

MozartSpaces Tutorial

Version 1.9

Written by Michael Wittmann,
updated by Laszlo Keszthelyi (laszlo@complang.tuwien.ac.at)

Examples and Exercises implemented by Laszlo Keszthelyi and Rene Formanek

Contents

1	Introduction	4
1.1	“Hello World!”-“Hello Space”	6
2	MozartSpaces Startup and Configuration	7
2.1	Startup	7
2.2	Configuration	7
2.3	Example Configuration	8
3	The Data Structures	10
3.1	The Container, an introduction	10
3.2	Entry	14
3.3	Coordinating the contained things	15
3.4	Selectors	21
3.4.1	VectorSelector	22
3.4.2	KeySelector	23
3.5	Methods to access data	24
3.5.1	write	24
3.5.2	read	24
3.5.3	take	25
3.5.4	destroy	26
3.5.5	shift	26
3.6	Examples for Implicit Coordinators	26
3.6.1	Example: The Lottery (RandomCoordinator)	26
3.6.2	Exercise: The Ticket Queue (FIFOCoordinator)	28
3.7	Examples for Explicit Coordinators	28
3.7.1	Example: The Student management (KeyCoordinator)	28
3.7.2	Exercise: Formula 1 Race (VectorCoordinator)	29
3.8	Summary:	29
4	Transactions	30
4.1	Exercise (Transactions)	32

4.2 Summary:	33
5 A completely new aspect: Aspects in XVSM	34
5.1 Local Aspects	34
5.2 Global Aspects	37
5.3 Summary:	38
6 Don't miss a thing: Notifications	39
6.1 Exercise (TicketQueue with Notifications)	40
6.2 Summary:	41
References	42
7 Bibliography	42
A Solutions for the exercises	I
A.1 The TicketQueue (see Exercise 3.6.2)	I
A.2 Forumula 1 Race (see Exercise 3.7.2)	III
A.3 Forumula 1 Race extended by Transactions (see Exercise 4.1)	VII
A.4 TicketQueue extended by Notifications (see Exercise 6.1)	XVIII

Chapter 1

Introduction

XVSM is standing for eXtensible Virtual Shared Memory, which is a middleware technology to store data objects in a space that can be shared with other peers. Data objects are written to and read from that common data storage.

The space-based approach has some nice advantages. Many implementations of such an approach offer the possibility to distribute data over multiple computers, which can help to be more fault-tolerant, when one participating computer fails or leaves the space. In this case, the data is not necessarily lost as it may be stored on another computer. Moreover, when multiple computers can access the same data storage, they can cooperatively work on this data. For example, you can implement a system where a space participant that takes one entry of the space and perform some action on it, while another participant waits for the result of this calculation. The two participants can be separated from each other in physical space as well as in time. This means, that each participant can run on its own computer and not on the same one, and they don't need to run at the same time, because the calculation result remains in the data space after writing it. Thus, the space participant that waits for the calculation result can access this entry even after the participant that calculated it is no longer present. It is also possible to let multiple computers work on the same space to cooperatively and concurrently work on the set of data to improve the performance to handle the stored data objects. XVSM in the current version offers the possibility to manage the data entries and the order and number of entries that are retrieved. For example, you can define a Coordinator to retrieve the entries in the opposite order they were written and another Coordinator to retrieve them in random order.

It also offers the possibility to perform actions within one transaction, this is to perform it as one atomic action that either executes successfully and thus the results are written to the space or when an error occurs, the whole transaction is rolled back and all changes are discarded. This reduces the danger that the data objects in the space are inconsistent after an error.

You can also register listeners on actions that are performed either on the data or on the space itself. If an action is performed that you are listening on, the corresponding method to react on this event is called. This can be used to react for example when a data entry is added or removed. Such a technique can be used to enhance the functionality of the system, even at runtime, as these handlers can be registered and unregistered even while the system is running.

As a precondition to understand how to use XVSM and the terminology it is recommended that you first read the Application scenarios document, which gives an overview about various programming techniques in Space-Based computing and in XVSM in particular. The tutorial helps you to start programming using MozartSpaces, which is an open source implementation of XVSM in Java. It is not intended that this tutorial is a complete documentation of the API, it only gives you advice how to program with XVSM. You can find the complete API reference at [1].

In the next sections, an introduction to the Container and the ready-to-use data structures that can be used within the container is given. Then, this document tries to show you how to select specific data in the space. After that, it shows you how to extend the functionality of XVSM by using Aspects. As a usage of Aspects, the usage of notifications is explained, which are used to tell the listening program if new data is available, data has changed or data has been deleted.

Please note that XVSM is prepared for various additional functions, including the distribution of the data over multiple peers, automatic data persistency, and many more. This tutorial uses the most recent Java implementation of XVSM, which is MozartSpaces version 1.0, provided by the Space Based Computing Group [2] at the Technical University of Vienna. In this version, these functions are not yet implemented. Thus the tutorial focuses on the functions that are working in this version, so you can try out all examples and exercises that are given in this tutorial.

Further please note that MozartSpaces only works with Java 5.0 or newer.

1.1 "Hello World!"-"Hello Space"

The "Hello World!" example is nearly mandatory for each new programming language or paradigm. Thus also here, an adaptation of this example is described. The details of it are explained in the subsequent chapters. The purpose of this example is to show the basics of the new paradigm and to have a fast overview of its usage. In the original example, the text "Hello World!" shall be printed on the screen. In the XVSM example, the text "Hello Space!" is used and as a further variation this text shall be written to a shared container from which it is read and printed on the screen:

```

1 public static void main(String[] args)
2 {
3     try
4     {
5         ICapi capi = new Capi();
6         // Create new Container using no transaction, no remote server,
7         // infinite container-size and no Coordinator
8         ContainerRef cref = capi.createContainer(null, null, null, ICapi.INFINITE, null);
9
10        /* Create a new AtomicEntry */
11        Entry entry = new AtomicEntry<String>("Hello Space!");
12        /* Write the entry to the container */
13        capi.write(cref, 0, null, entry);
14
15        // read and print the written value
16        Entry[] readEntries = capi.read(cref, 0, null, null);
17        System.out.println(((AtomicEntry) readEntries[0]).getValue());
18
19        /* Shutdown and clear space */
20        capi.shutdown(null, true);
21    }
22    catch (Exception e)
23    {
24        e.printStackTrace();
25    }
26 }

```

Listing 1.1: HelloWorld.cpp

The ICapi interface is the connection interface which offers you all methods to operate with MozartSpaces. For example, you use it to create or write to a container, or to access the Entries that are stored in a container. The details about containers and entries are explained in chapter 3. When you have instantiated the Capi (Core Application Programming Interface) class, which is the implementation of the ICapi interface, you can create a container. Then you write an Entry containing the String "Hello Space!" to that container. Afterwards, this Entry is read again from the container and written to System.out.

Chapter 2

MozartSpaces Startup and Configuration

2.1 Startup

A standalone instance of MozartSpaces can be started using `org.xvsm.server.Server` class. After downloading `MozartSpaces-1.0-alpha-all.jar` from the web-page (www.mozartspaces.org), MozartSpaces can be started using

```
1 java -cp MozartSpaces-1.0-alpha-all.jar org.xvsm.server.Server
```

Listing 2.1: Start server

Optionally, the position of the configuration file can be set as parameter:

```
1 java -cp MozartSpaces-1.0-alpha-all.jar org.xvsm.server.Server MozartSpacesConfiguratio.  
   properties
```

Listing 2.2: Start server with a specific configuration file

2.2 Configuration

MozartSpaces can be configured using a properties files. Per default, the file is called `spaces.prop`. MozartSpaces tries to find this file in the working directory during start up. If the file can not be found a new one is created containing default values for all properties. The position and the name of the configuration file can be modified as argument to the `Server` class.

Below some of the most important properties are described:

```
1 EnableRemoteAccess=true
```

Indicates whether the space shall be accessible via the network. This can be set to false if an embedded space shall only be accessible from the applications in the same virtual machine. For stand alone instances this property should always be set to true.

```
1 RemoteProtocols=TcpJava|TcpXML
```

A list of enabled protocols separated by "|". The names listed here will be used by MozartSpaces to find the uri, the listener, the sender and the marshaller for the transport (see below).

TcpJava means that a TCP connection is used to send/receive serialized java objects.

TcpXML means that a TCP connection is used to send/receive XML serialized messages.

```
1 TcpJava.uri=tcjava\://localhost|:0
```

The uri of the TcpJava transport. This uri (including port) should be set to the host name or the IP address of the machine on which MozartSpaces is running. If the port is set to 0 a random port will be used, i.e., for stand alone instances the port should be set.

```
1 TcpJava.listener=org.xvsm.remote.tcpconnection.TcpConnectedTransport
```

The implementation of the Listener used by TcpJava.

```
1 TcpJava.sender=org.xvsm.remote.tcpconnection.TcpConnectedTransport
```

The implementation of the sender used by TcpJava.

```
1 TcpJava.marshaller=org.xvsm.remote.marshaller.JavaMarshaller
```

The marshaller used to serialize the message before sending and deserialize the received messages.

```
1 DefaultAnswerToProtocol=TcpJava
```

The protocol which shall be used to send answers to requests. This property is important for clients which send requests to a remote XVSM instance.

2.3 Example Configuration

The following listing depicts an example configuration for a stand alone MozartSpace instance. The instance support TcpXML and TcpJava transports. The machine on which the instance is running has the IP address 192.168.1.1, TcpXML is listing on port 9876 and

TcpJava on port 4321. Both transport use The TcpConnctedTransport implementation which implements the listener as well as the sender interface.

```
1 EnableRemoteAccess=true
2 RemoteProtocols=TcpJava|TcpXML
3 TcpJava.uri=tcjava\://192.168.1.1|:4321
4 TcpJava.listener=org.xvsm.remote.tcpconnection.TcpConnectedTransport
5 TcpJava.sender=org.xvsm.remote.tcpconnection.TcpConnectedTransport
6 TcpJava.marshaller=org.xvsm.remote.marshaller.JavaMarshaller
7 TcpXML.uri=tcxml\://192.168.1.1|:9876
8 TcpXML.sender=org.xvsm.remote.tcpconnection.TcpConnectedTransport
9 TcpXML.listener=org.xvsm.remote.tcpconnection.TcpConnectedTransport
10 TcpXML.marshaller=org.xvsm.remote.marshaller.XMLMarshaller
11 DefaultAnswerToProtocol=TcpXML
12 SchedulerTimeoutDelay=500
13 ReplySenderThreads=100
14 SchedulerMaxThreads=1
15 EventProcessingThreads=100
```

Chapter 3

The Data Structures

There are several possibilities to read and write data from and to the XVSM space. As a precondition to understand what kind of data you'll be able to write to the space, it is necessary to show some basics about what the space actually is and how you can work with it.

3.1 The Container, an introduction

The Container is the place where the data is stored in. You can visualize the container in such a way:

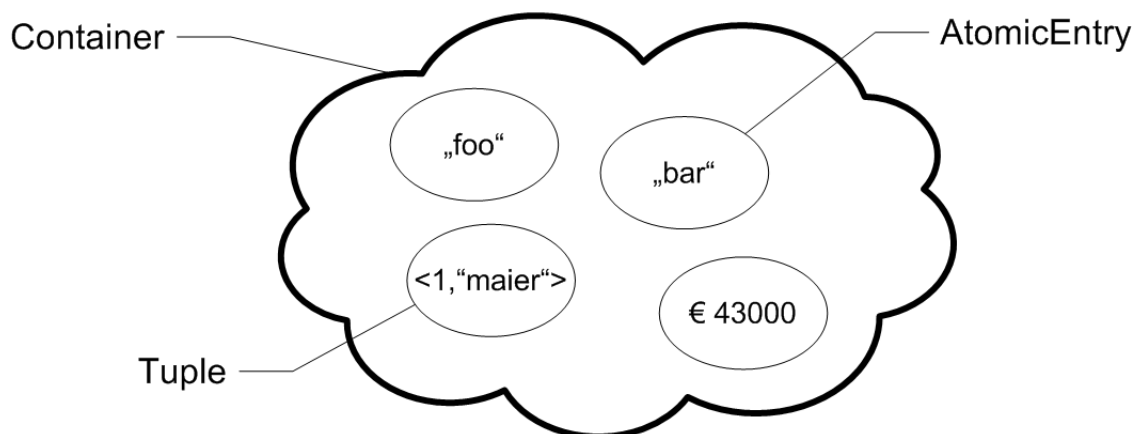


Figure 3.1: Overview of a container with objects

To give you additional information about the Entries, a table representation of the container is sometimes used. The arrows at the sides give the information of the order in which the Entries are written or read, which is especially useful when you use Implicit coordination (to be explained later). Such a table could look like this:

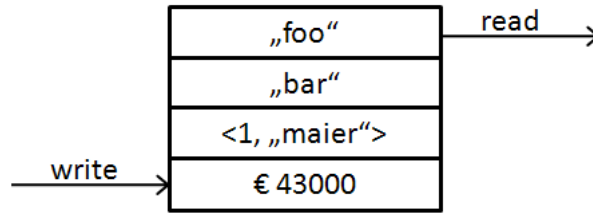


Figure 3.2: Table representation example of a container

The data items that are stored in a container are called “Entries”. An Entry can be either of type `Entry.EntryTypes.TUPLE` or `Entry.EntryTypes.ATOMICENTRY`. A Tuple contains other Entries, which can be either AtomicEntries or other Tuples. An AtomicEntry is a Generic Java class; when instantiating it, you can define the class that is contained within the AtomicEntry:

```
1 org.xvsm.core.Entry entry = new org.xvsm.core.AtomicEntry<String>("Hello World!");
```

Listing 3.1: Instantiation of an AtomicEntry

A Container can either be located on your local machine or on a remote machine:

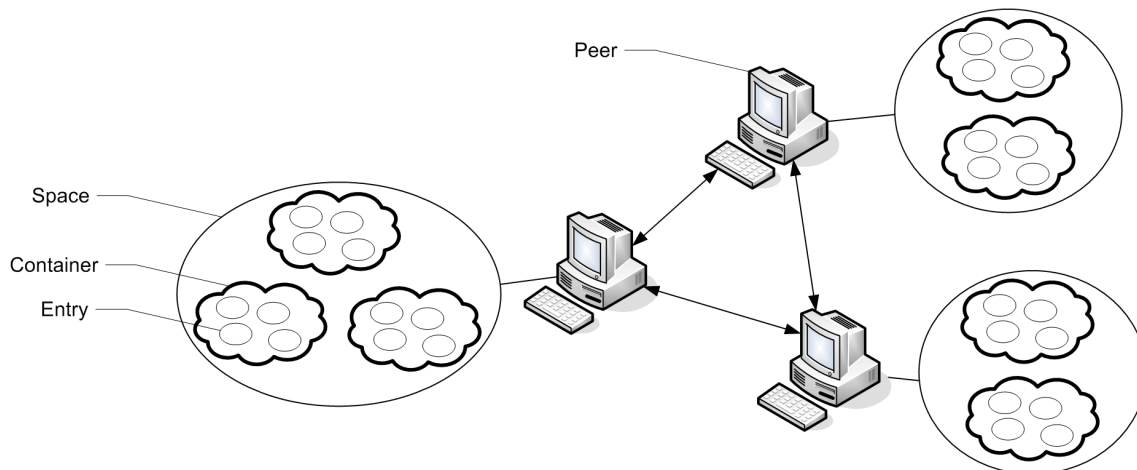


Figure 3.3: Communication of a set of peers, each with its own Space, Containers and Entries

The Capi class is an implementation of the ICapi interface, which is the main management class of the containers and XVSM itself. Using it, you can shutdown and restart the space, create and destroy containers, or read, write and destroy Entries in a container. First of all, you start your local space by calling

```
1 org.xvsm.interface.ICapi capi = new Capi();
```

Listing 3.2: Instantiation of Capi

Every time new Capi() is called on a computer, a local Space is created which is accessible by other peers by default on port 1234. Another possibility to create a Space on a computer is to start the standalone Server, which you can find in the org.xvsm.server package and which has its own main method. It only starts a Space to handle the requests by other peers but doesn't offer an interface to handle the Space on its own, thus only other peers can manage the content of the Space. Keeping this in mind, there are two possibilities to start a new Space: by instantiating Capi or by starting a standalone Server. To create a new container, use this method:

```
1 public ContainerRef createContainer(Transaction tx, URI site, String name, int size,
   ICoordinator... coordinators) throws XCoreException
```

Listing 3.3: Creating a new Container

To create a container, you need to pass the following parameters:

- Transaction: transactions are explained in chapter 4. Use null to perform the action with an implicit transaction (this means that the action is automatically committed)
- URI of the Container's site: this is the URI to access spaces on a remote peer. By default, the port to connect to the remote peer is 1234, but of course the peer could use another port. The URI of a remote Server is constructed the following way: `new URI("tcpjava://mycomputer.mydomain.com:1234")`. The default port can be changed in the space.prop configuration file. Use null to connect to your local machine.
- Container's name. It can be used if you want to obtain a container after restart (only if persistency is already implemented by the XVSM version you use) or from another peer. You can use null to create an unnamed container. But doing this way is only preferable if you don't want to use that container after a restart of the space or if it is not necessary to share the space with other peers in a convenient way, as otherwise it would be complicated to access that unnamed container, as you need to use the automatically generated id as container name, which may not be an easy name to remember.
- Container's maximum size: The maximum number of entries that the container can hold. If you try to write an Entry to a container which already holds the maximum number of Entries, the write operation blocks. There also exists a shift method, which – instead of blocking – deletes “disturbing” Entries before writing the new ones. If you don't want to have such a maximum size, use `IContainer.INFINITE_SIZE` as size parameter.

- List of Coordinators: You can define the internal coordination of the container. The `ICoordinator` interface is extended by the `IImplicitCoordinator` and `IExplicitCoordinator` interfaces. They are implemented by various classes, like the `KeyCoordinator` or `FifoCoordinator`. They are defining the sequence of the retrieved Entries when you call a method to read from the container. The different types of Coordinators will be described in Chapter 3.3. If you set the Coordination to null, the `RandomCoordinator` will be used, which retrieves the entries randomly.

The `createContainer` method returns a `ContainerRef`, which is in turn used to refer to this container. In the later examples, you will see that it makes sense to use multiple `ContainerRefs`. The `ContainerRef` is passed as parameter in most of the methods that are used to access data.

If you know that the container already exists or a `ContainerNameOccupiedException` (which extends `XCoreException`) is thrown on the creation, you can use the following method to get access to an existing (even remote) Container:

```
1 public ContainerRef lookupContainer(Transaction tx, URI site, String containerName)
   throws XCoreException
```

Listing 3.4: Lookup a new Container

The parameters of this method are similar to the ones used in `createContainer`. This way, you can access a container which already exists on another peer.

Both `createContainer` and `lookupContainer`, as well as most other methods in `MozartSpaces`, can throw an `XCoreException`, which is the generalization of most `XVSM`-specific Exceptions and also stands for an Exception that is thrown because of an internal error.

When you'd like to shut down the Capi peer, you can pass the URI where the peer is running. Thus, you can smoothly turn off a remote peer, which makes sense especially for the standalone Server. The `clearSpace` flag is used to clean the space before you shut down the peer. This functionality is useless until persistency is supported, but for future implementation it already exists:

```
1 void shutdown(URI site, boolean clearSpace) throws XCoreException
```

Listing 3.5: Shutting down space

Keep in mind that in the current implementation, no security mechanisms exist that keep a client away from restarting or clearing the space. Security mechanisms to prevent unauthorized users from performing these actions will be implemented in future releases of `MozartSpaces`.

Further note that currently, no persistency is implemented, so if you stop your peer, all information is lost, no matter if you set the `clearSpace` flag when shutting down the peer or not.

3.2 Entry

As already described, an `Entry` can be either a `Tuple` or an `AtomicEntry`. The `AtomicEntry` is a generic class, this means you can define what kind of data is stored in the entry:

```
1 Entry entry = new AtomicEntry<String>("Hello World!");
```

Listing 3.6: Instantiate an entry

In the current version of `MozartSpace`, the class that can be used as type parameter in the `AtomicEntry` must implement the `Serializable` interface. If you use the methods to access data, you'll see that in most cases you'll get an `Entry` returned, which is the superclass of `Tuple` and `AtomicEntry`, so you can't immediately get the value of the `Entry`. But `Entry` has a method `getEntryType()`, which returns one value of the `EntryTypes` enum that indicates what kind of `Entry` is received. Thus, using

```
1 if (readEntry.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
```

Listing 3.7: Verify if an entry is of type `AtomicEntry`

A `Tuple` entry implements the `Iterator` interface (`Iterator<Entry>`), thus it is ready-to-use to iterate over the entries that are contained within the tuple.

One major property of the `Entry` object is that when you instantiate it, you can pass a set of `Selectors` (which will be described in the next chapter):

```
1 protected Entry(Selector... sels)
```

Listing 3.8: Instantiate an `Entry` with selectors

So, for example

```
1 Entry entry = new AtomicEntry<String>(
2     "writeCar",
3     String.class,
4     new VectorSelector(i, 0));
```

Listing 3.9: Example of an `AtomicEntry`

creates a new `AtomicEntry` of type `String` with the value “writeCar” and uses the `VectorSelector` to write it to a `Vector` at the position `i`.

Please take a look at the complete example at the end of this chapter to better understand how the `Entries` are created and used in the container.

3.3 Coordinating the contained things

Coordinators are responsible for the order of the entries in the container that a programmer observes. You can access the `Container` either by using a `Selector` or without a `Selector`. A `Selector` is needed to refer to a specific `Entry` in the `Container`, like the `Entry` which holds the `id=3`. Generally speaking, `Coordinators` that implement the `ImplicitCoordination` interface have a view over all `Entries` in the `Space`, whereas those that implement the `ExplicitCoordination` interface have a view over all or a subset of the `Entries`, about which the `Coordinator` has stored additional info (like a key, an index,...). If you want to use the functionality of a specific `Coordinator` that you’ve specified when creating the container, you must pass the corresponding `Selector` when accessing `Entries` in this container (see also Chapter 3.4). If you leave the `Selector` away when accessing an existing `Entry`, the `RandomCoordinator` will be used. When writing to a container and you’d like to use the functionality of the implicit `Coordinator`, you don’t need to pass it a `Selector`. In the current version, the following possibilities exist to coordinate the order of the retrieved `Entries`:

Implicit (complete view over all `Entries`):

- **FifoCoordinator:** If you use the `FifoCoordinator` (FIFO stands for first-in-first-out), the entries are retrieved in the same order as you wrote them to the container. This means, if you write 5 entries to the container, then you call the `read(cref, timeout, transaction, fifoSel)` method, the entry that you wrote FIRST is returned, thus the name “first-in-first-out”. You can visualize this like a queue of people standing in front of the vendor’s desk.

```
1 FifoSelector sel = new FifoSelector();
2
3 // Create new Container using no transaction, no remote server/peer,
4 // container-size of 5 and a Fifo Coordinator
5 ContainerRef cref = capi.createContainer(null,
6     null,
7     null,
```

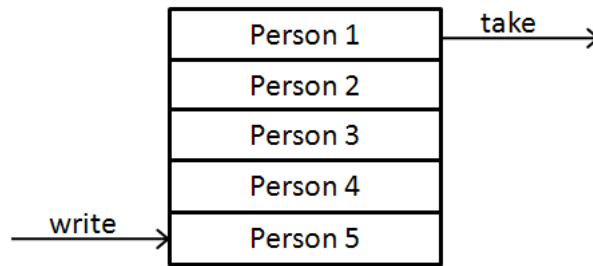


Figure 3.4: FifoCoordinator example

```

8     5,
9     new FifoCoordinator());
10
11  /* Fill the queue with the numbers 1 to 5*/
12  for (int i = 1; i <= 5; i++)
13  {
14      Entry entry = new AtomicEntry<Integer>(i);
15      capi.write(cref, 0, null, entry);
16  }
17  // read and print the first entry (1)
18  Entry [] readEntries = capi.read(cref, 0, null, sel);
19  System.out.println(((AtomicEntry) readEntries[0]).getValue());

```

Listing 3.10: FifoSelector

- **LifoCoordinator:** If the container uses the LifoCoordinator (LIFO stands for last-in-first-out), the entries are retrieved in the reverse order you wrote them. Thus, if you write 5 entries and then call the `read(cref, timeout, transaction, lifoSel)` method, the entry that was written LAST will be returned. You can compare this to a stack of plates – The last one you put on the stack will be the first one that you take from that stack.

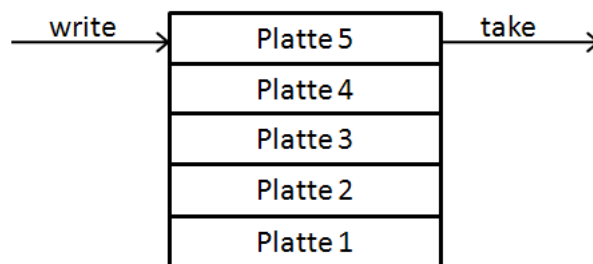


Figure 3.5: LifoCoordinator example

```

1  LifoSelector sel = new LifoSelector();
2
3  // Create new Container using no transaction, no remote server, container-
4  // size of 5 and a LifoCoordinator
5  ContainerRef cref = capi.createContainer(null,
6      null,
7      null,

```

```

8     5,
9     new LifoCoordinator());
10
11  /* Fill the queue with the numbers 1 to 5*/
12  for (int i = 1; i <= 5; i++)
13  {
14      Entry entry = new AtomicEntry<Integer>(i);
15      capi.write(cref, 0, null, entry);
16  }
17  // read and print the last value (5)
18  Entry [] readEntries = capi.read(cref, 0, null, sel);
19  System.out.println(((AtomicEntry) readEntries[0]).getValue());

```

Listing 3.11: LifoCoordinator

- **LindaCoordinator:** This is a Coordinator that retrieves data using Linda template matching [3]. You write the Entries (which must be tuples for this kind of coordination) to the Container; when you want to retrieve specific data, you define a template Entry. This template is a standard Entry, as already used when writing the data. For example, you write the following tuples to the Container:

```

< <"hello", "world">, 1>
< <"hello", "world">, 2>
< <"hello", "world">, 3>
< <"hello", "world">, 4>
< <"hello", "world">, 5>

```

Now you define a template Entry, which looks like:

```

< <"hello", "world">, null>

```

Now, all entries are retrieved.

If you define as template this Entry:

```

< <"hello", "world">, 1>

```

then only one Entry will be retrieved, as there is only one Entry in the container that matches this template. Thus, the template Entry needs to have the same structure (in terms of the Tuple and AtomicEntry structure). You define the fields of the Entry where you require exact matching and null where you don't require exact matching of the Entry.

```

1  //Create new Container using no transaction, no remote server,
2  //container-size of 5 and a LindaCoordinator
3  ContainerRef cref = capi.createContainer(null, null, null, 5,
4      new LindaCoordinator());
5
6  //Fill the container with <<"hello","world">,i> tuples where
7  //i = 1 to 5
8  for (int i = 1; i <= 5; i++)
9  {
10     Entry intEntry = new AtomicEntry<Integer>(i);
11     Entry entry = new Tuple(new Tuple(
12         new AtomicEntry<String>("hello"),

```

```

13  new AtomicEntry<String>("world")
14  ),new AtomicEntry<Integer>(i));
15  capi.write(cref, 0, null, entry);
16  }
17
18  //this template retrieves all entries as the last field in the tuple
19  //is null, but it is the only difference between the written tuples
20  Entry template1 = new Tuple(new Tuple(
21    new AtomicEntry<String>("hello"),
22    new AtomicEntry<String>("world")
23  ), null);
24  LindaSelector sel1 = new LindaSelector(Selector.CNT_ALL, template1);
25
26  Entry [] readEntries1 = capi.read(cref, 0, null, sel1);
27  printAllEntries(readEntries1);
28
29  //this template retrieves only the Entry with the 1 as last field in
30  //the tuple
31  Entry template2 = new Tuple(new Tuple(
32    new AtomicEntry<String>("hello"),
33    null
34  ),new AtomicEntry<Integer>(1));
35
36  LindaSelector sel2 = new LindaSelector(Selector.CNT_ALL, template2);
37
38  Entry [] readEntries2 = capi.read(cref, 0, null, sel2);
39  printAllEntries(readEntries2);

```

Listing 3.12: LindaCoordinator. The printAllEntries method is not listed here – it only prints the Entries’ content one after the other.

