interface provided by the `ICapi3` plugin contract with the CAPI-3 operation result classes. However, the `ISelectionManager` plugin contract is included as well, since its `Select` method has dependencies on transactions as well as coordinations contract projects. The `ISelectionManager` is used to wire the coordinators and orchestrate the entry selection. The `ICoordinationLockHandler` plugin contract is used to manage general locks for the coordinators per operation.

6.1.2 Plugin Project Instances

Figure 6.2 illustrates the XVSM.NET architecture from the plugin framework's point of view. This illustration contains all used regular plugin contracts (blue boxes) and plugin part contracts (green boxes). The plugin contracts of the plugin part contracts' factory plugins are omitted for a simpler visualization. Both solid and dashed directed blue associations denote the dependencies among plugin or plugin part contracts, as the target contract is directly used by the default implementation of the source contract. The solid directed blue and green associations stand for the control flow of Capi Take and Write operations between characteristic plugins and plugin parts. The gray `Client` box denotes the client code from developers that use XVSM.NET and interact with it through the `IXVSMCore` plugin contract. While the blue associations stand for calling plugins, the green associations indicate that a result will be returned from the called plugin or plugin part. Additional plugins that are provided by the plugin framework and used in the plugin implementation are not included in this illustration for reasons of clarity and comprehensibility.

To give a better understanding of XVSM.NET's architecture, we discuss the control flow of typical space operations.

Container creation

This operation starts by creating a request at the client, which uses the `IXvsmCore` plugin to either directly handle the request or transmit it to a remote space, so that it can be handled there. `IXvsmCore` creates an open request at the `IRequestResponseHandler` that is used to correlate requests and responses, and is used to make client requests synchronous. After that, `IXvsmCore` forwards the request to the `ICapiServiceExecutor` plugin. The `ICapiServiceExecutor` uses the `ICapiServiceMapper` to lookup the `ICapiService` plugin responsible to handle the container creation operation request and issues the execution.

Before the container operation can be forwarded to the `ICapi3` plugin, depending on whether a transaction was provided, a transaction is looked up or newly created through the `ITransactionRegistry` and `ITransactionFactory` plugins. In case of a provided transaction, a sub-transaction is created and supplied to `ICapi3`'s create

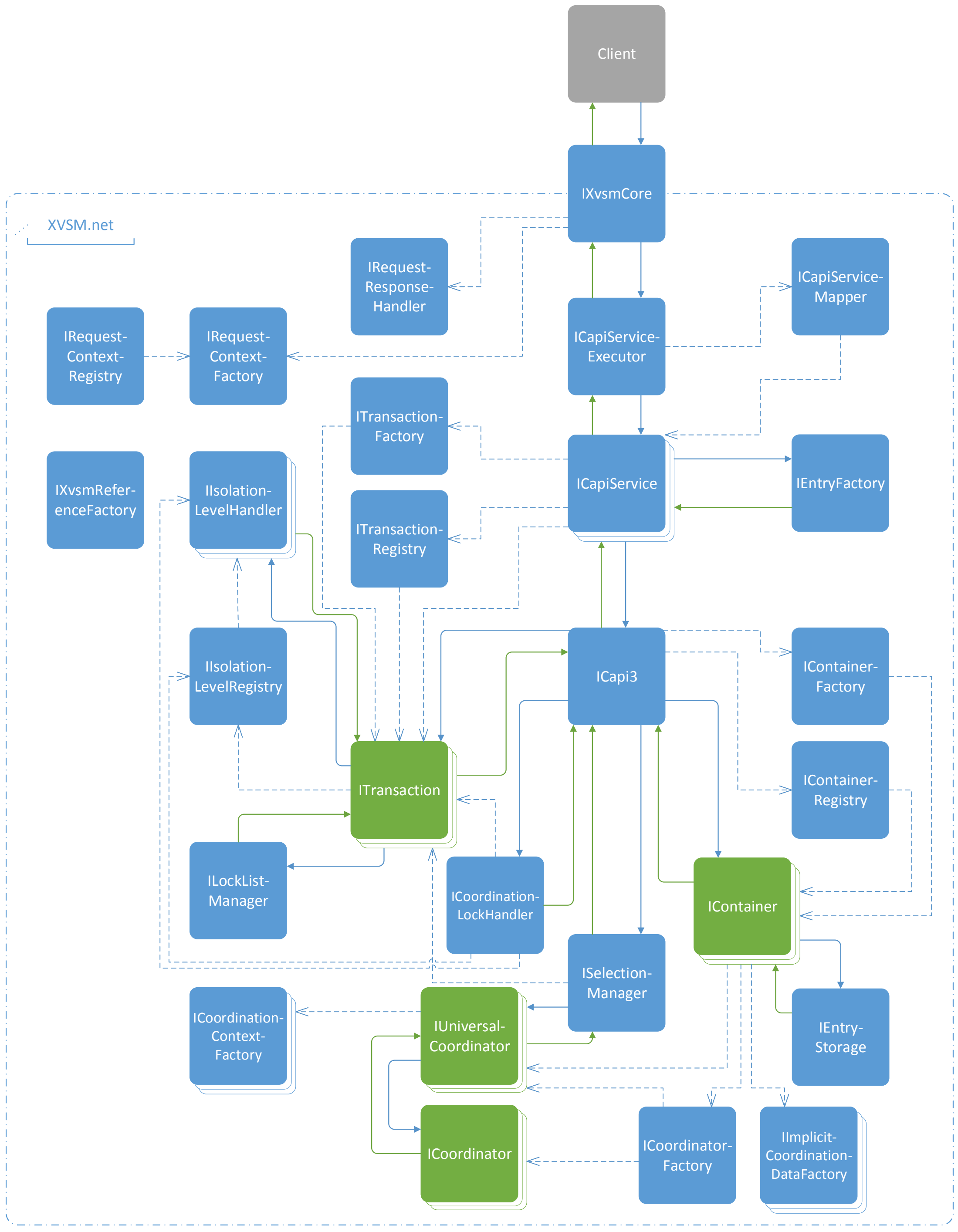


Figure 6.2: Dependency and architectural overview of XVSM.NET's Plugins and Plugin Parts architecture.

container operation with other information from the request, such as the container name, its used coordinators, and maximum size.

`ICapi3` uses the provided transaction for creating the container. This allows the container to be automatically removed in case of a transactional rollback. It also ensures that no other container can be created with the same name. The `ICoordinationLockHandler` is called as well to create initial locks for the coordinator, which can be used for future entry operations. After that, it uses the `IContainerFactory` and `IContainerRegistry` to create and register the container.

The newly created `IContainer` plugin part has access to the `IEntryStorage` plugin to store entries in the future. It creates the coordinators specified in the creation request through the `ICoordinatorFactory` and stores them to register future written entries.

The goal of the container creation operation is now reached, and the control flow returns back all the way to the `IXvsmCore` plugin to resolve the open request of the `IRequestResponseHandler` plugin and give control back to the client, which now should resume execution.

Writing entries

The first part of this operation is very similar to the previously described operation to create containers, with the major difference that another `ICapiService` plugin is used. This plugin is responsible to forward the write operation to the `ICapi3` plugin and uses the `IEntryFactory` plugin to wrap them in an internal entry representation.

When the `ICapi3` plugin is invoked for the write operation, it also creates a sub-transaction to ensure consistency in case of a rollback. It uses the `IContainerRegistry` to lookup the container plugin part the entries should be written to. However, before the entries can be written, the `ICoordinationLockHandler` is invoked again to acquire all relevant coordinator and container locks. If this fails, the write operation is rescheduled to be executed later. In case it succeeds, the entries are written to the container and stored with the `IEntryStorage` plugin. The container automatically adds coordination data to entries that are missing them if possible.

Analogue to the container creation, since the operation's goal is reached, the control flow is given back to the client.

Taking entries

The first part of this operation also does not differ much from the container creation operation until the `ICapi3` plugin is executed to take entries from the container. Like the other operations, it creates a sub-transaction for consistency and looks up the container through the `IContainerRegistry` plugin. It also accesses the `ICoordination-LockHandler` to acquire the container and coordinator locks required for this operation. If this fails, the operation is rescheduled as well. If it is successful, however, the entry references are selected through the `ISelectionManager` plugin. The `ISelection-Manager` orchestrates and uses the `IUniversalCoordinator` plugin parts for the selection, which internally use the `ICoordinator` plugin parts. In the end a list of entry references is returned to the `ICapi3` plugin which transactionally acquires them for the take operation and, if successful, takes them from the `IContainer` plugin part.

After the entries are successfully taken from the container, and the operation is finished, the previously mentioned behavior of giving back the control to the client is performed as well.

6.1.3 Utility Project Instances

There are two utility projects: `XVSM` and `XVSM.StandardCoordinators`. The former contains helper methods to manage an XVSM.NET space. It includes the `XvsmFactory` class to create a new space, and the `SpaceConfig` to provide a fluent interface to configure it.

The latter project contains concrete coordinator classes that are internally mapped to coordinator plugins. They are used to specify coordinators when creating containers, provide coordination data when writing new entries, or specifying selection semantics by a coordinator-specific selector.

Listing 6.1 gives an example on how to use these classes in order to start a space, create a container, and write and take entries.

```
using (var space = XvsmFactory.CreateSpace()
    .ConfigFileBasePath("myConfigLocation")
    .InstantiateSpace()) {
    var c1 = space.Capi.CreateContainer(
        "c1", coordinators: new []{new FifoCoordinator(true)});

    var entry1 = new Entry("e1",
       FifoCoordinator.NewCoordinationData());
    var entry2 = new Entry("e2",
       FifoCoordinator.NewCoordinationData());

    space.Capi.Write(new [] {entry1, entry2}, c1);

    space.Capi.Take<string>(c1, new[]
       {FifoCoordinator.NewSelector(2)});
}
```

Listing 6.1: Example showing basic XVSM.NET usage.

In the following sections we are going to illustrate individual parts of Figure 6.2 to give a better understanding of the behavior of the involved plugins and plugin parts.

6.2 Transactions

In this section we are going to take a deeper look at the transactional model of XVSM and the new concepts and implementation. Figure 6.3 shows the relevant plugin contracts of the transaction mechanism. `ITransactionFactory` and `ITransactionRegistry` plugins are used to create and lookup `ITransaction` plugin parts. Transactions, implemented through the `ITransaction` plugin part contract, provide a consistent access on space references and allow to specify actions that are executed on specific events in the lifecycle of the transaction. For instance, it is possible to provide a reference such as `xvsm://localhost/c1/xyz` and acquire a write lock. The transaction ensures that no other write lock with the same reference exists, if the acquisition was successful.

The `ILockListManager` is internally used by the `ITransaction` plugin part to keep track on the acquired locks and their types. The logic that decides which kind of lock should be acquired is contained in the `IIsolationLevelHandler` plugins, which can be retrieved through the `IIsolationLevelRegistry`. The `ICoordination-LockHandler` plugin contract makes use of `ITransaction` for entry and container operations and, thus, is explained with more detail in Section 6.3.

6.2.1 Transactions in XVSM

Transactions in XVSM are specified in the CAPI-2 layer and are typically used by operations of the CAPI-3 layer for transactional safety. XVSM's transactional model is based on transactions of database theory, and may be described by the *ACID properties*, which stand for *atomicity*, *consistency*, *isolation* and *durability* [BN97]:

Atomicity Operations in a transaction are seen from the outside as a single operation whose resulting state is becoming visible only after committing. In case of an error the intermediate changes are not visible and it appears to the outside as if the operation has not been invoked at all.

Consistency This property guarantees that the state changes of the system will lead from one consistent state to another consistent state when the transaction commits. Intermediate states may be inconsistent but if a transaction aborts or rollbacks, the state will be returned to a previous consistent state.

Isolation Concurrent execution of operations may have an unexpected impact on each other and could lead to race conditions. Transactions should therefore isolate the intermediate state changes, so that operations that are running in different transactions are only applied to consistent states or intermediate states of their own transaction.

Durability After a transaction has been committed, this property guarantees that the changes of the transaction are permanent.

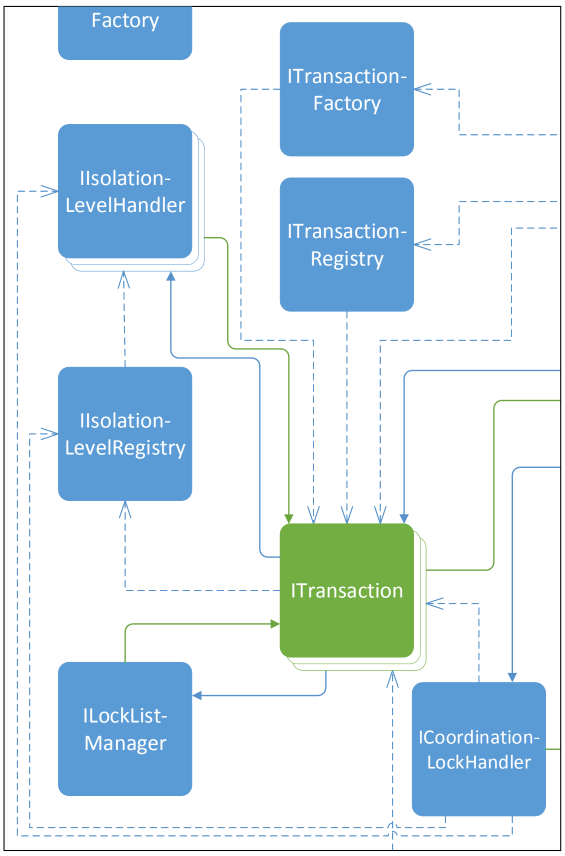


Figure 6.3: Extract of the dependency and architectural overview showing components relevant for transactions.

Similar to the JavaSpaces specification [Sun02], XVSM does not strictly demand persistence for the *durability* property, but rather only requires that committed transactional changes are permanent as long as the middleware core is running.

The formal model describes two different types of transactions: *User transactions* and *sub-transactions*. User transactions are manually created through the CAPI interface. They are used to group domain-related CAPI calls in a single transaction. The internal CAPI operations in XVSM, however, make use of sub-transactions, which are child transactions that are associated with exactly one user transaction. Sub-transactions may have many parallel running *sibling* transactions but they do not support further child transactions. Hence, the formal model uses nested transactions with a fixed depth of two layers, which is illustrated in Figure 6.4.

6.2.2 Nested Transactions

XVSM.NET does not differentiate between the transaction types like the formal model, but uses a closed nested transaction model instead that supports unrestricted transaction

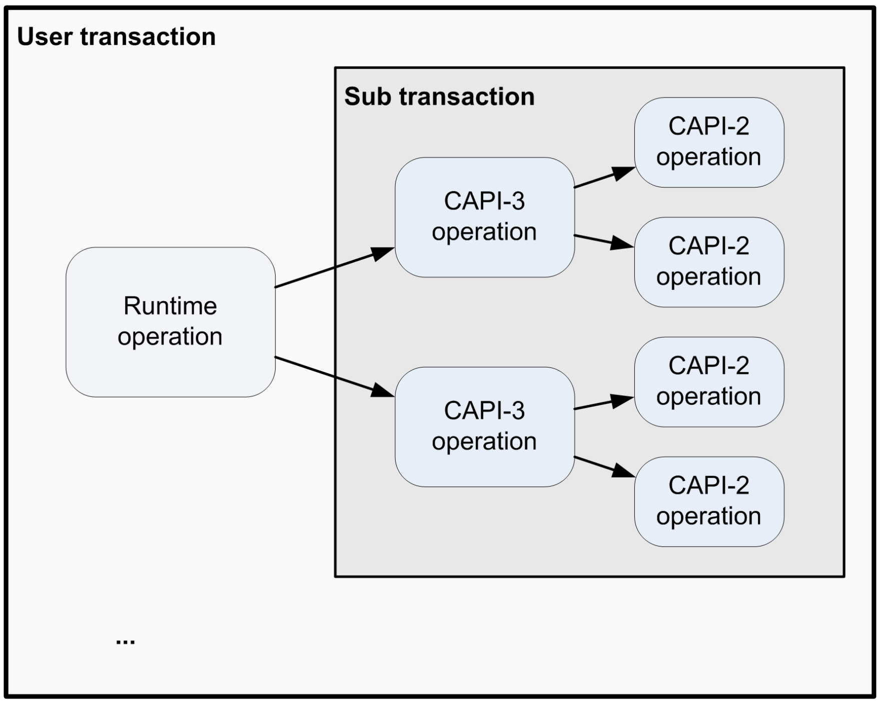


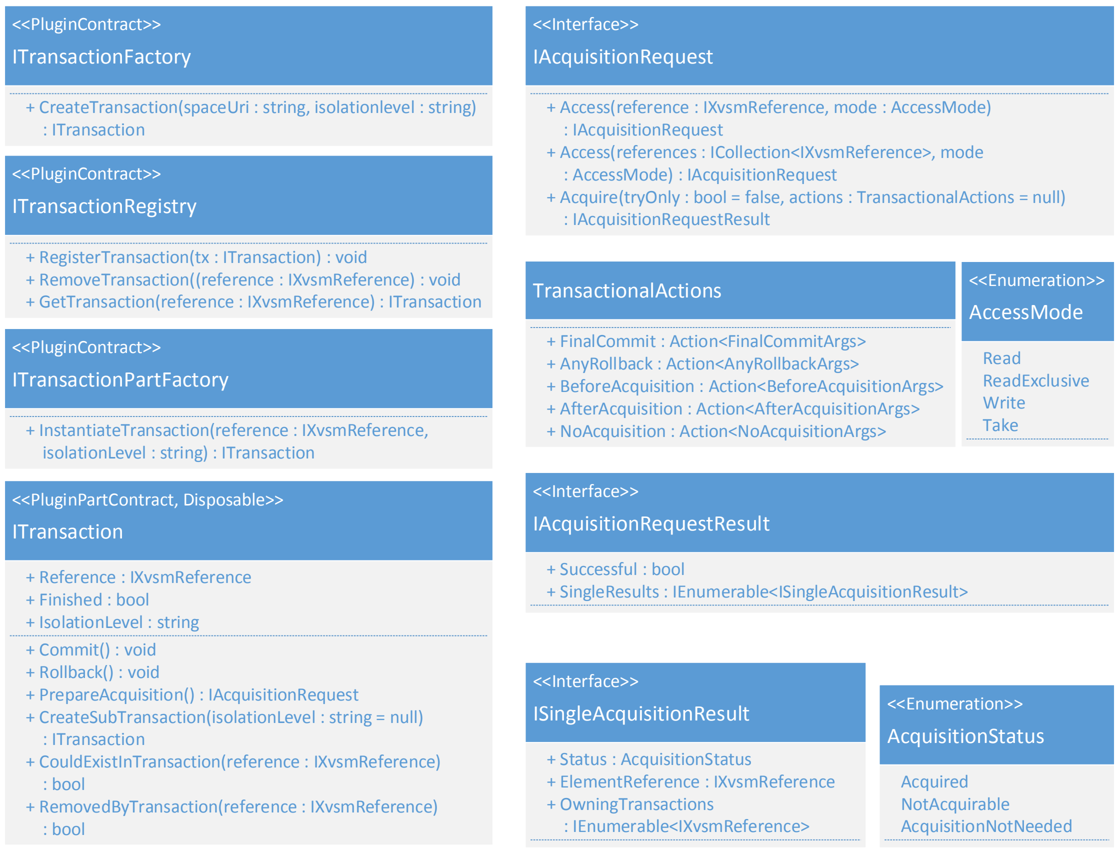
Figure 6.4: Transactions in the XVSM formal model [Cra10].

levels to incorporate both transaction types within a single type. User transactions correspond to top-level transactions in the hierarchy, and sub-transactions correlate with children of the top-level transactions. Transactions with an even lower level are not defined in the formal model as they are not required for the core XVSM functionality but they are possible in XVSM.NET. Unrestricted levels provide more flexibility with finer-grained partial rollbacks and help to further decouple the plugins, as plugins now may use an additional transaction level and therefore may be unaffected of partial rollbacks of other plugins. For instance, in MozartSpaces the pre- and post-aspects as well as the corresponding CAPI-3 operation all make use of the same sub-transaction instance. If an error occurs in one of the pre-aspects that results in a rollback of the sub-transaction, previous pre-aspects will be rollbacked as well and the further execution of the operation will not be possible. With unrestricted transaction levels it is possible to use an additional layer of child transactions that may be rollbacked without forcing the whole operation to fail and rollback.

6.2.3 Transaction Contracts

In this sub-section we are taking a look at the concrete contracts of the XVSM.NET transaction model. Even though these contracts were developed with the pessimistic locking model in mind, they were designed to be still flexible enough for a possible optimistic locking approach in the future. Hence, in this sub-section we are focusing on the contracts and not on their concrete implementation in plugins and plugin parts (cf. Section 6.2.8).

`ITransaction` is the central contract that is used for all transaction-related operations. Its main purpose is to acquire access to space references and register actions that are executed on transactional events. This is realized through the `IAcquisitionRequest` interface and the `TransactionalActions` class. `ITransactionFactory` provides functionality to create top-level `ITransaction` instances, which are then registered in the `ITransactionRegistry` for lookup purposes. *Sub-transactions*, on the other hand, are not created by the `ITransactionFactory` but instead are instantiated through a parent `ITransaction` instance with its `CreateSubTransaction` method. Details of the contracts are shown in Figure 6.5. The stereotypes *«PluginContract»*, *«PluginPartContract»*, and *«Disposable»* are used in this chapter to simplify the UML class diagrams by depicting plugin contracts, plugin part contracts, and `IDisposable` interfaces.



<<PluginContract>>
ITransactionFactory
+ CreateTransaction(spaceUri : string, isolationlevel : string) : ITransaction
<<PluginContract>>
ITransactionRegistry
+ RegisterTransaction(tx : ITransaction) : void
+ RemoveTransaction((reference : IXvsmReference) : void
+ GetTransaction(reference : IXvsmReference) : ITransaction
<<PluginContract>>
ITransactionPartFactory
+ InstantiateTransaction(reference : IXvsmReference, isolationLevel : string) : ITransaction
<<PluginPartContract, Disposable>>
ITransaction
+ Reference : IXvsmReference
+ Finished : bool
+ IsolationLevel : string
+ Commit() : void
+ Rollback() : void
+ PrepareAcquisition() : IAcquisitionRequest
+ CreateSubTransaction(isolationLevel : string = null) : ITransaction
+ CouldExistInTransaction(reference : IXvsmReference) : bool
+ RemovedByTransaction(reference : IXvsmReference) : bool
<<Interface>>
IAcquisitionRequest
+ Access(reference : IXvsmReference, mode : AccessMode) : IAcquisitionRequest
+ Access(references : ICollection<IXvsmReference>, mode : AccessMode) : IAcquisitionRequest
+ Acquire(tryOnly : bool = false, actions : TransactionalActions = null) : IAcquisitionRequestResult
TransactionalActions
+ FinalCommit : Action<FinalCommitArgs>
+ AnyRollback : Action<AnyRollbackArgs>
+ BeforeAcquisition : Action<BeforeAcquisitionArgs>
+ AfterAcquisition : Action<AfterAcquisitionArgs>
+ NoAcquisition : Action<NoAcquisitionArgs>
<<Enumeration>>
AccessMode
Read
ReadExclusive
Write
Take
<<Interface>>
IAcquisitionRequestResult
+ Successful : bool
+ SingleResults : IEnumerable<ISingleAcquisitionResult>
<<Interface>>
ISingleAcquisitionResult
+ Status : AcquisitionStatus
+ ElementReference : IXvsmReference
+ OwningTransactions : IEnumerable<IXvsmReference>
<<Enumeration>>
AcquisitionStatus
Acquired
NotAcquirable
AcquisitionNotNeeded

Figure 6.5: UML class diagram of the transaction-related contracts.

ITransactionFactory

`ITransactionFactory` is used to create top-level transactions, which is the responsibility of the XVSM runtime in CAPI-4 through the `CreateTransaction` method. The `spaceUri` parameter is used for the creation of the transaction's space reference. The factory is responsible to use a unique reference in the space. The default implementation of `ITransactionFactory` therefore makes use of a numeric integer counter to create a space unique identifier which is then provided to the `IXvsmReferenceFactory`, along with the space URI, to create the unique reference. The `isolationLevel` parameter is the string representation of the transactional isolation level. XVSM.NET also provides a type-safe access to the isolation levels through the `XvsmConstants.IsolationLevel` class in the core contract project. Nevertheless, the isolation level is always represented as a string in the contracts to ensure compatibility for additional isolation levels in the future. The `ITransactionFactory` uses an instance of the `ITransaction` plugin part factory for quick instantiations of `ITransaction` plugin parts with its method `InstantiateTransaction`.

ITransactionRegistry

The `ITransactionRegistry` contract provides methods to register, retrieve and remove transactions, which is also the responsibility of the XVSM runtime. These methods are not used for sub-transactions.

ITransactionPartFactory

`ITransactionPartFactory` is a plugin part factory that has direct access to an implementation of the `ITransaction` plugin part contract and is therefore responsible for creating new `ITransaction` instances by providing all plugin dependencies.

ITransaction

The `ITransaction` plugin part contract is a central and important contract of XVSM-.NET. It is used in the majority of the `ICAPI3` methods to provide transactional safety for its operations. `ITransaction` was designed to be used for locking space references and registering actions that are executed on certain transaction events.

`CreateSubTransaction` instantiates a new sub-transaction with the same `ITransaction` plugin part contract. Because the containing contract of this method is no plugin part factory, the plugin framework will not automatically transform the instantiated object into a plugin part. Hence, depending on the concrete `ITransaction` plugin part implementation, the instantiated object is a plugin part or a simple object, implementing the plugin part contract. Since the top-level transaction is guaranteed to be a plugin part, due to its instantiation through the `PluginPartFactory`, it is possible to create a custom plugin aspect that transforms the object result of this method to a plugin part as well, assuming plugin aspects are not prevented by the configuration of the plugin framework. The optional `isolationLevel` parameter can be used to specify a different isolation level for the sub-transaction. By default the sub-transaction will inherit the isolation level of its parent transaction. Even though this parameter is only rarely used in the default implementations, it was added to support coordinator lock semantics and will be discussed in Section 6.3.6.

The contract's `Commit` method is used to finalize the transaction recursively along with its sub-transactions. When a top-level transaction is committed, all *final-commit actions* that were provided in the `TransactionalActions.FinalCommit` property in the transaction hierarchy are invoked. All locks that were acquired in the transaction hierarchy should be either transferred to the parent transaction or, for top-level transactions, are propagated to the environment and will be accessible for all transactions. The `Rollback` method is working in a similar way to the `Commit` method and recursively rollbacks the transaction and its sub-transactions when being invoked. However, whenever a transaction is rollbacked, its own registered *any-rollback actions*, provided through `TransactionalActions.AnyRollback`, and those of already committed sub-transactions are invoked. For instance, if a deeply nested, already committed sub-transaction with a hierarchical depth of 5 has an any-rollback action that deletes a space container, the container will still be deleted if a transaction with a depth of 2 from the same hierarchy is rolled back. The `ITransaction` plugin part contract extends from *System.IDisposable* to control the transaction's lifecycle through C#'s using-statement. The transaction's `Commit` and `Rollback` methods are used to manually control its lifecycle. However, if the using-statement is left without invoking these lifecycle methods, it automatically invokes the `Rollback` method of the transaction. Listing 6.2 shows a typical usage of an `ITransaction` plugin part.

```
using (var stx = tx.CreateSubTransaction())
{
    stx.PrepareAcquisition()
        .Access(entryReference, Write)
        .Acquire();

    stx.Commit();
}
```

Listing 6.2: Example showing basic transaction operations.

`PrepareAcquisition` returns an instance of an `IAcquisitionRequest` object that is aware of the current transaction and provides acquisition functionality in the transaction's context. This is a variant of the builder pattern [Hel+00] that allows to specify the acquisition details through a fluent interface before finally executing the acquisition.

The two methods `CouldExistInTransaction` and `RemovedByTransaction` are used to gain insight into the current transactional state of the provided element reference. The former method returns `true` if there is a possibility that the concerning element exists or could be rollbacked to exist again in the transaction's context. In any other case it should return false. The latter method, however, returns true if the concerning element is guaranteed to be removed when the transaction hierarchy is committed and false otherwise. At first glance, the methods might seem redundant and negations of each other, but when we look closely we should find that there is a slight difference. The first method judges with respect to the global transactional state of the element but the second method only concerns the current transaction hierarchy.

The ITransaction plugin part contract also provides several read-only properties that provide information about the transaction. Reference returns the transaction's own space reference and IsolationLevel can be used to retrieve the string representation of its isolation level. The boolean Finished property only returns true if the transaction does not support further invocation of the Commit or Rollback methods.

AccessMode

The *access modes* that are defined in the AccessMode enumeration specify the different transactional acquisition types and correspond to the three basic XVSM operations read, write and take with the addition of ReadExclusive. ReadExclusive was added to provide an access mode that demands exclusive read access in contrast to the shared read access mode (Read) and without implying that the element was added (Write) or removed (Take) from the space.

IAcquisitionRequest

IAcquisitionRequest is used to prepare a single transactional acquisition request with its belonging transaction. The Access method signatures are used to specify the target references and their required access modes. After all target references have been set, the Acquire method may be called to carry out the acquisition request atomically and return with an IAcquisitionRequestResult object. The method contains two optional parameters, tryOnly and actions. The boolean parameter tryOnly may be used to simulate the acquisition without making any transactionally visible changes and influencing other transactions. This parameter was added for performance improving algorithms that check the acquisition possibility of elements without carrying out the acquisition. The second parameter actions is an object of the class Transaction-alActions and may provide custom actions for transactional events. These actions could be stored for future commit or rollback invocations, or are directly executed within the acquisition request.

IAcquisitionRequestResult

The IAcquisitionRequestResult interface is a simple container for the individual acquisition requests and therefore only contains two properties: The Successful property indicates whether all requested individual acquisitions have been obtained, or not. Detailed information about the individual acquisitions is stored in the SingleResults property as IAcquisitionSingleResult objects.

AcquisitionStatus

The `AcquisitionStatus` enumeration is used to specify the outcome of the individual acquisitions. It provides the following items:

Acquired: The element has been successfully acquired with the specified access modes.

NotAcquirable: It was not possible to acquire the element with the specified access mode.

AcquisitionNotNeeded: The element has not been acquired because it is already accessible with the specified access mode in this transaction.

IAcquisitionSingleResult

The result of every single acquisition is stored in a separate `IAcquisitionSingleRe-sult` instance. Its `Status` property provides an enum of the type `Acquisition-Status` that represents the outcome of the element acquisition and the property `ElementReference` contains the reference of the corresponding element. In the case of failed acquisition, the property `OwningTransactions` contains the reference of the transactions that hold the locks for the element.

