Department of Electronic and
Computer Engineering

MSc in Distributed Computing Systems Engineering

Master Thesis

Design and implementation of a component for information publishing in a distributed component based system

Frank Müller

Supervisor: Roger Prowse

Table of Contents

1. Introduction

One of the most important terms of today’s computer science is ‚ubiquitous connectivity’. This means that you can use electronic services, like the Internet, anywhere at any time. However there are still places, where this is not possible, or only in a limited way. For example in vehicles we normally cannot get information, write emails or do our work. Of course we could use portable computers, but only with less performance or comfort. From the point of view of a vehicle manufacturer there is one decisive problem about offering such technology to the passengers of a vehicle - the car lives much longer than the software. Permanently new services are created, which the buyer of a car could use. This means that the software must be updated during the lifetime of the car. The best way would be to update the software during runtime, without the need to drive to a garage.

The result of these considerations is a project, which is called COSIMA (**CO**mponent **S**ystem **I**nformation and **M**anagement **A**rchitecture). It is a component-based architecture, which allows loading, unloading, moving and updating software components dynamically during runtime without restarting the system. Chapter 3.1 of this thesis describes this architecture. Another problem that has to be solved by this project is the publication of information in a dynamic system. Information, which is created by newer components, should also be usable by older components in the system.

The aim of this thesis is to develop a component for COSIMA, which allows the distribution of information in a dynamic component based system. Information should be identifiable by all components in the system. Load balancing and failure-tolerance are important points that have to be considered. A prototype of this component will be implemented in Java. The inter-component communication will be realised with RMI (Remote Method Invocation) [Mue02]. The new component will be called **ObjectSpace**. The dissertation will show that ObjectSpace can be developed in any programming language and that the communication can be built with different protocols. So the implementation is only a reference implementation, which shows the mechanisms in form of a prototype.

2. About this dissertation

Source code, statements and the names of classes and objects are written in the font `Courier New`. Parts of programs, names of variables or methods, which are mentioned in the text, are also written in this font. Important items in the text are printed **bold** to highlight them.

3. Background of the project

It will be shown in the analysis that the best way to distribute information in a distributed system will be the development of a new component. The component will run in a component-based system called COSIMA (**CO**mponent **S**ystem **I**nformation and **M**anagement **A**rchitecture). The following chapter will describe this system.

3.1. COSIMA

COSIMA describes an architecture, which is shown in the following diagram.