- **RandomCoordinator:** This is the standard Coordinator (if you don’t use a Coordinator at all). The entries are returned in a random way. It is used by default when you don’t pass a Selector for reading, deleting etc. of entries (see Chapter 3.4). You can compare this to a bag with lottery numbers:

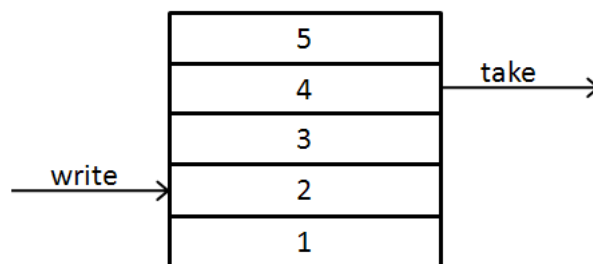


Figure 3.6: RandomCoordinator example

```

1  ContainerRef cref = capi.createContainer(null, null, null, 5, null);
2
3  /* Fill the queue with the numbers 1 to 5*/
4  for (int i = 1; i <= 5; i++)
5  {
6    Entry entry = new AtomicEntry<Integer>(i);
7    capi.write(cref, 0, null, entry);

```

```

8 }
9 // read and print a random value (something from 1 to 5)
10 Entry [] readEntries = capi.read(cref, 0, null, null);
11 System.out.println(((AtomicEntry) readEntries[0]).getValue());

```

Listing 3.13: RandomCoordinator (default)

Explicit (you need to give extra information when writing or reading the Entry):

- **KeyCoordinator:** When you use the KeyCoordinator, you can define a key when you write an entry. When you want to find this very entry again, you use the same key to get exactly this entry. So, for example, you have an AtomicEntry which holds an object of type Student, which has several properties. When you write the entry to the container, you use the Student's matriculation number as key. When you later want to find this Student in the container, you need to use the KeySelector together with the Student's matriculation number, which is the key.

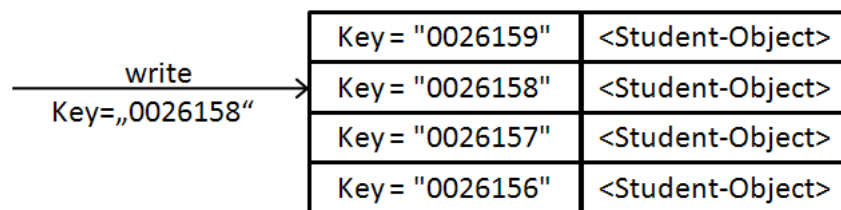


Figure 3.7: KeyCoordinator example

The name of the key must be a String, the key itself is generic, enabling to use any Java object, which is in turn passed as the type parameter. The key (see class KeyType in JavaDoc) has to be specified at the time the container is created. It is possible to specify several keys but it is not possible to add, remove or modify the key's name or type later on.

```

1 /* Create new Container using no transaction, no name, an infinite */
2 /* container-size a KeyCoordinator with one predefined key*/
3 ContainerRef cref = capi.createContainer(null, null, null,
4     IContainer.INFINITE_SIZE,
5     new KeyCoordinator(new KeyType("MatrNr", Integer.class)));
6
7 /* Create a Student instance and write it into the container */
8 Student writeStudent = new Student(1000, "Max", "Muster", 20);
9
10 /* Create new AtomicEntry using KeySelector */
11 Entry entry = new AtomicEntry<Student>(writeStudent, Student.class,
12     new KeySelector<String>("Surname", writeStudent.getSurname()));
13 capi.write(cref, 0, null, entry);
14
15 /* Get entry using KeySelector from the container */

```

```

16 Entry [] readEntries = capi.read(cref, 0, null,
17     new KeySelector<String>("Surname", "Muster"));
18
19 if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
20 {
21     Student readStudent = ((AtomicEntry<Student>) readEntries[0]).getValue();
22     System.out.println("MatNr.:    " + readStudent.get_MatNr());
23     System.out.println("Forename:  " + readStudent.get_Forename());
24     System.out.println("Surname:   " + readStudent.get_Surname());
25     System.out.println("Age:       " + readStudent.get_Age());
26 }

```

Listing 3.14: KeyCoordinator

- **VectorCoordinator:** The VectorCoordinator is used when you want to address the Entries by index. You can compare this with the position of Racing cars.

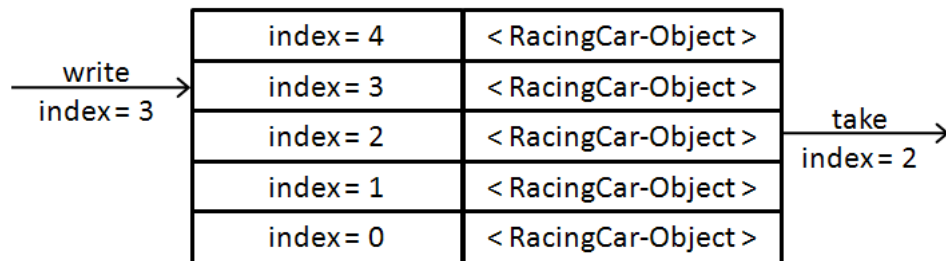


Figure 3.8: VectorCoordinator example

```

1 ContainerRef cref = capi.createContainer(null, null, null,
2     IContainer.INFINITE_SIZE,
3     new VectorCoordinator());
4
5 /* Create 5 Strings and write them to the container */
6 for (int i = 1; i <= 5 ; i++)
7 {
8     Entry entry = new AtomicEntry<String>(i+" . Person", String.class,
9         new VectorSelector(VectorSelector.APPEND, 0));
10    capi.write(cref, 0, null, entry);
11 }
12 /* Get entry using VectorSelector from the container */
13 Entry [] readEntries = capi.read(cref, 0, null, new VectorSelector(2, 1));
14
15 if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
16 {
17     System.out.println(((AtomicEntry<String>) readEntries[0]).getValue());
18 }

```

Listing 3.15: VectorCoordinator

One property of the VectorCoordinator which might be a bit tricky for beginners is that when you write an Entry to an index where another Entry already resides, this existing Entry is moved by one (its index is automatically incremented). All Entries' subsequent indexes are also incremented by 1. You can imagine this like

pushing all Entries with higher indexes one position further:

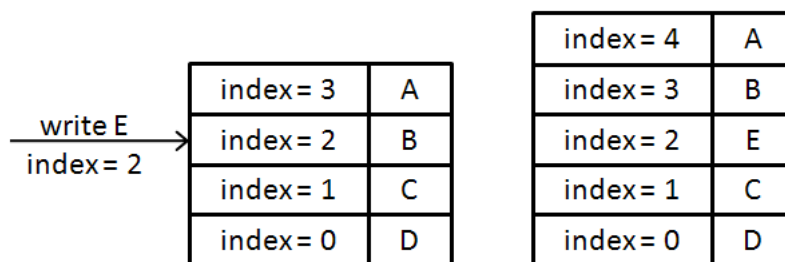


Figure 3.9: Write tentry to VectorCoordinator

Note that B was stored with index 2, but now it is stored with index 3. On the other hand, when you destroy an Entry from a container that is coordinated by a VectorCoordinator, the indexes of the Entries that have a higher index than the destroyed one are decremented by one:

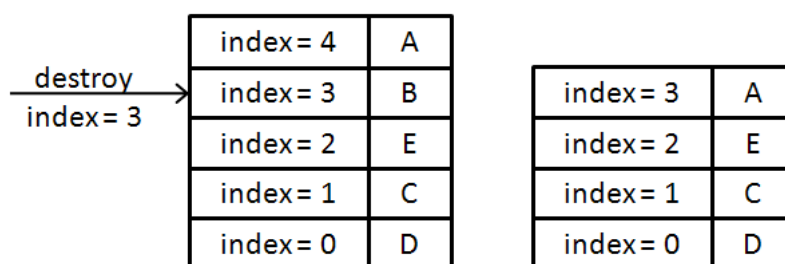


Figure 3.10: Destroy entry in VectorCoordinator

The Entry A was stored with index 4, after the destroy command, it was moved to index 3. Thus you need to keep in mind that the indexes of Entries may be changed when adding/removing Entries.

3.4 Selectors

The Selector is the counterpart to the Coordinator. It tells additional information to the Coordinator when you want to read or delete data, like the key value that you're searching for or the number of values that you'd like to retrieve. For each Coordination type, there exists the appropriate Selector.

The Selector class is abstract and offers two constructors:

```
1 public Selector()
2 public Selector(int count)
```

Listing 3.16: Selector

The count parameter is the number of entries that you'd like to retrieve using this Selector. If you leave away the count parameter, only one Entry is returned; when using the KeySelector, the Entry that has this key is returned. If you would like to have all entries of the container that fulfil the selection, use `Selector.CNT_ALL` as count parameter. Note that in the Linda example in chapter 3.3, I assumed the usage of the `Selector.CNT_ALL` parameter. When you use the LindaSelector without a count parameter, only one Entry is returned, if a matching one exists. If you create an Entry and write it to the container, you may pass a Selector as parameter (see Chapter 3.2). As Implicit Coordinators have the view over all Entries in the container, you don't need to pass it explicitly when writing the Entry to the container – the Implicit Coordinator manages the Entry anyway. When you pass an Explicit Selector together with its required additional info, the Explicit Coordinator (which has a view over all or a subset of Entries in the complete container) will manage this Entry, together with the Implicit Coordinator (as it has the view over ALL Entries, even those that are managed by an Explicit Coordinator). Thus, for example, if you created the container and passed it a FifoCoordinator and a KeyCoordinator, then you can write an Entry with a KeySelector, but still will be able to retrieve it using the FifoSelector.

The currently implemented implicit selectors only offer the constructors mentioned above, as the sequence of the retrieved entries is given implicitly (remember, the implicit selectors currently are: FIFO, LIFO, Linda, Random). The explicit selectors on the other hand offer a bit more information to the Coordinator, therefore, they are explained separately.

3.4.1 VectorSelector

```
1 public VectorSelector(int index, int count)
2 public VectorSelector(int count)
```

Listing 3.17: VectorSelector

The VectorSelector has the additional parameter index, which indicates the number of Entries in the container that you'd like to access. Note that if you write an Entry to the container, the numbers of the other entries with higher and equal numbers are shifted by one, which means that each index of an Entry that has a higher index is incremented by 1. For example:

```

1 Entry entry = new AtomicEntry<String>(
2   "The first one",
3   String.class,
4   new VectorSelector(1, 0));

```

Listing 3.18: Inserting an Entry with VectorSelector

3.4.2 KeySelector

```

1 public class KeySelector<T> extends Selector {
2   public KeySelector(String keyName, T keyValue) ...

```

Listing 3.19: KeySelector

The KeySelector is a generic class, you need to define which object type the key will be. The keyValue parameter will be the same type that you defined when you created the KeySelector instance:

```

1 KeySelector<String> keySel = new KeySelector<String>("Name", "Maier");

```

Listing 3.20: Instantiate KeySelector

Using this KeySelector, you could pass it as parameter to the Entry when you write it:

```

1 Entry entry = new AtomicEntry<Account>(
2   new Account(10000),
3   Account.class,
4   keySel);

```

Listing 3.21: Inserting an Entry with KeySelector

or you can use it when you want to retrieve this entry:

```

1 Entry[] readEntries = capi.read(cref, 0, null, keySel);

```

Listing 3.22: KeySelector read Entry

Note that the KeySelector doesn't have a parameter for the count, as the key is unique and thus, either only one or no entry can be retrieved with this key. If you want to write an Entry to the container with a key that already exists, an exception is thrown. Please further note that the shift method doesn't throw this exception, as it first would remove the clashing Entry.

Please take a look at the complete example at the end of this chapter to better understand how to use the KeySelector.

3.5 Methods to access data

In this chapter, the used `ContainerRef` objects are the same instances that you got when you created the container. For now, ignore the `Transaction` parameter, when programming, you can use `null` instead of the `tx`. Transactions will be explained in chapter 3.

3.5.1 write

```
1 public void write(ContainerRef cref, long timeout, Transaction tx, Entry... entries)
    throws XCoreException
```

Listing 3.23: write

By using this method, you can write one or more entries to the space. Also here, you use the `ContainerRef` object to refer to the corresponding container. When the container's size is limited, and there are already as much entries in the container as allowed (the number of entries in the container is equal to the maximum capacity), and you want to write further entries to the container, this operation blocks. The write operation waits until an entry is taken or destroyed or – if given – until the timeout is reached. If you'd like to use a `Selector` together with a specific `Entry` (e.g. a `KeySelector`), this `Selector` already is passed when you created the `Entry` (see Chapter 2.4)

3.5.2 read

```
1 public Entry[] read(ContainerRef cref, long timeout, Transaction tx, Selector... sel)
    throws XCoreException
```

Listing 3.24: read

The `read` method returns a number of `Entries` from the container, without destroying them. You need to use a `Selector` that is corresponding to the `Coordination`. If you leave away the `Selector` parameter, `RandomSelector` is used. The implicit as well as the explicit `Coordinators` can have a `count` parameter in the constructor, which tells the container the number of entries that you'd like to retrieve (see chapter 2.4 about `Selectors`). Thus, the return value of the `read` is an array of `Entries`, because you can define a `Selector` to read and return multiple `Entries` at once. If you don't use the parameter to tell the number of entries that should be read, you get one entry (if there is at least one in the container) or none. One important feature that can be used for synchronization is the fact, that if there is currently no entry in the container that can be retrieved, the `read` method blocks and waits for an `Entry`. And that's the reason why there is a `timeout` parameter:

It tells the read method to wait at least the timeout in seconds. So giving a timeout of `Timeout.NO_TIMEOUT`, you have a non-blocking read method call – it either reads an entry or if there is none, it immediately returns an `Exception`. The opposite would be to use the `Timeout.INFINITE_TIMEOUT`, which waits forever, if no one ever writes a suitable entry to the container.