TransactionalActions

Instances of the `TransactionalActions` class are used to register C# `Action<>` objects to execute custom logic on certain transactional events. This is used, for instance, to fully delete an entry of a container only after committing the top-level transaction that is used for a take operation.

The `TransactionalActions` class contains five properties that may be used to provide custom handling of transactional events. The `FinalCommit` and `AnyRollback` properties were already mentioned in the `ITransaction` section. An action registered in the `FinalCommit` property will only be invoked when a top-level transaction commits. If any other transaction in the hierarchy commits instead, the invocation will be postponed for the commit of the top-level transaction. When any transaction in the hierarchy rollbacks, on the other hand, actions registered in the `AnyRollback` properties of the rollbacked transactions are invoked right away. The actions in the `BeforeAcquisition` and `AfterAcquisition` properties are only invoked after a compatibility check of the `IAcquisitionRequest`'s required locks and the lock availability. The action in the `BeforeAcquisition` is then invoked right before the locks are acquired, whereas the action of the `AfterAcquisition` property is invoked afterwards. Provided that the `IAcquisitionRequest`'s required locks cannot be obtained, then the action of the `NoAcquisition` property is invoked in return. All actions are invoked with an individual class parameter that might contain metadata of the concrete event. Yet, the parameters are not used in the current XVSM.NET implementation and are only included for an easier future integration. The advantage of using these events in a

`TransactionalActions` object is that the actions should be executed within a special locked transactional environment and may be used to protect against side-effects from parallel running transactions. However, this relies to a great degree on the concrete implementation of `ITransaction` and `IAcquisitionRequest`.

6.2.4 Locking Contracts

In this sub-section, we are now showing the locking constructs that are used by the `PessimisticTransaction` plugin implementation. In the spirit of the XVSM.NET architecture, the locking-related parts were also developed in plugins over contracts. Figure 6.6 illustrates the locking-relevant contract elements from the `XVSM.Contract.Transaction.PessimisticLocking` namespace:

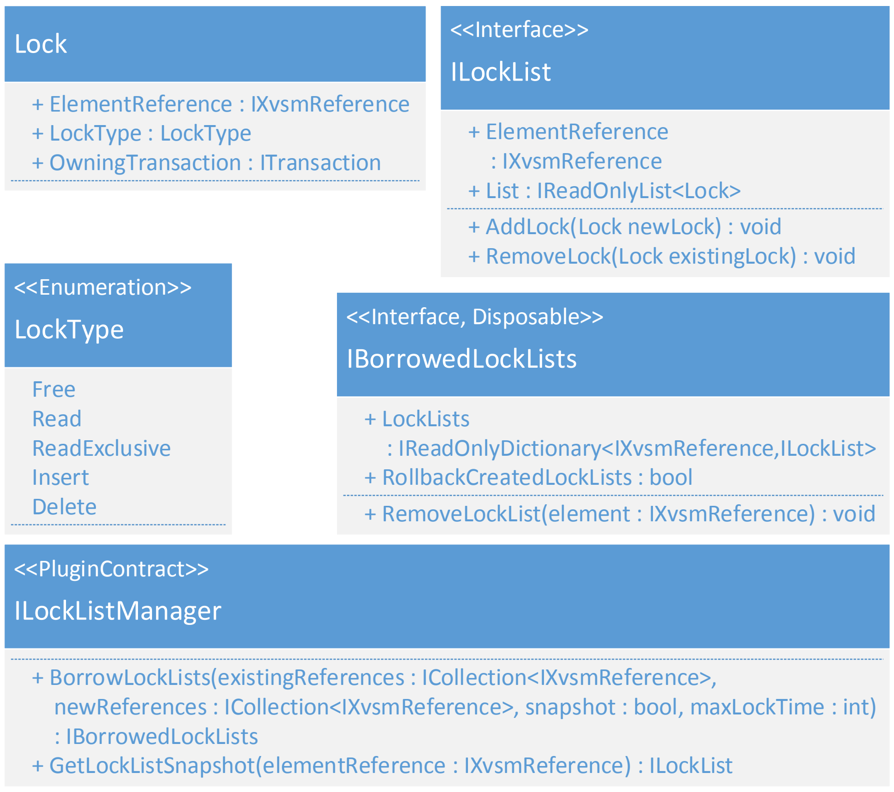

Lock
+ ElementReference : IXvsmReference
+ LockType : LockType
+ OwningTransaction : ITransaction
<<Interface>>
ILockList
+ ElementReference : IXvsmReference
+ List : IReadOnlyList<Lock>
+ AddLock(Lock newLock) : void
+ RemoveLock(Lock existingLock) : void
<<Enumeration>>
LockType
Free
Read
ReadExclusive
Insert
Delete
<<Interface, Disposable>>
IBorrowedLockLists
+ LockLists : IReadOnlyDictionary<IXvsmReference,ILockList>
+ RollbackCreatedLockLists : bool
+ RemoveLockList(element : IXvsmReference) : void
<<PluginContract>>
ILockListManager
+ BorrowLockLists(existingReferences : ICollection<IXvsmReference>, newReferences : ICollection<IXvsmReference>, snapshot : bool, maxLockTime : int) : IBorrowedLockLists
+ GetLockListSnapshot(elementReference : IXvsmReference) : ILockList

Figure 6.6: UML class diagram of locking related interfaces and contracts.

Lock & LockType

The `Lock` class is used to represent a single transactional lock on a space reference that has been acquired through a transaction. Every instance of the `Lock` class stores the reference of the target element, as well as the type of the lock, also referred to as *lock type*, in the read-only properties `ElementReference` and `LockType`. The third property `OwningTransaction` is of type `ITransaction` and contains the transaction that acquired the lock. However, this third property is not set for locks that have the `LockType` property set to `Free`, since they are accessible by any transaction and have no owner. Due to implementation details regarding concurrency, it was decided to explicitly set the lock type to `Free` instead of having no lock type set. The other elements of the `LockType` enumeration `Read`, `ReadExclusive`, `Insert` and `Delete` correspond to the access modes from the transaction contracts (see Section 6.2.3) and define the resulting locks of the acquisition operations. Please note that the `Take` access mode leads to a `Delete` lock type. The concrete semantics of the different lock types in combination with the isolation levels are discussed in Section 6.2.5. All lock types with the exception of `Free`, however, have an owning transaction that must be always set in the `Lock` instance.

For better readability, we will use the terms *free-lock*, *read-lock*, *readexclusive-lock*, *insert-lock*, and *delete-lock* from here on for *Lock* instances with the corresponding `LockType` value.

All properties of the `Lock` class are read-only, thus the instances of the class are immutable. If an acquisition with a different access mode, hence different lock type, is processed, a new `Lock` instance has to be created.

ILockList

As instances of `Lock` are immutable and may only contain one lock type and one owning transaction, it must be possible to have multiple `Lock` instances associated with the same space reference. Otherwise, it would be unfeasible to fully define the lock state of a space reference. For instance, in the example in Listing 6.3 a top-level transaction `tx` and its child transaction `stx` both access the same element in the space. With only a single `Lock` instance it would not be possible to cleanly rollback the acquisition made through the `stx` transaction as the prior owning transaction would have been overridden. This also applies for shared read-locks by transactions from a different transaction hierarchy.

```
// Up till now, element ref1 has only a free-lock associated.
tx.PrepareAcquisition().Access(ref1, Read).Acquire();
var stx = tx.CreateSubTransaction();
stx.PrepareAcquisition().Access(ref1, Take).Acquire();
stx.Rollback();
```

Listing 6.3: Acquisition example of a transaction hierarchy with two levels.

Hence, multiple immutable instances of the `Lock` class are used to describe the full lock state of a single element in the space and they are accumulated as *lock lists* by the `ILockList` data structure.

`ILockList` is an interface that provides the methods `AddLock` and `RemoveLock` for simple lock list manipulations. It also provides the read-only property `ElementRef-erence` to retrieve the space reference of the concerning element, and the read-only property `List` to access all the locks associated with the element. New locks that are added through the `AddLock` method are appended at the end of the list. Thus, the order of the locks represent their time of acquisition.

ILockListManager & IBorrowedLockLists

The instance of the `ILockListManager` plugin contract, called *lock list manager*, is the central point that holds and provides the lock lists. The contract provides two methods to retrieve lock lists, `BorrowLockLists` and `GetLockListSnapshot`. The `GetLockListSnapshot` method is the simpler method of the two, as it only returns a single existing lock list that represents the lock state of the provided element reference. The returned lock list is a *snapshot*, and thus immutable and could show outdated lock information. After the lock list has been retrieved, additional locks might have been added or existing locks might have been removed from the lock list. However, these changes will not be reflected in the snapshot. This retrieval method is therefore only used for quick and uncritical checks of the state of the lock list. This is used for optimistic checks where a computationally expensive algorithm should be executed only if there is a chance that a later acquisition succeeds.

The method `BorrowLockLists`, on the other hand, grants a more sophisticated access to the lock lists. With this method, the lock lists are accessed in a bulk mode by providing multiple references at once. Hence, already existing lock lists are retrieved by providing the corresponding space references in the `existingReferences` collection, whereas lock lists for elements that do not yet have a lock list must be separately provided in the `newReferences` collection. This differentiation was a design decision that serves as a simplification for concurrency handling. The `snapshot` parameter of the method defines whether the resulting lock lists should be exclusively *borrowed* (`false`), or if only snapshot copies should be retrieved (`true`). If set to true, the method provides similar semantics as the `GetLockListSnapshot` method. It allows to decide whether an acquisition would be possible without actually executing it. The last and optional parameter `maxLockTime` allows to override the configured maximum timeout in milliseconds when entering a lock.

The result of this method is an instance of the disposable `IBorrowedLockLists` interface, which contains the requested existing lock lists, as well as new lock lists for the specified new references. The lock lists are all stored in the property `LockLists` of a dictionary type and are accessible by the element's reference as the key. The `IBorrowed-LockLists` interface also provides the method `RemoveLockList` to remove the lock list with the specified element reference from the lock list manager. The method is used when the corresponding element is removed from the space. The concrete removal of the lock lists will be delayed until the *IBorrowedLockLists* instance is disposed. Since new lock lists are automatically created by providing new references to the `BorrowLockLists` method, the `IBorrowLockLists` interface provides a possibility to undo the creation by setting the `RollbackCreatedLockLists` boolean property to `true`. As soon as the

`IBorrowedLockLists` instance is disposed, the newly created lock lists are removed again if this property is set.

The lock list manager is responsible to guarantee exclusive access to the lock lists by the current `IBorrowedLockLists` instance. No other thread should be able to access the same existing lock lists (cf. Figure 6.6). If a concurrent access occurs, the thread should be blocked and depending on the configuration an exception can be raised. The concrete thread-safe implementation is presented in Section 6.2.7. This exclusive access lasts until the `IBorrowLockLists` instance is disposed by the caller. However, this only applies to existing lock lists and not newly created ones. The protection of newly created lock lists is complex and error-prone in combination with rollbacked transactions. This issue will be discussed in Section 6.2.8, as it requires more information about specific aspects of the concrete implementation.

6.2.5 Isolation Level Contracts

The *isolation level contracts* are used to provide the semantics that connect the access modes with the full lock state of an element in the space. Figure 6.7 illustrates elements of the isolation level contracts.

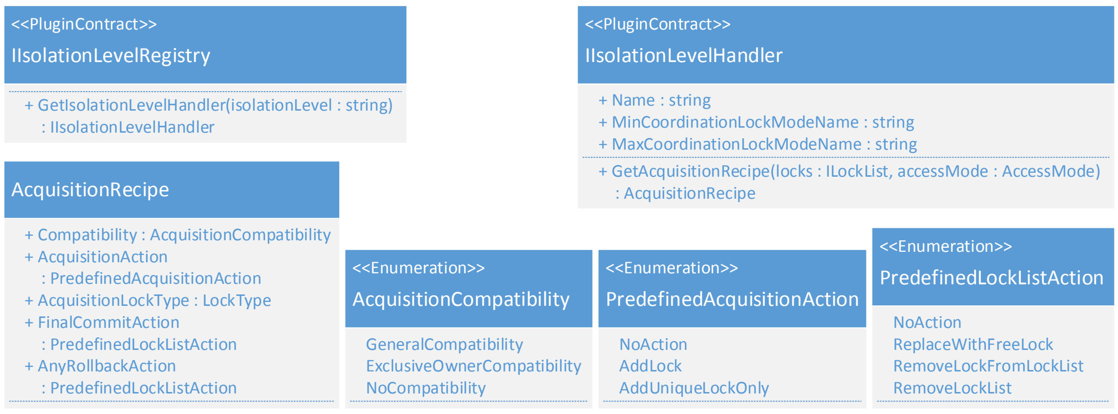

<<PluginContract>>
IIsolationLevelRegistry
+ GetIsolationLevelHandler(isolationLevel : string) : IIsolationLevelHandler
<<PluginContract>>
IIsolationLevelHandler
+ Name : string
+ MinCoordinationLockModeName : string
+ MaxCoordinationLockModeName : string
+ GetAcquisitionRecipe(locks : ILockList, accessMode : AccessMode) : AcquisitionRecipe
AcquisitionRecipe
+ Compatibility : AcquisitionCompatibility
+ AcquisitionAction : PredefinedAcquisitionAction
+ AcquisitionLockType : LockType
+ FinalCommitAction : PredefinedLockListAction
+ AnyRollbackAction : PredefinedLockListAction
<<Enumeration>>
AcquisitionCompatibility
GeneralCompatibility
ExclusiveOwnerCompatibility
NoCompatibility
<<Enumeration>>
PredefinedAcquisitionAction
NoAction
AddLock
AddUniqueLockOnly
<<Enumeration>>
PredefinedLockListAction
NoAction
ReplaceWithFreeLock
RemoveLockFromLockList
RemoveLockList

Figure 6.7: UML class diagram of the isolation-related contracts.

IIsolationLevelRegistry

The plugin contract `IIsolationLevelRegistry` is a registry that is used as a central point to lookup *isolation level handlers* with the `GetIsolationLevelHandler` method by providing the string representation of the isolation level. The method then returns an instance of `IIsolationLevelHandler`. Every transaction or sub-transaction stores a reference to an isolation level handler that corresponds to the string representation provided from the transaction instance. When a top-level transaction is created, the globally configured isolation level string representation is used by default, but it can be overridden with a different isolation level if it is provided as an argument. When a sub-transaction is created, the isolation level is inherited from its parent by default, however, it is also possible to specify a custom isolation level with its instantiation method. Thus, transactions of the same transaction hierarchy can have different isolation levels. This mechanism is used when a custom isolation level is provided for CAPI operations. This allows, for instance, to execute two read operation with different isolation levels.

IIsolationLevelHandler & AcquisitionRecipe

The `IIsolationLevelHandler` plugin contract provides the concrete semantics of an isolation level in XVSM.NET. XVSM.NET comes with multiple default implementations that are presented in Section 6.2.6.

Its `Name` property corresponds to the transactional isolation level's string representation and is equal to the lookup string that is used in the registry for the `IIsolation-LevelHandler`. The two string properties `MinCoordinationLockModeName` and `MaxCoordinationLockModeName` define the lower and upper boundary for *coordination locks* in the `ICoordinationLockHandler`. Coordination locks are acquired through the `ICoordinationLockHandler` plugin for coordinator operations such as read, take, or write on the involved coordinators and container. The lock mode boundaries are used to alter the behavior of the `ICoordinationLockHandler` plugin. The semantics of the lock modes are shown in Section 6.3.6. The only method of the `IIsolationLevelHandler` plugin contract is the `GetAcquisitionRecipe` method, which is used to return a kind of *recipe* that contains the semantics for an acquisition with an instance of the class `AcquisitionRecipe`. This class is used by the `ITransaction` implementation when an acquisition request is handled to decide whether an element reference can be acquired with the provided access mode. If an acquisition is possible, the `AcquisitionRecipe` class also provides rules that are interpreted by the `ITransaction` implementation to modify the lock list and register transactional actions.

An instance of the `AcquisitionRecipe` class is used as a container that holds the five properties that are used for the lock acquisition. The `Compatibility` property is an enumeration of the type `AcquisitionCompatibility` and is used to define constraints of the lock acquisition with the following values: `GeneralCompatibility`, `ExclusiveOwnerCompatibility` and `NoCompatibility`. While `GeneralCom-patibility` denotes that there are no compatibility constraints on the lock acquisition, `NoCompatibility`, on the other hand, implies that the lock acquisition is not possible under any circumstances. This only applies to locks that have an owner, which excludes locks with the `Free` lock type. In between the two values lies `Exclusive-OwnerCompatibility`, which restricts the lock acquisition to the owner of all the locks in the lock list. This means that every lock must be either owned directly or through its ancestors by the transaction that tries to carry out the acquisition, or otherwise the acquisition would not be possible. The type of the `AcquisitionAction` property is the enumeration with the name `PredefinedAcquisitionAction` and is used to describe how the acquisition of the lock should be performed. It contains the values `NoAction`, `AddLock` and `AddUniqueLockOnly`. The first value `NoAction` denotes that no acquisition action should be performed. This value is used if the acquisition is not compatible at all or if the acquisition action is just not required. The value `AddLock` represents that a new lock with the lock type, specified in the property `AcquisitionLockType`, should be created and added to the lock list to finally carry out the lock acquisition. The final value `AddUniqueLockOnly` is semantically equal to `AddLock` with the additional constraint that the action will only be performed if the resulting lock would be unique in the lock list in terms of lock type and owning transaction. This is used for shared locks that would otherwise lead to duplicates in the lock list. As already mentioned, the `AcquisitionLockType` property contains the lock type that is used for a possible newly created lock. Hence, the instance of `IIsolationLevelHandler` is responsible to correctly map the access mode to the lock type of the lock. The two remaining properties `FinalCommitAction` and `AnyRollbackAction` are used to register predefined actions that are only executed on the corresponding event. The action in the `FinalCommitAction` property will be executed when the top-level transaction in the transaction hierarchy commits, while the `AnyRollbackAction`, on the other hand, will be executed when any transaction of the transaction hierarchy will rollback. Hence, these actions are executed analogously to the actions in `TransactionalActions` presented in Section 6.2.3. However, in this case the actions are limited by the values of the enumeration `PredefinedLockLis-tAction` to simplify the implementation of the `IIsolationLevelHandler` plugin contract. The possible predefined actions are `NoAction`, `ReplaceWithFreeLock`, `RemoveLockFromLockList` and `RemoveLockList`. The value `NoAction` trivially describes that no operation will be performed. With the `ReplaceWithFreeLock` value an action is executed that removes the previously created lock and adds a new free-lock instead. The value `RemoveLockFromLockList` describes an action that is equivalent to the previously stated value `ReplaceWithFreeLock` with the difference that it excludes the subsequent addition of the free-lock. This value should only be used if the lock list contains at least two locks because an empty lock list should be avoided. The final value `RemoveLockList` is used to invoke the `RemoveLockList` method on the corresponding `IBorrowedLockList` instance to remove the lock list for good.

6.2.6 Isolation Level Implementation

XVSM.NET comes with four default plugins that implement the `IIsolationLevel-Handler` plugin contract and correspond to the *read uncommitted*, *read committed*, *repeatable read*, and *serializable* isolation levels from the database theory. The plugins are referred to as *ReadUncommittedPlugin*, *ReadCommittedPlugin*, *RepeatableReadPlugin* and *SerializablePlugin* to tell them apart from the conceptual isolation levels. The main task of these plugins is to provide instances of `AcquisitionRecipe` that correspond to their isolation level. They also provide minimum and maximum coordination lock modes, which will be covered later in Section 6.3.6. Hence, we will solely focus on the creation of these recipes in this sub-section. UML state diagrams will be used as an illustration of the allowed lock acquisitions and transaction operations. The plugins are discussed in the order of their isolation level, starting with the one with the lowest guarantee of consistency [Gra+76].

ReadUncommittedPlugin

The ReadUncommittedPlugin is located in the `XVSM.Plugin.IsolationLevelHandler.ReadUncommitted` namespace and corresponds to the read uncommitted isolation level. As the lowest isolation level of the included four, it does not provide protection against inconsistencies due to effects of other concurrent operations but it still protects space references from entering illegal lock states. For instance, it is not possible to successfully issue a take-acquisition on the same element twice.

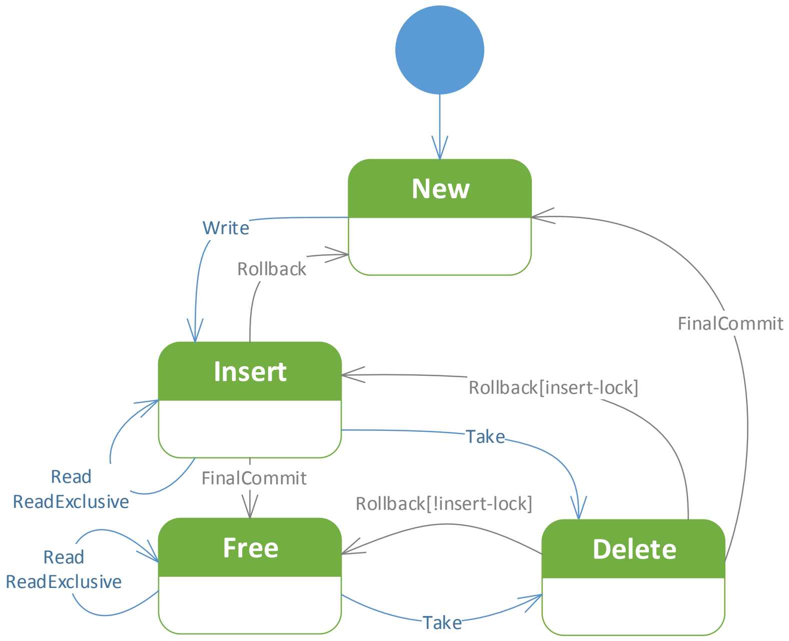


Figure 6.8: UML state diagram illustrating the lock state of a space element for the read uncommited isolation level.

Figure 6.8 shows the possible logical lock states of a space element for the ReadUncommittedPlugin. It uses logical states that do not explicitly exist in the implementation in terms of single `LockType` values, but they represent the compound state of a space element's lock list. For instance, the *New* state corresponds to a removed or non-existing lock list. Every other state in the state diagram corresponds to a lock list with one or more locks. Only one lock of the lock list, however, defines the state: The lock with the highest priority `LockType` value. This leads to the following priority order of locks: free-lock, read-lock, readexclusive-lock, insert-lock, and delete-lock. Hence, a lock list with a free-lock and a delete-lock corresponds to the *Delete* state.

There are two types of state transitions in the state diagram: *acquisition transitions* (blue color) and *transaction transitions* (gray color). Acquisition transitions visualize acquisition operations by the following events: *Read*, *ReadExclusive*, *Write* and *Take*. These events correspond to the access modes that are provided to `IAcquisitionRequest` instances (cf. Section 6.2.3). The transaction transitions, on the other hand, correspond to transitions that change the transactions' own states. The events *FinalCommit* and *Rollback* belong to these transitions. The *FinalCommit* event corresponds to committing the top-level transaction in the transaction hierarchy. The *Rollback* event occurs when the transaction is rollbacked that currently holds the defining lock of the state. It is possible to have multiple *rollback transitions* in a row when a transaction is rollbacked that holds multiple locks. For instance, a rollback of a transaction that holds both an insert-lock and an delete-lock of the same lock list would lead to two rollback transitions from the state *Delete* over the state *Insert* to the final state *New*. Please also note that the guard of the rollback transition from the *Delete* state ensures that the lock list also contains an insert-lock. This is important for the transitions in the state diagram as no ambiguity is allowed. The rollback transitions may be seen as *undo* operations of the acquisitions that led to the current state. Hence, almost all rollback transitions are implemented by the use of the `RemoveLockFromLockList` value in the `AnyRollbackAction` property of the recipe. The only exception is a rollback from the *Insert* state, since in addition to creating an insert-lock, a lock list had to be created first. Here the `RemoveLockList` value is used to not only remove the insert-lock but also the lock list. The guards applied to the rollback transitions are used to distinguish the possible previous states. The guard *[insert-lock]* ensures that the lock list also contains an insert-lock, while the other used guard *[!insert-lock]* ensures that the lock list has no insert-lock. Acquisition transitions that are not specified in the state diagram all lead to a failing acquisition and an instance of `AcquisitionRecipe` with the `NoCompatibility` value from the `AcquisitionCompatibility` enumeration.

The only possible transition from the *New* state is a write-acquisition, since read-, readexclusive-, or take-acquisitions all demand an existing space element, which is not guaranteed without an insert-lock. The write-acquisition leads to the creation of such an insert-lock by the use of the `PredefinedAcquisitionAction AddLock`. The *Insert* state along with the *Free* state both provide transitions for all acquisition operations with the exception of write-acquisitions. Allowing the acquisitions for all transactions is a unique behavior of the ReadUncommittedPlugin, as it does not protect from inconsistencies that come from reading temporary created space elements or even removing them. This phenomena is called *dirty reads*. It is even possible to issue a take-acquisition on a space element in the *Insert* state from a different transaction, since the owner of the insert-lock is not checked. As a consequence of this behavior, the ReadUncommittedPlugin does not create read- or readexclusive-locks at all. Hence, read- and readexclusive-acquisitions do not lead to different states and do not create new locks in the lock list. This is realized with the `NoAcquisition` value in the `AcquisitionAction` property of the recipe. If the top-level transaction of the previous write-acquisition is committed, the insert-lock will be replaced by a free-lock. This corresponds to the `FinalCommitAction` value `ReplaceWithFreeLock`. The other transitions with the *FinalCommit* event all lead to the *New* state, as their current states all contain a delete-lock. As with any other transition that leads to the *New* state, the `RemoveLockList` value is used to fully remove the lock list. Take-acquisitions are allowed when it is guaranteed that the space element still exists, which is the case in the *Free* and *Insert* states. These states have transitions with the *Take* event that lead to the *Delete* state, by simply creating a delete-lock in their corresponding lock lists with the `AddLock` value in the recipe.

Even though the ReadUncommittedPlugin might not be usable for most situations, due to its lacking consistency qualities, it can be used to gather insight of other transactions and from a conceptual point of view, it can be seen as the starting point for the isolation level handlers.

ReadCommittedPlugin

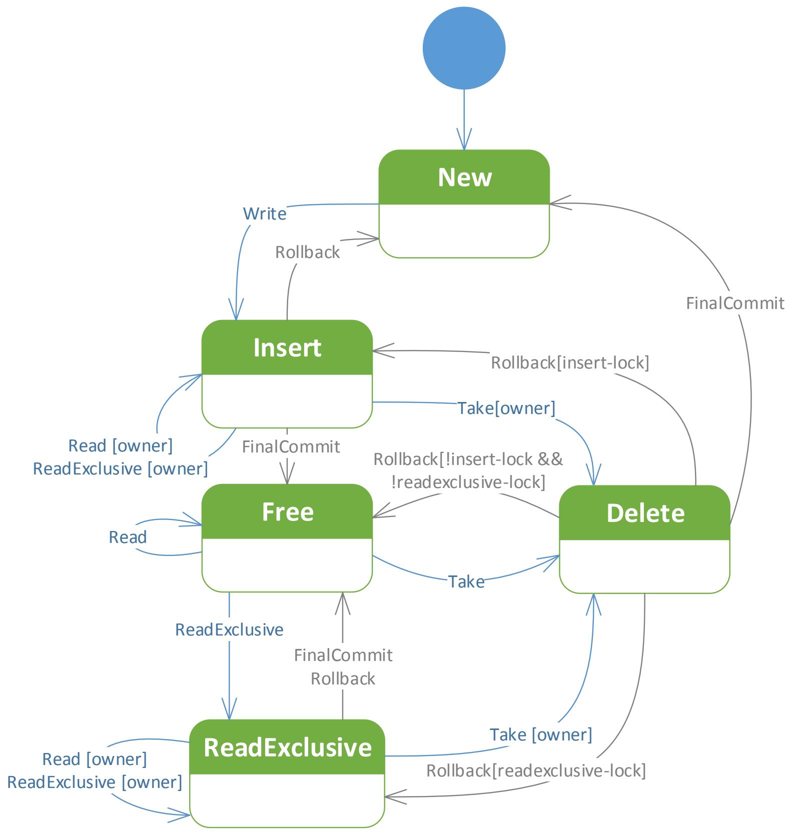


Figure 6.9: UML state diagram illustrating the lock state transitions for the read commited isolation level.

The ReadCommittedPlugin is located in the `XVSM.Plugin.IsolationLevelHandler.ReadCommitted` namespace and is based on the ReadUncommittedPlugin. Figure 6.9 shows that the state diagram for the read committed implementation is an extension of read uncommitted with additional constraints. All transitions from Figure 6.8 also exist here but many of them have an additional guard now. The guard *[owner]* is located on several acquisition transitions. It depicts that the event will only fire the transition if the transaction of the acquisition is the owner of the source state's defining lock. This is a constraint that guarantees that only the owning transaction of the current lock is allowed to carry out the acquisition. This is an important difference between this plugin and the ReadUncommittedPlugin, as it protects from the dirty reads phenomena. It is not possible anymore to access or remove space elements from a different transaction hierarchy that have not been committed yet. Therefore, the acquisition transitions from the *Insert* state have this guard applied to protect the space element from other transaction hierarchies until it is committed or removed. This guard is realized with the `ExclusiveOwnerCompatibility` value in the `Compatibility` property of the recipe.

This presented guard is not the only difference to the previous plugin. The state *ReadExclusive* was introduced. From the state *Free* it is possible to acquire a readexclusive-lock to reach the new state. This is realized with the `AddLock` value in the recipe. From there, further read-, readexclusive-, or take-acquisitions are only allowed for the owning transaction of the readexclusive-lock. However, further read- and read-exclusive transitions are overruled by the existing readexclusive-lock anyway. Thus, these two transitions are realized with the `NoAcquisition` value in the recipe. The take-transition, on the other hand, might lead to the *Delete* state with the creation of a delete-lock. The delete-lock is created, as usual, with the `AddLock` value.

The *Delete* state may be entered from three different states now: *Insert*, *Free*, and *ReadExclusive*. According to the definition of the states in the figure, every one of these states has a different defining lock in their lock lists. The *Insert* state has an insert-lock, the *Free* state a free-lock and the *ReadExclusive* state a readexclusive-lock included in their lock list. Hence, the logical *Delete* state describes the lock list state of three different possible lock lists.