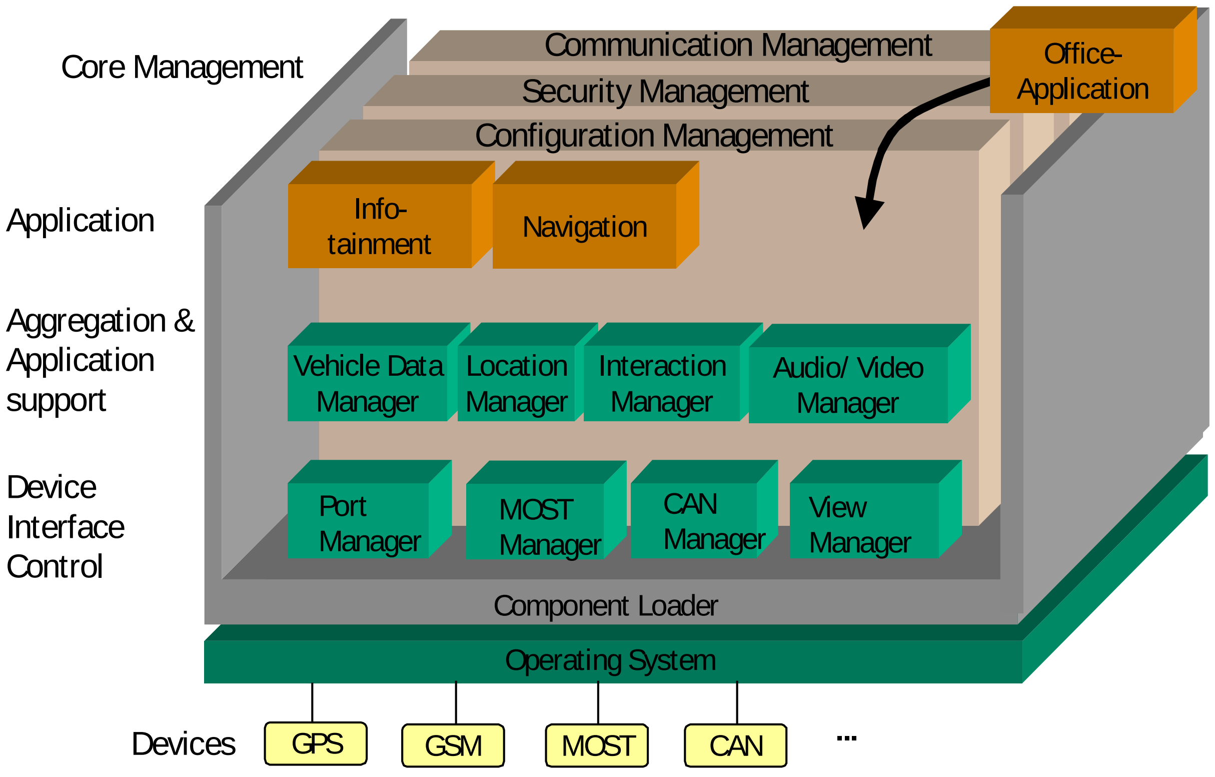

Figure 1 COSIMA - COmponent System Information and Management Architecture

The system is based on components [Stuem99]. The components can be divided in three essential layers. The lowest layer is the **device-interface-control layer** that provides a remote interface to the attached hardware systems. This makes the hardware devices available to the whole system. The next layer is the **application-support layer** that combines information from the device interface control components to a higher, technology independent level. The sum of the APIs of these components can be seen as a vehicle API being the main interface for application developers. The top layer is the **application layer** that provides the application logic and allows users to interact with the system.

All components run in an environment (**Component Loader**) that loads or unloads components during runtime. Load balancing, dependencies, security etc. is handled by the core-management. The component loader can be started more than once in a vehicle or in the infrastructure. The result of this is a distributed system for optimising load balancing. By using this sort of runtime environment it is possible to update components during runtime. This is a very important point, especially in vehicles, which live much longer than software. It is also possible to add new components or remove components during runtime. By this the user can use new services every time.

The new component ObjectSpace will be developed to run in COSIMA as a dynamic component. The analysis will show that components in COSIMA have to meet specific requirements, like for example being moveable during runtime. It will also be shown that these requirements need a specified design of the component.

3.2. Motivation

The following scenario explains the problem, which will be solved by this project.

We think about an existing component called `MessageViewer`. This component allows the user to read or write emails of a POP3-account. We imagine that the user gets this component together with his car.

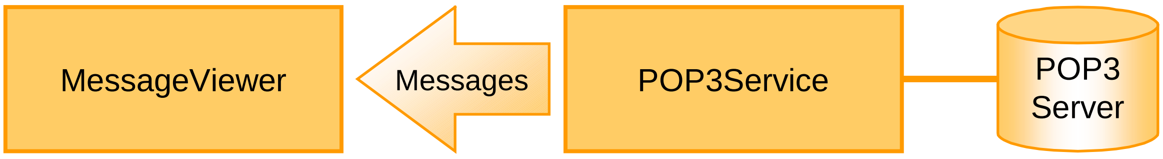

Figure 2 A well known service offers its information

After some time he gets a new component, which can be used to receive and send SMS-Messages, so he loads it in his car. If he then receives a SMS, the `MessageViewer` does not know anything about SMS and cannot present it to the user.

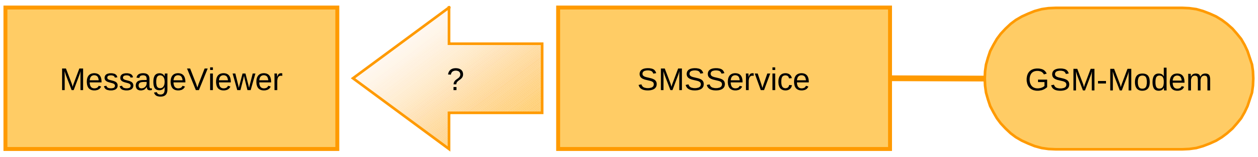

Figure 3 A new service offers its still unknown information

So the problem is to **identify information**, which was not known, when the component was developed. This is an essential problem in all dynamic information systems – to describe information in such an abstract way, that the software can identify it. The idea is to create an additional component, which can be seen as a sort of information-bus. Components publish information with the help of this bus, without knowing which other components are interested in this information.