Example: One time with `size = 2` and the second time without `size` given. Also one read with a `Timeout` given and one without. As transaction, `null` is used.

```

1 //Create new Container using no transaction , container-size of 5 and a
2 //Fifo Coordinator
3 ContainerRef cref = capi.createContainer(null , null , null ,
4     5, new FifoCoordinator());
5
6 /* Fill the queue with the numbers 1 to 5 */
7 for (int i = 1; i <= 5; i++)
8 {
9     Entry entry = new AtomicEntry<Integer>(i);
10    capi.write(cref , 0, null , entry);
11 }
12 Entry [] readEntries = capi.read(cref , 0, null , new FifoSelector());
13 /* ... prints 1 */
14 System.out.println(((AtomicEntry<Integer>) readEntries[0]).getValue());
15
16 /* Read from the container with FifoSelector of size 2 */
17 readEntries = capi.read(cref , 0, null , new FifoSelector(2));
18 /* ... prints 1 */
19 System.out.println(((AtomicEntry<Integer>) readEntries[0]).getValue());
20 /* ... prints 2 */
21 System.out.println(((AtomicEntry<Integer>) readEntries[1]).getValue());
22
23 /* Read from the container with timeout */
24 readEntries = capi.read(cref , 5, null , new FifoSelector());
25 /* ... prints 1 */
26 System.out.println(((AtomicEntry<Integer>) readEntries[0]).getValue());

```

Listing 3.25: Reading an entry using the `FifoSelector`.

3.5.3 take

```

1 public Entry [] take(ContainerRef cref , long timeout , Transaction tx , Selector... sel)
   throws XCoreException

```

Listing 3.26: `take`

This method is the same as first calling `read` and then `destroy`: The read `Entry` is immediately destroyed after reading in one atomic step. This functionality is also commonly referred to as “consuming read”. Also the `take` method could block and thus has a `timeout` parameter.

3.5.4 destroy

```
1 public void destroy(ContainerRef cref, long timeout, Transaction tx, Selector... sel)
    throws XCoreException
```

Listing 3.27: destroy

The destroy method adheres to the same rules as the read method. Depending on the container's Coordination type, an entry is destroyed, but without returning the destroyed entry. Note that also, when destroying entries, the method blocks until a suitable entry is found in the container. One might think that it is not very meaningful to wait for an entry just to immediately destroy it, but even though there are better techniques to perform this task, one might use this technique to notify a single listener. The listener calls `destroy(cref, Timeout.INFINITE_TIMEOUT)` and thus is blocked. The notifier writes an entry to the container and doing so, the entry is destroyed and the listener is unblocked. As already said, there are better possibilities in XVSM to perform such tasks – in most cases, you'll have entries when you call destroy, so this method won't block.

3.5.5 shift

```
1 public void shift(ContainerRef cref, Transaction tx, Entry... entries) throws
    XCoreException
```

Listing 3.28: shift

The shift method has the same behaviour as the write method, but instead of blocking if the container is already full, the shift method destroys an entry automatically. Thus, no timeout parameter is needed – it just never blocks. The entries that are shifted out of the container depend on the Coordinators of the Container and the Selectors used in the passed Entries. The shift method decides what Entry to destroy, in order to be able to write the new Entry.

3.6 Examples for Implicit Coordinators

3.6.1 Example: The Lottery (RandomCoordinator)

In this example, we would like to show how an Austrian Lottery¹ (in German it is called: "Lotto 6 aus 45") can be programmed. As we assume that the lottery really picks the

¹for non-Austrians: you have 45 numbered balls, the moderator picks 6 out of them + 1 additional. If you have guessed the 6 correctly, you've won the major amount; also you get quite a lot of money if you have 5 correct + the 1 additional.

balls randomly, we take the RandomCoordinator with a size of 45. First, we write the 45 numbered balls to the container. Then we take the 6 + 1 balls out of the container. At the end, we list the taken balls.

```

1 package org.xvsm.tutorial.lottery;
2
3 import java.net.URI;
4 import java.net.URISyntaxException;
5
6 import org.xvsm.coordinators.RandomCoordinator;
7 import org.xvsm.core.AtomicEntry;
8 import org.xvsm.core.Capi;
9 import org.xvsm.core.ContainerRef;
10 import org.xvsm.core.Entry;
11 import org.xvsm.interfaces.ICapi;
12 import org.xvsm.internal.exceptions.XCoreException;
13 import org.xvsm.selectors.RandomSelector;
14 import org.xvsm.selectors.Selector;
15
16
17 /*
18  * Example 2.5.1: Lottery (RandomCoordinator)
19  * This example writes the numbers (balls) from 1 to 45 to the container
20  * and then takes 6 + 1 numbers (balls) out of the container randomly.
21  *
22  * @author Formanek, Keszthelyi
23  */
24 public class Lottery
25 {
26
27     public static void main(String[] args) throws XCoreException, URISyntaxException
28     {
29         /* Create new Capi instance */
30         ICapi capi = new Capi();
31
32         /* Create new RandomSelector instance */
33         Selector sel = new RandomSelector();
34
35         /*
36          * Create new Container using RandomCoordinator, RandomSelector and a
37          * container-size limit of 45
38          */
39         ContainerRef cref = capi.createContainer(null, new URI("tcpjava://localhost:4321"),
40             "lottery", 45, new RandomCoordinator());
41
42         /* Write the 45 numbers to the container */
43         for (int i = 1; i < 46; i++)
44         {
45             /* Create a new AtomicEntry */
46             Entry entry = new AtomicEntry<Integer>(i);
47             /* Write the entry to the container */
48             capi.write(cref, 0, null, entry);
49         }
50
51         /* Take 6 + 1 numbers from the container */
52         System.out.print("Regular Numbers: ");
53         for (int j = 0; j < 7; j++)

```

```
53 {
54     /* Take an entry from the container using the RandomSelector */
55     Entry [] readEntries = capi.take(cref, 0, null, sel);
56
57     System.out.print(((AtomicEntry<Integer>) readEntries[0]).getValue());
58     if (j == 5)
59     {
60         System.out.print("\nBonus: ");
61     }
62     else if (j < 6)
63     {
64         System.out.print(", ");
65     }
66 }
67
68 /* Remove container from space */
69 capi.destroyContainer(null, cref);
70
71 System.exit(0);
72 }
73 }
```

Listing 3.29: The Austrian Lottery

3.6.2 Exercise: The Ticket Queue (FIFOCoordinator)

A bunch of people is standing in a queue to buy cinema tickets. One person after the other is served by the shop assistant. A person who wants to buy a ticket must pay and wait for the ticket to be printed. This takes some time (we assume that this is a quite fast action and takes 2 seconds in total). Please program a Capi peer that handles the persons in the queue one after the other. As person objects, use an extra class “Person” that has a `payTicket()` and a `waitForTicket()` method. Both methods only wait for 1 second. In later examples, this exercise will be enhanced. The shop assistant only gets the first person in the queue, receives the payment and prints the ticket. Afterwards, the person object in the queue leaves the queue. You don’t need to have a shop assistant class, just write it in the main method of the main class.

3.7 Examples for Explicit Coordinators

3.7.1 Example: The Student management (KeyCoordinator)

The Technical University of Vienna would like to have a new student management system developed using XVSM. Each Student has a matriculation number (MatrNr), so we’ll use

a `KeyCoordination`, where the key is the `MatrNr`. As a quick test, we'll write 5 `Student` objects and read them by retrieving them by their `MatrNr`.

3.7.2 Exercise: Formula 1 Race (`VectorCoordinator`)

You're the IT professional in managing the Race results of a Formula 1 Race. You have the task to implement a piece of software (using `XVSM` of course!) to show the current race position. The values of the entries are `RacingCar` objects, whereas they have the property "driverName". When someone accesses the container to retrieve the current status of the race, of course, all cars should be listed together with their current position. For simplicity, just write 10 cars to the container. In the first round, car number 6 crashes and thus is removed from the race. In the second round, car 4 overtakes car 3. As this is a short race, let's say the race is over after this lap. List the current racing positions at the end of each lap. Also this example will be improved later. Each `Racing` car is running on a separate `Peer` (thus, one peer creates and holds the container, whereas all the other peers are connecting and using this container then).

3.8 Summary:

In this chapter, you've learnt the most important basics to handle a container and a container's content. The basic access to the container is done using the `Capi` class, which offers methods to create, lookup and destroy a container and to read, write, destroy and shift entries from/to the container. An entry can be either a tuple or an atomic entry. A tuple can hold either other tuples or atomic entries. An atomic entry is a `Generic` that holds the data of interest. You also learnt how to set the internal coordination of the container and how to select specific data in the container. A typical use case scenario is that you'd like to create a new container and pass it the internal coordination type. Then, you'd like to create some entries and set the information in the entries. Afterwards, you'll write the entries to the container and then read or take them from the container. At the end, most probably you'll shutdown the `Capi` to unlink from the container.

Chapter 4

Transactions

We assume that most programmers are familiar with the concept of transactions: A transaction is helping to collect various actions to perform them as one single, atomic step to avoid consistency problems of the data storage. In the XVSM implementation, the transaction is performed as a pessimistic transaction; this means a transaction builds up locks on the data.

The lifecycle of a transaction from the user's perspective is as follows: Create a new transaction Perform various actions If everything went fine, commit the transaction to write the changes to the data storage If there was an error, rollback the changes to discard all actions you made.

In XVSM, if you start a transaction and then only perform read operations, a read lock is obtained. But as soon as you perform a write operation (or another operation that changes the container's content), the container or the accessed Entries (depending on the isolation level implemented by the MozartSpaces version you use) in the container are locked exclusively for this one transaction. There are multiple read locks allowed, but only one write lock at a time. All other applications or application parts that would like to access the container but don't use this transaction are blocked, no matter if they want to read or write to that container, until the transaction is committed or rolled back. It depends on the used Coordinator, if the complete container must be locked or if it is sufficient to only lock the accessed Entries. For example, the FifoCoordinator will lock the complete container, because the sequence the Entries will be returned, must remain the same while the transaction, whereas using a RandomCoordinator, the locks will be obtained only on the accessed Entries.

You create a transaction by calling

```
1 ICapi capi = new Capi();  
2 Transaction tx = capi.createTransaction((URI)site, (long)timeout);
```

Listing 4.1: Create Transaction

As you remember from the previous chapter, this Transaction object is passed as parameter in various operations (read, destroy, write, ...). So, if you call the write operation with the Transaction tx, the container resp. the corresponding part of the container is locked only for this transaction. Concurrent access to the container that is in conflict with the write operation will be blocked. Only the methods that use this tx transaction are executed without blocking. One thing that you need to keep in mind: if you don't set a transaction (you use null as parameter), internally, an implicit transaction is created. This means, internally, a transaction object is created – thus, also here, concurrent actions run isolated to each other, although the action is committed automatically. The site parameter is the URI, where the container is located at (see Chapter 2.1) and the timeout parameter can give a maximum validity time of the Transaction. If the given time has elapsed, all changes within this Transaction are rolled back and all further actions with this Transaction are rejected.

You commit the transaction by calling:

```
1 capi.commitTransaction(tx);
```

Listing 4.2: Commit Transaction

This means all your changes are written to the container and the container is unlocked then. The transaction object no longer is valid then.

You roll the changes in this transaction back by calling:

```
1 capi.rollbackTransaction(tx);
```

Listing 4.3: Rollback Transaction

This means that the actions that you've done so far are discarded. A rollback might be intended if some error occurred, so most probably, the rollback statement will be standing in an error handling part.

```
1 ICapi capi = new Capi();
2 tx = capi.createTransaction(null, Capi.INFINITE_TIMEOUT);
3 tx2 = capi.createTransaction(null, Capi.INFINITE_TIMEOUT);
4 try {
5
6     /* Create new Container using a container-size of 3 */
7     ContainerRef cref = capi.createContainer(null, null, null, 3, new FifoCoordinator());
8
9     /* Write 3 entries to the container */
10    /* After the first write action the container is locked */
11    for (int i = 1; i <= 3; i++) {
12        Entry entry = new AtomicEntry<Integer>(i);
13        capi.write(cref, 0, tx, entry);
14    }
15
16    capi.commitTransaction(tx);
```

```

17  /* The container is not locked anymore */
18
19  capi.read(cref, 0, tx2, new RandomSelector(3));
20  /* The container is still not locked */
21  capi.destroy(cref, 0, tx2, new RandomSelector());
22  /* The container is locked now */
23
24  throw new Exception();
25
26 } catch (Exception e) {
27     try {
28         if (tx != null) {
29             capi.rollbackTransaction(tx2);
30         }
31         /* The container is not locked */
32     }

```

Listing 4.4: Transaction handling

4.1 Exercise (Transactions)

Extend the exercise 2.7.2 (Formula 1): We'd like to let some cars run for a random amount of time. So we'd start a Capi peer for each car, let it wait for some time and then re-register in the container within a transaction to have a consistent container content. As this is a bit tricky, we'll give you a guide how to solve this exercise:

1. Add the properties "runtime" and "lapNumber" to each car.
2. For each car, start a new Capi peer. When having started it, let the peer wait for a random time between 3 and 5 seconds:

```

1 Random rnd = new Random();
2 int time = rnd.nextInt(2) + 3;  //(random between 0 and 2) plus 3

```

Listing 4.5: Random time

3. If the Time is up, start a transaction.
4. Add the time that you've waited to your runtime.
5. Remove the car that actually has the transaction from the container.
6. Read all actual positions of the cars, and get the runtime for each car. If you've found a runtime that is higher, whereas the next runtime is lower than your runtime, then this is your position.
7. Write the car object to the new position (remember that the entries in the vector with higher index than the one that you use will be shifted down automatically).