The ReadCommittedPlugin may be used for much more situations than the ReadUncommittedPlugin because of its protection against dirty reads. The support of a separate state for readexclusive-acquisitions is also useful for some XVSM.NET operations. For instance, the CAPI-3 operation to lock a container is implemented by issuing such a readexclusive-acquisition. One thing this isolation level does not protect from is the non-repeatable read phenomena. It is possible to successfully create a read-acquisition through a transaction that if repeated later with the same transaction might not work anymore. The space element could have been removed in the meantime, since the previous read-acquisition would not protect it from being removed by a different transaction. The following isolation level will cover this phenomena.

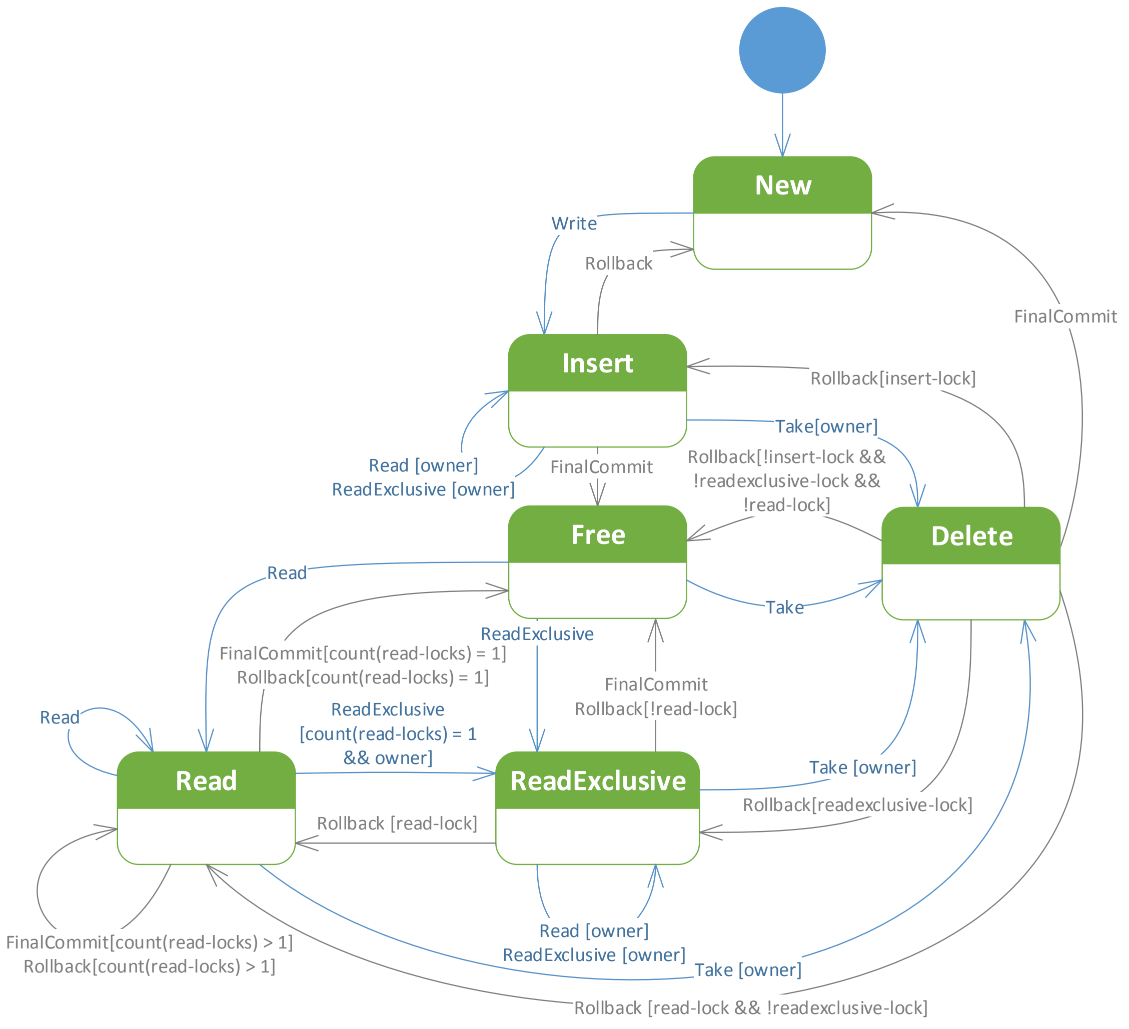

New
Write
Rollback
FinalCommit
Insert
Rollback[insert-lock]
Take[owner]
Read [owner]
ReadExclusive [owner]
FinalCommit
Rollback[!insert-lock && !readexclusive-lock && !read-lock]
Free
Delete
Read
Take
ReadExclusive
FinalCommit[count(read-locks) = 1]
Rollback[count(read-locks) = 1]
FinalCommit
Rollback[!read-lock]
Read
ReadExclusive [count(read-locks) = 1 && owner]
Take [owner]
Read
ReadExclusive
Rollback[readexclusive-lock]
Rollback [read-lock]
Read [owner]
ReadExclusive [owner]
FinalCommit[count(read-locks) > 1]
Rollback[count(read-locks) > 1]
Take [owner]
Rollback [read-lock && !readexclusive-lock]

Figure 6.10: UML state diagram illustrating the lock state transitions for the repeatable read isolation level.

RepeatableReadPlugin

With the RepeatableReadPlugin, which is located in the `XVSM.Plugin.Isolation-LevelHandler.RepeatableRead` namespace, the state diagram from the previous ReadCommittedPlugin is extended further in Figure 6.10. The changes from the previous state diagram come from the addition of shared read-locks that led to the new logical state *Read*.

The *Read* state is reachable with a read-acquisition from the *Free* state. This is the only way to reach this state by an acquisition transition. This reflects the implementation behavior that it is only possible to create the first read-lock in the lock list if the lock list solely contains a free-lock. Any other lock would either make the acquisition unnecessary or incompatible. Hence, the *Read* state corresponds to a lock list containing a free-lock and read-locks.

From the *Read* state there are two possible acquisition transitions with the following events: *ReadExclusive* and *Read*. The transition with the *ReadExclusive* event is only allowed when the lock list contains only a single read-lock that is also owned by the same transaction. This transition may then be used in situations where it is required to turn the shared access of a space element into an exclusive access. The acquisition transition with the *Read* event is a little bit different to the other transitions that correspond to the read-acquisition. It allows to create additional read-locks if the executing transaction does not own any read-lock yet. This is the single use of the `PredefinedAcquisitionAction` value `AddUniqueLockOnly`. This value ensures that the read-lock is only added to the lock list if it does not contain any other read-lock with the same owning transaction.

The multiple transaction transitions illustrate the different behavior depending on the concrete amount of read-locks in the lock list. When the lock list only contains a single read-lock, both types of transaction transitions will lead to the *Free* state as the read-lock will be removed.

The RepeatableReadPlugin implements the recommended isolation level of the formal model, as it reasonable extends the ReadCommittedPlugin by adding shared read-locks. In addition to the protection against the dirty read phenomena, it protects against the non-repeatable read phenomena, as well. However, there is another phenomena, called *phantom reads*, that could still occur with this isolation level and is covered by the last isolation level handler plugin.

SerializablePlugin

The SerializablePlugin, which is located in the `XVSM.Plugin.IsolationLevelHandler.Serializable` namespace, is the last of the four isolation level handler plugins. It has exactly the same state diagram as the RepeatableReadPlugin in Figure 6.10, as it creates instances of the *AcquisitionRecipe* with the same semantics. The concrete implementation actually injects the RepeatableReadPlugin and delegates the `GetAcquisitionRecipe` method to it. The only difference to the RepeatableReadPlugin is the provided value in the `MinCoordinationLockModeName` property, which modifies the default acquired coordination lock through the `ICoordinationLockHandler` plugin. This can be seen as a locking mechanism on a higher order that leads to less concurrency, especially for modifying operations, but improves the consistency. This protects against the phantom reads phenomena, which occurs, for instance, when one transaction commits while another transaction is still running and executes the same read operation twice with different results, once before and once after the other transaction commits. This could occur due to the changes from the committing transaction that are visible right away after it is finished. Hence, a repeated access to the space element would lead to different results in the same transaction. With the SerializablePlugin, operations that lead to this phenomena would be blocked and rescheduled by the runtime.

Nevertheless, it is still possible to access different coordinators concurrently from the same container since this isolation level handler only increases the locks that are acquired on the coordinators.

However, even though XVSM.NET supports this isolation level, it is a purely experimental plugin with limited use cases due to its aggressive locking strategy. It was developed for experimental purposes and because the acquisition of coordination locks through the `ICoordinationLockHandler` was already in place and required for the implementation of the Vector coordinator. The Vector coordinator has unique locking semantics and only allows modifying operations, such as write, take, or delete from a single transaction hierarchy. This will be covered in Section 6.3.

6.2.7 Locking Implementation

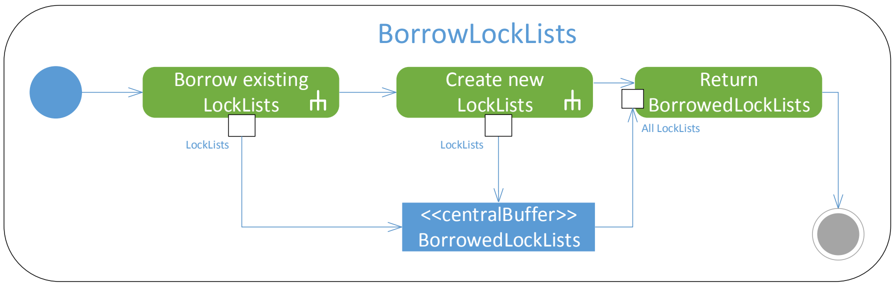


Figure 6.11: UML activity diagram showing the general functionality of the `BorrowLockLists` method.

The default plugin implementing the locking contracts (cf. Section 6.2.4) is simply called `LockListManager` and is located in the `XVSM.Plugin.LockListManager` namespace. The main challenge of developing this plugin was to create a thread-safe implementation while maintaining concurrency as much as possible. The implementation uses C#'s `ConcurrentDictionary` class from the `System.Collections.Concurrent` namespace to store the lock lists with their corresponding space element reference as a lookup key. Future references to this concrete instance will be called *LockListDictionary* in this sub-section.

The characteristic method of the plugin is the `BorrowLockLists` method, which is used by the transaction implementation (cf. Section 6.2.8) to create new lock lists and get exclusive access to the existing ones. As illustrated in Figure 6.11, the method is dividable into two major tasks: the handling of existing lock lists, and the handling of new lock lists. These two tasks are visualized by separate activity diagrams and will be covered later in this sub-section. The input parameters were omitted in this figure for better readability. When both activities are finished, their preliminary resulting lock lists are provided to a new `BorrowedLockLists` instance that implements `IBorrowedLockLists`.

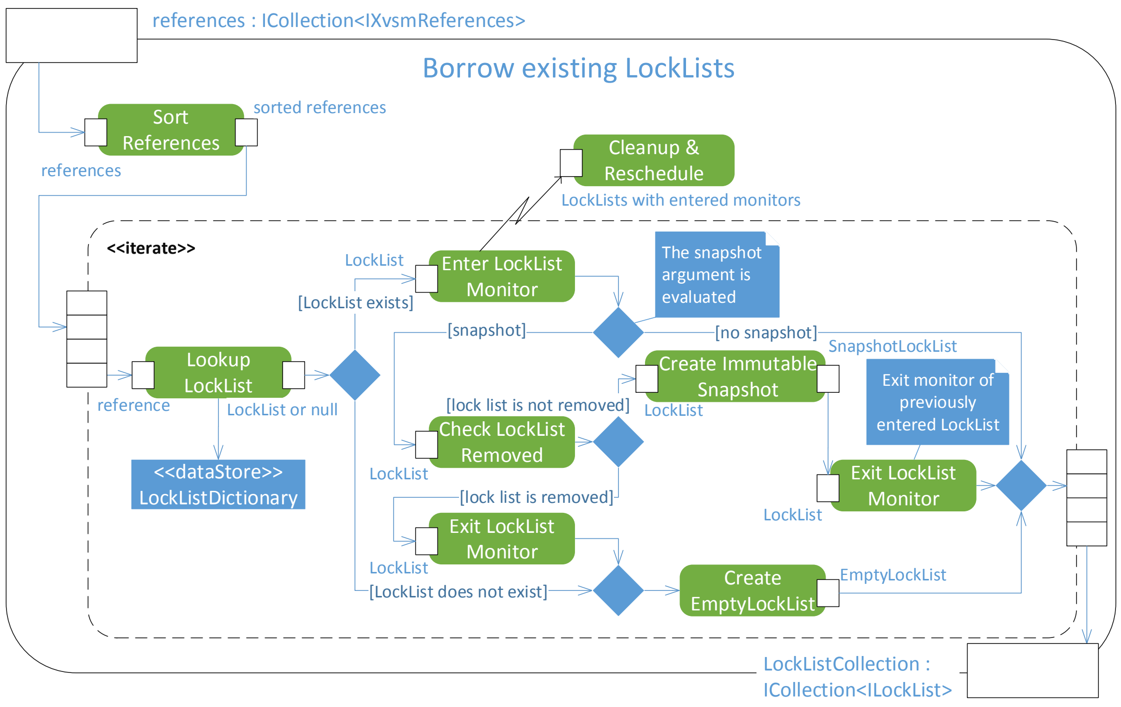


Figure 6.12: UML activity diagram visualizing the algorithm to acquire exclusive access to existing lock lists.

Exclusive Access to Existing Lock Lists

This sub-task is illustrated as an UML activity in Figure 6.12. Its function is to check the provided space references for existing lock lists and acquire exclusive access for them. As this is a bulk task, the provided parameter is a collection of space references and is visualized in the input parameter of the activity. It should be noted that the implementation method also contains the parameters `snapshot` and `maxLockTime`. However, they were omitted as parameters in the activity diagram for clearness, but they will be still covered in this sub-section. The action *Sort References* is used to create a deterministic ordering of the references as a deadlock prevention mechanism. The provided references are iterated in the sorted order and their corresponding lock lists are tried to be looked up in the LockListDictionary. When a lock lists was successfully retrieved, its monitor lock will be entered. As many monitor locks may not be released until later when disposing the `IBorrowedLockLists` instance, without sorting, it would be possible to acquire locks in a circular dependent way and run into a deadlock. Thus, the sorting was used to guarantee that the monitors are entered without circular dependencies and deadlocks. When it still takes too long to enter the monitor lock, an exception is thrown and the runtime needs to reschedule the operation. The default maximum time before the exception is raised can be configured through the `LockEnterTimeout` property of the `LockListManagerConfigContract`. However, it is also possible to override the default maximum time by providing the maximum lock time as parameter.

After the lock lists' monitor lock was successfully entered, the `snapshot` argument is evaluated, which defines whether the original lock list should be directly used or an immutable copy of it. In the latter case the lock list is checked whether it was marked as "removed". It is possible that the lock list was removed by another transaction before its monitor lock could be entered. For such situations, its monitor lock will be exited right away and an empty immutable lock list is used instead. This is the same behavior as if no lock list is associated with a given element reference. Hence, the transaction implementation must act accordingly for such situations. If the lock list, however, was not marked as "removed", the lock list is copied in an immutable instance of the type `SnapshotLockList`. The monitor of the retrieved lock list will be exited too as exclusive access is not required anymore.

As illustrated in the activity diagram of Figure 6.11, all the resulting lock lists will be temporarily stored until the sub-task of handling new lock lists is finished too.

Create new LockLists

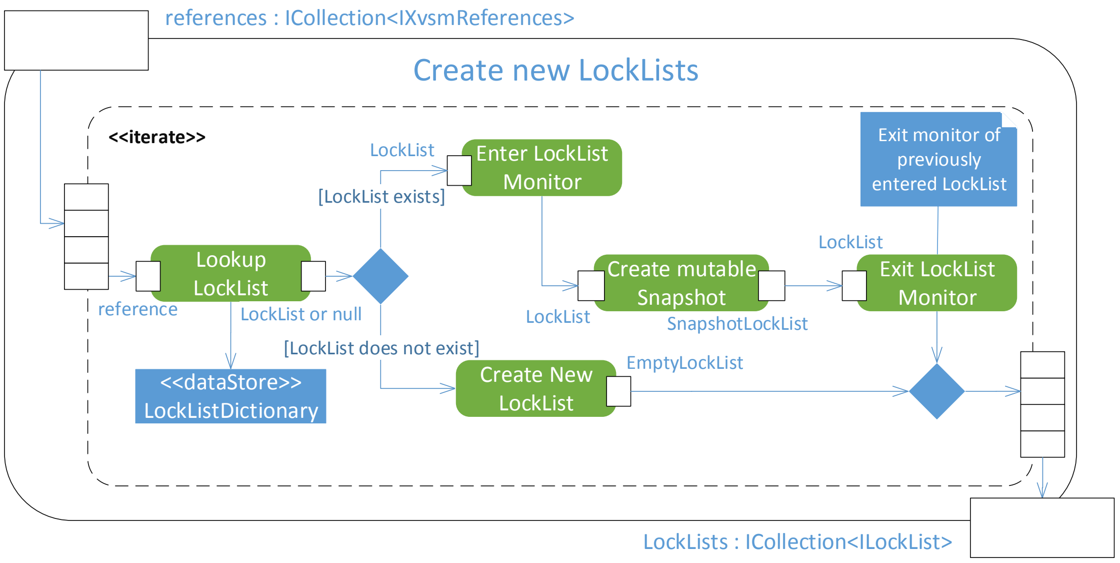


Figure 6.13: UML activity diagram depicting the algorithm to enter monitors from lock lists.

The second sub-task is illustrated as an activity as well. Figure 6.13 shows the process of creating new lock lists by providing element references. For every single reference, a lookup is issued on the LockListDictionary to prevent the creation of duplicate lock lists for the same reference. When an existing lock list was found indeed, a mutable copy will be created but is not yet registered at the LockListDictionary. At first, its monitor is entered to prevent concurrent access. Right after, a regular lock list copy is created and the monitor is exited again. The creation of a mutable lock list copy is different from the usual immutable snapshot lock lists that were previously used, because now the calling transaction of the `BorrowLockLists` method might not be aware that a snapshot was returned. Hence, the addition and removal of locks in this returned lock list should not directly lead to an exception. When finally the lock list is added to the LockListDictionary, an exception will be thrown if there is still another existing lock list with the same reference. The reason why this sub-task even tests for existing lock lists and not simply always returns new lock lists is to allow the calling transaction a more precise reaction by avoiding raising exceptions. If simply an empty mutable lock list was returned, the transaction could not directly react at all until the exception is thrown when the `IBorrowedLockLists` instance is disposed and the addition of the new lock lists failed. Listing 6.4 illustrates the situation where this problem could arise:

```
tx1.PrepareAcquisition().Access(ref1, Write).Acquire();
// -> successful

tx2.PrepareAcquisition().Access(ref1, Write).Acquire();
// -> exception or unsuccessful
```

Listing 6.4: Duplicate write-acquisition of the same element.

Both transactions `tx1` and `tx2` access the same reference `ref1` for write access. However, as new space references need to be unique, the acquisition of `tx2` will not work. The problem is that the second acquisition must not lead to an exception in this case but should simply return an unsuccessful result. Hence, the implementation returns the mutable snapshot of the existing lock list to provoke an incompatibility with the isolation level and therefore failing fast.

BorrowedLockLists

All the lock lists that have been retrieved or created in the previous sub-tasks are put in a `BorrowedLockLists` instance that implements the *IBorrowedLockLists* interface. This instance will be returned to the calling transaction. When the instance is disposed, several operations are executed:

- Existing lock lists that were marked to be removed are unregistered in the LockListsDictionary.
- The monitor locks of the existing and entered lock lists are exited.
- Newly created lock lists are registered in the LockListDictionary, but only if the `RollbackCreatedLockLists` property is not set to `true`.

6.2.8 Pessimistic Transaction Implementation

The default plugin part for the *ITransaction* contract is a pessimistic nested locking implementation called PessimisticTransaction and is located in the `XVSM.Plugin.Pessimis-`

ticTransaction namespace. This plugin part makes use of the previously presented plugins implementing the locking contracts and isolation level contracts.

Commit and Rollback

The implementation of the commit method is visualized in Figure 6.14, showing the commit of the top-level transaction of a hierarchy with three levels.

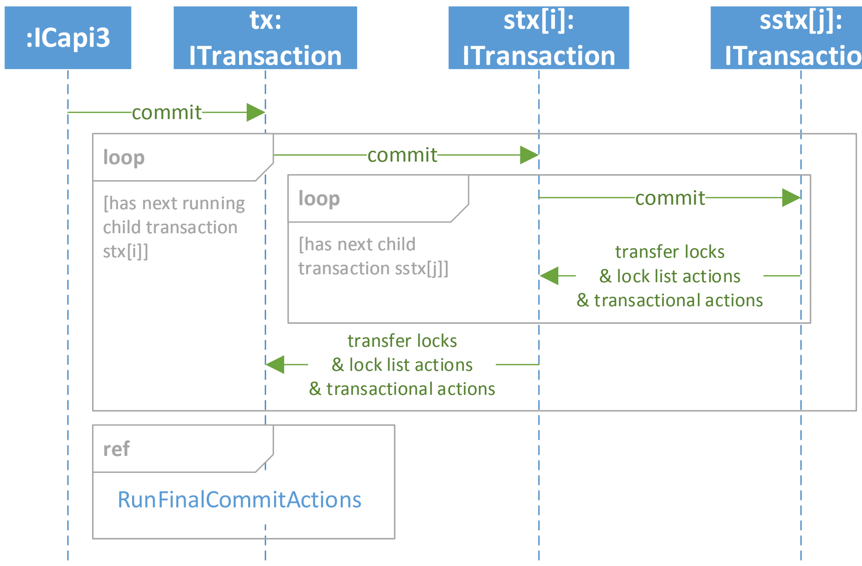

:ICapi3
tx: ITransaction
stx[i]: ITransaction
sstx[j]: ITransaction
commit
loop
[has next running child transaction stx[i]]
commit
loop
[has next child transaction sstx[j]]
commit
transfer locks & lock list actions & transactional actions
transfer locks & lock list actions & transactional actions
ref
RunFinalCommitActions

Figure 6.14: UML sequence diagram showing the committing process of a transaction hierarchy with three levels.

The commit method is recursively propagated to all the running sub-transactions. The deepest running child transaction starts to transfer its locks, registered lock actions and transactional actions to its parent transaction. This process is repeated until the top-level transaction contains these stored elements from the full transaction hierarchy. With these elements, the top-level transaction initiates the execution of the FinalCommit event, which is illustrated in the separate Figure 6.15.

Before any action is invoked or applied, all lock lists of the transaction hierarchy are borrowed again to obtain exclusive access. Since all requested lock lists are already existing, the `newReferences` parameter of the borrowing method is left `null`. As this borrowing request incorporates the re-borrowing of all lock lists that have ever been used by any transaction in the hierarchy, this request might take some time and is therefore executed with priority. This second borrowing of the same lock lists is important to guarantee transactional safety for the registered lock list actions but also to provide a protected execution environment for the transactional actions. These actions are invoked before borrowed lock lists are returned, which provides them the guarantee that the lock lists may not be modified in the meantime.

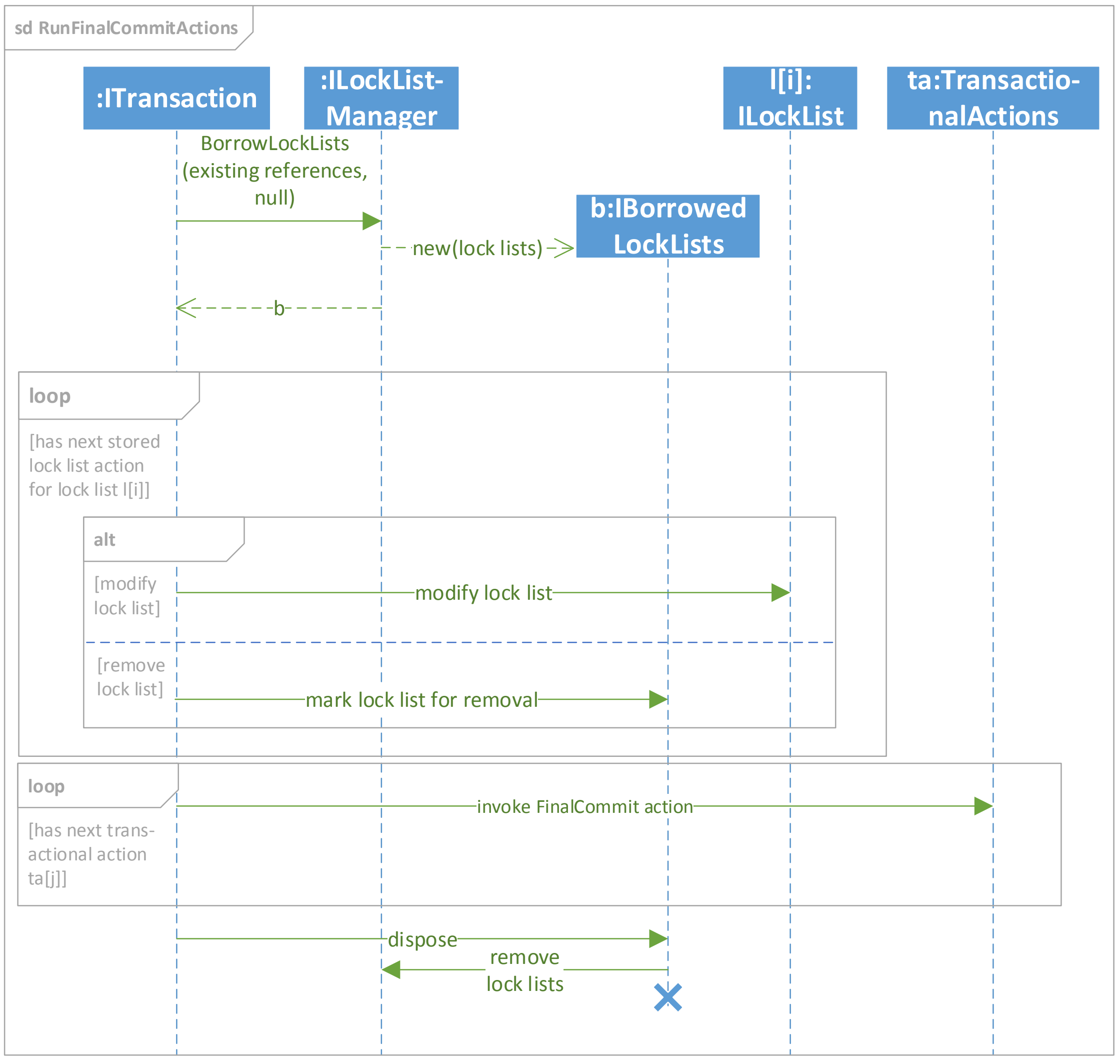


Figure 6.15: UML sequence diagram illustrating the execution of registered final commit actions.

Initiating a rollback on a transaction hierarchy functions in a similar way as a commit. Instead of the registered final commit action, the registered rollback actions are executed and these actions are executed by every rollbacked transaction individually and not only the top-level transaction.

Lock Acquisition

The method `PrepareAcquisition` returns an instance of an `IAcquisitionRe-quest` implementation. When the acquisition request is set up and its `Acquire` method is called, the `ITransaction` instance is called back again to carry out the acquisition.

Figure 6.16 visualizes an example acquisition process and shows the interaction of the lock list manager and isolation level handler plugins in a simplified form.

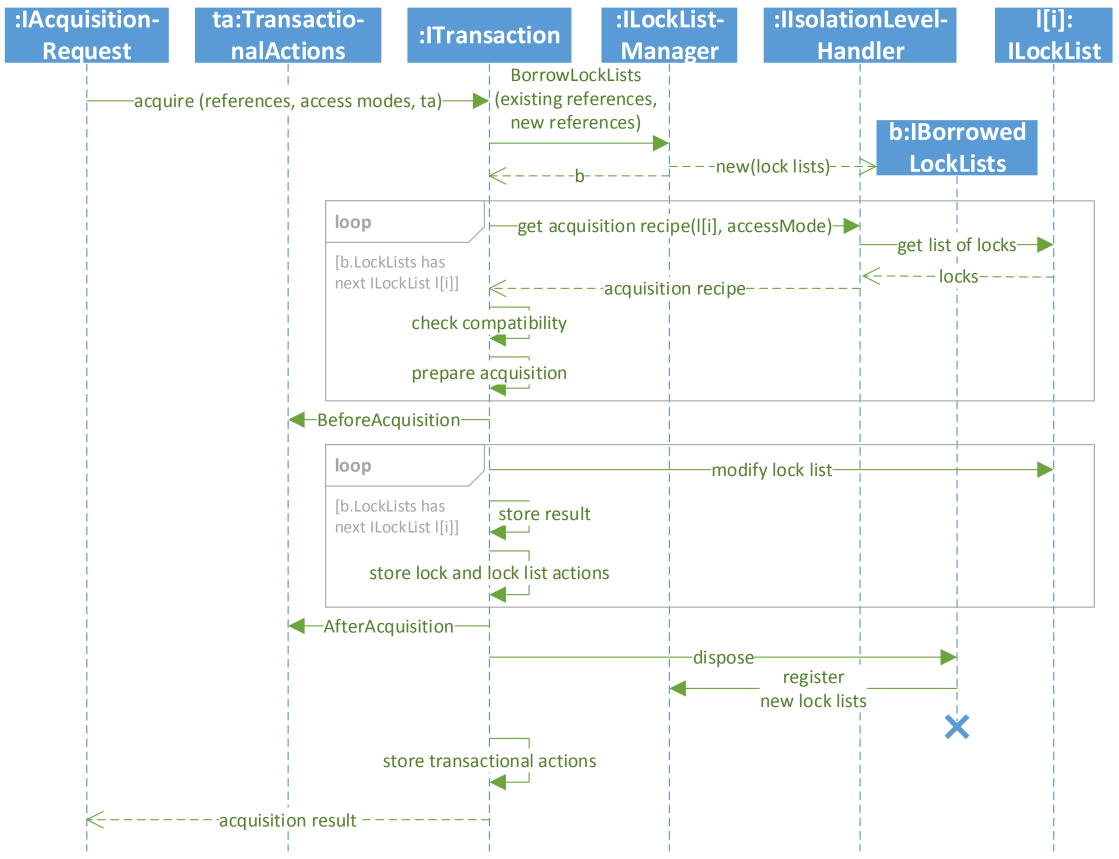


Figure 6.16: UML sequence diagram illustrating a successful lock acquisition.