4. Approach

This chapter describes the process, which is used to develop ObjectSpace. The process is beneficial to the development of dynamic components in a system like COSIMA [Mue00].

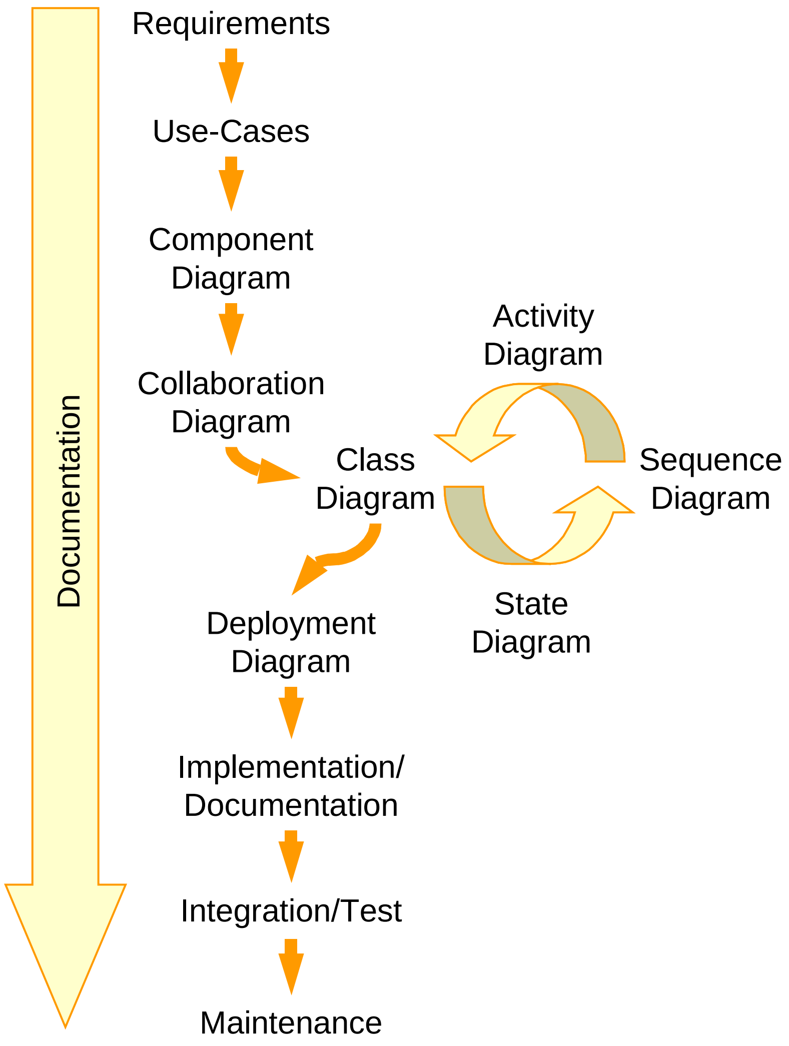

Figure 4 Development-process of components

The development of dynamic components can be divided in three essential steps. The first step is the **analysis**, which contains the establishment of requirements and the analysis of those, which leads to the use-cases. Component and collaboration diagrams can be used to get the borders of the component, the dependencies to other components and the service interface, which contains the functionality in the system. To find the borders of a component also means that this phase will show, whether the component has to be distributed, and by this divided in several parts. Additional requirements, which are specific for this sort of dynamic system inside vehicles, have to be considered. At the end of this phase we will get the architecture of the component and the collaboration with other components in the system.

The next step is an iterative development process called the **design**. During this phase the “inner” of the component is built. By using UML diagrams [Scha99] in an iterative way, the classes and their functionality can be found.