Don't forget that the car only needs to take 2 rounds, but of course you can use more rounds – you just need to keep the number of rounds for each car the same ...

8. Every time a car reaches the finish line, write the actual list of race positions together with the runtimes. Don't forget that cars that already have passed the finish line don't change their racing time any more.

4.2 Summary:

Transactions are fundamental if there are many peers (or threads) that use containers to avoid that through concurrent write access, the content may become inconsistent. A transaction helps you to keep the container consistent, as it serializes the transactions as if one were executed after the other. So normally, you'd like to start a transaction, read or write from/to the container and commit. All other application parts that want to access that part of the container but don't use this transaction are blocked.

Chapter 5

A completely new aspect: Aspects in XVSM

An Aspect in XVSM lets you implement an extension to the existing XVSM functionality. Features like automatic persistency or notification can be implemented using Aspects. You also can use Aspects to add logging, authentication and many more features to the existing XVSM implementation. The idea behind Aspects is to have various pre- and post-methods on each action that the Capi class is able to perform, like a `preWrite` and a `postWrite` method. You implement these pre- and post-methods to perform some special action that is done automatically before or after you call one of the following methods of the `ICapi` interface. You need to divide Aspects into two groups: Local Aspects and Global Aspects. Local Aspects are aspects that are connected to actions on a specific container, while Global Aspects refer to the space itself or to all containers. Each possibility to listen on a certain performed action is called `IPoint` (standing for `Interceptor Points`). Adhering to the just mentioned Local and Global Aspects, the corresponding `IPoints` are called `LocalIPoint` and `GlobalIPoint`. You not only have the possibility to create one Aspect to perform an action, but you can also add multiple Aspects to perform various actions in sequence. This possibility will be explained at the end of Chapter 4.1.

5.1 Local Aspects

Local Aspects offer the possibility to overwrite the pre- and post-Methods for the following actions:

- `addAspect()`
- `removeAspect()`

- read()
- destroy()
- take()
- write()
- shift()

If you want to create a new local aspect, you have to extend the abstract `LocalAspect` class, if you want to create a new global aspect, you have to extend the abstract `GlobalAspect` class. They already contain the needed pre- and post-methods that per-se don't do anything. If you implement a new Aspect, you override these methods to add new functionality to the method. An example for a new Local Aspect is:

```

1 public class LoggingAspect extends LocalAspect
2 {
3     static final long serialVersionUID = 0;
4     public void postWrite(ContainerRef cref ,
5                           Transaction tx ,
6                           List<Entry> entries , Properties p)
7     throws AspectNotOkException , AspectRescheduleException ,
8            AspectSkipException
9     {
10        try
11        {
12            FileOutputStream fs = new FileOutputStream("Logfile.log", true);
13            ObjectOutputStream os = new ObjectOutputStream(fs);
14            Iterator it = entries.iterator();
15            while (it.hasNext())
16            {
17                /* Write the value of the AtomicEntry to the logfile */
18                os.writeObject(((AtomicEntry) it.next()).getValue());
19            }
20            os.close();
21        }
22        catch (Exception e)
23        {
24            e.printStackTrace();
25        }
26    }
27
28    public void postRead(ContainerRef cref ,
29                        Transaction tx ,
30                        List<Entry> entries ,
31                        List<Selector> selectors , Properties p)
32    throws AspectNotOkException , AspectRescheduleException ,
33           AspectSkipException
34    {
35        try
36        {
37            FileOutputStream fs = new FileOutputStream("Logfile.log", true);
38            ObjectOutputStream os = new ObjectOutputStream(fs);
39            Iterator it = entries.iterator();
40            while (it.hasNext())

```

```

41     {
42         /* Write the value of the AtomicEntry to the logfile */
43         os.writeObject(((AtomicEntry) it.next()).getValue());
44     }
45     os.close();
46 }
47 catch (Exception e)
48 {
49     e.printStackTrace();
50 }
51 }
52 }

```

Listing 5.1: Logging Aspect

In the listing above, you extend the `LocalAspect` class and override the various methods and therefore add new functionality. The methods could look like this:

```

1 public void postWrite(
2 ContainerRef cref ,
3 Transaction tx ,
4 List<Entry> entries ,
5 Properties contextProperties)
6 throws AspectNotOkException ,
7 AspectRescheduleException ,
8 AspectSkipException

```

Listing 5.2: Overwrite LocalAspect Methode

The `cref` and `tx` parameters should already be clear. You also can give a list of `Entries` that are used in this action, to perform some special task with these entries, as you can see in the listing. The `contextProperties` are free to your disposal, if you want to pass additional information to the aspect; it is passed when calling the `ICapi.setAspectContext(Properties props)` method. Some exceptions could be thrown, it depends on the implementer of the aspect if they are used or not.

After you created the new aspect, you add it to the `Capi` by calling

```

1 LocalAspect aspect = new LoggingAspect();
2 List<IPoint> p = new ArrayList<IPoint>();
3 p.add(LocalIPoint.PostWrite);
4 p.add(LocalIPoint.PostRead);
5 capi.addAspect(cref, p, aspect);

```

Listing 5.3: Create and add a new LocalAspect

You pass a list of `IPoints` to indicate the actions where the aspect offers some special functionality and the aspect itself. The reason that you need to pass a number of `IPoints` is that you can choose if you want to perform only a part of the actions that an Aspect would offer. Imagine for example a Logging Aspect which offers Logging possibility for each Pre-/Post method, but you only would like to use it for the write action. Instead of

writing a new Aspect, you use the IPoints PreWrite and PostWrite and that's it. As explained in the introduction of Chapter 4, you have the possibility to add multiple Aspects to let them work one after the other. You only need to call the `add` method with the Aspect in the same order as you'd like them to be executed.

5.2 Global Aspects

Using Global Aspects, you can listen to the following actions, additionally to those already mentioned in Local Aspects:

- `createContainer()`
- `destroyContainer()`
- `transactionCreate()`
- `transactionCommit()`
- `transactionRollback()`
- `coreShutdown()`

The usage of a Global Aspect is similar to the usage of a Local Aspect. The only differences are the methods that are offered and that a Global Aspect is associated with a space, not with a specific container. So, for example you overwrite the following method:

```
1 public void preContainerCreate(String containerName, int size, Properties
   contextProperties, ICoordinator... coordinators) throws AspectNotOkException,
   AspectRescheduleException, AspectSkipException
```

Listing 5.4: Overwrite GlobalAspect Methode

and then, you add an Aspect as follows:

```
1 URI uri = new URI(...); //URI to a space (may be null for local)
2 GlobalAspect aspect = new SecurityAspect();
3 List<IPoint> p = new ArrayList<IPoint>();
4 p.add(GlobalIPoint.PreCoreShutdown);
5 capi.addAspect(uri, p, aspect);
```

Listing 5.5: Adding an Aspect

```
1 public static void main(String[] args)
2 {
3
4     try
5     {
6         ICapi capi = new Capi();
7         /* Create new Container using a container-size of 3 */
8         ContainerRef cref = capi.createContainer(null, null, null, 3,
9             new FifoCoordinator());
```

```

10
11     LocalAspect aspect = new LoggingAspect();
12     /* Create the IPoints for the aspect */
13     List<LocalIPoint> p = new ArrayList<LocalIPoint>();
14     p.add(LocalIPoint.PostWrite);
15     p.add(LocalIPoint.PostRead);
16     capi.addAspect(cref, p, aspect);
17
18     /* Write 3 entries to the container */
19     for (int i = 1; i <= 3; i++)
20     {
21         Entry entry = new AtomicEntry<Integer>(i);
22         capi.write(cref, 0, null, entry);
23     }
24     capi.read(cref, 0, null);
25
26     capi.shutdown(null, true);
27 }
28 catch (Exception e)
29 {
30     e.printStackTrace();
31 }
32 }

```

Listing 5.6: Adding the LoggingAspect to Capi

You also have the possibility to use the LocalAspects together with GlobalAspects-. In this case, the actions that are specific for LocalAspects are performed on ALL containers and not only a specific one. Using this functionality, you can for example implement a method that writes a log entry whenever an entry is read from ANY container in your space.

5.3 Summary:

Using Aspects, you can easily add new functionality to your XVSM system. You can define some actions that are done either before or after an operation is done with the container or space. So for example you can add the functionality that before a write, the permissions of the writer is checked. This way, security checks can be introduced. For a new Aspect, you overwrite the pre- and post-methods to add new functionality and then, you register the Aspect in the Capi.

Chapter 6

Don't miss a thing: Notifications

Using Aspects, a very important feature was added to the XVSM implementation: Notifications. With notifications, you can listen on various actions and be called back if such an action is performed. For example: You have a listening class that would like to be notified if a write is called (and thus, a new entry is added). This class must implement the `NotificationListener` interface and so it must implement the method `handleNotification(final Operation operation, final Entry... entries)`.

Then you need to register this class as listener for the call of the write method:

```
1 URI capi.createNotification(  
2     cref,  
3     this, //the class to be notified  
4     Operation.Write);
```

Listing 6.1: Create Notification

There exist the following notification targets, which should be self-explanatory:

```
1 Operation.Write  
2 Operation.Read  
3 Operation.Take  
4 Operation.Shift  
5 Operation.Destroy
```

Listing 6.2: Notification Targets

Removing Notifications You can remove notifications by removing the according `IPoints` of the notifications aspect. For example the code in listing 44 will remove the notifications regardless of the operation on which the notifications was registered. Of course you can remove a certain notifications on an operation, by specifying only that operation.

```
1 // create a container and register a notification
2 // nl is the NotificationListener implementation
3 ContainerRef cref = capi.createContainer(null, defaultURI, null, ICapi.INFINITE, (
    ICoordinator) null);
4 URI aspectURI = capi.createNotification(cref, nl, Operation.Write);
5 // write an entry
6 capi.write(cref, 0, null, new AtomicEntry<String>("ein string"));
7
8 //the notification fires
9
10 // remove the notification aspect
11 capi.removeAspect(cref, java.util.Arrays.asList(LocalIPoint.PostWrite),
12     aspectURI);
13
14 capi.write(cref, 0, null, new AtomicEntry<String>("ein string"));
15 // since the aspect has been removed the notification
16 // does not fire
```

Listing 6.3: Remove Notification

Putting all together, you can find that with the usage of notifications, you can much more efficiently program some of the examples/exercises of the recent chapters. Just to take one example: In the TicketQueue example, you've created one main peer which handles the Persons in the queue one after the other. Now imagine that you start one peer for each person and one peer for the shop assistant (which now is a new class). Both the Person and the Shop assistant implement the NotificationListener interface. When the Shop assistant is instantiated, it is registered as listener on a request to pay a ticket. The Person is registered as listener of the ticket when it has paid the ticket to receive the printed ticket. And as you might expect, there is a final exercise:

6.1 Exercise (TicketQueue with Notifications)

Try to program the just-described TicketQueue Example. You will need to have

- A Ticket and a Payment object that are used to notify the other party to do something.
- A second container to store these Ticket and Payment entries independently from the FIFO queue of the persons. You may name this container "SalesDesk".
- A couple of Persons which have a `payTicket()` method – the `waitForTicket()` method is not needed anymore, as you now have a `handleNotification()` method which tells the Person that the Ticket is printed. In the `payTicket()` method, write the Payment object to the SalesDesk container and register the Person as listener for a call of the write method. In the `handleNotification` method, you need

to take care if really a Ticket was returned (imagine multiple ShopAssistants and Persons at a time writing to the container – if you don't check if the Ticket is "yours", the person could either "steal" Payment or Tickets from others). If it really is a ticket, take it from the container, give a `System.out.println` that you've got the ticket and leave the queue ("destroy" the Person object from the container and shutdown this peer).

- A single ShopAssistant class which implements NotificationListener. In its `handleNotification` method, take the Payment from the SalesDesk container, give a `System.out.println()` that you're printing the ticket and wait for 1 second. Then store the Ticket object in the container. Again, in the `handleNotification` method, take care that you only take Payment objects, not Ticket objects.

6.2 Summary:

A Notification is used if your class is interested in a special event in the Container, for example if an Entry was written or read. This Entry can be passed to the `notify` method in order to enable the listening class to perform some action using this Entry. Using Notifications and looking at the last example, the character of Space-based computing is shown best: Instead of imperatively telling the ShopAssistant or the Person to do something, you write a request to the space instead. The other party takes the request, performs the requested action and writes the result back. If the requested party is overloaded or too slow – no problem: you add one more of such participants; they collaboratively work on the requests in the container. With the possibility to spread the container over multiple peers and with help of transactions, you can implement load balancing scenarios in the easiest way. Moreover, you can add or remove working peers on-the-fly.