When `ITransaction` is invoked by the `IAcquisitionRequest` instance, it uses the lock list manager to borrow the lock lists for the provided references. The references are divided between write-acquisitions and the rest, in order to provide separate lists for the borrowing method. The lock list manager then returns with an `IBorrowedLockLists` instance that contains the requested lock lists for exclusive access.

These lock lists are iterated and an acquisition recipe is requested for each one from the transaction's isolation level handler. When the resulting acquisition recipe attests compatibility for the single acquisition request, the recipe is stored until the other recipes of the remaining lock lists are checked too. The temporary storing is required since either all individual acquisition requests must be performed or none. In this successful example, however, all acquisition recipes attest compatible acquisitions.

The second iteration that finally carries out the acquisition is surrounded by the invocation of the `TransactionalActions` instance's registered events. The lock list is modified in the specified way of the concrete acquisition recipe and the results are stored to be later returned together. When a new lock was created and the acquisition recipe's lock list actions for the `FinalCommit` or `AnyRollback` events are set, the lock and the lock actions are stored in an internal list of the transaction. When the event occurs, the corresponding lock list actions will be executed.

If the compatibility test in the first iteration failed for any single acquisition, the second iteration as well as the two invocations of the transactional actions would be omitted and instead a single invocation of the `NoAcquisition` transactional action would be executed. Disposing the *IBorrowedLockLists* instance makes sure the borrowed existing lock lists are unlocked and can be borrowed by other transactions. The newly created lock lists are registered in the `ILockListManager` instance. Before returning the results, the `TransactionalActions` instance is stored as well in the transaction, since it might contain custom actions for the `FinalCommit` or `AnyRollback` events.

6.2.9 NoTransaction Implementation

The `NoTransaction` plugin implementation in the namespace `Xvsm.Plugin.No-Transaction` is an alternative that provides the functionality for the transaction contracts. It does not use locking to protect and isolate elements and therefore does not use the isolation level handlers as well. It is a very simple implementation of the transaction contracts' interfaces that should be used in scenarios where no transactional safety is needed.

Nested instances and the transactional actions, however, are supported by this implementation and work similar to the PessimisticTransaction plugin. The commit is propagated to every running sub-transaction, which leads to the transfer of the registered transactional actions to the top-level transaction. The top-level transaction iterates through the transactional actions and invokes the final commit actions. The rollback functions, similarly as the invocation of the rollback method, are also propagated to all running sub-transactions. The plugin implementation respects the transaction contracts to allow its usage as a drop-in replacement for the pessimistic transaction implementation.

6.3 Coordination

In the formal model the coordination is introduced in the CAPI-3 layer, which is responsible for entries and containers. It provides methods for retrieving and storing entries in a structured way. In this section we are taking a look at the CAPI-3 layer of XVSM and show how the coordination mechanism of XVSM is realized in XVSM.NET. Figure 6.17 depicts the involved plugin and plugin part contracts.

The central contract, that acts as an entry point, is the `ICapi3` plugin contract. It provides an API according to the CAPI-3 layer of the formal model. Its main responsibility is to provide entry operations with coordination support in a consistent, transactional manner. Thus, the plugin and plugin part contracts `ICapi3` depends on are used to achieve this functionality.

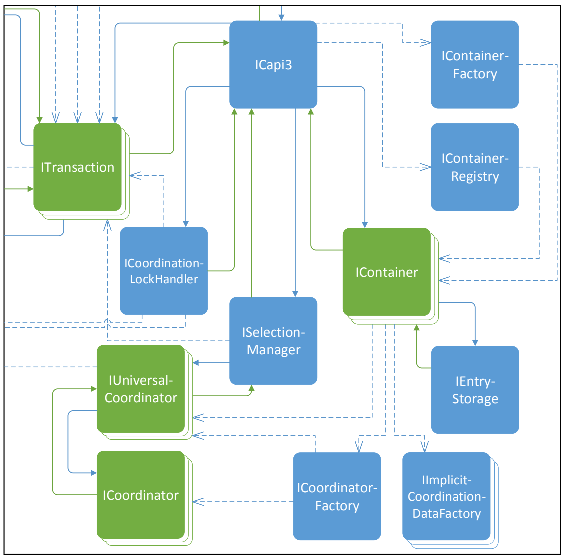


Figure 6.17: Extract of the dependency and architectural overview showing coordination-relevant components.

Containers are managed by `ICapi3` through the `IContainerFactory` and `IContainerRegistry` plugins. The resulting `IContainer` plugin parts are used by `ICapi3` to store and retrieve entries. To achieve this, the `IContainer` instances delegate the storage responsibility of entries to the `IEntryStorage` plugin. Coordinators are instantiated by `IContainer` plugin parts according to the provided information of `ICapi3`'s create container operation. When entries are stored, their coordination data are registered at the coordinators. In some cases the `IContainer` plugin part uses the `IImplicitCoordinationDataFactory` to create the coordination data automatically before registering it.

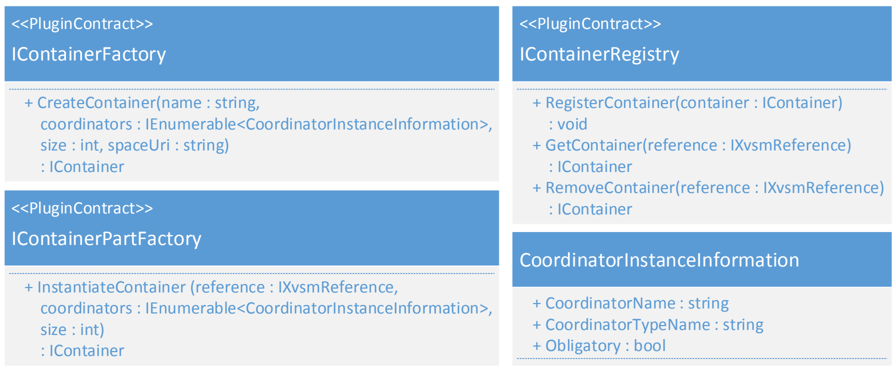
For `ICapi3`'s entry selection operations, such as read, take, or delete, the `ISelectionManager` plugin is used to orchestrate the selection by wiring the `IUniversalCoordinator` plugin parts and executing the selection. They, on the other hand, delegate their operations to the concrete `ICoordinators` plugin parts.

`ITransaction` plugin parts are used to create a sub-transaction to acquire locks and execute the operation in a consistent manner. The `ICoordinationLockHandler` plugin is used to acquire general coordination-specific locks per operation.

In this section we show the concrete specification of these plugin and plugin part contracts as well as provide details of their implementation characteristics.

6.3.1 Container Contracts

Containers are plugin parts with the contract `IContainer`. As a plugin part, its instantiation is performed by a plugin part factory with the `IContainerPartFactory` contract. However, this plugin part factory is not directly used by CAPI-3 to instantiate containers, but instead the `IContainerFactory` plugin is used, which internally delegates the instantiation requests to the plugin part factory. This separate plugin was introduced to allow the simultaneous usage of multiple container implementations. For instance, it is possible to create a container plugin part that does not support bounded XVSM containers but instead is optimized for unbounded containers. For situations where a bounded container is requested, on the other hand, a different implementation should be used. The `IContainerFactory` plugin is responsible to select the implementation that fits the situation best. This can be realized through the plugin framework's dynamic plugin selection mechanism that allows to choose the concrete plugin through its metadata. Figure 6.18 shows an UML class diagram visualizing the contracts that are used for the container creation and retrieval.



<<PluginContract>>
IContainerFactory
+ CreateContainer(name : string, coordinators : IEnumerable<CoordinatorInstanceInformation>, size : int, spaceUri : string) : IContainer
<<PluginContract>>
IContainerPartFactory
+ InstantiateContainer (reference : IXvsmReference, coordinators : IEnumerable<CoordinatorInstanceInformation>, size : int) : IContainer
<<PluginContract>>
IContainerRegistry
+ RegisterContainer(container : IContainer) : void
+ GetContainer(reference : IXvsmReference) : IContainer
+ RemoveContainer(reference : IXvsmReference) : IContainer
CoordinatorInstanceInformation
+ CoordinatorName : string
+ CoordinatorTypeName : string
+ Obligatory : bool

Figure 6.18: UML class diagram showing contracts for the container creation and retrieval.

The `CreateContainer` method of the `IContainerFactory` contract requires the container name, a collection defining the coordinator instances, the maximum number of entries and the space URI as arguments. The specified name and space URI are used to create a space reference with the `IXvsmReferenceFactory` plugin. The newly created reference, along with the remaining parameters, are delegated to the `InstantiateContainer` method of the concrete plugin part factory. The integer defines the maximum number of entries for bounded containers but may also be used to specify unbounded containers when the value is zero. The provided collection of `CoordinatorInstanceInformation` objects specifies the associated coordinators of the container. These instances contain the name and type of the coordinator as strings and define whether the usage of the coordinator is required or optional. With all this information, it is possible for the container to arrange the instantiation of the coordinators and manage their instances. The created container may then be registered at the registry plugin `IContainerRegistry` for future lookups by the CAPI-3 layer.

Containers in XVSM.NET represent the containers of XVSM. They are an additional abstraction layer for the space and provide a custom orchestration of useable coordinators. Figure 6.19 shows the contracts of the containers and related elements in XVSM.NET.

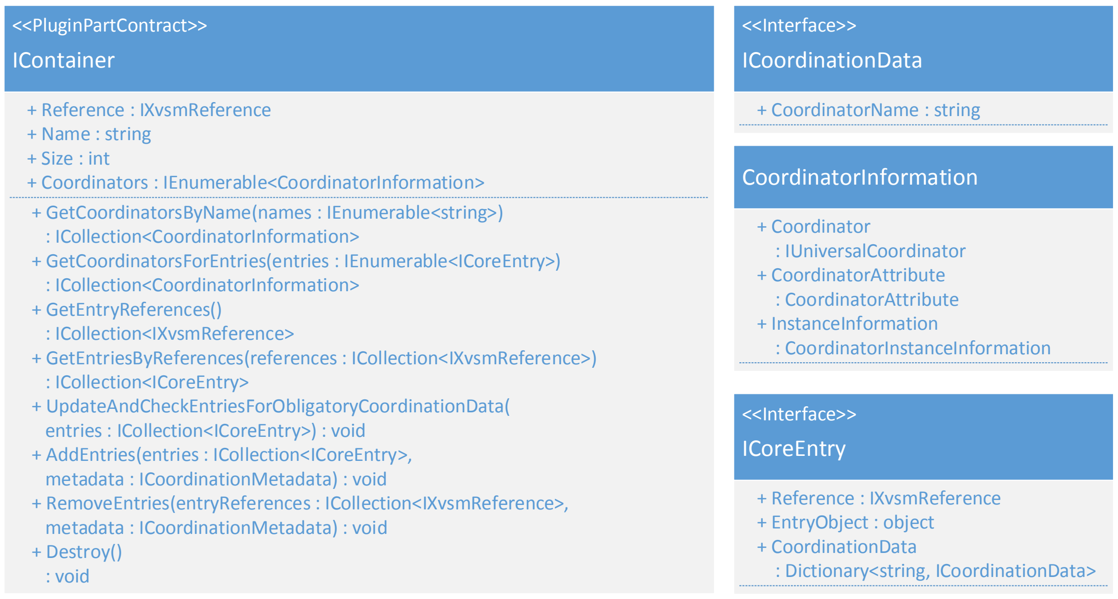

<<PluginPartContract>>
IContainer
+ Reference : IXvsmReference
+ Name : string
+ Size : int
+ Coordinators : IEnumerable<CoordinatorInformation>
+ GetCoordinatorsByName(names : IEnumerable<string>) : ICollection<CoordinatorInformation>
+ GetCoordinatorsForEntries(entries : IEnumerable<ICoreEntry>) : ICollection<CoordinatorInformation>
+ GetEntryReferences() : ICollection<IXvsmReference>
+ GetEntriesByReferences(references : ICollection<IXvsmReference>) : ICollection<ICoreEntry>
+ UpdateAndCheckEntriesForObligatoryCoordinationData(entries : ICollection<ICoreEntry>) : void
+ AddEntries(entries : ICollection<ICoreEntry>, metadata : ICoordinationMetadata) : void
+ RemoveEntries(entryReferences : ICollection<IXvsmReference>, metadata : ICoordinationMetadata) : void
+ Destroy() : void
<<Interface>>
ICoordinationData
+ CoordinatorName : string
CoordinatorInformation
+ Coordinator : IUniversalCoordinator
+ CoordinatorAttribute : CoordinatorAttribute
+ InstanceInformation : CoordinatorInstanceInformation
<<Interface>>
ICoreEntry
+ Reference : IXvsmReference
+ EntryObject : object
+ CoordinationData : Dictionary<string, ICoordinationData>

Figure 6.19: UML class diagram illustrating the container contract.

The arguments that have been provided to the plugin part factory of the container may be accessed through the container's properties to give information about the container. However, the `Coordinators` property uses a slightly different type that includes the previously provided `CoordinatorInstanceInformation` object as well as objects of the instantiated coordinator. The instantiated coordinator is accessible through the `Coordinator` property and implements the `IUniversalCoordinator`, which are described in Section 6.3.3. The attribute of the coordinator is also stored for a quick access to the coordinator's metadata without the need of C#'s reflection API. As already mentioned, the container is responsible to instantiate coordinator instances according to the specified coordinator names and types. However, the container implementation does not directly instantiate the coordinators but uses an instance of the `ICoordinator-Factory` plugin. The task of this factory is to select the correct coordinator plugin and instantiate it through its corresponding plugin part factory.

The methods `GetCoordinatorsByName` and `GetCoordinatorsForEntries` provide functionality to retrieve coordinator information objects from the container. The former method simply returns coordinators with the specified names and is used by the CAPI-3 layer before entries are selected by the retrieved coordinators. The latter method returns all coordinators that are relevant for the entries and is called whenever entries are written to the container. Entries may be registered on different coordinators, thus the information for coordinators that are relevant to at least one entry is returned. This coordinator information is useful to acquire coordinator-specific locks before the entries are written to the container.

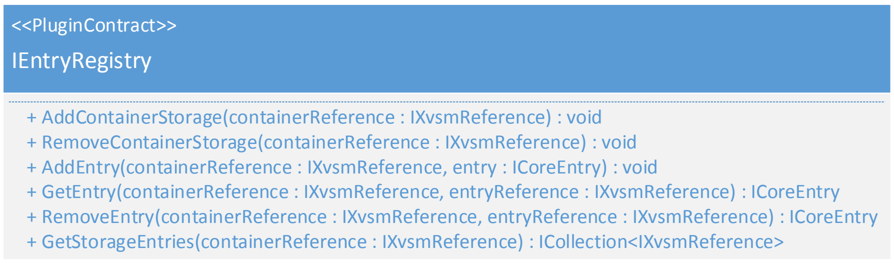
The method `GetEntryReferences` returns the entry references of all stored entries in the container, which are otherwise only accessible through the coordinators. The entry references may then be used with the bulk method `GetEntriesByReferences` to retrieve the corresponding entry objects. These returned instances are objects of the type `ICoreEntry`, which includes the entry reference, its data object and the coordination data. This coordination data was included with the entry object to provide additional coordination-relevant information for the coordinators. The method `UpdateAndCheckEntriesForObligatoryCoordinationData`, as the name implies, checks if the provided entries contain the required coordination data of obligatory coordinators and automatically creates implicit coordination data if possible.

Entries are added with the method `AddEntries` by providing a collection of `ICoreEntry` instances and an `ICoordinationMetadata` object. The container is responsible to store the entries and provide their coordination data to the coordinators. `ICoordinationMetadata` is used to provide additional metadata for the operation. The concrete specializations of the interface are shown in Section 6.3.3. Containers must provide a `RemoveEntries` method to remove stored entries. When they are removed from the container, they must be unregistered from the coordinators as well.

The remaining method in the container interface is the `Destroy` method that is used to quickly remove the container with all stored entries and associated coordinators.

6.3.2 Entry Storage

The implementation of the default container plugin does not directly store the entries but uses an instance of the `IEntryStorage` plugin. All container objects share the same instance of this plugin. Figure 6.20 shows the plugin contract of the `IEntryStorage` plugin. The container plugin invokes the `AddContainerStorage` method with its reference to create a personal storage location. Entries may be stored, retrieved or removed by providing the container reference that also identifies the storage location. With the `RemoveContainerStorage` method, all remaining entries in the storage location can be removed at one go.


<<PluginContract>>
IEntryRegistry

+ AddContainerStorage(containerReference : IXvsmReference) : void
+ RemoveContainerStorage(containerReference : IXvsmReference) : void
+ AddEntry(containerReference : IXvsmReference, entry : ICoreEntry) : void
+ GetEntry(containerReference : IXvsmReference, entryReference : IXvsmReference) : ICoreEntry
+ RemoveEntry(containerReference : IXvsmReference, entryReference : IXvsmReference) : ICoreEntry
+ GetStorageEntries(containerReference : IXvsmReference) : ICollection<IXvsmReference>

Figure 6.20: UML class diagram showing the `IEntryStorage` plugin contract.

It was decided to create this separate storage plugin for flexibility of the storage implementation. This was created with persistence in mind. A custom entry storage that saves the entries in a database does not need an additional container implementation with this separate plugin. This is especially useful as the container incorporates logic for the coordinator management that is also required for an alternative persistent storage.

6.3.3 Coordinator Contracts

Coordinators of XVSM correspond to plugin parts of the `ICoordinator` plugin part contract, which are referred to as *coordinators* as well. The development of a flexible and type-safe concept for the coordinator contracts led to the contract hierarchy in Figure 6.21.

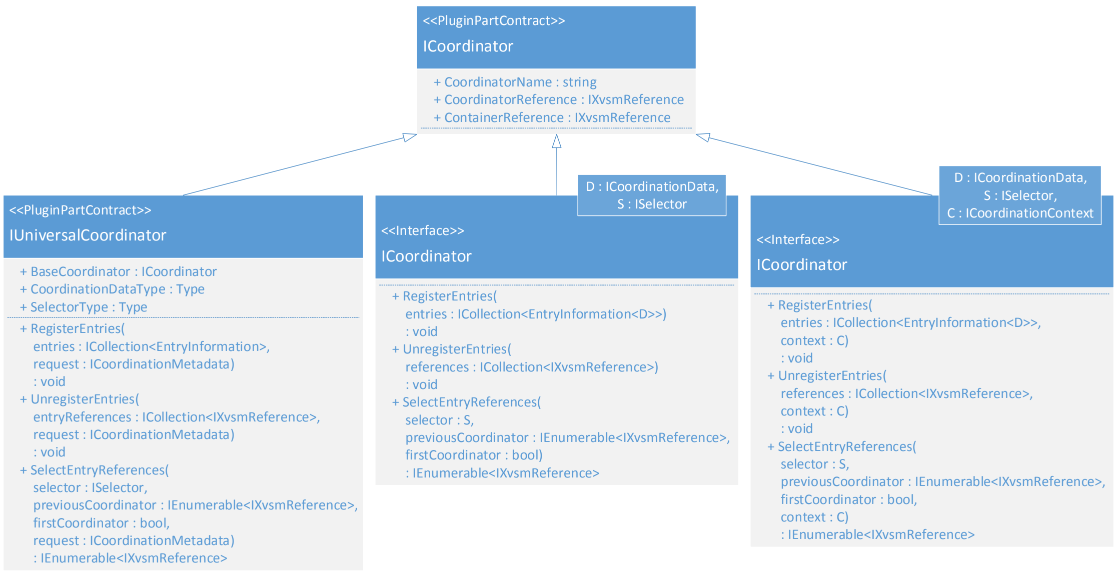

<<PluginPartContract>>
ICoordinator
+ CoordinatorName : string
+ CoordinatorReference : IXvsmReference
+ ContainerReference : IXvsmReference
<<PluginPartContract>>
IUniversalCoordinator
+ BaseCoordinator : ICoordinator
+ CoordinationDataType : Type
+ SelectorType : Type
+ RegisterEntries(
entries : ICollection<EntryInformation>,
request : ICoordinationMetadata)
: void
+ UnregisterEntries(
entryReferences : ICollection<IXvsmReference>,
request : ICoordinationMetadata)
: void
+ SelectEntryReferences(
selector : ISelector,
previousCoordinator : IEnumerable<IXvsmReference>,
firstCoordinator : bool,
request : ICoordinationMetadata)
: IEnumerable<IXvsmReference>
D : ICoordinationData,
S : ISelector
<<Interface>>
ICoordinator
+ RegisterEntries(
entries : ICollection<EntryInformation<D>>)
: void
+ UnregisterEntries(
references : ICollection<IXvsmReference>)
: void
+ SelectEntryReferences(
selector : S,
previousCoordinator : IEnumerable<IXvsmReference>,
firstCoordinator : bool)
: IEnumerable<IXvsmReference>
D : ICoordinationData,
S : ISelector,
C : ICoordinationContext
<<Interface>>
ICoordinator
+ RegisterEntries(
entries : ICollection<EntryInformation<D>>,
context : C)
: void
+ UnregisterEntries(
references : ICollection<IXvsmReference>,
context : C)
: void
+ SelectEntryReferences(
selector : S,
previousCoordinator : IEnumerable<IXvsmReference>,
firstCoordinator : bool,
context : C)
: IEnumerable<IXvsmReference>

Figure 6.21: UML class diagram showing the coordinator contract inheritance without interface members.

The non-generic plugin part contract `ICoordinator` is the base interface of all coordinators. The class diagram shows a plugin part contract and two simple interfaces extending the base interface. These interfaces are generic and could not be realized as true plugin part contracts since the current implementation of the plugin framework does not support generic plugin or plugin part contracts. However, it is possible to use a generic interface in combination with a non-generic base interface that is exported as the plugin part contract. The corresponding plugin part factory simply returns instances of the base interfaces, which then can be manually casted to the concrete generic interface when used. The various coordinator plugin parts must implement one of the generic interfaces as they provide type-safe access to coordination-related objects. *Universal coordinator's*, which are plugin parts with the `IUniversalCoordinator` contract, are internally used as wrappers for the coordinator plugin parts with the generic interfaces. They are responsible to translate general coordinator requests to comply with the concrete generic coordinators and their generic type parameters. The sequence diagrams in Figures 6.22 and 6.23 exemplarily illustrates this delegation process. The `IContainer` plugin part is responsible to register entries on the relevant coordinators, while the `ICapi3` plugin retrieves the entry references from the coordinators.

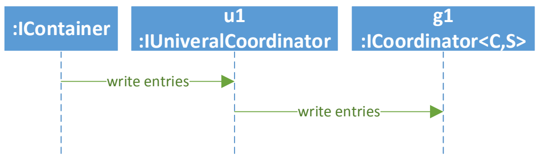


Figure 6.22: The delegation of write operations by `IUniversalCoordinator` plugin parts.

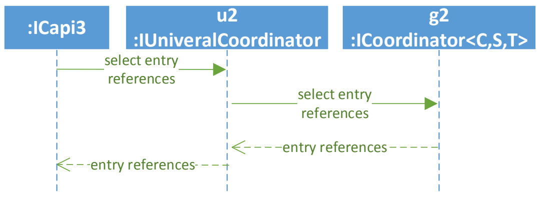


Figure 6.23: Showing the delegation of entry reference selection operations by `IUniversalCoordinator` plugin parts.

Invocations on the coordinator with the `IUniversalCoordinator` interface are delegated to its corresponding generic coordinator implementation. Thus, every instance of `IUniversalCoordinator` stores the corresponding generic coordinator object that was passed through the `ICoordinator` parameter from its instantiation method.

The `RegisterEntries` and `UnregisterEntries` methods of the coordinator interfaces are used to keep track of entry operations. The `RegisterEntries` method is called by the container to provide the coordinator with the coordination data of newly written entries. The generic `EntryInformation<>` objects that are handed over contain the references of the entries and their coordination data. In XVSM.NET the responsibility for the storage of the coordination data is placed on the coordinators. Here, XVSM.NET behaves similar to MozartSpaces and differs from the formal model as it is beneficial to leave the storage of the coordination data at the coordinators. This allows the coordinators to store the coordination data in data structures optimized for entry retrieval operations. For instance, a Label coordinator could store the coordination data in a data structure optimized for lookup operations. The `UnregisterEntries` method is invoked for all removal operations on the container. The references of the entries are provided as parameters to allow the coordinators to remove the stored coordination data.

With the method `SelectEntryReferences` it is possible to retrieve and filter entries with the semantics of the coordinator. An `ISelector` instance is provided that contains metadata about the selection, such as the count or coordinator name. The concrete type in the generic interface may contain additional selection data that the coordinator could use. The coordinators are used to retrieve previously registered entries through specific selectors.

Every coordinator must have a `CoordinatorAttribute` applied to provide metadata about the coordinator. The `Deterministic` property defines whether the results of the coordinator's `SelectEntryReferences` method are repeatable from the same state. This metadata is useful for selection algorithms that can use this information to quickly abort the operation when a required entry from a deterministic coordinator is not available. However, the property does not take concurrent state modifications into account for non-determinism. Details about the selection algorithm are presented in Section 6.3.5. The

`Implicit` property of the attribute denotes whether the coordinator requires manually provided coordination data or not. The container implementation is responsible to check if the provided entries contain the required coordination data. Thus, coordinators that require certain metadata of entries for storage have the property set to `false` to force the provision of manually provided coordination data. When entries without coordination data should be managed by a specific coordinator that has the `Implicit` property set to `false`, the container will abort the operation with an exception. However, there is an approach to customize the process of creating implicit coordination data for coordinators with the `Implicit` property set to `true`. If it is possible to automatically extract the required coordination data from the entry object, then the corresponding instance of the `IImplicitCoordinationDataFactory` plugin contract is used to create the coordination data. Instances of the `IImplicitCoordinationDataFactory` plugin contract are automatically wired through their applied `ImplicitCoordinationDataFacto-ryAttribute`, which contains the property `CoordinationDataType` that specifies the type of the coordination data the `IImplicitCoordinationDataFactory` can create. The container is responsible for the instantiation of these factory plugins and invokes their single method `CreateImplicitCoordinationData` with the coordinator reference, its name, and the entry object as arguments to create the custom coordination data. The final property `LockMode` defines what additional locking is required for the coordinator to function properly. The concrete lock modes of the `CoordinatorLockMode` type are discussed in Section 6.3.6.

With the generic `ICoordinator` interface, the developer of a coordinator simply needs to specify the concrete types that are used for the coordinator as the casting and mapping of the types is handled by other dedicated plugins. In the following, the generic type parameters and their translation operations are described:

ICoordinationData: The coordination data of an entry is stored in instances of this type and is provided to the coordinator when entries are registered. It is possible to simply use the `ICoordinationData` interface as the generic type parameter for coordinators that require no custom coordination data. Some coordinators, on the other hand, require custom coordination data and therefore may provide a custom class with additional properties. This is useful, since little data is provided by default to the coordinator for every entry, as can be seen in Figure 6.24. The non-generic `EntryInformation` class is provided to the universal coordinator, which then casts the included `CoordinationData` property to the generic type of the corresponding generic coordinator instance. The casted type, along with the entry reference, is then packed in a new generic instance of the `EntryInformation` class and provided to the generic coordinator instance. Without a custom coordination data class, the entry reference must be sufficient entry data for the coordinator to conclude the entry registration.

ISelector: For every entry selection operation, a selector object is provided to the relevant coordinator. The selector object contains relevant selection information for

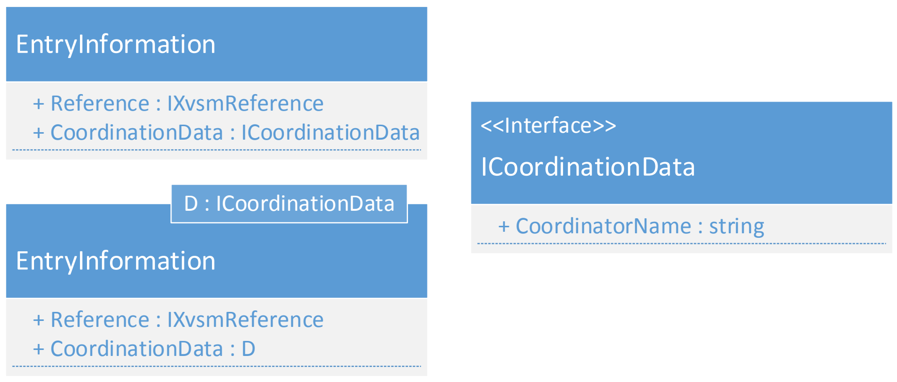

EntryInformation
+ Reference : IXvsmReference
+ CoordinationData : ICoordinationData
D : ICoordinationData
EntryInformation
+ Reference : IXvsmReference
+ CoordinationData : D
<<Interface>>
ICoordinationData
+ CoordinatorName : string

Figure 6.24: UML class diagram showing classes relevant for the entry registration.

the coordinator. Some coordinators do not need any custom selection information and therefore directly use the `ISelector` interface as the generic type parameter. Other coordinators, however, do require custom selection data and use a custom class with additional properties for the selector. The `ISelector` interface contains the coordinator name and the selection count for the coordinator as properties. The universal coordinator is again responsible to cast the selector instance to the concrete type and provide the resulting instance to the relevant coordinator.

ICoordinationContext: This interface is used for the optional generic type parameter to provide extended coordination information for the coordinator. In this case, it only makes sense to use a specialization of the `ICoordinationContext` interface as type parameter, since the base interface does not contain any elements. XVSM.NET already comes with a specialization of this interface, which is illustrated in Figure 6.25. The interface `ITransactionalContext` can be used by

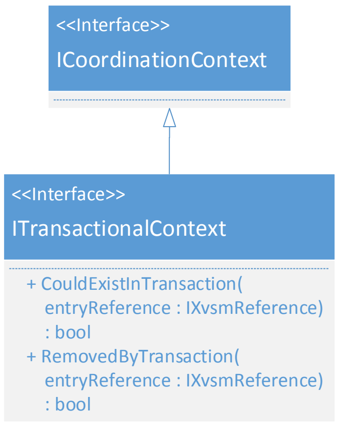
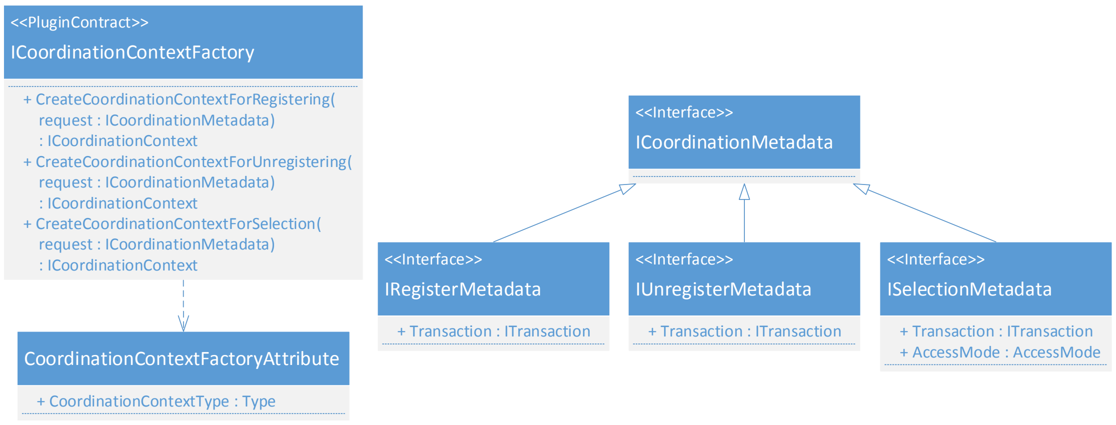


Figure 6.25: UML class diagram showing the `ITransactionalContext` interface.

the coordinator to gain insight of the transactional states of the entries. The two included methods correspond to the equally named methods of the ITransaction interface presented in Section 6.2.3. It would have also been possible to simply provide the used transaction object in the ITransactionalContext instance and let the coordinators directly access the methods through the transaction object, but this would have made the transaction object accessible to the coordinator. This is a consequence of the design decision to separate the coordinators' logic from the internal plugins and plugin parts of XVSM.NET. This coordination context mechanism is extensible and allows to provide custom context objects that extend the ICoordinationContext interface. Context objects are instantiated through plugins implementing the ICoordinationContextFactory plugin contract that are invoked through the universal coordinator instances. Figure 6.26 illustrates the plugin contract used to instantiate context instances and the metadata provided to the factories. It was decided to leave the casting of the concrete



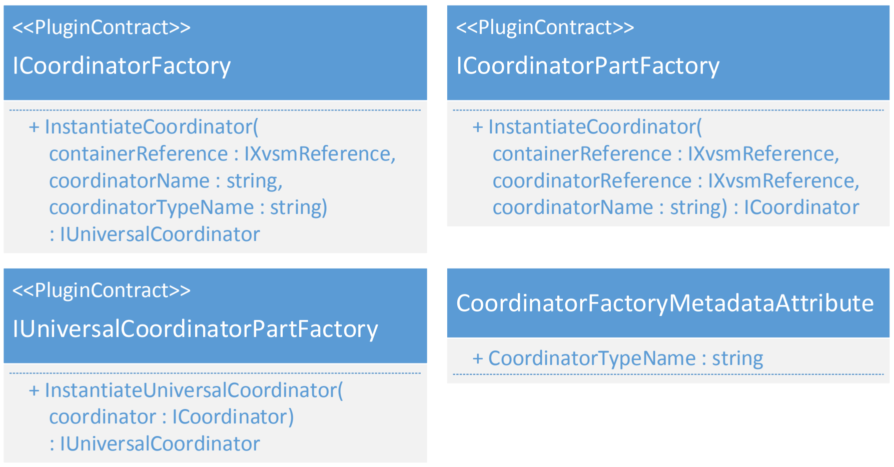
<<PluginContract>>
ICoordinationContextFactory
+ CreateCoordinationContextForRegistering(request : ICoordinationMetadata) : ICoordinationContext
+ CreateCoordinationContextForUnregistering(request : ICoordinationMetadata) : ICoordinationContext
+ CreateCoordinationContextForSelection(request : ICoordinationMetadata) : ICoordinationContext
CoordinationContextFactoryAttribute
+ CoordinationContextType : Type
<<Interface>>
ICoordinationMetadata
<<Interface>>
IRegisterMetadata
+ Transaction : ITransaction
<<Interface>>
IUnregisterMetadata
+ Transaction : ITransaction
<<Interface>>
ISelectionMetadata
+ Transaction : ITransaction
+ AccessMode : AccessMode

Figure 6.26: UML class diagram showing the plugin contract to instantiate custom context objects with related elements.

metadata types to the factories, since later added factories will probably require specialized metadata types anyway. The universal coordinator is responsible to lookup the correct context factory with the plugin service locator by checking the CoordinationContextFactoryAttribute of the context factories. The context factory with an attribute that has the correct context type stored in its CoordinationContextType property is then used for the context instantiation. The universal coordinator casts the resulting context object to the required type and provides it along with the other arguments to the generic coordinator. The provided IRegisterMetadata, IUnregisterMetadata, and ISelectionMetadata methods all contain the ITransaction instance used for the operation, which is required to create the ITransactionalContext instance.

6.3.4 Coordinator Instantiation

The instantiation of the coordinators through the `IContainer` implementation requires several steps, thus, the responsibility is sourced out to a separate plugin that is accessed through its `ICoordinatorFactory` plugin contract. This plugin is responsible to lookup and instantiate the generic coordinator implementation suitable for the specified type, before instantiating and returning a wrapping instance of the `IUniversalCoordinator` interface. The generic coordinator types are not hardcoded in XVSM.NET but make use of a string identifier stored in the plugin part factories' metadata. Figure 6.27 shows the concrete contracts involved in the instantiation.



<<PluginContract>>
ICoordinatorFactory
+ InstantiateCoordinator(containerReference : IXvsmReference, coordinatorName : string, coordinatorTypeName : string) : IUniversalCoordinator
<<PluginContract>>
ICoordinatorPartFactory
+ InstantiateCoordinator(containerReference : IXvsmReference, coordinatorReference : IXvsmReference, coordinatorName : string) : ICoordinator
<<PluginContract>>
IUniversalCoordinatorPartFactory
+ InstantiateUniversalCoordinator(coordinator : ICoordinator) : IUniversalCoordinator
CoordinatorFactoryMetadataAttribute
+ CoordinatorTypeName : string

Figure 6.27: UML class diagram showing plugin contracts required for the instantiation of coordinators.

When the instantiation method of the `ICoordinatorFactory` plugin is called, it creates the coordinator-specific `ICoordinatorPartFactory` based on the provided coordinator type. To lookup the right plugin part factory, the `ICoordinatorPartFactory` implementations must have the `CoordinatorFactoryMetadataAttribute` applied with a string specifying the coordinator type. The instantiation request of the coordinator is delegated to the `ICoordinatorPartFactory`, which instantiates the concrete coordinator and passes the container and coordinator references along with the coordinator name to the coordinator.

After the coordinator has been instantiated and returned to the `ICoordinatorFactory` plugin, the plugin invokes the `IUniversalCoordinatorPartFactory`'s method to wrap the generic coordinator implementation with an `IUniversalCoordinator` plugin part. The `IUniversalCoordinator` instance is returned to the container.

6.3.5 Selection Manager

The plugin implementing the `ISelectionManager` plugin contract, called *Selection-Manager*, contains the single method `Select` to execute selection operations and is called by the `ICapi3` plugin. Figure 6.28 depicts its signature. Its first parameter `selectionData` provides pairs of the used selector and the corresponding coordinator information object, which is used by the coordinator orchestration required for the selection. As the second and final parameter, selection metadata is provided to the selection manager. Through the metadata the operation's transaction as well as its access mode are retrieved by the SelectionManager.

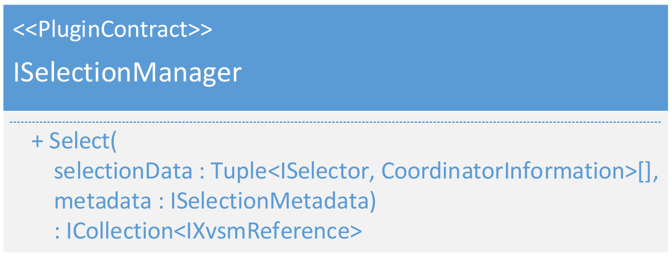

<<PluginContract>>
ISelectionManager
+ Select(
selectionData : Tuple<ISelector, CoordinatorInformation>[],
metadata : ISelectionMetadata)
: ICollection<IXvsmReference>

Figure 6.28: UML class diagram showing the plugin contract of the SelectionManager.

The SelectionManager has several tasks related with the selection of entry references. One of its tasks is to link the coordinators together in the requested evaluation order. Since the coordinators are not generally responsible to enforce the count constraints or ensure the transactional availability of the entry references, these tasks also lie in the responsibility of the SelectionManager. To handle all these tasks, the SelectionManager makes use of a variant of the mediator and pipe-and-filter [Mon+97] style by using `CoordinatorMediator` instances that handle the communication between the coordinators. Thanks to C#'s support of generators through the `yield` keyword [KPS12], a streaming-based coordination implementation has been realized.

The first coordinator yielding entry references is the only coordinator for the selection that has the `firstCoordinator` boolean argument set to `True`. The yielded entry references may then be passed individually by the first mediator instance to the next coordinator and so on. In the default implementation, the first mediator takes a special role among all the mediators. It is responsible to evaluate the transactional availability of the yielded entry references and filter them when necessary. XVSM.NET uses a similar double-checking locking strategy as MozartSpaces by using an optimistic approach that might lead to re-execution of the selection operation, but preserves concurrency, since alternative pessimistic locking strategies would limit concurrent execution significantly [Bar10]. It is important to note that the first mediator only checks if the entry references are available. The transactional acquisition of the entry references is left to the selection methods of `ICapi3` in Section 6.3.9. The transactional availability is checked by applying a try-only acquisition of the entry reference on the provided transaction instance with the supplied access mode. When the entry reference is available it is simply yielded to the next coordinator. However, when the acquisition check fails, an `EntryLockedException` exception is thrown. Since the exception extends from `Capi3Exception`, a `Locked` operation result could be returned when none of the following conditions are satisfied:

1. The selector of the first coordinator has `COUNT_MAX` for the count property. This is the direct consequence of the used selector count property and conforms to the required behavior.

2. The entry reference could not exist in the transaction of the operation. Since the unregistering of entries is delayed until the *FinalCommit* event, the coordinators contain references of entries that were removed in one transaction. With the check for this condition, the mediator automatically filters these entry references.

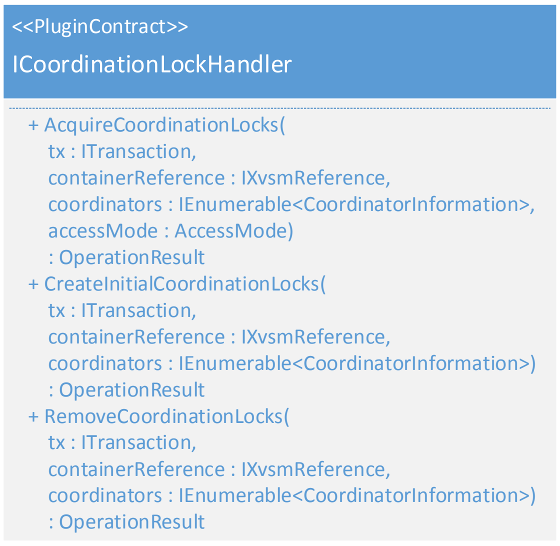
3. The selector of the first coordinator has a discrete value for the selector count property and the first coordinator is not deterministic. This check is an optimization that leads to less `Locked` operation results in some situations. It makes use of the condition that some coordinators do not guarantee a deterministic result and therefore simply filters the unavailable entry reference. However, if such a filtered entry reference leads to too few available entries at a later point, a `Locked` operation result is still returned instead of a `Delayable`.

Checking the transactional availability of entry references is not the only task of mediators, they are also responsible to enforce the selector count properties. This time the task is not solely subject to the top most mediator but is the responsibility of all mediators. Every yielded entry reference of the previous coordinator that is not filtered by the mediator is locally counted in the mediator. When the selector count property is discrete, the responsible mediator stops selection when the discrete count has been reached or raises an exception when the coordinator yields too few available entry references. The possible exception that is raised is a `CountNotMetException`, which is a `Capi3Exception` and results to an operation result with a `Delayable` value.

6.3.6 Concurrency & Locking

Methods of the generic coordinator implementations may be invoked concurrently. This applies especially to the three methods that are used to register, unregister or retrieve entry references. In XVSM.NET, the responsibility to cope with this concurrency lies in the coordinators themselves by default. In most coordinators of the default implementation, this is simply solved by using data structures that are optimized for concurrent usage, such as the `ConcurrentDictionary`. Since the coordinators are unaware of the concrete transaction and used isolation level, it is sufficient to simply execute the invoked methods without special transactional handling. Ensuring transactional correctness is the responsibility of the plugins requesting the coordinator operations, such as `ICapi3` and `IContainer`.

Even though XVSM.NET leaves the concurrency handling to the coordinators, it provides an automatic locking mechanism for entry operations tied to the used transaction. The so-called *coordination lock modes* define the concrete locking behaviors, which are carried out by the *coordination lock handler* plugin implementing the `ICoordinationLock-Handler` plugin contract, shown in Figure 6.29. This is realized in the plugin by using



```
<<PluginContract>>
ICoordinationLockHandler

+ AcquireCoordinationLocks(
    tx : ITransaction,
    containerReference : IXvsmReference,
    coordinators : IEnumerable<CoordinatorInformation>,
    accessMode : AccessMode)
    : OperationResult
+ CreateInitialCoordinationLocks(
    tx : ITransaction,
    containerReference : IXvsmReference,
    coordinators : IEnumerable<CoordinatorInformation>)
    : OperationResult
+ RemoveCoordinationLocks(
    tx : ITransaction,
    containerReference : IXvsmReference,
    coordinators : IEnumerable<CoordinatorInformation>)
    : OperationResult
```

Figure 6.29: UML class diagram showing the `ICoordinationLockHandler` plugin contract.

the transaction system to acquire locks on coordinator and container references. This mechanism was added as an abstraction layer to support coordination-specific locking semantics. The reason was to introduce an indirection that allows to acquire coordinator locks for the semantics of the Vector coordinator, which differs from the other coordinators in this regard and requires a stricter lock mode. However, container locks are also acquired to realize CAPI-3's container locking functionality through this plugin. In addition, it allows the realization of the serializable isolation level. If the serializable isolation level is used in the transaction of the coordination method, the coordination lock mode is increased to the provided minimum boundary of the corresponding isolation level handler.

The coordination lock handler provides three methods to manage the locks on the references. The methods `CreateInitialLocks` and `RemoveLocks` are invoked by the default container plugin to ensure that locks for the container's and coordinators' references are created or removed in alignment with the container's life cycle. The remaining method `AcquireLocks` is then invoked for entry operations with the access mode defining the concrete operation type as additional parameter. Depending on the provided access mode and the used coordination lock mode, specific acquisition requests are issued for the container and coordinator references, as can be seen in Figure 6.30.

The sequence diagram shows the behavior of the coordination lock handler for entry operations that lead to invocation of its `AcquireLocks` method. For better readability, the sequence diagram simplifies the retrieval of the minimum and maximum coordination lock modes by omitting the involved isolation level handler. In the concrete implementation the isolation level handler of the running transaction is accessed to obtain the lock modes

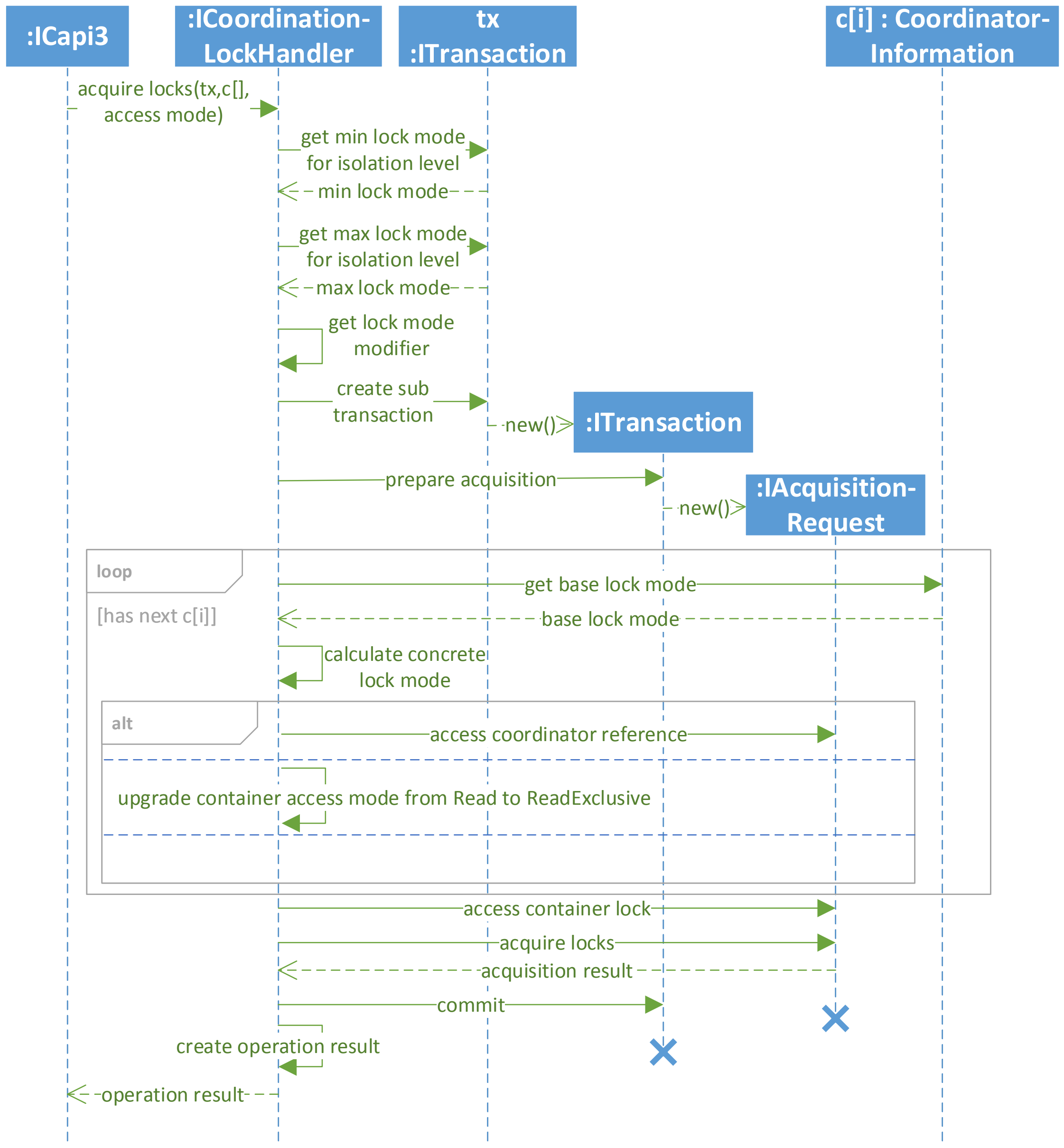


Figure 6.30: UML sequence diagram showing the behaviors of the coordination lock handler's `AcquireLocks` method regarding lock acquisition.

through its `MinCoordinatorLockModeName` and `MaxCoordinatorLockModeName` properties. These lock modes serve as lower and upper bound constraints for the base lock modes specified in the coordinators' attributes. The lock modes have a total order, ascending from requiring no locking to locking the whole space container for access through a single transaction. If the base lock mode is below the minimum coordination lock, for instance, the minimum lock mode is used in the coordination lock handler instead of the base lock mode. The coordinator information objects are iterated and the coordinator attribute is accessed to retrieve the base coordination lock mode. This process was simplified in the sequence diagram as well. With the base coordination lock mode and its boundary constraints, the concretely used lock mode is calculated and applied depending on the provided access mode. The following enumeration shows the supported lock modes in the same ascending order that also applies to the boundary constraints of the base lock mode:

- **FullConcurrency:** All public coordinator methods might be concurrently accessed by different threads and in different transactions. No acquisition is issued for the coordinator's reference. This is the default coordination lock mode that is used in most coordinators.
- **FullConcurrencyWithReadLocks:** All public coordinator methods now acquire shared read-locks on the coordinator references.
- **SingleWrite:** The methods used to register or unregister entries at a coordinator now acquire an exclusive lock on the coordinator's reference while the retrieval method issues no read lock. This means that separate transactions within different transaction hierarchies may not have concurrent write access on the coordinator until the lock holding transaction commits and releases the exclusive lock. As an optimization, this lock mode acquires no read-lock for retrieval operations.
- **SingleWriteOrSharedRead:** This lock mode extends the `SingleWrite` lock mode to acquire shared read-locks for executing the retrieval methods of the coordinators. This is the minimum lock mode of the serializable isolation level and protects against phantom reads. This is realized by practically locking the full coordinator for modifications in a single transaction hierarchy.
- **SingleWriteOrSingleRead:** This coordination lock mode even further minimizes concurrent access by acquiring readexclusive-locks not only for methods modifying coordinators but also for the execution of the retrieval method.
- **SingleContainerAccess:** This most severe coordination lock mode changes the acquisition type of the container's reference into a readexclusive-lock and therefore fully locks the container and all its coordinators for a single transaction hierarchy.

After the access operations for the coordinators' references have been prepared, the container reference gets prepared as well. Depending on the type of operation and used lock modes, the container reference either issues a single read-acquisition or readexclusive-acquisition. When the acquisition could be successfully applied, the sub-transaction is committed and a successful operation result is returned. However, when the acquisition failed, an unsuccessful operation result is returned to the `ICapi3` plugin, which then aborts the operation.

The provided transaction is used to create a new sub-transaction with a repeatable read isolation level that is used for the acquisition operations if the transaction `tx` has a lower isolation level. In the sequence diagram, the transaction `tx` has a lower isolation level such as read committed, thus a further sub-transaction is created. The transaction `tx` corresponds to the sub-transaction of the XVSM formal model, hence the transactions are further nested in this case. If the transaction `tx` had a repeatable read isolation level, it would have been directly used by the coordination lock handler. It makes use of the repeatable read isolation level since lower isolation levels do not support read locks.

6.3.7 Coordinator Implementations

XVSM.NET comes with several default coordinator plugins to provide basic XVSM functionality. The coordinators with implementation details are briefly described in the following:

AnyCoordinator

The AnyCoordinator plugin is the simplest coordinator provided by default in XVSM.NET. It requires no custom coordination data and simply stores the entry references in a data structure for quick lookups. The register or unregister methods store or remove the provided entry references in its local storage, which is then used by the selection method for filtering. The entry references from the previous coordinator that are not stored in the AnyCoordinator instance are removed from the result. When the coordinator is used as the first coordinator, simply all stored entry references are returned. Since the filtering mechanism of the coordinator is applied to each entry reference individually, the coordinator is able to return the result as a stream. In general, the coordinator is indeterministic since its semantics allow to return the entry references in arbitrary order, which could lead to different results for selections with a discrete maximum element count. Regarding concurrency and locking, the coordinator uses the default coordination lock mode and allows full concurrent access to all methods.

FifoCoordinator

The FifoCoordinator plugin stores entry references in a queue data structure to retain the *first-in-first-out* order in which the entries are written to the container. The coordinator requires no custom coordination data for the written entries. The register method simply stores the entry references in the same Fifo order as the method is invoked. Since invocations might occur concurrently through different transactions, it is not guaranteed that all entries, written in a single transaction, are consecutively stored, their partial order, though, is guaranteed. When entries are retrieved through the selection method, the entry references from previous coordinators in the coordinator chain are collected first to be filtered and returned in the Fifo order. Entry references that are not stored in the FifoCoordinator are simply filtered from the result. The coordinator does not support streaming as the entry references of previous coordinators are collected first. The result of the FifoCoordinator is deterministic, since all selected entry references must be included for a semantically correct result. Regarding concurrency and locking, the coordinator uses the default coordination lock mode and allows full concurrent access to all methods.

LifoCoordinator

The LifoCoordinator plugin uses the same concept as the FifoCoordinator with the difference that it stores the entries in a stack data structure to represent the *last-in-first-out* order of the written entries. Every other aspect of the plugin is identical to the FifoCoordinator.

RandomCoordinator

The RandomCoordinator plugin acts identical to the AnyCoordinator for the registering or unregistering of entries. It requires no custom coordination data and stores the entry references in a data structure optimized for quick lookups. However, the selection method differentiates from its AnyCoordinator pendant since it needs to return the result in a shuffled order. The previous coordinator's results need to be collected and filtered first, before they are shuffled to return a permutation. The shuffling algorithm is implemented in the plugin and uses a modern version of the *Fisher-Yates shuffle* from [Knu97]. Since the results of the previous coordinators are captured first, the coordinator does not support streaming. Its selection results are indeterministic as all possible permutations of the result are semantically correct results. All methods of the coordinator can be invoked concurrently from different transactions.

KeyCoordinator

The KeyCoordinator plugin stores entries with associated strings, here called *keys*. Every entry that is registered in the coordinator must have a single unique key assigned to it, which is provided by custom coordination data. Registering multiple entries with the same key leads to exceptions and aborts the operation. For the selection a single key must be specified in the used selector, which limits the coordinator to return only entries with the specified key associated. As a consequence of the key's unique constraint, the method only returns a single available entry reference at most. This unique constraint leads to a rather complex handling of the registering and unregistering of entry references. Since it is possible to remove an entry from the container within a non-committed transaction hierarchy, the coordinator must have insight of specific transactional information, which is established with the `ITransactionalContext` object. The coordinator allows to store multiple entry references for a single key when there is no possibility that these entries could coexist after committing or rollbacking any involved transactions. When entries are selected, the possible entry references are determined first before using them to filter all other entry references from the previous coordinator's result. The KeyCoordinator is deterministic since at most one transactionally available entry reference may be returned and no other result would be semantically correct for the coordinator. Even though all methods of the coordinator may be invoked in parallel, concurrent execution of register or unregister operations of entries with the same key is not possible.

LabelCoordinator

The LabelCoordinator can be seen as a relaxed variant of the KeyCoordinator that associates entry references with strings as well. However, it uses so-called *labels* instead of keys that do not have to be unique in the coordinator. Every registered entry still requires exactly one associated label, which is provided by custom coordination data. The used selector contains a single specified label that is used as a filter to only return elements with the same associated label. In contrast to the KeyCoordinator, the LabelCoordinator does not need transactional insight of provided entry references in order to store them, since this is only required to enforce the uniqueness constraint of keys. This leads to a simpler implementation than its KeyCoordinator counterpart. The selection algorithm, however, works identical as it internally retrieves all possible entry references for the provided label from the local storage, before it consecutively filters the results of the previous coordinator. Consequently, this allows the support of streaming. When multiple entries have the same label, all entries are returned in an arbitrary order. Hence, the coordinator is indeterministic. Concerning the concurrency, the LabelCoordinator allows concurrent access on its methods and has no restriction on registering entry references with the same label concurrently.

VectorCoordinator

The VectorCoordinator allows to store and retrieve entry references by indices. Entry references are stored in a list and can be either inserted at a specified index or appended at the end. The concrete insertion point is defined with an integer through the custom coordination data. When entries are selected, a custom selector implementation is used to provide the required index. Whenever entry references are added or removed, the indices are shifted in order to ensure continuous numeration. The realization of this constraint leads to the usage of a more severe coordination locking mode. Thus, the VectorCoordinator uses the `SingleWrite` coordination lock mode that prevents execution of the register and unregister methods through different transaction hierarchies. However, if the same VectorCoordinator with multiple entries is accessed from one transaction that removes an entry, read access from other transaction hierarchies is possible as long as the selected entry references can be acquired. For instance, if an entry is removed within one running transaction hierarchy, another concurrently running transaction hierarchy cannot acquire read access to the same entry with the default isolation level. Internally, the coordinator stores the entry references in a list data structure that allows quick insertions at random positions. The selection method iterates through the list and returns a sub-list for the specified selection index and count. When the coordinator is not the first coordinator, the result of the previous coordinator is used to ensure that all entry references of the sub-list are available.

LindaCoordinator

The LindaCoordinator imitates the coordination behavior of the original Tuple Space with the Linda coordination language [Gel85]. Instead of tuples, the template matching algorithm compares objects and their fields. Analog to the MozartSpaces implementation, the entry classes and their fields need to be annotated with attributes in order to be used for the template matching. The `XvsmQueryableAttribute` is applied to classes with an optional property that allows automatic indexation of their fields. For manual indexation the `IndexAttribute` may be applied to fields. The implementation of the template matching algorithm is conceptionally equivalent to the MozartSpaces implementation with the exception that it allows the matching of entries with different class types as long as their indexed fields are compatible. The coordinator requires no explicit coordination data provided by the developer, since entry objects themselves are sufficient. However, the default implicit coordination data objects do not include the entry object. This issue is solved through a concrete plugin implementation of the `IImplicitCoordinationDataFactory` plugin contract that provides the `LindaCoordinationData`. The coordinator applies the template matching consecutively at the entry references and yields the matches. The implementation allows full concurrent access on all its methods.

LeastUsedCoordinator

The LeastUsedCoordinator orders the stored entry references dynamically, depending on the number of their read accesses. Whenever entries are selected in the coordinator for readonly operations, the internal counters of the resulting entry references are incremented. When entries are selected, the coordinator sorts the resulting entry references by their internal counter in ascending order. Hence, the entries that are least accessed are returned first. The coordinator requires neither custom coordination data nor a custom selector but uses a plugin aspect to increment the access counters. The formal model recommends the usage of an accountant function that is invoked for every successful read operation on the coordinators. This accountant function was neither directly implemented in MozartSpaces nor in XVSM.net, since it is not needed for regular coordinators. The LeastUsedCoordinator is the only coordinator coming with XVSM.net that could have made use of this function. Nonetheless, it is possible in XVSM.net to implement the LeastUsedCoordinator by simulating the accountant function through a plugin aspect on the `ICapi3` plugin. This aspect intercepts the invocation of `ICapi3`'s `ReadEntries` method and increments the internal counters for successfully read entries. Furthermore, the coordinator is fully thread-safe and uses the default locking mode. Since its selection method returns the entry references in a specific order, it is deterministic and does not support streaming. It is deterministic even though parallel read access could change the specific order, since the deterministic property is defined that a re-evaluation of the selection from the same state would also produce this result.

Table 6.1 summarizes the properties of the coordinators.

	Deterministic	**Implicit**	**Streaming**	**Lock Mode**
Any	✗	✓	✓	`FC`
Fifo	✓	✓	✗	`FC`
Lifo	✓	✓	✗	`FC`
Random	✗	✓	✗	`FC`
Label	✗	✗	✓	`FC`
Key	✓	✗	✓	`FC`
Vector	✓	✗	✗	`SW`
Linda	✗	✓	✓	`FC`
LeastUsed	✓	✓	✗	`FC`

Table 6.1: Summary of the default coordinator plugins and their properties.