Chapter 7

Bibliography

- [1] Mozartspaces. Website. <http://www.mozartspaces.org>; last visited on September 20th 2008.
- [2] Space based computing group. Website. <http://www.complang.tuwien.ac.at/eva/SBC-Group/sbcGroupIndex.html>; last visited on September 22th 2008.
- [3] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

Appendix A

Solutions for the exercises

(only available for those who really tried hard to solve the problem)

A.1 The TicketQueue (see Exercise 3.6.2)

```
1 package org.xvsm.tutorial.ticketqueue;
2
3 import java.net.URI;
4 import java.net.URISyntaxException;
5
6 import org.xvsm.core.AtomicEntry;
7 import org.xvsm.core.Capi;
8 import org.xvsm.core.ContainerRef;
9 import org.xvsm.core.Entry;
10 import org.xvsm.interfaces.ICapi;
11 import org.xvsm.interfaces.ICoordinator;
12 import org.xvsm coordinators.FifoCoordinator;
13 import org.xvsm.interfaces.container.IContainer;
14 import org.xvsm.internal.exceptions.XCoreException;
15 import org.xvsm.selectors.FifoSelector;
16
17
18 /*
19  * Example 2.5.2: Ticket Queue (FifoCoordinator)
20  * This example writes a couple of persons to the container.
21  * Every person in the ticket queue has to pay the ticket and
22  * wait for it to be printed.
23  *
24  * @author Formanek, Keszthelyi
25  */
26 public class TicketQueue
27 {
28     private static int CUSTOMERS = 5;
29
30     public static void main(String [] args) throws XCoreException, URISyntaxException
31     {
32         /* Create new Capi instance */
33         ICapi capi = new Capi();
```

```

34
35  /* Create new FifoSelector instance */
36  FifoSelector sel = new FifoSelector();
37
38  /*
39   * Create new Container using FifoCoordinator, FifoSelector and an
40   * infinite container-size
41   */
42  ContainerRef cref = capi.createContainer(null, new URI("tcpjava://localhost:4321"),
43      "queue", IContainer.INFINITE_SIZE, new FifoCoordinator());
44
45  /* Fill the queue with a couple of persons */
46  for (int i = 1; i <= CUSTOMERS; i++)
47  {
48      /* Create a new AtomicEntry */
49      Entry entry = new AtomicEntry<Person>(new Person(i));
50      /* Write the entry to the container */
51      capi.write(cref, 0, null, entry);
52  }
53
54  /* Handle the persons of the queue */
55  for (int i = 1; i <= CUSTOMERS; i++)
56  {
57      /* Take an entry from the container using the FifoSelector */
58      Entry[] readEntries = capi.take(cref, Capi.INFINITE_TIMEOUT, null, sel);
59      if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
60      {
61          Person actualPerson = ((AtomicEntry<Person>) readEntries[0]).getValue();
62          /* start paying the Ticket */
63          actualPerson.payTicket();
64          /* start printing the Ticket */
65          actualPerson.waitForTicket();
66      }
67  }
68
69  /* Remove container from space */
70  capi.destroyContainer(null, cref);
71
72  System.exit(0);
73 }

```

Listing A.1: TicketQueue.java

```

1  package org.xvsm.tutorial.ticketqueue;
2
3  import java.io.Serializable;
4
5  /*
6   * Example 2.5.2: Ticket Queue (FifoCoordinator)
7   * This class provides all needed actions of a Person.
8   *
9   * @author Formanek, Keszthelyi
10  */
11 public class Person implements Serializable
12 {
13
14     int waitingPosition;

```

```
15
16 /**
17  * Constructor
18  *
19  * @param waitingPosition
20  */
21 public Person(int waitingPosition)
22 {
23     this.waitingPosition = waitingPosition;
24 }
25
26 /**
27  * Simulate payment
28  */
29 public void payTicket ()
30 {
31     try
32     {
33         System.out.println("The " + waitingPosition + ". person pays the ticket.");
34         Thread.sleep(1000);
35     }
36     catch (Exception e)
37     {
38         e.printStackTrace();
39     }
40 }
41
42 /**
43  * Simulate waiting for the ticket
44  */
45 public void waitForTicket ()
46 {
47     try
48     {
49         System.out.println("The " + waitingPosition + ". person waits for the ticket.");
50         Thread.sleep(1000);
51     }
52     catch (Exception e)
53     {
54         e.printStackTrace();
55     }
56 }
57
58 }
```

Listing A.2: Person.java

A.2 Forumula 1 Race (see Exercise 3.7.2)

```
1 package org.xvsm.tutorial.formula1race;
2
3 import java.net.URI;
4 import java.net.URISyntaxException;
5
```

```

6 import org.xvsm coordinators.VectorCoordinator;
7 import org.xvsm.core.AtomicEntry;
8 import org.xvsm.core.Capi;
9 import org.xvsm.core.ContainerRef;
10 import org.xvsm.core.Entry;
11 import org.xvsm.interfaces.ICapi;
12 import org.xvsm.interfaces.ICoordinator;
13 import org.xvsm.interfaces.container.IContainer;
14 import org.xvsm.internal.exceptions.XCoreException;
15 import org.xvsm.selectors.Selector;
16 import org.xvsm.selectors.VectorSelector;
17
18
19 /*
20  * Example 2.6.2: Formula 1 Race (VectorCoordinator)
21  * This example writes 10 RacingCar objects to the container. In the first lap
22  * the 6th positioned car has an accident and is removed from the container.
23  * In lap 2 car number 4 overtakes number 3.
24  *
25  * @author Formanek, Keszthelyi
26  */
27 public class Formula1Race
28 {
29
30     // only needed for nice printing
31     final protected static int MAX_LAPS = 2;
32
33     /**
34      * @param args
35      * @throws URISyntaxException
36      * @throws XCoreException
37      */
38     public static void main(String [] args) throws XCoreException, URISyntaxException
39     {
40         String driver [] = {"Lewis Hamilton", "Fernando Alonso", "Kimi Raikonen", "Felipe
41             Massa", "Nick Heidfeld", "Robert Kubica",
42             "Heikki Kovalainen", "Giancarlo Fisichella", "Nico Rosberg", "Alexander Wurz"};
43
44         /* Create new Capi instance */
45         ICapi capi = new Capi();
46
47         /* Create new Container using VectorCoordinator and VectorSelector */
48         ContainerRef racingPositions = capi.createContainer(null, new URI("tcpjava://
49             localhost:4321"), "positions", IContainer.INFINITE_SIZE, new VectorCoordinator
50             ());
51         ContainerRef retiredCars = capi.createContainer(null, new URI("tcpjava://localhost
52             :4321"), "retired", IContainer.INFINITE_SIZE, new VectorCoordinator());
53
54         /* Create 10 RacingCar instances and write them to the container */
55         for (int i = 0; i < driver.length; i++)
56         {
57             RacingCar writeCar = new RacingCar(driver[i]);
58             /* Create a new entry */
59             Entry entry = new AtomicEntry<RacingCar>(writeCar, RacingCar.class, new
60                 VectorSelector(VectorSelector.APPEND, 0));
61             /* Write entry to the container */
62             capi.write(racingPositions, 0, null, entry);
63         }
64     }
65 }

```

```

59     printCurrentPositioning(capi, racingPositions, retiredCars, 0);
60
61     /*
62     * LAP 1 Car number 6 has an accident and is removed from the list
63     */
64
65     /* Read and destroy entry 6 in the container */
66     Entry [] readEntries = capi.take(racingPositions, 0, null, new VectorSelector(6, 1));
67
68     /* Write the read entry to the retiredCars container */
69     if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
70     {
71         RacingCar readCar = ((AtomicEntry<RacingCar>) readEntries[0]).getValue();
72         Entry entry = new AtomicEntry<RacingCar>(readCar, RacingCar.class, new
73             VectorSelector(0));
74         capi.write(retiredCars, 0, null, entry);
75     }
76
77     printCurrentPositioning(capi, racingPositions, retiredCars, 1);
78
79     /*
80     * LAP 2 Car number 4 overtakes number 3
81     */
82
83     /* Read and destroy entry 4 in the container */
84     readEntries = capi.take(racingPositions, 0, null, new VectorSelector(4, 1));
85
86     if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
87     {
88         /* Write back previously read entry at index 3 */
89         RacingCar readCar = ((AtomicEntry<RacingCar>) readEntries[0]).getValue();
90         Entry entry = new AtomicEntry<RacingCar>(readCar, RacingCar.class, new
91             VectorSelector(3, 0));
92         capi.write(racingPositions, 0, null, entry);
93     }
94
95     printCurrentPositioning(capi, racingPositions, retiredCars, 2);
96
97     /* Remove container from space */
98     capi.destroyContainer(null, racingPositions);
99     capi.destroyContainer(null, retiredCars);
100
101     System.exit(0);
102 }
103
104 /*
105 * Print current positioning
106 */
107 public static void printCurrentPositioning(ICapi capi, ContainerRef racingPositions,
108     ContainerRef retiredCars, int lap) throws XCoreException
109 {
110     // only some nice print-formatting
111     if (lap == 0)
112         System.out.println("*** Starting Grid ***");
113     else if (lap == MAX_LAPS)
114         System.out.println("*** Lap " + lap + " - Final Lap ***");
115     else

```

```

114     System.out.println("*** Lap " + lap + " ***");
115
116     /* Read the current positioning */
117     Entry[] readEntries = capi.read(racingPositions, 0, null, new VectorSelector(
118         VectorSelector.CNT_ALL));
119
120     int i = 1;
121     for (Entry readEntry : readEntries)
122     {
123         if (readEntry.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
124         {
125             RacingCar car = ((AtomicEntry<RacingCar>) readEntry).getValue();
126             System.out.println("Pos " + i++ + " - " + car.getDriver());
127         }
128     }
129
130     /* Read all retired cars */
131     readEntries = capi.read(retiredCars, 0, null, new VectorSelector(Selector.CNT_ALL));
132
133     for (Entry readEntry : readEntries)
134     {
135         if (readEntry.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
136         {
137             RacingCar retiredCar = ((AtomicEntry<RacingCar>) readEntry).getValue();
138             System.out.println("Retired " + retiredCar.getDriver());
139         }
140     }
141
142     System.out.println("*** ***** ***** ***\n");
143 }

```

Listing A.3: Formula1Race.java

```

1 package org.xvsm.tutorial.formula1race;
2
3 import java.io.Serializable;
4 import java.lang.Exception;
5
6
7 /*
8  * Example 2.6.2: Formula 1 Race (VectorCoordinator)
9  * This class stores all needed information of a Racing-Car.
10 *
11 * @author Formanek, Keszthelyi
12 */
13 public class RacingCar implements Serializable
14 {
15
16     private String driver = null;
17
18     /**
19     * Constructor
20     *
21     * @param driver
22     */
23     public RacingCar(String driver)
24     {

```

```
25     this.driver = driver;
26 }
27
28 /**
29  * @return driver
30  */
31 public String getDriver()
32 {
33     return driver;
34 }
35 }
```

Listing A.4: RacingCar.java

A.3 Formula 1 Race extended by Transactions (see Exercise 4.1)

```
1 package org.xvsm.tutorial.transactions;
2
3 import java.net.URI;
4 import java.net.URISyntaxException;
5
6 import org.xvsm coordinators.VectorCoordinator;
7 import org.xvsm.core.AtomicEntry;
8 import org.xvsm.core.Capi;
9 import org.xvsm.core.ContainerRef;
10 import org.xvsm.core.Entry;
11 import org.xvsm.core.notifications.Operation;
12 import org.xvsm.interfaces.ICapi;
13 import org.xvsm.interfaces.NotificationListener;
14 import org.xvsm.interfaces.container.IContainer;
15 import org.xvsm.internal.exceptions.XCoreException;
16
17 import org.xvsm.selectors.Selector;
18 import org.xvsm.selectors.VectorSelector;
19 import org.xvsm.transactions.Transaction;
20
21 /*
22  * Exercise 3.1: Transactions (Formula 1 extension)
23  * This class represents the server and coordinates
24  * the clients so they can start all at the same time.
25  *
26  * @author Formanek, Keszthelyi
27  */
28 public class Formula1RaceTransaction implements NotificationListener
29 {
30     public static final String RACING_POSITIONS_CONTAINER = "RacingPositions";
31     public static final String COORDINATION_CONTAINER = "Coordination";
32     public static final int MAX_LAPS = 2;
33     public static final int MAX_CARS = 10;
34     public static final int WAIT_FOR_CLIENTS = 15;
35
36     private ICapi capi = null;
```

```

37 private URI serverURI = null;
38 private ContainerRef racingPositions = null;
39 private ContainerRef coordination = null;
40 private boolean running = false;
41 private int registeredCars = 0;
42
43 public Formula1RaceTransaction()
44 {
45 }
46
47
48 /**
49  * Create a new race and wait X sec (= WAIT_FOR_CLIENTS) for
50  * clients to register. After the expiration of this term place
51  * a message to the coordination-container, telling the clients
52  * to start the race. After the last client has finished print the
53  * results.
54  *
55  * @throws XCoreException
56  * @throws URISyntaxException
57  */
58 public void openRace() throws XCoreException, URISyntaxException
59 {
60     /* Create new Capi instance */
61     capi = new Capi();
62     serverURI = new URI("tcpjava://localhost:4321");
63
64     /* Create new Container using VectorCoordinator */
65     racingPositions = capi.createContainer(null, serverURI, RACING_POSITIONS_CONTAINER,
66         MAX_CARS, new VectorCoordinator());
67     coordination = capi.createContainer(null, serverURI, COORDINATION_CONTAINER,
68         IContainer.INFINITE_SIZE, new VectorCoordinator());
69
70     /* Wait for clients */
71     for (int i = WAIT_FOR_CLIENTS; i > 0; i--)
72     {
73         System.out.println("Waiting " + i + " seconds for Clients");
74         try
75         {
76             Thread.sleep(1000);
77         }
78         catch (Exception e)
79         {
80             e.printStackTrace();
81         }
82     }
83
84     /* read how many cars have registered */
85     registeredCars = capi.read(racingPositions, 0, null, new VectorSelector(Selector.
86         CNT_ALL)).length;
87
88     System.out.println("Found " + registeredCars + " registered cars");
89
90     /* start race if one or more cars have registered */
91     if (registeredCars > 0)
92     {
93         System.out.println("Starting Race!");
94         running = true;
95     }
96 }