6.3.8 Writing Entries

CAPI-3, as the coordination layer, provides the functionality to write entries in containers, which is realized with the `WriteEntries` method of `ICapi3`. The method signature includes the active transaction, the reference of the container, and the entry objects as parameters. The entry objects are provided in a collection of `ICoreEntry` instances, which contain the entry reference and data along with explicit coordination data. Figure 6.31 depicts the behavior of the method on a successful execution.

The sequence diagram focuses on a successful execution for an uncluttered visualization. The method's implementation starts by using the container reference to lookup the corresponding container instance from the container registry plugin. When no container was found, the method execution will be aborted with a `NotOk` operation result. In the case of a successful lookup, the transaction instance is used to create a new sub-transaction to pool the following transaction acquisitions together for a better isolation. Only when the sub-transaction is committed, all acquisitions are propagated to its parent transaction. In the case of a possible abortion, the parent transaction will not be affected by the temporary acquisitions. The entries with their coordination data are included in the `UpdateAndCheckEntriesForObligatoryCoordinationData` method invocation of the container. The container then checks whether they contain the required coordination data and automatically inserts implicit coordination data instances, if possible. The detailed behavior of the method has been covered in Section 6.3.1. Missing coordination data will lead to the abortion of the CAPI-3 method with the returning of the `NotOk` value. Assuming the entries do have all required coordination data, the next step is to retrieve information about the coordinators used for the entries. This information is returned in the form of the type `CoordinatorInformation` and includes the reference of the coordinator and its `CoordinatorAttribute`. The coordinator information objects are provided to the `CoordinationLockHandler` along with the container reference and the sub-transaction instance. Depending on the used coordination lock modes, the `CoordinationLockHandler` instance acquires transactional locks in the context of the provided sub-transaction. If the `CoordinationLockHandler`

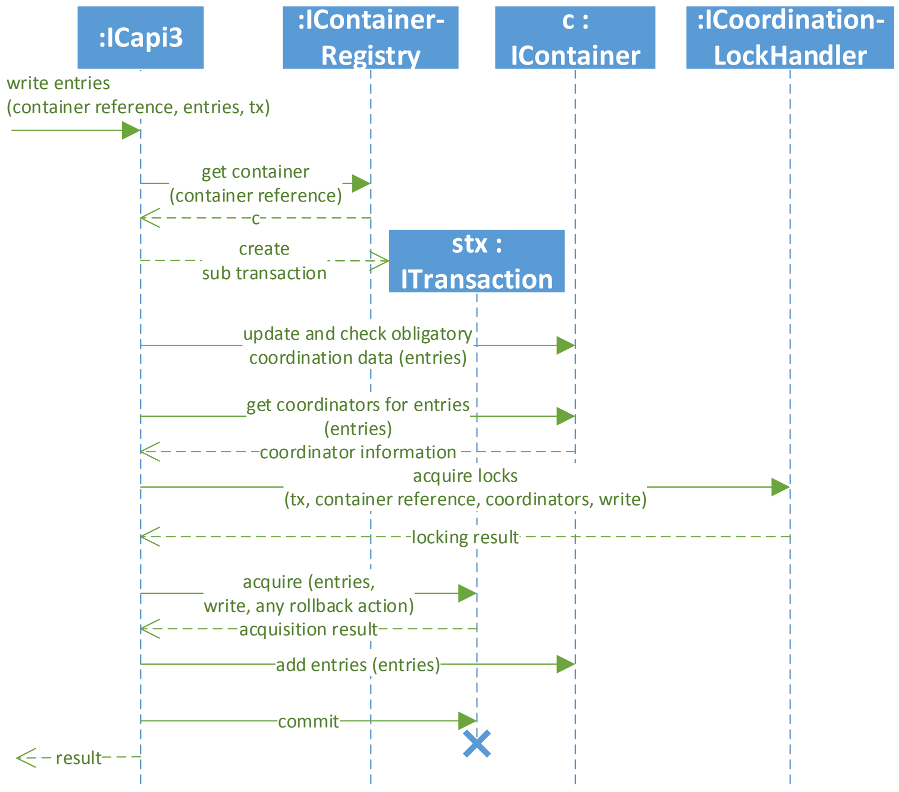

:ICapi3
:IContainer-Registry
c : IContainer
:ICoordination-LockHandler
write entries (container reference, entries, tx)
get container (container reference)
c
create sub transaction
stx : ITransaction
update and check obligatory coordination data (entries)
get coordinators for entries (entries)
coordinator information
acquire locks (tx, container reference, coordinators, write)
locking result
acquire (entries, write, any rollback action)
acquisition result
add entries (entries)
commit
result

Figure 6.31: UML sequence diagram showing a successful operation of writing entries in CAPI-3.

cannot successfully acquire the locks, though, the CAPI-3 method will be aborted and return with the `Locked` value.

After successful acquisitions of the `CoordinationLockHandler`, the entry references must be acquired as well. The sequence diagram simplifies the visualization of the entry acquisition for better readability and excludes the `IAcquisitionRequest` object. Even though technically, this acquisition could fail as well, the entries are assumed to use new, unused references and therefore should always be successfully acquirable. Right after the successful acquisition, the entries are stored and registered through the container. In addition to storing the entries, the container is responsible to register the entries at the coordinators as well, which is depicted in Figure 6.32. Depending on the concrete semantics of the coordinators, it is possible that the registration produces an error and must be aborted. This is realized by raising `Capi3Exceptions` that are caught in the CAPI-3 methods and lead to an abortion with an unsuccessful operation result depending on the concrete exception. As such an abortion could happen after some entries might be already stored and registered, a compensation action must be used for such situations. When the entries are acquired, an action for the *AnyRollback* event is provided to compensate the changes of the CAPI-3 method invocation. Since the lock

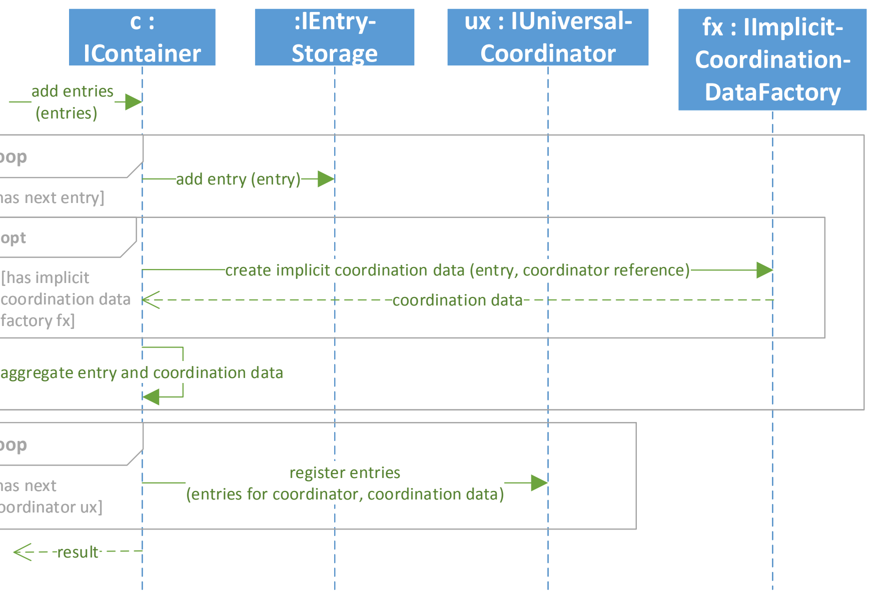

c : IContainer
:IEntry-Storage
ux : IUniversal-Coordinator
fx : IImplicit-Coordination-DataFactory
add entries (entries)
loop
[has next entry]
add entry (entry)
opt
[has implicit coordination data factory fx]
create implicit coordination data (entry, coordinator reference)
coordination data
aggregate entry and coordination data
loop
[has next coordinator ux]
register entries (entries for coordinator, coordination data)
result

Figure 6.32: UML sequence diagram showing a successful writing of entries to a container.

acquisitions are compensated automatically, the only necessary compensation task left is to remove and unregister the entries. To realize this, the `RemoveEntries` method of the container is invoked with all entry references as parameters. However, assuming the storing and registering of the entries went successfully, the sub-transaction is committed and the CAPI-3 method returns with a successful operation result with the `Ok` value.

6.3.9 Selecting Entries

CAPI-3 provides several methods that internally select entries. The `ReadEntries`, `TakeEntries`, `DeleteEntries` and `TestEntries` methods are all based on the same internal principle and share their implementation. Figure 6.33 shows a successful application of the `TakeEntries` algorithm implementation as an example. Nevertheless, we will discuss the algorithm differences in the other selection operations as well.

The `TakeEntries` method is invoked with the container reference, the active transaction and the used selectors for the entry retrieval. The algorithm starts by using the container reference to retrieve the container implementation from the container manager. If the container reference could not be found, the whole method is aborted with a `NotOk` value as operation result.

If the container was successfully retrieved, it is used to gather information about the required coordinators by providing their names. The coordinator information objects are later used for the CoordinationLockHandler to possibly acquire locks on the coordinator references. A new sub-transaction is created that allows to atomically commit or rollback transactional acquisitions of this method and will be used for any further task requiring

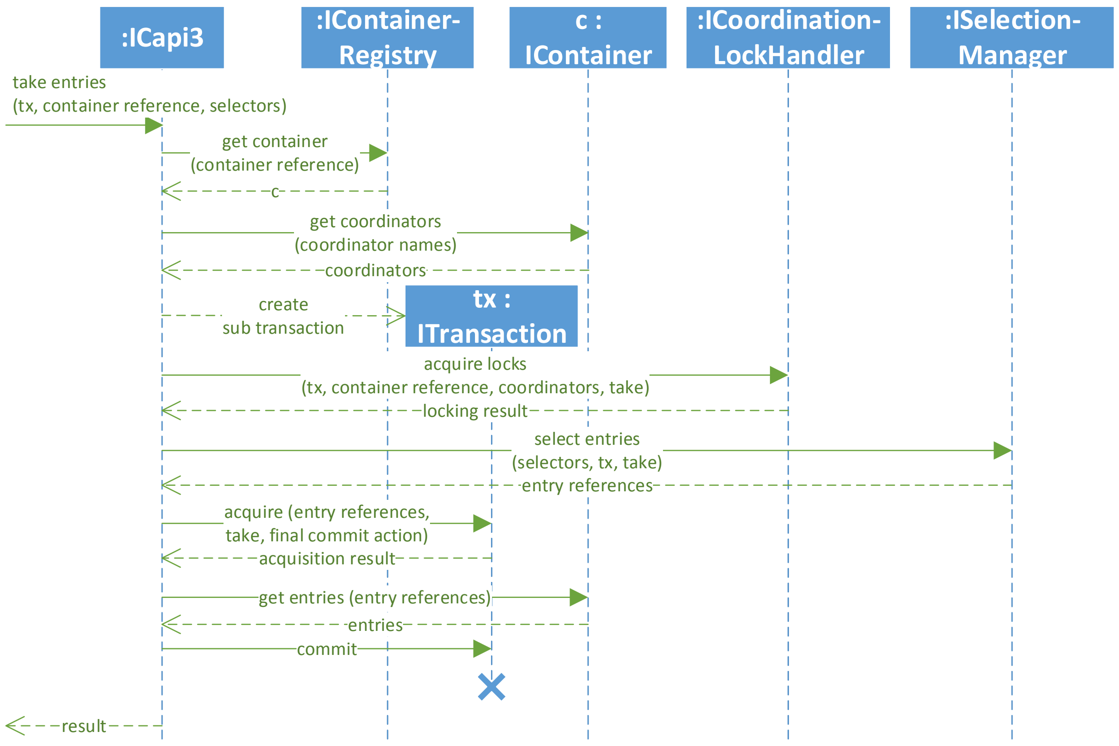

:ICapi3
:IContainer-Registry
c : IContainer
:ICoordination-LockHandler
:ISelection-Manager
take entries (tx, container reference, selectors)
get container (container reference)
c
get coordinators (coordinator names)
coordinators
create sub transaction
tx : ITransaction
acquire locks (tx, container reference, coordinators, take)
locking result
select entries (selectors, tx, take)
entry references
acquire (entry references, take, final commit action)
acquisition result
get entries (entry references)
entries
commit
result

Figure 6.33: UML sequence diagram showing a successful take operation in CAPI-3.

a transaction instance.

The CoordinationLockHandler is now invoked, with the container reference, the coordinator information and the sub-transaction instance. The concrete type of the selection operation is mapped to an access mode value and provided to the CoordinationLockHandler, since the CoordinationLockHandler needs to be aware of the selection type. `Take` or `Delete` selection types are mapped to the `Take` access mode and all other selection types to the `Read` access mode. If the CoordinationLockHandler is not able to acquire all locks successfully, the execution of the selection method is aborted with an unsuccessful operation result. When the coordination locks could be acquired though, the SelectionManager plugin is used to select the entry references by providing the transaction instance, the selectors and the mapped access mode. It is possible that a `Capi3Exception` could get raised, which will lead to an abortion as well. Assuming the entry references were successfully selected, they must be transactionally acquired. To realize this, the selection method creates an acquisition request and atomically acquires all entry references with the mapped access mode. For `Take` or `Delete` selections, the acquisition request includes a transactional action as well. Only when the FinalCommit event takes place, the entries will be unregistered from the coordinators and removed from the container. We now assume that the entry references have been successfully acquired. Depending on the selection type, the entry instances are looked up through the container's GetEntries method. The lookup is only issued for Read or Take selection types, Test and Delete simply return the count of affected entry references. Finally, the sub-transaction is committed and the operation result is returned.

As we have discussed the method's behaviour when the acquisition of the entry references has been successfully executed, we should now take a look at the case when the acquisition fails. Since the selection method can be executed by concurrent threads, it is possible that entry references are not available anymore when they are transactionally acquired. This is a consequence of the optimistic locking approach used by the SelectionManager. A trivial solution is to simply retry the operation at a later point by returning with a `Locked` operation result and let the runtime reschedule it.

6.4 Runtime

The CAPI-4 layer defines the XVSM runtime and is responsible for scheduling and executing XVSM operations either on a local or remote XVSM core. Since this thesis turns its focus on the modular structure of XVSM and the implementations of the core plugins of the CAPI layers below the runtime, its implementation of the contracts deviate from the formal model for a simpler implementation and to show the benefits of the modular architecture.

6.4.1 Overview

The plugins that are involved in the runtime are depicted in Figure 6.34. The `IXvsmCore` plugin can be described as the bridge between the client and the space. It provides access to an `ICapi` and `IRequestContextFactory` implementation via its two readonly properties. `ICapi` is no plugin or plugin part but provides an interface comparable to MozartSpaces' and the formal model's CAPI interface. XVSM.NET provides a Peer Model implementation of the `IXvsmCore` plugin that uses the Peer Model for communication and coordination of request execution. Details of the Peer Model implementation are covered in Section 6.4.2. The `IRequestContextFactory` plugin may be used to create a request context object that can be passed to several methods of the CAPI interface.

Since the methods of the CAPI interface are synchronous, a blocking implementation is required, which is realized through the `IRequestResponseHandler` plugin. It provides a method that consumes requests and blocks until another of its methods is invoked with the corresponding response object.

The instance of the `ICapiServiceExecutor` plugin is able to process arbitrary requests by dynamically delegating the request to the registered concrete service plugin of the corresponding `ICapiService` plugin. This is implemented by using the Plugin Service Locator and matching the concrete type of the request object with the defined metadata attributes on the `ICapiService` plugins through the `ICapiServiceMapper` plugin. Every `ICapiService` plugin handles exactly one unique request type and returns

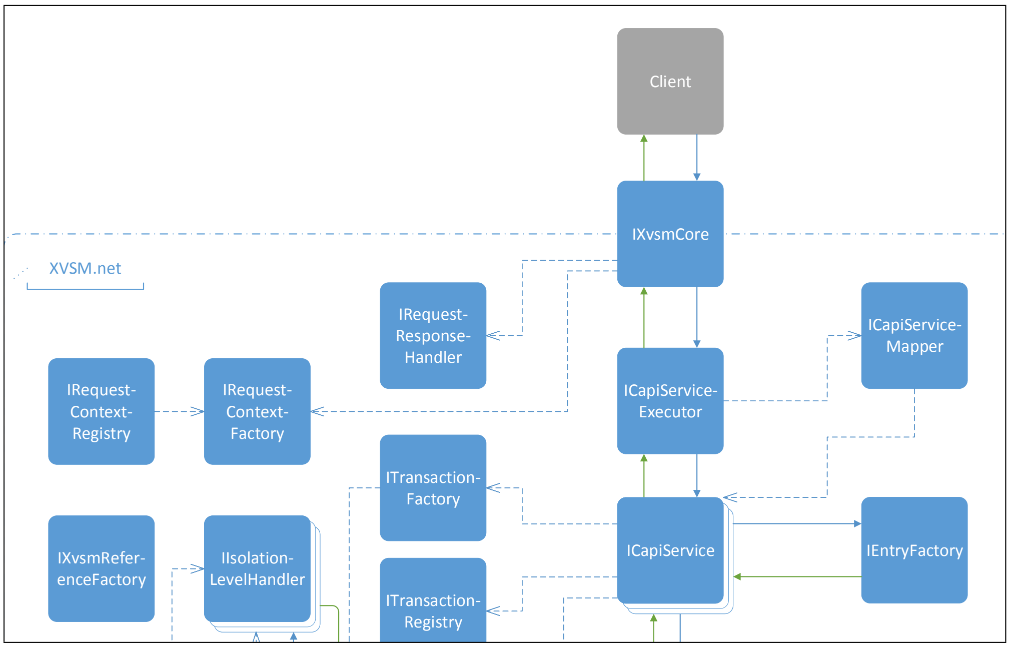


Figure 6.34: Extract of the dependency and architectural overview showing runtime-relevant components.

an object with a corresponding response type. Request and response objects have a one-to-one relation and are associated by having the same request id. The service plugins are stateful and keep shared instances of the `ICapi3`, `ITransactionRegistry`, `IContainerRegistry` and other space-relevant plugins.

The `ICapi` interface, as shown in Listing 6.5, was designed for developers that are familiar with the MozartSpaces' `CAPI` realization. Thus, C#'s asynchronous programming language features, such as `async` and `await` were not used. However, as C# supports default arguments in method signatures, the number of explicit method overloads could be reduced.

```
// Container Operations
IContainerReference CreateContainer(string name = null, string spaceUri = null, int
     size = XvsmConstants.ContainerSize.UnboundContainer,
     IEnumerable<AbstractCoordinator> coordinators = null, ITransactionReference
     transactionReference = null, string isolationLevel = null, IRequestContext
     requestContext = null);

IContainerReference LookupContainer(string name, string spaceUri = null, long
     timeOutInMilliseconds = XvsmConstants.RequestTimeout.TryOnce,
     ITransactionReference transactionReference = null, string isolationLevel = null,
     IRequestContext requestContext = null);

void LockContainer(IContainerReference containerReference, ITransactionReference
     transactionReference = null, string isolationLevel = null, IRequestContext
     requestContext = null);

void DestroyContainer(IContainerReference containerReference, ITransactionReference
     transactionReference = null, string isolationLevel = null, IRequestContext
     requestContext = null);

```

```
// Entry Operations
void Write(IEnumerable<Entry> entries, IContainerReference containerReference, long
    timeOutInMilliseconds = XvsmConstants.RequestTimeout.TryOnce,
    ITransactionReference transactionReference = null, string isolationLevel = null,
    IRequestContext requestContext = null);

IEnumerable<TEntry> Read<TEntry>(IContainerReference containerReference,
    IEnumerable<ISelector> selectors = null, long timeOutInMilliseconds =
    XvsmConstants.RequestTimeout.TryOnce, ITransactionReference transactionReference =
    null, string isolationLevel = null, IRequestContext requestContext = null);

IEnumerable<Entry> Read(IContainerReference containerReference, IEnumerable<ISelector>
    selectors = null, long timeOutInMilliseconds =
    XvsmConstants.RequestTimeout.TryOnce, ITransactionReference transactionReference =
    null, string isolationLevel = null, IRequestContext requestContext = null);

IEnumerable<TEntry> Take<TEntry>(IContainerReference containerReference,
    IEnumerable<ISelector> selectors = null, long timeOutInMilliseconds =
    XvsmConstants.RequestTimeout.TryOnce, ITransactionReference transactionReference =
    null, string isolationLevel = null, IRequestContext requestContext = null);

IEnumerable<Entry> Take(IContainerReference containerReference, IEnumerable<ISelector>
    selectors = null, long timeOutInMilliseconds =
    XvsmConstants.RequestTimeout.TryOnce, ITransactionReference transactionReference =
    null, string isolationLevel = null, IRequestContext requestContext = null);

int Test(IContainerReference containerReference, IEnumerable<ISelector> selectors =
    null, long timeOutInMilliseconds = XvsmConstants.RequestTimeout.TryOnce,
    ITransactionReference transactionReference = null, string isolationLevel = null,
    IRequestContext requestContext = null);

int Delete(IContainerReference containerReference, IEnumerable<ISelector> selectors =
    null, long timeOutInMilliseconds = XvsmConstants.RequestTimeout.TryOnce,
    ITransactionReference transactionReference = null, string isolationLevel = null,
    IRequestContext requestContext = null);

// Transaction Operations
ITransactionReference CreateTransaction(long timeOutInMilliseconds =
    XvsmConstants.RequestTimeout.TryOnce, string isolationLevel = null, string
    spaceUri = null, IRequestContext requestContext = null);

void CommitTransaction(ITransactionReference transactionReference, IRequestContext
    requestContext = null);

void RollbackTransaction(ITransactionReference transactionReference, IRequestContext
    requestContext = null);

// Misc
void Shutdown(string spaceUri = null, IRequestContext requestContext = null);
```

Listing 6.5: The method signatures of the `ICapi` interface.

6.4.2 Peer Model Implementation

As already mentioned in Section 3.2, the Peer Model and its .NET implementation PeerSpace.NET are used for the runtime implementation. We will describe the used peers, sub-peers, and wirings in the following.

As depicted in the Figure 6.35, a working XVSM.NET setup with the PeerSpace plugin uses two different kind of peers, the *RuntimePeer* and *CorePeer*. The RuntimePeer acts as a facade that encapsulates the runtime and scheduling logic. The CorePeer is a sub-peer that is responsible for the execution of space operations.

Figure 6.35 shows the wiring of the RuntimePeer that moves the request object (RQ1) from its PIC to the CorePeer's PIC. Please note that only two exemplary wirings of the

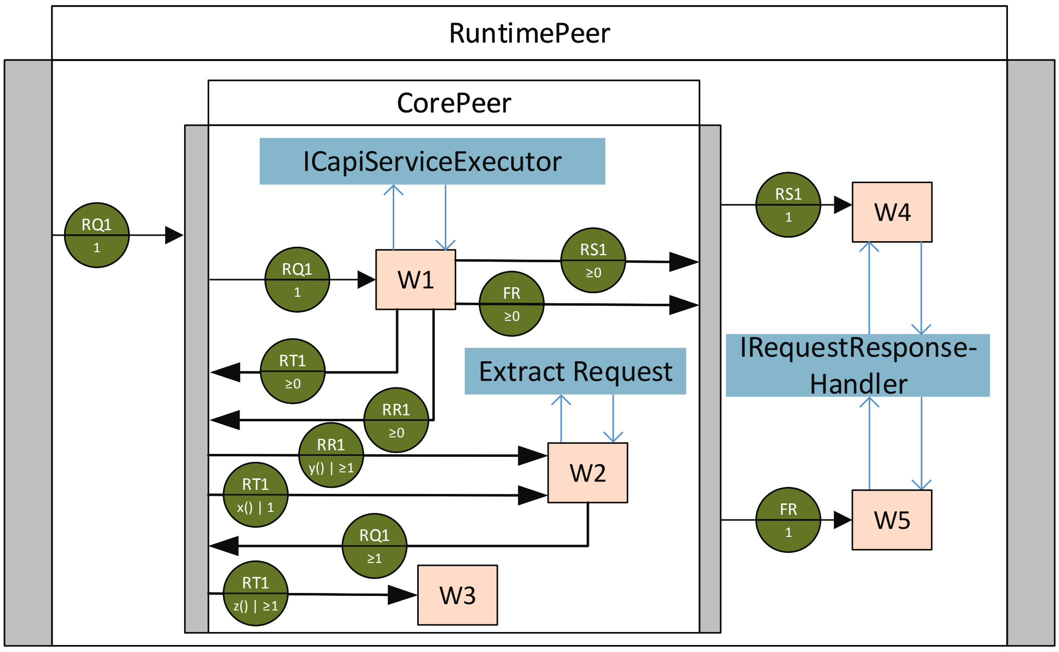


Figure 6.35: The runtime Peer Model implementation, showing CAPI (W1), rescheduling (W2, W3), and response wirings (W4, W5) for an embedded space.

CorePeer are depicted in the figure to ease visualization. The wiring W1 is an exemplary CAPI wiring that takes the request object RQ1 from the CorePeer's PIC and passes it to the `ICapiServiceExecutor` plugin for execution. The result of the execution is returned to the service with additional meta information about the execution. The execution can lead to three different types of outcome: it can be successful, it can fail, and it can be rescheduled. Depending on the concrete outcome, the execution response contains different information that is interpreted by the wiring's service. For instance, if RQ1 leads to a successful execution, its response object (RS1) is emitted and moved to the CorePeer's POC in addition to emitting a reschedule token (RT1) describing the succeeded operation to its PIC. The type of the required reschedule token would be included in the execution's meta information. The reschedule token is emitted to implement basic rescheduling semantics required for the XVSM runtime. As an example, this allows the implementation to automatically retry a take operation after a previously locked entry gets available again. In comparison with MozartSpaces, however, the rescheduling implementation is not as sophisticated, thus, leaving room for improvement.

The execution of the CAPI service might fail and could lead to a failed response that should immediately notify the blocking client. In this case a failed response object FR is emitted by the wiring's service execution context and moved to the CorePeer's POC; the emission of a reschedule token is not required in this case. Since the CAPI-3 layer of XVSM not only defines succeeding and failing operations, but also temporarily failing operations (`Delayable`, `Locked`) that should be rescheduled, depending on the

CAPI execution result, the original request object RQ1 is wrapped, along with metadata describing the required triggers to retry the operation, in a retry request object (RR1). This retry request object is emitted and moved to the CorePeer's PIC. PeerSpace.NET's support for time-to-live (TTL) properties on entries are used on retry request objects to implement the timeout support, as the request object might include a maximum timeout the client should be blocked. In addition to the retry request objects with a TTL, a failed response object with a time-to-start (TTS) is emitted that becomes visible to the space after the retry request has timed out. In case of a timed out request, the failed response object is handled and the failure is reported to the client. However, if the request object could be successfully handled within its TTL, the failed response object is discarded when it becomes visible to the W5 wiring.

The second wiring in the Figure 6.35 (W2) contains two input links that are used to take the retry request objects RR1 along with the reschedule token RT1 from the PIC. The conditions $x()$ and $y()$ of the guards use the metadata of the rescheduled request and the reschedule token to decide whether the requests should be retried. An example for the conditions would be a rescheduled insert operation due to a full container. In this case $x()$ could be satisfied when RT1 was emitted due to a take or delete operation, and $y()$ is satisfied if it awaits such a removal operation. If all guards are satisfied, the original request object RQ1 is extracted from RR1 and moved back to the CorePeer's PIC in order to be consumed by the W1 wiring again for retrial. Since a single reschedule token could lead to the retrial of multiple rescheduled request, the wiring takes all RR1 request objects that satisfy the guards.

To prevent the immediate retrial of a rescheduled request because of a pre-existing reschedule token, the Wiring W3 removes reschedule tokens that cannot be used at the moment to reschedule requests, which is checked by the $z()$ function.

The wirings W4 and W5 are used to handle the response objects. When W4 consumes a response object RS1 or W5 consumes a failure response FR, they invoke the `IRequestResponseHandler` plugin with the consumed entry. The waiting client will be notified directly through the `IRequestResponseHandler` and continues execution.

The wiring W1 may be seen as a *wiring type* or placeholder for various wirings that differ on the request object type and the response object type. Since we allow to dynamically register `ICapiService` plugins, the wirings are dynamically created as well. The `CapiServiceHandlerMetadata` attributes of the plugins are used to retrieve the request and response object types per plugin. The wiring W2 is an exemplary wiring. The concrete implementation uses three different wirings that kick-start the retrial depending on concrete triggers: `retryAfterEntryRemoval`, `retryAfterEntryInsert`, and `retryAfterCommitOrRollback`. When a removal or insert operation is successfully executed, the first two wirings, or after a transaction is committed or rolled back, the last wiring, restores the request objects of rescheduled requests that are waiting for these triggers.

Setups with a remote space core require two RuntimePeers. The RuntimePeer running on a client host acts as a proxy between the client code and a remote RuntimePeer. Requests are serialized and transferred over the network in order to be processed by the RuntimePeer and passed to its CorePeer.

Analogue to the dynamically created wirings of the CorePeer, there are two types of wirings in the RuntimePeer responsible for the remote communication and one in the CorePeer. The first wiring of the RuntimePeer is used to move request objects to the PIC of the remote RuntimePeer and providing the address of the client host's peer in the metadata. The remote host executes the request as if the request object would have been locally written to its PIC. However, instead of invoking the `IRequestResponseHandler`, an additional wiring type in the CorePeer moves the response object back to the client host's PIC. The second wiring type of the RuntimePeer for remote communication consumes the response object from the PIC and resolves the `IRequestResponseHandler` instance blocking the client's request.

6.4.3 Request Context

The request context in XVSM provides a shared state that is passed from the CAPI-4 API to aspects and coordinators. It allows, for instance, aspects to access the data and modify it. Thus, it is a shared memory between aspects and coordinators. Instead of passing a request context object through the call-stack from CAPI-4 to the coordinators, XVSM.NET uses its plugin framework.

```
public void SimpleRequestContextExample() {
  using (var ctx =
     requestContextRegistry.RegisterRequestContext()) {
    // store data in the request context
    ctx.RequestContext.SetProperty("myProp", "test");
    MyMethod();
  }
}
private void MyMethod() {
  var myProp = requestContextRegistry.RequestContext["myProp"];
  // myProp == "test"
}
```

Listing 6.6: Example showing usages of the request context through accessing the default `IRequestContextRegistry` plugin.

The `ISharedPluginMemory` plugin is used as the foundation for the `IRequestContextRegistry` and `IRequestContextFactory` plugins. The latter plugin is simply used to create empty request context objects. However, the implementation of the `IRequestContextRegistry` makes use of the `ISharedPluginMemory` plugin to associate the request context to a shared plugin memory. To access the stored request context through the `ISharedPluginMemory` plugin, a shared identifier must be used.