```

```

92
93     /* register to be notified on WRITES in the coordination-container */
94     capi.createNotification(coordination, this, Operation.Write);
95
96     /* place message telling clients to start race */
97     /* create transaction */
98     Transaction tx = capi.createTransaction(serverURI, Capi.INFINITE_TIMEOUT);
99     /* create new entry appending to vectorcontainer */
100    Entry startMessage = new AtomicEntry<Message>(new Message(Message.START_RACE),
101        Message.class, new VectorSelector(VectorSelector.APPEND, 0));
102    /* write entry with transaction */
103    capi.write(coordination, Capi.INFINITE_TIMEOUT, tx, startMessage);
104    /* commit transaction */
105    capi.commitTransaction(tx);
106
107    /* wait until the race has finished */
108    while (running);
109 }
110
111 /* wait for WRITE-notifications */
112 public void handleNotification(Operation operation, Entry... entries)
113 {
114     try
115     {
116         /* lookup how many entries are already in the container */
117         int numOfEntries = capi.read(coordination, Capi.INFINITE_TIMEOUT, null, new
118             VectorSelector(VectorSelector.CNT_ALL)).length;
119
120         /* if there are registeredCars + 1 entries in the coordination-container then all
121            * clients have finished. print the results and quit.
122            */
123         if (numOfEntries == registeredCars + 1)
124         {
125             running = false;
126
127             System.out.println("Race finished!\n Printing result:");
128             /* Read the current ranking */
129             Entry[] readEntries = capi.read(racingPositions, 0, null, new VectorSelector(
130                 VectorSelector.CNT_ALL));
131
132             int i = 1;
133             for (Entry readEntry : readEntries)
134             {
135                 if (readEntry.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
136                 {
137                     CarInfo car = ((AtomicEntry<CarInfo>) readEntry).getValue();
138                     System.out.println("Pos " + i++ + " - " + car.getDriver() + ", Time: " + car
139                         .getRunTime());
140                 }
141             }
142
143             System.out.println("*** ** Shutting down now!");
144
145             /* Destroy containers */
146             capi.destroyContainer(null, racingPositions);
147             capi.destroyContainer(null, coordination);

```

```

146     System.out.println("Bye!");
147     System.exit(0);
148 }
149 } catch (XCoreException e)
150 {
151     // TODO Auto-generated catch block
152     e.printStackTrace();
153 }
154 }
155
156 /**
157  * @param args
158  * @throws URISyntaxException
159  * @throws XCoreException
160  */
161 public static void main(String [] args)
162 {
163     Formula1RaceTransaction f1Race = new Formula1RaceTransaction();
164
165     try
166     {
167         /* start a new race */
168         f1Race.openRace();
169     } catch (XCoreException e)
170     {
171         e.printStackTrace();
172     } catch (URISyntaxException e)
173     {
174         e.printStackTrace();
175     }
176 }
177 }

```

Listing A.5: Formula1RaceTransaction.java

```

1 package org.xvsm.tutorial.transactions;
2
3 import java.io.Serializable;
4
5 /*
6  * Exercise 3.1: Transactions (Formula 1 extension)
7  * This class is used to coordinate the server and its clients.
8  *
9  * @author Formanek, Keszthelyi
10 */
11 public class Message implements Serializable
12 {
13     public static final int START_RACE = 0;
14     public static final int CAR_FINISHED = 1;
15
16     private int messageType = -1;
17
18     /**
19      * Constructor
20      *
21      * @param messageType
22      */
23     public Message(int messageType)

```

```
24 {
25     this.messageType = messageType;
26 }
27
28
29 /**
30  * Return message-type
31  *
32  * @return messageType
33  */
34 public int getMessageType()
35 {
36     return messageType;
37 }
38 }
```

Listing A.6: Message.java

```
1 package org.xvsm.tutorial.transactions;
2
3 import java.io.Serializable;
4
5 /**
6  * Exercise 3.1: Transactions (Formula 1 extension)
7  * This class stores all needed information for a Racing-Car.
8  *
9  * @author Formanek, Keszthelyi
10 */
11 public class CarInfo implements Serializable
12 {
13     private String driver = null;
14     private int runTime = 0;
15     private int lapNumber = 0;
16
17     /**
18      * Constructor
19      *
20      * @param driver Name of driver
21      */
22     public CarInfo(String driver)
23     {
24         this.driver = driver;
25     }
26
27
28     /**
29      * Return the driver's name
30      *
31      * @return driver
32      */
33     public String getDriver()
34     {
35         return driver;
36     }
37
38
39     /**
40      * Return the current runtime
```

```
41  *
42  * @return runTime
43  */
44  public int getRunTime()
45  {
46      return runTime;
47  }
48
49
50  /**
51   * Return the current lapnumber
52   *
53   * @return lapNumber
54   */
55  public int getLapNumber()
56  {
57      return lapNumber;
58  }
59
60  /**
61   * Add time to runTime
62   *
63   * @param time
64   */
65  public void addTime(int time)
66  {
67      runTime += time;
68  }
69
70  /**
71   * Increment lapnumber
72   *
73   */
74  public void incrementLap()
75  {
76      lapNumber++;
77  }
78 }
```

Listing A.7: CarInfo.java

```
1 package org.xvsm.tutorial.transactions;
2
3 import java.net.URI;
4 import java.net.URISyntaxException;
5 import java.util.Arrays;
6 import java.util.List;
7 import java.util.Random;
8 import java.util.Vector;
9
10 import org.xvsm.core.AtomicEntry;
11 import org.xvsm.core.Capi;
12 import org.xvsm.core.ContainerRef;
13 import org.xvsm.core.Entry;
14 import org.xvsm.core.notifications.Operation;
15 import org.xvsm.interfaces.ICapi;
16 import org.xvsm.interfaces.NotificationListener;
17 import org.xvsm.internal.exceptions.InvalidTransactionException;
```

```

18 import org.xvsm.internal.exceptions.TimeoutExpiredException;
19 import org.xvsm.internal.exceptions.XCoreException;
20 import org.xvsm.selectors.Selector;
21 import org.xvsm.selectors.VectorSelector;
22 import org.xvsm.transactions.Transaction;
23
24 /*
25  * Exercise 3.1: Transactions (Formula 1 extension)
26  * This class represents the car (client) that registers to
27  * the server by placing its carinfo into the |"racingPositions|"
28  * container. After receiving a notification from the server about
29  * starting the race, the simulation is started. Each round the carInfo
30  * is taken from the container, updated and written back to the container.
31  * This is done by using one transaction and on error retried until the
32  * transaction could successfully be committed.
33  *
34  * @author Formanek, Keszthelyi
35  */
36 public class RacingCar implements NotificationListener
37 {
38     private ICapi capi = null;
39     private ContainerRef racingPositions = null;
40     private ContainerRef coordination = null;
41     private URI serverURI = null;
42
43     private boolean running = true;
44     private String driver;
45
46     /**
47      * Constructor
48      *
49      * @param driver Name of driver
50      */
51     public RacingCar(String driver)
52     {
53         this.driver = driver;
54     }
55
56     /**
57      * Register car by placing its carinfo into
58      * the |"racingPositions|" container.
59      *
60      *
61      * @throws XCoreException
62      * @throws URISyntaxException
63      */
64     public void registerCar() throws XCoreException, URISyntaxException
65     {
66         capi = new Capi();
67         serverURI = new URI("tcpjava://localhost:4321");
68
69         /* lookup containers */
70         racingPositions = capi.lookupContainer(null, serverURI, Formula1RaceTransaction.
71             RACING_POSITIONS_CONTAINER);
71         coordination = capi.lookupContainer(null, serverURI, Formula1RaceTransaction.
72             COORDINATION_CONTAINER);
72         System.out.println("Found Container");
73

```

```

74  /* register to receive notification about the race-start.
75  * because we don't expect other messages at the beginning, we only need a
       notification once.
76  */
77  capi.createNotification(coordination, this, Operation.Write);
78
79  /* create transaction */
80  Transaction tx = capi.createTransaction(serverURI, Capi.INFINITE_TIMEOUT);
81  /* create new carinfo-entry appending it to vectorcontainer */
82  Entry carInfo = new AtomicEntry<CarInfo>(new CarInfo(driver), CarInfo.class, new
       VectorSelector(VectorSelector.APPEND, 1));
83  /* write carinfo */
84  capi.write(racingPositions, Capi.INFINITE_TIMEOUT, null, carInfo);
85  /* commit transaction */
86  capi.commitTransaction(tx);
87
88  System.out.println("Registered Car!");
89
90  /* wait until race has finished */
91  while (running);
92  }
93
94  /**
95   * Simulate the race
96   *
97   * @throws XCoreException
98   */
99  public void simulateRace() throws XCoreException
100  {
101  System.out.println("Simulating!");
102
103  for (int lap = 1; lap <= Formula1RaceTransaction.MAX_LAPS; lap++)
104  {
105  Random rnd = new Random();
106
107  int time = rnd.nextInt(2) + 3;
108
109  System.out.println("Lap: " + lap + "waiting " + time + "sec.");
110  try
111  {
112  /* simulate lap */
113  Thread.sleep(time * 1000);
114  } catch (InterruptedException e)
115  {
116  }
117
118  boolean reposition = true;
119
120  /* retry until transaction has successfully committed */
121  while (reposition)
122  {
123  /* create transaction */
124  Transaction tx = capi.createTransaction(serverURI, Capi.INFINITE_TIMEOUT);
125
126  try
127  {
128  /* take current positions */

```

```

129     Entry entries [] = capi.take(racingPositions, 2500, tx, new VectorSelector(
130         Selector.CNT_ALL));
131
132     /* search for the carinfo matching this driver */
133     for (Entry entry : entries)
134     {
135         CarInfo car = ((AtomicEntry<CarInfo>) entry).getValue();
136
137         if (car.getDriver().compareTo(driver) == 0)
138         {
139             /* update carinfo */
140             car.addTime(time);
141             car.incrementLap();
142             break;
143         }
144     }
145
146     /* new vector for easier sorting */
147     Vector<CarInfo> newPosition = new Vector<CarInfo>();
148
149     /* resort entries */
150     for (Entry entry : entries)
151     {
152         int i = 0;
153         /* find position for entry */
154         for (; i < newPosition.size(); i++)
155         {
156             CarInfo car = newPosition.get(i);
157
158             if (car.getRuntime() > ((AtomicEntry<CarInfo>) entry).getValue().
159                 getRuntime())
160             {
161                 newPosition.add(i, ((AtomicEntry<CarInfo>) entry).getValue());
162                 break;
163             }
164         }
165
166         /* in case the vector is empty or we reached the end of
167          * vector in the iteration above, append entry */
168         if (i == newPosition.size())
169         {
170             newPosition.add(((AtomicEntry<CarInfo>) entry).getValue());
171         }
172     }
173
174     /* write entries */
175     for (CarInfo car : newPosition)
176     {
177         /* create new entry for carinfo */
178         Entry carInfo = new AtomicEntry<CarInfo>(car, CarInfo.class, new
179             VectorSelector(VectorSelector.APPEND, 1));
180
181         /* write carinfo */
182         capi.write(racingPositions, 2500, tx, carInfo);
183     }
184
185     /* commit transaction */
186     capi.commitTransaction(tx);

```

```

184     /* leave loop */
185     reposition = false;
186 } catch (XCoreException e)
187 {
188     /* is exception an InvalidTransactionException? */
189     if (e instanceof InvalidTransactionException)
190     {
191         /* try to rollback transaction */
192         try
193         {
194             capi.rollbackTransaction(tx);
195         } catch (XCoreException ex)
196         {
197             // discard any XCoreExceptions here
198         }
199     }
200     /* is exception an TimeoutExpiredException? */
201     else if (e instanceof TimeoutExpiredException)
202     {
203         /* try to rollback transaction */
204         try
205         {
206             capi.rollbackTransaction(tx);
207         } catch (XCoreException ex)
208         {
209             // discard any XCoreExceptions here
210         }
211     }
212     /* in all other cases print error and exit */
213     else
214     {
215         e.printStackTrace();
216         System.exit(1);
217     }
218
219     /* Penalty in case of transaction/timeout-exception
220     * This is needed to get a better propability that
221     * one client can commit its transaction and minimize
222     * concurrency problems.
223     */
224     int penalty = rnd.nextInt(10) * 100;
225     System.out.println("Timeout while waiting!\nPenalty waiting " + penalty + "ms"
226         );
227
228     try
229     {
230         Thread.sleep(penalty);
231     } catch (InterruptedException e1)
232     {
233     }
234 }
235 }
236
237 /* Simulation ended. Place Finished-message into coordination-container */
238 System.out.println("Finished! Placing message...");
239 /* create transaction */
240 Transaction tx = capi.createTransaction(serverURI, Capi.INFINITE_TIMEOUT);

```

```

241  /* create new entry appending to vectorcontainer */
242  Entry carInfo = new AtomicEntry<Message>(new Message(Message.CAR_FINISHED), Message.
      class, new VectorSelector(VectorSelector.APPEND, 1));
243  /* write entry with transaction */
244  capi.write(coordination, Capi.INFINITE_TIMEOUT, tx, carInfo);
245  /* commit transaction */
246  capi.commitTransaction(tx);
247
248  System.out.println("Simulation done! Bye!");
249  System.exit(0);
250 }
251
252 /*
253  * wait for notification about race-start
254  *
255  *
256  * @see org.xvsm.interfaces.NotificationListener#handleNotification(org.xvsm.core.
      notifications.NotificationContext)
257  */
258 public void handleNotification(Operation operation, Entry... entries)
259 {
260     /* inspect first entry */
261     if (((AtomicEntry<Message>) entries[0]).getValue().getMessageType() == Message.
          START_RACE)
262     {
263         try
264         {
265             simulateRace();
266         } catch (XCoreException e)
267         {
268             // TODO Auto-generated catch block
269             e.printStackTrace();
270         }
271     }
272 }
273
274 /**
275  * @param args
276  * @throws URISyntaxException
277  * @throws XCoreException
278  */
279 public static void main(String[] args) throws XCoreException, URISyntaxException
280 {
281     if ((args.length != 1))
282     {
283         System.out.println("Usage: RacingCar name\n" +
284                             "name ... Driver-Name");
285         System.exit(1);
286     }
287
288     RacingCar racingCar = new RacingCar(args[0]);
289
290     /* register car for race */
291     racingCar.registerCar();
292 }
293 }