It was considered passing a request identifier as a parameter to all methods that could possibly access the request context, but this approach was discarded since the original idea of using the plugin mechanism to access the request context was to avoid adding such a separate parameter to all possible method signatures. Thus, instead of manually passing such an identifier, the *thread identifier*, which is globally accessible but unique per request, was used. When a CAPI-3 method is executed, the thread identifier is registered for the provided request context until the method execution is finished. After that the registered thread identifier is removed. For situations where sub-threads are spawned, it is possible to register them for the same request context until their execution is finished. The implementation uses C#'s `IDisposable` interface for a simple and explicit way to unregister the thread identifier, as can be seen in Listing 6.6.

This listing shows an example where a string `test` is stored in the request context under the key `myProp`. The default implementation of the request context object allows to access properties through C#'s custom *Indexer* feature, which is shown in the listing.

In addition, the request context object implementation also provides prefixes, analogue to the MozartSpaces implementation, which can be specified in an overload for the methods `SetProperty` and `GetProperty`.

CHAPTER 7

Evaluation

In this chapter, we evaluate the introduced XVSM implementation regarding its extensibility properties at first by comparing them with other XVSM implementations. A performance analysis is conducted with focus on lower CAPI layers. The stated requirements from Chapter 4 are analyzed regarding their fulfilment. Finally, the chapter concludes with a retrospective analysis concerning the suitability of the plugin framework.

7.1 Extensibility

This section begins by providing scenario descriptions on how to implement concrete extensions with the introduced XVSM implementation. It concludes by comparing its architecture with other XVSM implementations.

7.1.1 Persistence

The plugin architecture in XVSM.NET provides a flexible, extensible storage mechanism, which allows the integration of different persistency implementations in the storage layer. There are persistency implementations that were already loosely coupled in MozartSpaces and should be easily integrable into XVSM.NET. For instance, there is an implementation that uses a separate MozartSpaces CAPI-3 layer to execute space operations on a database [Bar10]. Since the CAPI-3 layer is a simple plugin in XVSM.NET, a custom CAPI-3 database plugin is an equivalent solution to the MozartSpaces implementation, which is based on a separate CAPI-3 module. Then there is an approach that uses XVSM aspects [Mei11] for persistency. As XVSM aspects are a higher-level extension mechanism defined in XVSM, an equivalent implementation of the extension for the .NET platform could support this persistency extension, when XVSM's aspect specification will be implemented. However, the newest persistency extension for MozartSpaces [Zar12] is more difficult to integrate than replacing several

XVSM layer implementations or using XVSM aspects, as a more integrated approach was used due to efficiency and effectiveness. In the following, we analyze the concrete integration points of the persistency extension and provide suggestions for an XVSM.NET implementation.

In MozartSpaces the runtime was adjusted to load the persistence configuration files and initialize the persistence context. In XVSM.NET the persistence context may be realized through a singleton plugin. It could make use of the plugin configuration system and initialize the persistence context in its constructor. MozartSpaces CAPI interface method `createContainer` was extended to provide an additional parameter to use an *in-memory* persistence implementation. Of course, it is also possible in XVSM.NET to adjust the interface, however, it is recommended to use the request context dictionary to provide such a parameter, which may then be accessed by the persistence context plugin. If syntactic sugar for the parameter is desired, it is still possible to create a wrapper around the CAPI interface that adds the parameter to the request context dictionary.

The container implementation of MozartSpaces made use of a `ConcurrentHashMap` to store entries in-memory. Consequently, it had to be adjusted to use a persistent storage instead. The `DefaultContainerManager` of MozartSpaces was adjusted as well. It used to keep the state of all containers in-memory and, therefore, was updated to persist container metadata, such as the container name, ID, and maximum supported entry count. Coordinators are not fully persisted, but are recreated for restoration. The states of optional or obligatory coordinators are recreated by custom persistence lifecycle hooks in the coordinators. The FifoCoordinator (implemented in `DefaultQueueCoordinator`), for instance, restores its `StoredMap` in the `postRestoreContent` hook to recreate the queue data structure with the entries in the correct FIFO order. Hence, coordinators included in MozartSpaces had to be adjusted for the persistence integration as well. To prevent reloading of all entries into memory, they use a lazy entry implementation to only load the entry when needed.

When the characteristics in XVSM.NET's architecture are taken into account, a different approach may be used for persistency extension with comparable properties. XVSM.NET might provide a custom `IContainerRegistry` plugin that stores the container metadata similar to MozartSpaces. For restoration, it could retrieve the container metadata from the persistent storage and recreate the container. The `IEntryStorage` plugin could be exchanged with a custom implementation that uses the persistent context to store entries. The concrete implementation of the `ICoreEntry` interface can be replaced with a lazy implementation that retrieves the requested data from the persistence context. Since the `ICoreEntry` distinguishes between coordination data and the entry payload, it is even possible to prefetch the coordination data, which is used by most coordinators, and load the payload only when requested. This approach has the advantage that no changes to the coordinators are required for persistency support. Nevertheless, it is still possible to provide the persistence context plugin to the constructor of the coordinator via its part factory to enable custom persistence restoration per coordinator. As the coordinator part factory is a plugin, it can inject the persistence context plugin and pass it to the coordinator constructor.

The persistence extension in MozartSpaces provides two different transaction implementations: *buffered* and *mapped transactions*. The concept of the mapped transactions is to use database transactions for the pessimistic transaction system required by the space. This has the advantage that it is simple to integrate and moves transactional logic from the space implementation to the database. However, mapped transactions may only be used when the used database supports transactions with compatible semantics, which are long-running transactions with read uncommitted isolation level [Zar12]. The alternative, buffered transactions, do not have such requirements for the database. They defer the database interaction until the space transaction is committing by keeping a log of database operations that should be executed then. If the transaction is rollbacked instead, the log is simply discarded. Both transaction implementations should be realizable in XVSM.NET. A new database transaction must be started whenever a new user transaction is created. A plugin aspect that is invoked after the creation of the transaction plugin part by the transaction factory could start the corresponding database transaction and register transactional actions that mirror commit and rollback for the database transaction. It must be possible to retrieve the database transaction from an operation that runs under a space transaction. Consequently, the database transaction may be registered in the persistence context under the transaction reference. Thus, operations of the entry storage may then lookup the database transaction in order to perform the persistence operation under the said transaction. Figure 7.1 shows all recommended changes for a persistency extension with mapped transactions. When we take a look at a possible implementation for the buffered transactions, it requires similar integration points. The plugin aspect may be used to register the final commit transactional action to start a database transaction and perform the stored operations under it. The operations may be stored either by a custom `IEntryStorage` plugin or by a plugin aspect for the `IEntryStorage` plugin.

As can be seen, XVSM.NET's architecture allows an easily pluggable persistence extension for the storage layer. Since typical coordinators do not need to be adjusted for persistence support, the development of custom coordinators is simplified. In comparison with MozartSpaces the default code base of XVSM.NET does not require hardcoded dependencies to the persistency implementation. Other modular persistency implementations for other XVSM middlewares such as XCOSpaces or TinySpaces that are based on XVSM aspects have performance and integration drawbacks according to [Zar12].

7.1.2 Nested Transactions in Aspects

XVSM aspects are a dynamic and easy to use extension mechanism. The current implementations in XVSM frameworks are well-suited for use cases where no tight transaction integration is required, such as a notification extension. However, with the existing mechanism it is not possible to use a third transaction layer that provides access to already acquired resources from the sub-transaction. This would allow partial rollbacks and therefore increase the functionality of XVSM aspects.

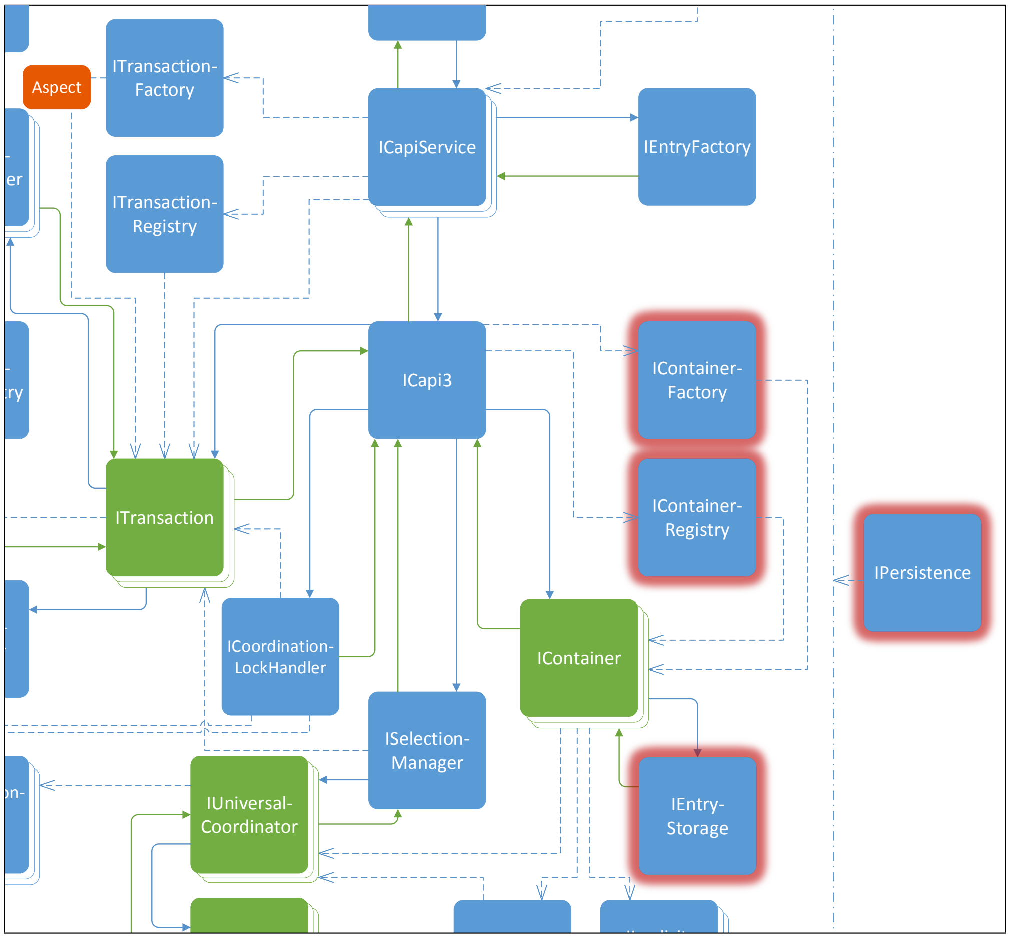


Figure 7.1: Extract of a possible XVSM.NET persistence extension with highlighted changes.

Since XVSM.NET supports arbitrarily nested transactions, partial rollbacks are possible in a future aspect implementation without further implementation. An additional advantage of this mechanism is the improved isolation of aspects. Since they can use their own transaction, their implementation may be simplified.

7.1.3 Custom Coordinators

The ability to support multiple coordinators is one of the core concepts of XVSM. The variety of coordinators allows developers to choose the best fitting coordinator compilation for the used domain. However, sometimes it might be more efficient to develop a custom coordinator for a specific problem. Thus, XVSM describes custom coordinators in its formal model and XVSM frameworks such as MozartSpaces, XCOSpaces, or TinySpaces provide support for custom coordinators. One of the major goals of XVSM.NET design was to simplify the creation of custom coordinators but still support the streaming-based coordination approach that was introduced in [Bar10]. In Table 7.1, we are comparing the XVSM implementations MozartSpaces (MS), XCOSpaces (XCO), TinySpaces, and XVSM.net regarding the coordinator responsibilities. The table depicts coordinators with a responsibility by using a checkmark (✓) or a cross (✗) if the task is handled by a different part of the framework. "N/A" is used to visualize that the responsibility is not handled at all in the framework.

	MS	**XCO**	**TinySpaces**	**XVSM.net**
Registering Entries	✓	✓	✓	✓
Unregistering Entries	✓	✓	✓	✓
Selecting Entries	✓	✓	✓	✓
Counting Entries	✓	✓	✓	✗
Transactional Availability Check	✓	N/A	N/A	✗
Handle First Coordinator	✓	✓	✗	✗
Acquire Coordinator Locks	✓	N/A	N/A	✗
Transactional Log Registration	✗	✓	✓	✗

Table 7.1: Coordinator responsibilities in XVSM middlewares.

Coordinators in all four implementations must handle basic coordination tasks to register, unregister, and select entries. All implementations except XVSM.net require a coordinator to only return the requested amount of entries. In XVSM.net the `CoordinatorMediator` iterates over the streamed entry references returned from the corresponding coordinator and ensures that the required amount of entries are passed to the next coordinator or layer. The TinySpaces and XCOSpaces implementations check the entry count without taking transactional visibility into account. For instance, when an entry is registered in a coordinator with a running user transaction, it still might get returned by a selection through another transaction. If the selection uses a take operation, the entries cannot be acquired with the default XVSM isolation level and lead to a rescheduling of the operation, even though another matching entry could still satisfy the selector count. MozartSpaces avoids this issue by manually checking the transactional entry availability in the coordinators. In TinySpaces and XVSM.net the first coordinator is not required to have a special handling, because the entries from the container are simply passed to the first coordinator, in contrast to the implementation in XCOSpaces and MozartSpaces. In MozartSpaces the transactional availability check is only performed for the first coordinator as a performance optimization. Hence, every existing coordinator in MozartSpaces had a special handling when used as the first coordinator. For the Vector coordinator, coordination locks are acquired in MozartSpaces and XVSM.net, but only MozartSpaces requires the coordinator to manually acquire the lock. In XVSM.net this is realized through the `ICoordinationLockHandler` plugin. Coordinator locks are not available in TinySpaces as it has no Vector coordinator implementation. In XCOSpaces no coordinator locks are acquired for Vector coordinators, hence violating access to a Vector coordinator's entries leads to rescheduling of the operation. The registration of transactional logs for entries by acquiring entry locks is automatically executed for coordinators in MozartSpaces and XVSM.NET. Hence, their coordinators are not required to provide special handling after a transactional rollback or commit.

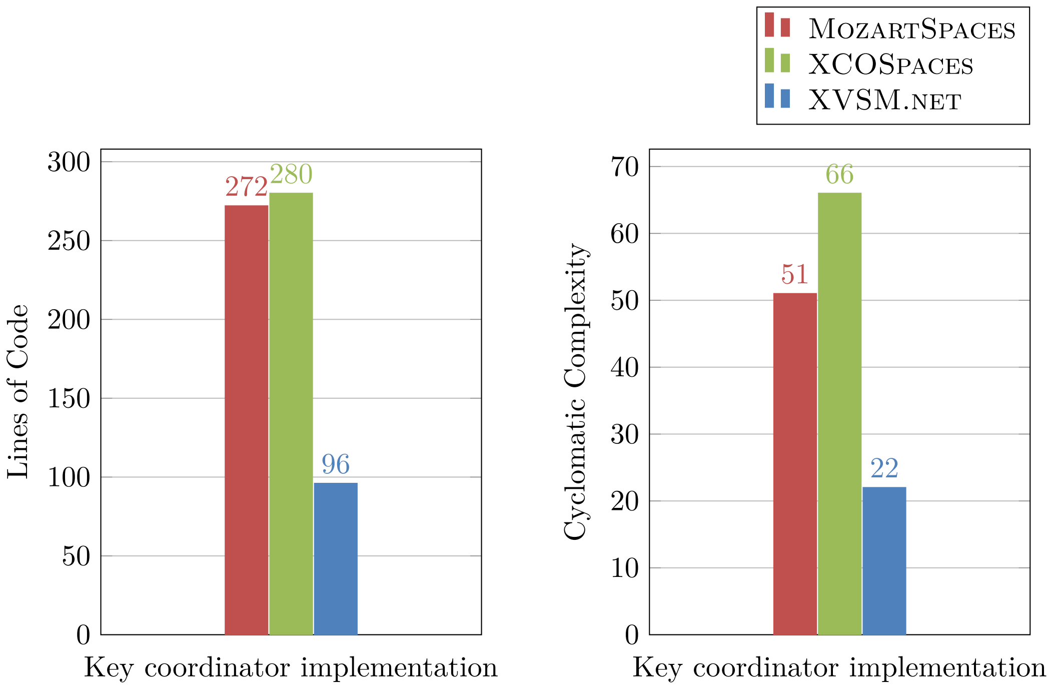


Figure 7.2: Key coordinator implementation measurements.

Figures 7.2 and 7.3 compare Key coordinator and Vector coordinator implementations regarding their complexity in MozartSpaces, XCOSpaces, and XVSM.NET. The open source code quality and static analysis platform SonarQube 6.7 [26] was used for the measurement of the code. TinySpaces is missing from this comparison since its code base would have to be updated to be analyzed by SonarQube. The charts of both figures show the lines of code (LOC) and the *Cyclomatic Complexity* after McCabe [McC76]. Both are popular metrics used for object-oriented programming languages according to [Nuñ+17]. Cyclomatic Complexity counts the number of linear independent paths in source code. The usage of many and complicated control structures increases the Cyclomatic Complexity.

Both measurements of the Key coordinator and Vector coordinator substantiates the previous responsibility comparison of coordinators in different XVSM middlewares. It shows the benefits of the new architecture in regard to simplification of coordinators.

7.1.4 Security

MozartSpaces supports a dynamic, data-driven access control mechanism [CK12; Cra+12] called *XVSM Access Control Model*. It uses a policy language that resembles

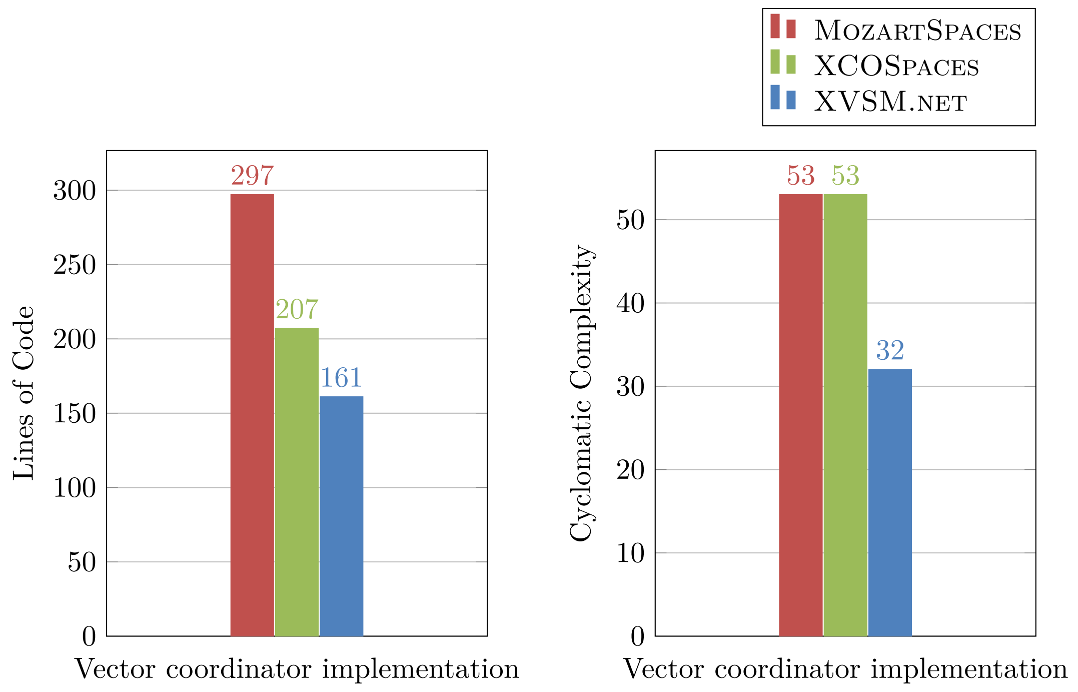


Figure 7.3: Vector coordinator implementation measurements.

XACML [Mos+05] and that makes use of the flexible coordination system provided by XVSM. The policy rules are stored in a container and may be accessed through a custom coordinator. When the space is started, the policy container is created by MozartSpaces' container manager. The container implementation conducts the authorization check for all entries before the entry selection operation is invoked by the coordinators. The authorization result is passed to the coordinator chain, where the first coordinator is responsible to check the authorization result. When the selected entries are retrieved from the coordinators, the authorization result is checked again by the container after handling the entry lock acquisition. When entries are written to the container, their authorization is checked by the container implementation, right before storing them.

A possible implementation in XVSM.net could automatically create the policy container on its first usage. This is possible since the plugin architecture allows the injection of the `IContainerFactory` plugin. When entries are written to the container, a custom plugin aspect could execute the authorization check and throw an exception to resemble the behavior of the MozartSpaces implementation. The check in the entry retrieval may be realized through a custom `ISelectionManager` plugin that uses an authorization-aware `CoordinatorMediator`. Of course, the authorization result may not be passed manually, but could be evaluated by a CAPI-3 plugin aspect that adds the result to the request context object. This allows the modified `CoordinatorMediator` to access it without passing it manually through the call-chain.

7.1.5 Comparison

Due to the pluggable architecture with recomposition support of XVSM plugins, plugin parts, and plugin aspects, all layers can be classified with the configurable adaption type S3 (see Section 2.1.3) and potentially, even the dynamic adaption type D2 if recomposition handling is implemented for the layers. The easiest way to change an implementation in XVSM.net is by providing a custom assembly that replaces a predefined one. Since this is possible without recompilation of the whole middleware, all modules would be classified with S3, naturally. However, XVSM.net even provides recomposition support through its underlying extension framework MEF. Even though recomposition may require manual recomposition handling, it is possible through the plugin framework.

The recomposition mechanism not only allows to change plugin or plugin part implementations at runtime. It makes it also possible to add plugin aspects. This allows us to modify or mutate the existing code to an unknown degree. Thus, we may classify the modules with a mutable adaption type. Even though the platform of XVSM.net supports recomposition, it still needs to be enabled and manually orchestrated. Hence, we classify each property with S3 that is possible to be increased up to D2. When the runtime is fully implemented with XVSM aspect support, the *interception* property will be improved to D2.

Table 7.2 extends the comparison from Section 2.6 with XVSM.net. MozartSpaces and XCOSpaces are abbreviated as MS and XCO. Thanks to the plugin framework, all plugins of XVSM.net have an S3 adaption type without additional adjustments. The table shows the main layers, where XVSM.net stands out against the other frameworks because of its consistent classification as S3 with an option to D2. Thus, changing the behavior of XVSM.net can be realized by configuring other used plugin throughout its whole implementation. This properties reflect its flexibility regarding extensions as was shown in this chapter.

	XAP	**River**	**MS**	**XCO**	**TinySpaces**	**XVSM.net**
Storage	S3	D1	S3	S1	S2	S3 & (D2)
Transaction	S3	D1	S1	S1	S2	S3 & (D2)
Coordination	S1	S1	S3	S2	S2	S3 & (D2)
Runtime	S1	S1	S2	S3	S2	S3 & (D2)
Interception	S3	N/A	D2	D2	D2	S3 & (D2)

Table 7.2: Extended comparison of space-based middleware regarding extensibility properties.

7.2 Performance & Memory Measurments

In this section we take a look at performance and memory characteristics of XVSM.net in comparison with MozartSpaces. The CAPI-3 layer is used for the comparison, as an early evaluation showed that a lot of performance overhead is produced by the entry serialization operation in the PeerSpace.NET implementation that is used in CAPI-4 of XVSM.net. Since this thesis is focused on CAPI-3 anyway, we restrict the comparison on this layer as well.

To avoid benchmarking pitfalls, we use the benchmarking tools JMH [27] for MozartSpaces, and BenchmarkDotNet [28] for XVSM.net to execute the performance measurements. The tools simplify the development of benchmarks by taking care of warmups, multiple iterations, and calculating the result. The performance benchmarks were conducted on a desktop computer running Windows 10 Pro (1709) in 64-bit on an Intel Core i5-2500 CPU with 16GB of DDR3 RAM. The tests were run for multiple times to ensure that they can be reproduced and no unexpected system activity influenced the results. The Java Runtime Environment 1.8.0 was used for the MozartSpaces tests and its pendant for the .NET platform, the Common Language Runtime (4.0), was used for XVSM.net.

All performance tests had five warmup iterations and five additional iterations that were used for the benchmark results.

7.2.1 Write Performance

This test compares the performance of entry write operations in different transactional scenarios. Thus, we distinguish between the following three scenarios:

1. Use a separate user transaction per write operation (denoted as *OwnTx*).
2. Use a separate sub transaction per write operation (denoted as *SameTx*).
3. Use the same sub transaction to write all entries (denoted as *OneTx*).

These three scenarios were chosen to distinguish the influence that the used transactional scenarios have for both frameworks. In addition to the MozartSpaces and the regular XVSM.net implementation, an additional variant of XVSM.net with the NoTransaction implementation from Section 6.2.9 is compared. This is used to show how a minimal overhead of the transaction system affects the performance.

In all scenarios 10,000 entries are written to an unbounded container with a single Fifo coordinator by using the default isolation level repeatable read. The space and the container are created in a setup method before every measurement iteration.

Figure 7.4 shows the results of the benchmarks. It can be clearly seen that MozartSpaces easily out-performs XVSM.net. Performance profiling has shown for XVSM.net that big parts of its computation time is spent for garbage collecting and internal lock handling, especially when the default pessimistic transactional locking plugin is used. The gap between the *OwnTx* and *SameTx* scenarios with the *OneTx* scenario indicates the overhead of creating 10,000 transactions. Since, the NoTransaction plugin also

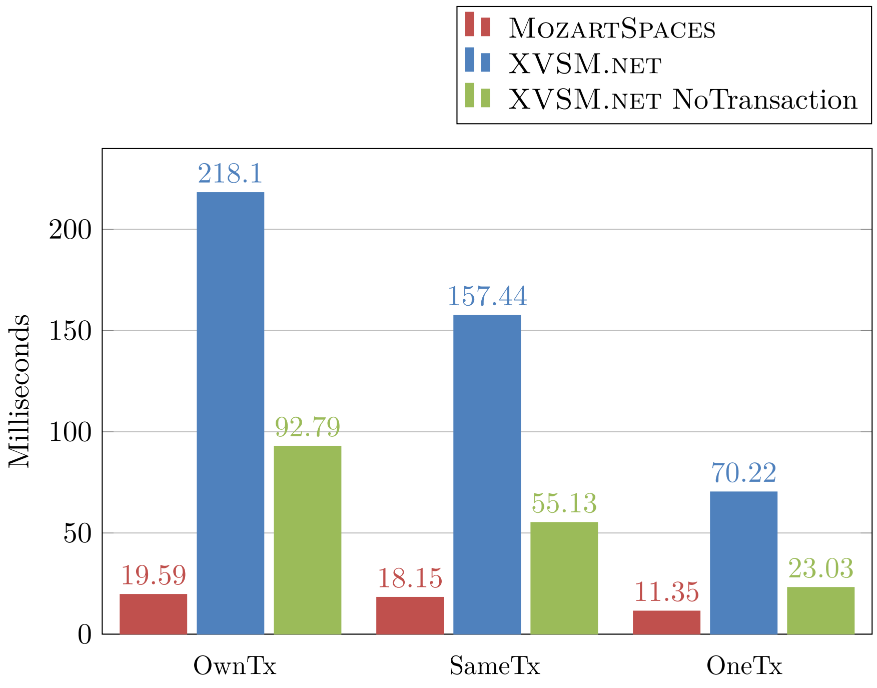


Figure 7.4: Performance of Write operations on a Fifo coordinator.

instantiates lightweight transactions, even though without locking and isolation support, its performance is also increased by writing the entries in a batch within the same transaction.

In addition to the previously shown results, a concurrent version of the benchmark for the *OwnTx* scenario was created and its results are depicted in Figure 7.5. For these performance measurements, a container with a Label coordinator is created and 10,000 entries are written to it. The measurements were taken with a different number of parallel jobs that would evenly divide the number of entries to be written among them. For instance, the single running job of the first data point is required to write all 10,000 entries by it self, but for the second data point the two jobs individually need to write 5000 entries. In Java, the jobs are executed on the common `ForkJoinPool` with a `Completable Future` and on C# through `Parallel Tasks`. In both cases the higher-level abstraction was used to allow the platform to optimize the execution.

As can be seen in the figure, MozartSpaces still outperforms XVSM.NET in terms of raw performance, however, XVSM.NET benefits more from concurrently executed jobs. MozartSpaces also gains performance improvements if two jobs are executed in parallel, but executing three or four job decreases the performance again. In contrast, XVSM.NET increasingly benefits with a higher number of parallel running jobs, at least up to four running jobs. With the increasing number of parallel running jobs, the relative benefit decreases which leads to the conclusion that not all parts of XVSM.NET involved in the write operation are scaling for concurrent execution.

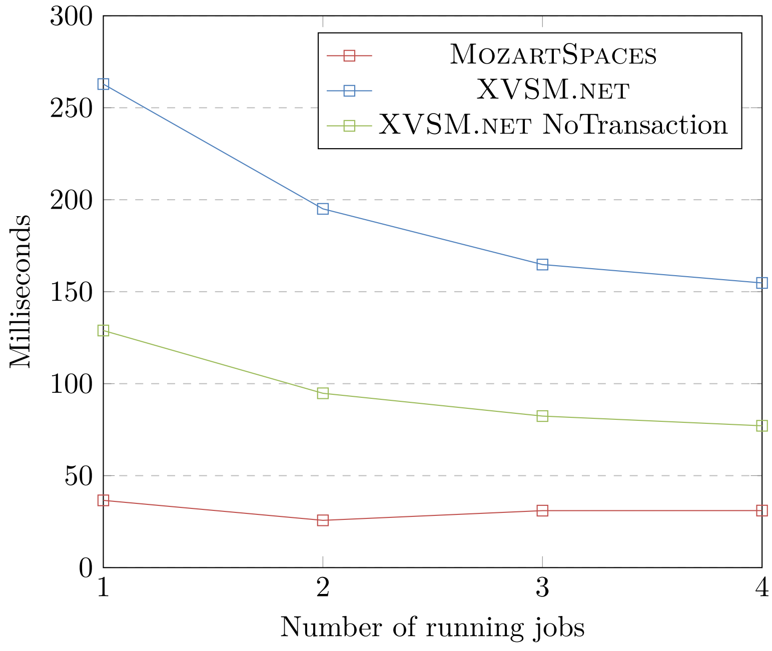


Figure 7.5: Performance of concurrent Write operations on a Label coordinator.

7.2.2 Read Performance

The performance of read operations in XVSM.NET was measured in a similar fashion to the write performance from the previous section. Analogue to the three scenarios from the write performance, these three scenarios were derived for the measurements:

1. Use a separate user transaction per read operation (denoted as *OwnTx*).
2. Use a separate sub transaction per read operation (denoted as *SameTx*).
3. Use the same sub transaction to read all entries (denoted as *OneTx*).

To measure the read operation performance, an unbounded container with a Fifo coordinator is created. After the creation of the container, 10,000 entries are written to it. This procedure is performed in a setup method for every measurement iteration. In every iteration read operations are executed with the default isolation level repeatable read.

The results of the read performance measurements are depicted in Figure 7.6. Similar to the write performance, MozartSpaces outperforms XVSM.NET with the default transaction plugin for *OwnTx* and *OneTx* scenarios. Surprisingly, MozartSpaces' performance suffered significantly for the *SameTx* scenario. The performance measurement

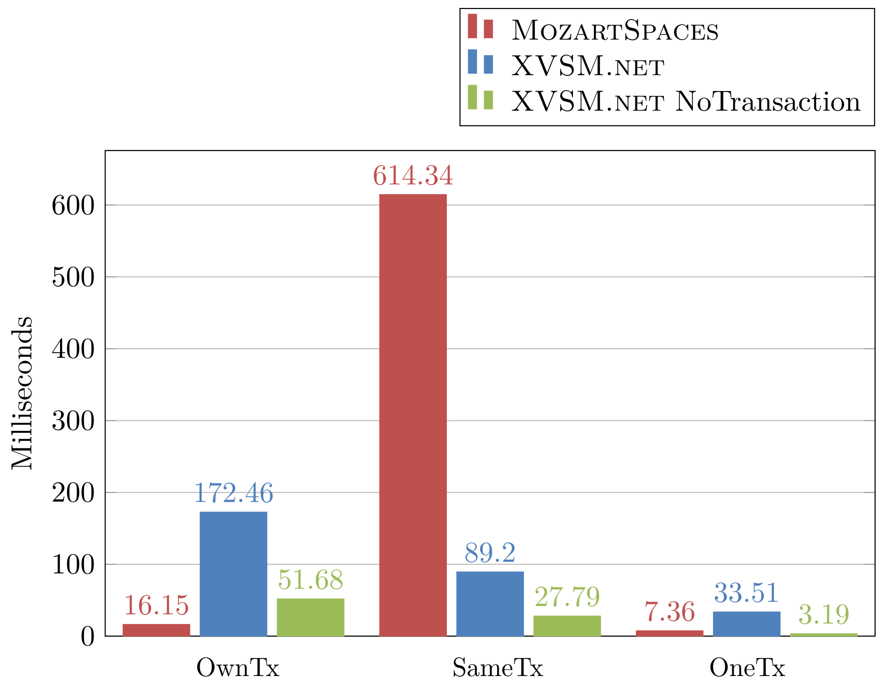


Figure 7.6: Performance of Read operations on a Fifo coordinator.

was repeated multiple times to rule out a possible measurement error, but the result was reproducible. Hence, a measurement error is not expected. JMH denotes a standard deviation for this measurement of 104.20 ms. It appears that the performance degradation is connected to the number of locks on the same entry, which was confirmed by further tests with a Label coordinator.

The concurrent read performance for the *OwnTx* scenario was measured too, as visualized in Figure 7.7. In this case, MozartSpaces produces its best performance when two jobs are running concurrently. For three or four concurrently running jobs, its performance degrades again. Both XVSM.NET variants, on the other hand, improve their performance gradually with more concurrently running jobs for the observed measurements. However, the relative performance improvement steadily degrades with more concurrently running jobs.

7.2.3 Memory Usage

To measure and compare the memory usage of XVSM.NET, we used dotMemory [29] from the Resharper utility, which was available through an academic license. For the Java pendant to measure the memory usage of MozartSpaces, VisualVM [30] from the Java SDK was used. Both tools were used to analyze the heap memory usage of the space when executing several write operations on a single unbounded container with a Fifo coordinator and the default isolation level repeatable read. Per individual write

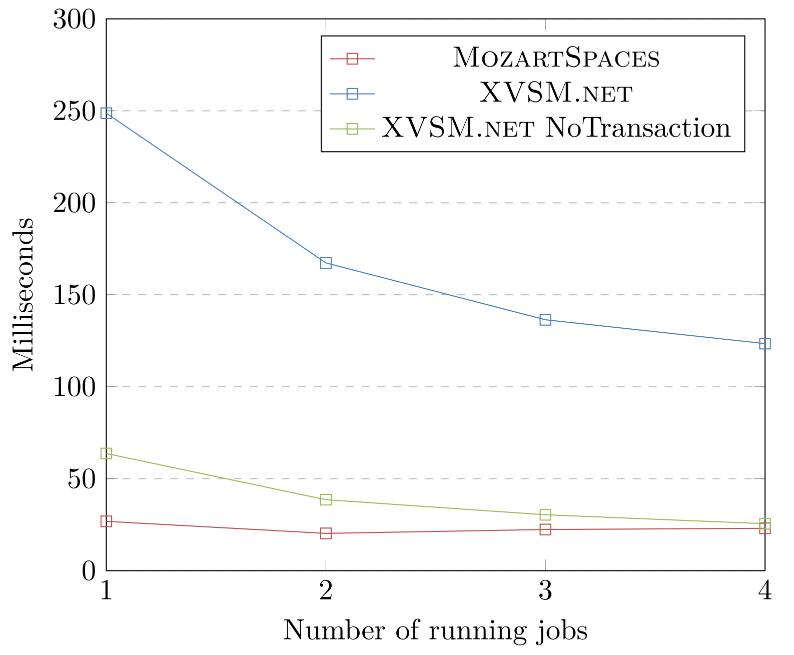


Figure 7.7: Performance of concurrent Read operations on a Label coordinator.

operation, a single user and sub transaction were created. To minimize the impact of Java's garbage collector the initial heap size was configured to 512MB and the maximum heap size to 2GB[1]

Figure 7.8 shows the resulting memory usage of XVSM.net and MozartSpaces for different write operations. The figure depicts the high memory usage of XVSM.net, which is a consequence of the plugin framework and its used libraries. However, the overall memory consumption is in an acceptable range, especially for spaces with a higher number of stored entries. As depicted in the figure, after executing 50,000 write operations, the difference between the memory consumption of XVSM.net and MozartSpaces shrinks to a smaller fraction of the overall memory consumption.

Overall, the memory usage of XVSM.net lies in a comparable range with MozartSpaces. The higher base usage was expected due to the extensive usage of plugins within the plugin framework.

7.2.4 Conclusion

By comparing the provided benchmarks, it is clear that XVSM.net is not at the same performance level as MozartSpaces at the moment. Even with the NoTransaction plugin, its performance lies behind MozartSpaces in most scenarios. However, its

[1]The memory test application was started with the arguments `-Xmx2G -Xms512M`.

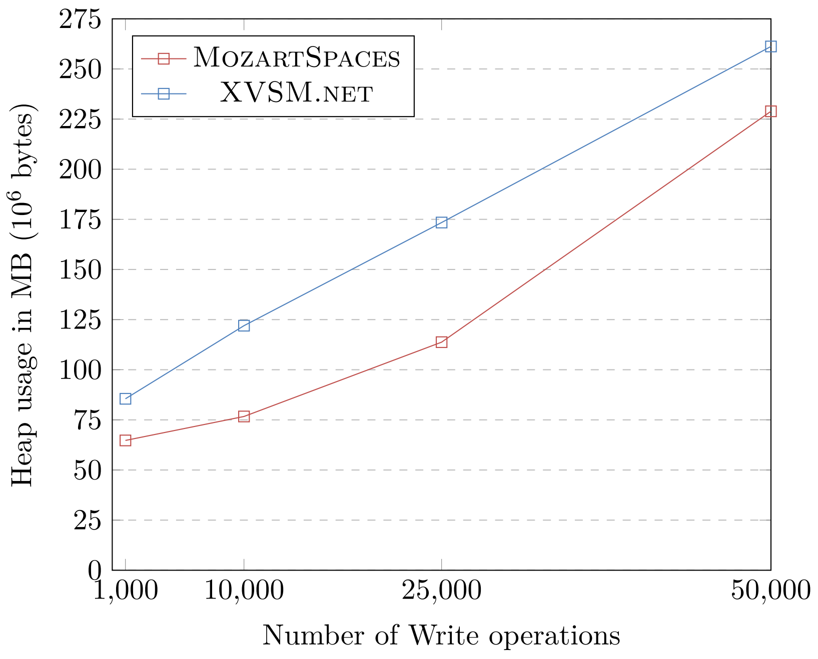


Figure 7.8: Memory usage of Write operations.

scalability characteristics show an advantage for multi-threaded environments with more than two threads.

Architectural implementation specifics, such as the use of proxy objects, are expected to have an impact on the overall performance of the framework. However, there are multiple optimization possibilities that could be incorporated in the future, such as using byte-code manipulation instead of relying on proxy objects.

7.3 Requirements

The requirements of XVSM.NET were defined in Chapter 4. In this section we will compare the defined requirements with the implemented solution.

7.3.1 Functional Requirements

The implementation of the lower CAPI layers 1 to 2 were mainly realized through the `IContainer`, `IEntryStorage`, and `ITransaction` plugins and plugin parts. As defined, the CAPI layers were not implemented as explicit layers but they provide the required functionality for the CAPI-3 layer implementation. The `ICapi3` plugin provides XVSM's required CAPI-3 functionality as an explicit layer. As can be seen from its contract, XVSM's requirements were satisfied, as it is possible to execute the required entry operations read, write, take, delete, and test, as well as the required container operations create, lookup, destroy, and lock.

When containers are created, it is possible to specify optional or obligatory coordinators, as required. A container plugin with support for bounded containers is provided in the solution as well.

The transaction implementation of the default `ITransaction` plugin part supports pessimistic locking with arbitrarily nested transactions. Hence, the required two level transaction system of XVSM's formal model can be realized. The provided default `IIsolationLevelHandler` support the required repeatable read and the optional read committed and read uncommitted isolation levels defined in the requirements. Due to XVSM.NET's architecture, multiple isolation levels are supported at a plugin level.

The required Any, Fifo, Lifo, Label, Key, Linda, and Vector coordinator from XVSM's formal model were all implemented. In addition the Random and LeastUsed coordinator were also implemented to show the flexibility of XVSM.NET in combination with the plugin framework. Other coordinators that are missing, such as the Query coordinator, can be added through a future extension.

A runtime implementation based on the Peer Model was implemented and shown in this thesis with a simple rescheduling mechanism. The rescheduling mechanism is not as advanced as the MozartSpaces implementation and would require more optimization as it retries operations more often than needed. The implementation of the runtime does not use the XVSMP protocol and it therefore lacks interoperability with other XVSM-based frameworks at the moment. The requirement of the blocking interface implementation with similar method signatures as the MozartSpaces' implementation was satisfied as can be seen in this thesis.

7.3.2 Non-functional Requirements

As already shown in this chapter, XVSM.NET makes use of a plugin framework with flexible extensibility features that allow to change or extend the behavior with a multitude of mechanisms without the need to recompile the source code.

The requirement to simplify the implementation of custom coordinators was fulfilled. Custom coordinator implementation was simplified in XVSM.NET by reducing the responsibility of the coordinators and focusing their implementations on the coordination logic, as was shown in this chapter.

As shown in Section 6.4, XVSM.NET provides a mechanism to handle failed operations and notifies gracefully the failed operations to the operation's issuing client.

As seen in this chapter, XVSM.NET supports concurrent execution of multiple operations on a single container as required. Even though the overall performance of XVSM.NET is not at the same level as MozartSpaces at the moment, it shows better support for multi-threaded scenarios, thus, fulfilling the scalability requirements.

All plugin and plugin part contracts are documented through C#'s XML documentation comments, which is an equivalent to the comment documentation Javadoc used in MozartSpaces. It allows to generate an API documentation from the source code. The contracts were designed to only need the minimal required parameters. For instance, the request context could be removed from many method signatures as a parameter thanks to the plugin framework, which allowed to inject the request context instead. C#'s features, such as default arguments, custom indexer, or the using statement through the `IDisposable` interface were used to further simplify the usage of the plugins and plugin parts.

Due to the many plugin and plugin part contracts used in XVSM.NET and its dependency injection support, creating tests for plugins is well supported throughout the framework. Through dependency injection, it is possible to provide mocks of other plugins or plugin parts by simply providing the instances in the constructor or setter methods of the plugin that should be tested. The usage of contracts helps in creating black-box tests against the plugin's or plugin part's specification.

7.4 Suitability of the Plugin Framework

Even though a lot of work was put in the development of the plugin framework throughout this thesis, it proved to be helpful in fulfilling the stated requirements and greatly supported the development of XVSM.NET. The plugin aspect feature and its incentive for high modularization increased the flexibility of XVSM.NET in terms of extensibility as shown in this chapter. The aspired adaption type S3 could be easily realized by using the plugin framework.

As the plugin framework is not depending on XVSM.NET it can be used for other applications and middlewares too. Applications that want to provide a highly modular, extensible architecture may profit from the usage of the plugin framework in a similar way as XVSM.NET did.

CHAPTER 8

Future Work & Conclusion

This chapter gives a summary over what has been accomplished with this thesis and provides suggestions for future research and development options.

8.1 Future Work

XVSM.NET marks a new foundation for XVSM on the .NET platform, but it is still at an early development stage. Further work for the runtime needs to be done. The runtime needs to be extended and updated to fully support the scheduling mechanisms defined in the formal model. It should be coordinated with the MozartSpaces development to provide an interoperable implementation that is not limited by a non-interoperable entry payload serialization mechanism. XVSM aspects are still missing from XVSM.NET and should be added with a future runtime plugin, although XVSM.NET is well-prepared for XVSM aspect implementation through its nested transactions and plugin aspect support. For an interoperable runtime implementation, it should be evaluated whether interoperable aspects might be a feasible option. Since, both the .NET and the Java platforms support JavaScript interpretation out of the box [Pon14] or via third party libraries, it should be considered as an interoperable alternative for writing XVSM aspects in the platform-native programming language.

Due to the performance impacts of the default transaction implementation, it should be evaluated whether a nested transaction variant of the MozartSpaces implementation could provide a viable alternative. An initial performance profiling has shown that big parts of the processing time is lost due to the complex locking mechanisms. Since MozartSpaces' implementation does not allow concurrent operation execution per container but still provides satisfying performance characteristics, we hope that a non-concurrent, nested transaction implementation provides a similar performance than MozartSpaces' current implementation. An alternative to the used proxy implementation that uses byte-code manipulation to integrate plugin aspects should be evaluated for general performance improvements of the plugin framework. More broadly, plugins for persistence, replication, security, the meta model, and the query coordinator should be developed and integrated with the proposed plugin architecture.

Parallel to the full .NET Framework, Microsoft has developed a more lightweight and modular version (.NET Core [31]) that has cross-platform support. Future work should examine whether the Core Framework supports all required dependencies for XVSM.NET and, if possible, port the source code to the new platform.

Java 9's JIGSAW module system [Jec17] could provide an interesting possibility for MOZARTSPACES to modularize its implementation. It is supported out of the box in Java 9 and provides some mechanisms of OSGI frameworks and MEF. It should be evaluated and compared with OSGI frameworks and other frameworks providing modularization concepts. However, the modular architecture of XVSM.NET should be considered for MOZARTSPACES for a refactoring of the core application.

During the creation of this thesis, a flexible extension to XVSM was proposed [Cra+17]. This extension uses an event-based approach and allows the execution of custom user-defined code after certain event conditions are triggered. In combination with the modular architecture presented in this thesis, this may lead to even simpler extension implementations for certain use cases. This is an interesting development for space-based middlewares and should be further looked into.

8.2 Conclusion

In this thesis we have presented a new XVSM implementation for the .NET Framework with a high focus on modularity and extensibility. We have introduced a plugin framework based on an established technology for the .NET platform as a foundation for the implementation of the space-based middleware presented in this thesis. Since the plugin framework is not depending on the XVSM implementation, it may be used for other applications too that could benefit from a modular and flexible architecture.

The middleware was designed to be highly adaptable and circumvent issues of other XVSM implementations. It has been shown that extensions, such as the persistence extension, do not need to be hardwired in the middleware implementation. The modular structure and easy extensibility was designed to encourage and motivate the development of experimental extensions. It also serves as the basis for separate middleware distributions including different plugins. For instance, a non-transactional high-performance distribution could be offered along the standard distribution.

A significant part of the thesis went into simplifying the coordinator implementations, since the knowledge of middleware internals that other XVSM implementations required increased the complexity to create custom coordinators and led to code duplication. The thesis presented coordinator classification properties and showed how they can be used in combination with a pluggable and modular architecture to extract redundant and complex code segments from the coordinators. We hope to see more custom coordinators in the future, due to the simplification.

The nested transaction implementation with support of concurrent container operations increased the flexibility of future XVSM aspect implementations at the cost of performance. Even though the performance was overall acceptable, it leaves room for optimizations. However, the increased flexibility for aspects due to the possibility of partial rollbacks benefits the extensibility of XVSM.

XVSM.NET satisfies the specified requirements and lays the foundation for future developments for space-based middleware on the .NET platform. Even though it is far from feature parity with MozartSpaces, its focus lies on mechanisms to achieve this goal.

List of Figures

List of Tables

List of Listings

Bibliography

[Arn99] K. Arnold. “The Jini architecture: dynamic services in a flexible network”. In: *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*. June 1999, pp. 157–162.

[AS04] S. Androutsellis-Theotokis and D. Spinellis. “A Survey of Peer-to-peer Content Distribution Technologies”. In: *ACM Comput. Surv.* 36.4 (Dec. 2004), pp. 335–371.

[Bar10] M.-S. Barisits. “Design and implementation of the next generation XVSM framework : operations, coordination and transactions”. MA thesis. TU Wien, 2010.

[BEI09] O. Ben-Kiki, C. Evans, and B. Ingerson. *YAML ain’t markup language (YAML)(tm) version 1.2*. Tech. rep. Sept. 2009.

[Ber96] P. A. Bernstein. “Middleware: A Model for Distributed System Services”. In: *Commun. ACM* 39.2 (Feb. 1996), pp. 86–98.

[BN97] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., 1997.

[Boo86] G. Booch. “Object-oriented Development”. In: *IEEE Trans. Softw. Eng.* 12.1 (Jan. 1986), pp. 211–221.

[Cis17] Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021*. Whitepaper. Feb. 2017.

[CK12] S. Craß and E. Kühn. “A Coordination-based Access Control Model for Space-based Computing”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC ’12. Mar. 2012, pp. 1560–1562.

[CKS09] S. Craß, E. Kühn, and G. Salzer. “Algebraic Foundation of a Data Model for an Extensible Space-based Collaboration Protocol”. In: *Proceedings of the 2009 International Database Engineering & Applications Symposium*. IDEAS ’09. Sept. 2009, pp. 301–306.

[Cra+12] S. Craß, T. Dönz, G. Joskowicz, and E. Kühn. “A Coordination-Driven Authorization Framework for Space Containers”. In: *2012 Seventh International Conference on Availability, Reliability and Security*. Aug. 2012, pp. 133–142.

[Cra+17] S. Craß, E. Kühn, V. Sesum-Cavic, and H. Watzke. "An Open Event-Driven Architecture for Reactive Programming and Lifecycle Management in Space-Based Middleware". In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Aug. 2017, pp. 189–193.

[Cra10] S. Craß. "A formal model of the Extensible Virtual Shared Memory (XVSM) and its implementation in Haskell : design and specification". MA thesis. TU Wien, 2010.

[Dön11] T. Dönz. "Design and implementation of the next generation XVSM framework : runtime, protocol and API". MA thesis. TU Wien, 2011.

[EAS07] W. Emmerich, M. Aoyama, and J. Sventek. "The Impact of Research on Middleware Technology". In: *SIGOPS Oper. Syst. Rev.* 41.1 (Jan. 2007), pp. 89–112.

[EFB01] T. Elrad, R. E. Filman, and A. Bader. "Aspect-oriented Programming: Introduction". In: *Commun. ACM* 44.10 (Oct. 2001), pp. 29–32.

[FAH99] E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces principles, patterns, and practice.* Jini technology series. Addison-Wesley, 1999.

[Gel85] D. Gelernter. "Generative Communication in Linda". In: *ACM Trans. Program. Lang. Syst.* 7.1 (Jan. 1985), pp. 80–112.

[Gra+76] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. "Granularity of locks and degrees of consistency in a shared data base". In: *IFIP Working Conference on Modelling in Data Base Management Systems.* Jan. 1976, pp. 365–394.

[Hel+00] R. Helm, R. E. Johnson, E. Gamma, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Braille Jymico Incorporated, 2000.

[Hev+04] A. R. Hevner, M. T. Salvatore, P. Jinsoo, and S. Ram. "Design science in information systems research". In: *MIS quarterly* 28.1 (Mar. 2004), pp. 75–105.

[Hir12] J. Hirsch. "An adaptive and flexible replication mechanism for MozartSpaces, the XVSM reference implementation". MA thesis. TU Wien, 2012.

[HN05] M. Hicks and S. Nettles. "Dynamic Software Updating". In: *ACM Trans. Program. Lang. Syst.* 27.6 (Nov. 2005), pp. 1049–1096.

[Jec17] A. Jecan. "Project Jigsaw". In: *Java 9 Modularity Revealed: Project Jigsaw and Scalable Java Applications.* Apress, 2017, pp. 17–30.

[Kar09] M. Karolus. "Design and implementation of XcoSpaces, the .Net reference implementation of XVSM : coordination, transactions and communication". MA thesis. TU Wien, 2009.

[Kic+97] G. Kiczales et al. "Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming.* Vol. 1241. Lecture Notes in Computer Science. June 1997, pp. 220–242.

[KJ04] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*. John Wiley & Sons, 2004.

[Knu97] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[KPS12] O. Kiselyov, S. Peyton-Jones, and A. Sabry. “Lazy v. Yield: Incremental, Linear Pretty-Printing”. In: *Programming Languages and Systems*. Dec. 2012, pp. 190–206.

[KRJ05] E. Kühn, J. Riemer, and G. Joskowicz. *XVSM (eXtensible Virtual Shared Memory) Architecture and Application*. Tech. rep. Space-Based Computing Group, Institute of Computer Languages, TU Wien, Nov. 2005.

[Küh+09] E. Kühn, R. Mordinyi, L. Keszthelyi, C. Schreiber, S. Bessler, and S. Tomic. “Aspect-Oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems”. In: *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*. OTM ’09. Nov. 2009, pp. 432–448.

[Küh+13] E. Kühn, S. Craß, G. Joskowicz, A. Marek, and T. Scheller. “Peer-Based Programming Model for Coordination Patterns”. In: *Coordination Models and Languages*. Vol. 7890. Lecture Notes in Computer Science. June 2013, pp. 121–135.

[Mar10] A. O. B. Marek. “Design and implementation of TinySpaces : the .NET micro framework based implementation of XVSM for embedded systems”. MA thesis. TU Wien, 2010.

[MC00] R. Monson-Haefel and D. Chappell. *Java Message Service*. O’Reilly & Associates, Inc., 2000.

[McC76] T. J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320.

[McK+04] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. “Composing adaptive software”. In: *Computer* 37.7 (July 2004), pp. 56–64.

[Mei11] T. Meindl. “XVSM Persistence : developing an orthogonal functional profile for the eXtensible Virtual Shared Memory”. MA thesis. 2011.

[MK11] R. Mordinyi and E. Kühn. “Coordination Mechanisms in Complex Software Systems”. In: *Next Generation Data Technologies for Collective Computational Intelligence*. Vol. 352. Studies in Computational Intelligence. 2011, pp. 3–30.

[MKS10] R. Mordinyi, E. Kühn, and A. Schatten. “Space-Based Architectures As Abstraction Layer for Distributed Business Applications”. In: *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*. CISIS ’10. Feb. 2010, pp. 47–53.

[Mon+97] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. “Architectural styles, design patterns, and objects”. In: *IEEE Software* 14.1 (Jan. 1997), pp. 43–52.

[Mor10] R. Mordinyi. “Managing complex and dynamic software systems with space-based computing”. PhD thesis. TU Wien, 2010.

[Mos+05] T. Moses et al. “Extensible access control markup language (xacml) version 2.0”. In: *Oasis Standard* 200502 (Feb. 2005).

[New07] J. Newmarch. *Foundations of Jini 2 programming.* Apress, 2007.

[Nuñ+17] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo. “Source code metrics: A systematic mapping study”. In: *Journal of Systems and Software* 128.Supplement C (June 2017), pp. 164–197.

[OSG03] OSGI Alliance. *Osgi Service Platform, Release 3.* IOS Press, Inc., 2003.

[Pon14] J. Ponge. “Oracle nashorn: A next-generation javascript engine for the JVM”. In: (Feb. 2014).

[Pra09] D. R. Prasanna. *Dependency Injection.* 1st. Manning Publications Co., 2009.

[Rau14] D. Rauch. “PeerSpace.NET”. MA thesis. TU Wien, 2014.

[Sch08] T. Scheller. “Design and implementation of XcoSpaces, the .Net reference implementation of XVSM : core architecture and aspects”. MA thesis. TU Wien, 2008.

[See12] M. Seemann. *Dependency injection in .NET.* Manning New York, 2012.

[SM03] S. M. Sadjadi and P. K. McKinley. “A survey of adaptive middleware”. In: *Michigan State University Report MSU-CSE-03-35* (2003).

[SP02] W. Schult and A. Polze. “Aspect-oriented programming with C# and .NET”. In: *Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). Proceedings. Fifth IEEE International Symposium on.* Apr. 2002, pp. 241–248.

[Sun02] Sun. *JavaSpaces Service Specification.* Tech. rep. Sun Microsystems, 2002.

[TBS10] M. Trofin, N. Blumhardt, and C. Szyperski. “The .NET Primitives for Open, Dynamic and Reflective Component Frameworks”. In: *Software Composition.* Vol. 6144. Lecture Notes in Computer Science. July 2010, pp. 138–153.

[Wal99] J. Waldo. “The Jini Architecture for Network-centric Computing”. In: *Commun. ACM* 42.7 (July 1999), pp. 76–82.

[Wat15] H. Watzke. “Lifecycle and memory management for extensible virtual shared memory (XVSM)”. MA thesis. TU Wien, 2015.

[Zar12] J. Zarnikov. “Energy-efficient persistence for extensible virtual shared memory on the android operating system”. MA thesis. TU Wien, 2012.

Online Resources

[1] M. Fowler. *Inversion of control containers and the dependency injection pattern.* Jan. 2004. URL: https://martinfowler.com/articles/injection.html (visited on March/17/2018).

[2] OW2 Consoritium. *JORAM: Java Open Reliable Asynchronous Messaging.* URL: http://joram.ow2.org (visited on March/23/2018).

[3] The Apache Software Foundation. *Apache Felix.* URL: http://felix.apache.org (visited on March/14/2018).

[4] DataNucleus. *DataNucleus Access Plattform.* URL: http://www.datanucleus.org (visited on March/23/2018).

[5] Various. *Guice.* URL: https://github.com/google/guice (visited on March/12/2018).

[6] GigaSpaces. *XAP, In-Memory Computing Platform.* URL: https://www.gigaspaces.com/product/xap (visited on March/23/2018).

[7] Various. *Spring Framework.* URL: https://projects.spring.io/spring-framework (visited on March/12/2018).

[8] GigaSpaces. *XAP 12.2 Documentation.* URL: https://docs.gigaspaces.com/xap/12.2/ (visited on March/12/2018).

[9] Various. *Hibernate ORM.* URL: http://hibernate.org/orm/ (visited on March/12/2018).

[10] The Apache Software Foundation. *Apache Cassandra.* URL: http://cassandra.apache.org (visited on March/12/2018).

[11] MongoDB, Inc. *MongoDB.* URL: https://www.mongodb.com (visited on March/12/2018).

[12] PTIOBE Software Quality Company. *TIOBE Index for January 2018.* Accessed: 2018-01-05. Jan. 2018. URL: https://www.tiobe.com/tiobe-index/ (visited on January/05/2018).

[13] JetBrains s.r.o. *Jetbrains Resharper.* URL: https://www.jetbrains.com/resharper/ (visited on March/23/2018).

[14] .NET Foundation and contributors. *NUnit.* URL: `http://nunit.org` (visited on March/23/2018).

[15] A. Egerton and D. Tchepak. *NSubstitute.* URL: `http://nsubstitute.github.io` (visited on March/23/2018).

[16] J. Kowalski, K. Christensen, and J. Verdurmen. *NLog.* URL: `http://nlog-project.org` (visited on March/23/2018).

[17] Xcoordination. *Xcoordination Application Space.* URL: `http://www.xcoordination.com/application_space` (visited on January/01/2018).

[18] Microsoft. *CCR Introduction.* URL: `https://msdn.microsoft.com/en-us/library/bb648752.aspx` (visited on January/01/2018).

[19] Castle Project. *Windsor.* URL: `http://www.castleproject.org/projects/windsor/` (visited on March/23/2018).

[20] Various. *Ninject.* URL: `http://www.ninject.org` (visited on March/23/2018).

[21] Various. *Spring.NET.* URL: `http://springframework.net` (visited on March/12/2018).

[22] Castle Project. *DynamicProxy.* URL: `http://www.castleproject.org/projects/dynamicproxy` (visited on March/23/2018).

[23] NHibernate Community. *NHibernate.* URL: `http://nhibernate.info` (visited on March/14/2018).

[24] A. Aubry. URL: `http://aaubry.net/pages/yamldotnet.html` (visited on March/23/2018).

[25] J. Zander. *Why isn't there an Assembly.Unload method?* May 2004. URL: `https://blogs.msdn.microsoft.com/jasonz/2004/05/31/why-isnt-there-an-assembly-unload-method/` (visited on March/17/2018).

[26] SonarSource S.A, Switzerland. *SonarQube.* URL: `https://sonarqube.org` (visited on March/23/2018).

[27] Various. *JMH.* URL: `http://openjdk.java.net/projects/code-tools/jmh/` (visited on January/01/2018).

[28] .NET Foundation and contributors. *BenchmarkDotNet.* URL: `http://benchmarkdotnet.org` (visited on January/01/2018).

[29] JetBrains s.r.o. *Jetbrains dotMemory.* URL: `https://www.jetbrains.com/dotmemory/` (visited on April/02/2018).

[30] Various. *VisualVM.* URL: `https://visualvm.github.io` (visited on April/02/2018).

[31] Various. *Choosing between .NET Core and .NET Framework for server apps.* Mar. 2018. URL: `https://docs.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server` (visited on March/28/2018).