```

Listing A.8: RacingCar.java

A.4 TicketQueue extended by Notifications (see Exercise 6.1)

```

1 package org.xvsm.tutorial.ticketqueuenotification;
2
3 import java.io.BufferedReader;
4 import java.io.InputStreamReader;
5 import java.net.URI;
6
7 import org.xvsm coordinators.FifoCoordinator;
8 import org.xvsm.core.AtomicEntry;
9 import org.xvsm.core.Capi;
10 import org.xvsm.core.ContainerRef;
11 import org.xvsm.core.Entry;
12 import org.xvsm.core.notifications.Operation;
13 import org.xvsm.interfaces.ICapi;
14 import org.xvsm.interfaces.container.IContainer;
15 import org.xvsm.interfaces.NotificationListener;
16 import org.xvsm.internal.exceptions.XCoreException;
17 import org.xvsm.selectors.FifoSelector;
18
19 /*
20  * Example 5.1: TicketQueue with Notifications
21  * This class represents the shop-assistent who
22  * waits for payments from customers and places
23  * the ticket.
24  *
25  * @author Formanek, Keszthelyi
26  */
27 public class ShopAssistent implements NotificationListener {
28     private ICapi capi = null;
29     private boolean quit = false;
30     private ContainerRef salesDesk = null;
31
32     /**
33      * Constructor
34      *
35      */
36     public ShopAssistent() {
37     }
38
39     /**
40      * Start shop-assistent. Create or lookup container, register as a
41      * notificationlistener, check if there are already payments, proceed first
42      * and wait then for notifications.
43      *
44      *
45      */
46     public void run() {
47         try {
48             capi = new Capi();
49
50             try {
51                 /* try to create container */
52                 capi.createContainer(null, new URI("tcpjava://localhost:4321"),
53                     ISalesDeskObject.SALES_DESK_CONTAINER,

```

```

54         IContainer.INFINITE_SIZE, new FifoCoordinator());
55     } catch (Exception ex) {
56         /* nothing to do here */
57     }
58
59     /* lookup container */
60     salesDesk = capi.lookupContainer(null, new URI(
61         "tcpjava://localhost:4321"),
62         ISalesDeskObject.SALES_DESK_CONTAINER);
63
64     /* register to receive notifications */
65     capi.createNotification(salesDesk, this, Operation.Write);
66     System.out.println("Registered to the Container");
67
68     /*
69      * check if there are already payments available after proceeding
70      * the first payment we will be notified about new writes and don't
71      * have to take care about other payments here anymore.
72      *
73      * We assume that clients don't leave without taken their ticket!
74      */
75     Entry[] entries;
76     try {
77         entries = capi.read(salesDesk, 0, null,
78             new FifoSelector());
79         ISalesDeskObject sdo = ((AtomicEntry<ISalesDeskObject>) entries[0])
80             .getValue();
81
82         /* is the entry an instance of payment */
83         if (sdo instanceof Payment) {
84             proceedPayment(salesDesk);
85         }
86     } catch (Exception e) {
87
88     }
89     System.out.println("Waiting for customers ...");
90
91     /* wait for correct user input to quit */
92     while (!quit) {
93         BufferedReader stdin = new BufferedReader(
94             new InputStreamReader(System.in));
95
96         if (stdin.readLine().compareTo("q") == 0) {
97             quit = true;
98         }
99     }
100 } catch (Exception e) {
101     /* exit on error */
102     e.printStackTrace();
103     System.exit(1);
104 }
105 }
106
107 /*
108  * check notification if it is an instance of payment, take it from the
109  * container if it is and write back a new ticket for the payment's owner.
110  * Ignore all other notifications.
111  */

```

```

112 public void handleNotification(Operation operation, Entry... entries) {
113     System.out.println("Received Notification!");
114     try {
115         /* take first entry from container using the fifo-selector */
116         Entry[] read;
117         try {
118             read = capi.read(salesDesk, 0, null,
119                 new FifoSelector(1));
120         } catch (Exception e) {
121             /* someone was faster, nothing to do */
122             return;
123         }
124         ISalesDeskObject sdo = ((AtomicEntry<ISalesDeskObject>) read[0])
125             .getValue();
126
127         /* Check if the object is a payment */
128         if (sdo instanceof Payment) {
129             proceedPayment(salesDesk);
130         }
131         else
132         {
133             System.out.println("Received Notification! — IGNORING");
134         }
135     } catch (Exception e) {
136         /* print any exception, we don't expect one */
137         e.printStackTrace();
138     }
139     System.out.println("Received Notification! — END");
140 }
141
142 /**
143  * take the payment from the container and write back a ticket for the
144  * payment's owner
145  *
146  *
147  * @param salesDesk
148  * @throws XCoreException
149  */
150 public void proceedPayment(ContainerRef salesDesk) throws XCoreException {
151     /* take first entry using the fifo-selector */
152     Entry[] entries = capi.take(salesDesk, Capi.INFINITE_TIMEOUT, null,
153         new FifoSelector());
154
155     Payment payment = (Payment) ((AtomicEntry<ISalesDeskObject>) entries[0])
156         .getValue();
157     System.out.println("ShopAssistent: Received Payment from ID: "
158         + payment.getOwner().getID());
159
160     /* create a new ticket */
161     Ticket ticket = new Ticket(payment.getOwner());
162     Entry entry = new AtomicEntry<ISalesDeskObject>(ticket,
163         ISalesDeskObject.class, new FifoSelector());
164     /* write ticket into the container */
165     capi.write(salesDesk, Capi.INFINITE_TIMEOUT, null, entry);
166     System.out.println("ShopAssistent: Placed Ticket for ID: "
167         + ticket.getOwner().getID() + " :: " + ticket.getClass());
168 }
169

```

```

170  /**
171   * @param args
172   */
173  public static void main(String[] args) {
174      ShopAssistent assi = new ShopAssistent();
175      assi.run();
176
177      System.out.println("Exiting!");
178      System.exit(0);
179  }
180 }

```

Listing A.9: ShopAssistent.java

```

1  /**
2   *
3   */
4  package org.xvsm.tutorial.ticketqueuenotification;
5
6  import java.net.URI;
7
8  import org.xvsm.coordinators.FifoCoordinator;
9  import org.xvsm.core.AtomicEntry;
10 import org.xvsm.core.Capi;
11 import org.xvsm.core.ContainerRef;
12 import org.xvsm.core.Entry;
13 import org.xvsm.core.notifications.Operation;
14 import org.xvsm.interfaces.ICapi;
15 import org.xvsm.interfaces.NotificationListener;
16 import org.xvsm.interfaces.container.IContainer;
17 import org.xvsm.selectors.FifoSelector;
18
19 /*
20  * Example 5.1: TicketQueue with Notifications
21  * This class represents the customer who places
22  * his payment on the salesdesk and waits until
23  * he receives his ticket.
24  *
25  * @author Formanek, Keszthelyi
26  */
27 public class Customer implements NotificationListener {
28
29     private int ID = 0;
30     private ICapi capi = null;
31     private boolean quit = false;
32     private ContainerRef salesDesk = null;
33
34     /**
35      * Constructor
36      *
37      * @param ID customer ID
38      */
39     public Customer(int ID)
40     {
41         this.ID = ID;
42     }
43
44     /**

```

```

45  * create/lookup container, register as a notificationlistener,
46  * place payment into the salesdesk-container and wait until the
47  * ticket has been received.
48  *
49  */
50  public void run()
51  {
52      try
53      {
54          capi = new Capi();
55
56          try
57          {
58              /* try to create container */
59              capi.createContainer(null, new URI("tcpjava://localhost:4321"), ISalesDeskObject
              .SALES_DESK_CONTAINER, IContainer.INFINITE_SIZE, new FifoCoordinator());
60          }
61          catch (Exception ex)
62          {
63              /* nothing to do here */
64          }
65
66          /* lookup container */
67          salesDesk = capi.lookupContainer(null, new URI("tcpjava://localhost:4321"),
              ISalesDeskObject.SALES_DESK_CONTAINER);
68
69          /* register to receive notifications */
70          capi.createNotification(salesDesk, this, Operation.Write);
71          System.out.println("Registered to the Container");
72
73          /* create payment */
74          Payment payment = new Payment(new Person(ID));
75          Entry entry = new AtomicEntry<ISalesDeskObject>(payment, ISalesDeskObject.class,
              new FifoSelector());
76          /* write payment */
77          capi.write(salesDesk, Capi.INFINITE_TIMEOUT, null, entry);
78          System.out.println("Placed Payment");
79
80          /* wait for ticket */
81          while(!quit)
82          {
83          }
84      }
85      catch (Exception e)
86      {
87          e.printStackTrace();
88          System.exit(1);
89      }
90  }
91
92  /* on notification verify if a ticket is the first
93  * entry in the salesdesk-container, in case it is
94  * taken from the container and exit.
95  */
96  public void handleNotification(Operation operation, Entry... entries)
97  {
98      System.out.println("Received Notification!");
99  }

```

```

100     try
101     {
102         /* take first entry from container using the fifo-selector */
103         Entry [] read;
104         try {
105             read = capi.read(salesDesk, 0, null,
106                 new FifoSelector(1));
107         } catch (Exception e) {
108             /* someone was faster, nothing to do */
109             return;
110         }
111         ISalesDeskObject sdo = ((AtomicEntry<ISalesDeskObject>) read[0]).getValue();
112
113         System.out.println("class :: " + sdo.getClass());
114         /* Check if the object is a payment and this customer is its owner */
115         if ((sdo instanceof Ticket) && (sdo.getOwner().getID() == ID))
116         {
117             /* take the ticket*/
118             read = capi.take(salesDesk, Capi.INFINITE_TIMEOUT, null, new FifoSelector(1));
119             System.out.println("Customer: Received my Ticket!" + read.length);
120
121             /* nothing to do anymore, exit */
122             System.exit(0);
123         }
124     }
125     catch (Exception e)
126     {
127         e.printStackTrace();
128     }
129 }
130
131 /**
132  * @param args
133  */
134 public static void main(String [] args)
135 {
136     /* check if an ID has been provided */
137     if ((args.length != 1))
138     {
139         printUsage();
140     }
141
142     int ID = 0;
143
144     /* parseInt, printUsage on fail */
145     try
146     {
147         ID = Integer.parseInt(args[0]);
148     } catch (NumberFormatException e)
149     {
150         printUsage();
151     }
152
153     /* ID must be > 0 */
154     if (ID > 0)
155     {
156         Customer customer = new Customer(Integer.parseInt(args[0]));
157

```

```
158     customer.run();
159 }
160 else
161 {
162     printUsage();
163 }
164 }
165
166 /**
167  * Print correct usage and exit
168  *
169  */
170 public static void printUsage()
171 {
172     System.out.println("Usage: customer id\n" +
173         "id ... Integer > 0");
174     System.exit(1);
175 }
176 }
```

Listing A.10: Customer.java

```
1 package org.xvsm.tutorial.ticketqueuenotification;
2
3 import java.io.Serializable;
4
5 /*
6  * Example 5.1: TicketQueue with Notifications
7  * This class is used to identify the customer who
8  * pays and waits for his ticket.
9  *
10 * @author Formanek, Keszthelyi
11 */
12 public class Person implements Serializable
13 {
14     /* customer-ID */
15     private int ID = 0;
16
17     /**
18      * Constructor
19      *
20      * @param ID customer-ID
21      */
22     public Person(int ID)
23     {
24         this.ID = ID;
25     }
26
27     /**
28      * Return customer's ID
29      *
30      * @return ID
31      */
32     public int getID()
33     {
34         return ID;
35     }
36 }
```

Listing A.11: Person.java

```
1 package org.xvsm.tutorial.ticketqueuenotification;
2
3 import java.io.Serializable;
4
5 /*
6  * Example 5.1: TicketQueue with Notifications
7  * This is an interface for objects placed in the "SalesDesk"-container
8  *
9  * @author Formanek, Keszthelyi
10 */
11 public interface ISalesDeskObject extends Serializable
12 {
13     public static final String SALES_DESK_CONTAINER = "SalesDesk";
14
15     public Person getOwner();
16 }
```

Listing A.12: ISalesDeskObject.java

```
1 package org.xvsm.tutorial.ticketqueuenotification;
2
3 /*
4  * Example 5.1: TicketQueue with Notifications
5  * This class implements ISalesDeskObject and represents a ticket placed by a
6  * shopassistant.
7  *
8  * @author Formanek, Keszthelyi
9  */
10 public class Ticket implements ISalesDeskObject
11 {
12     Person person; // Customer who placed this payment
13
14     /**
15      * Create a payment
16      *
17      * @param person ... The person who has payed for the Ticket
18      */
19     public Ticket(Person person)
20     {
21         this.person = person;
22     }
23
24     /**
25      * Takes a person as parameter, compares it with the owner
26      * of this ticket and returns true if the comparison was true.
27      *
28      * @param person
29      *
30      * @return boolean
31      */
32     public boolean isOwner(Person person)
33     {
34         return this.person.equals(person);
35     }
36 }
```

```
35
36 /**
37  * return this payment's owner
38  *
39  * @return person
40  */
41 public Person getOwner()
42 {
43     return person;
44 }
45 }
```

Listing A.13: Ticket.java

```
1 package org.xvsm.tutorial.ticketqueuenotification;
2
3 /*
4  * Example 5.1: TicketQueue with Notifications
5  * This class implements ISalesDeskObject and represents a payment placed by customer.
6  *
7  * @author Formanek, Keszthelyi
8  */
9 public class Payment implements ISalesDeskObject
10 {
11     private Person person; // Customer who placed this payment
12
13     /**
14      * Create a payment
15      *
16      * @param person
17      */
18     public Payment(Person person)
19     {
20         super();
21         this.person = person;
22     }
23
24     /**
25      * return this payment's owner
26      *
27      * @return person
28      */
29     public Person getOwner()
30     {
31         return person;
32     }
33 }
```

Listing A.14: Payment.java