The last step is the **implementation and integration** of the component in the system. The deployment diagram can be used to show how the components have to be distributed in the system. The theory of COSIMA says that every component can run everywhere in the system, yet outside the vehicle, anywhere in the infrastructure. In reality this is not always possible. Some components have to run on specified computers regarding their dependencies on and requirements to hard- and software. This shall be shown in the diagram. The last steps are implementation, integration and test. These phases are the same as in the development of normal applications. The test can be divided in two parts. The first is the white-box test, which is normally made during the implementation. Parts of the software are tested regarding details of the implementation. The black-box test can be made by using the service interface, which is a remote interface. So the component can be tested by “dummy-components”, which use the service interface for known tests, like the load-test.

The documentation is made in parallel to the whole development process. So the development is nearly the same as in normal applications, but we always have to think about the dynamic system. Information cannot simply be transferred to other components. We have to think about communication and load balancing. Apart from that it is important that the component has to be moveable during runtime and scalable to an unlimited number of users.

4.1. Phases to develop the new component

The following chapter describes the phases and their aims.

Requirements
In this phase the requirements will be defined. This will show, what the new component should do.

Analysis
In the first step of this phase, the use-cases will be determined. By the use-cases and the analysis of the dependencies to other software or hardware components the service interface of the new component can be identified. From this the architecture of the component can be built. The architecture is platform and language independent; therefore, the next step is to think about the used technologies. How should the persistence be built? How can language independence be ensured? Should a server database be used? Etc.

Design
The design-phase identifies the fine-grained structure of the component. Classes, interfaces and their methods will be determined.

Implementation
During this phase the component is implemented.

Integration and Test

The component is integrated into the system. A black-box test will show the behaviour during usage. The white-box test is made during the implementation.

Maintenance

Maintenance is a very important part of COSIMA. It is the main reason why COSIMA was developed. Maintenance will not be considered in this project.

Documentation

The documentation is made from the beginning of the project.

5. ObjectSpace

The following chapter describes the development of ObjectSpace. It will show the requirements, analysis, design and test.

5.1. Requirements

The ObjectSpace component has to meet the following requirements.

1. A component should be built for the existing system COSIMA. This means that the design has to achieve specified requirements, determined by COSIMA. The component has to be loadable, updateable and moveable during runtime. This has to be considered in the design of the component.

2. The component should allow the distribution of information in a sort of broadcast or multicast. Information that is published should be available for every component in the system.

3. Information should be defined in a language and platform independent way, so that it can be published and subscribed by components written in any programming language.

4. Information should be identifiable from its description. It should not be classified in fixed categories to preserve the dynamics of the system.

5. The component should be optimised for vehicles. It has to meet specified requirements like security or safety.

6. Failure tolerance should be considered. The component will be an important part of the system. If it fails, it can powerfully influence the functioning of the whole system. This is a very important point within the design of the component.

7. The component should allow storing information persistently. Information for example emails should be stored, so that they are available even after a restart of the system. It is necessary that for this persistence any form of storage can be used for example databases, files or chip-cards. Apart from that, failure tolerance must be considered in this part.

8. The component should inform other components, if the structure of information is changed (publisher-subscriber-pattern).

9. Load balancing should be considered.

10. Scalability should be considered.

5.2. Analysis

In the analysis we will **review the requirements**. This chapter will discuss how the requirements can be met. It will show that different solutions could be used to solve specified problems. The result of the analysis is the architecture of ObjectSpace. The architecture describes the structure of ObjectSpace in a language and platform independent form. Chapter 5.2.2 will show that the requirements can be met by developing a new component.

5.2.1. Requirements determined by COSIMA

COSIMA is an architecture, which is built especially for vehicles. It is a dynamic architecture, which allows components to be updated or moved during runtime. This behaviour brings additional requirements.

1. The component has to be loadable during runtime of the system
2. The component has to be unloadable during runtime of the system
3. The component has to be updateable during runtime
4. The component has to be moveable during runtime
5. Failure-tolerance has to be considered
6. Scalability has to be considered

These requirements are important and have to be paid attention to in the following.

5.2.2. Information publishing

The following diagram shows the aim to publish information in the system. On the right is the information publisher, which offers the system information. On the left, there are information subscribers, which are interested in this information. One essential point is that a particular subscriber is only interested in specified information, for example a component that can present emails to the user is interested only in messages and not in vehicle specific data.

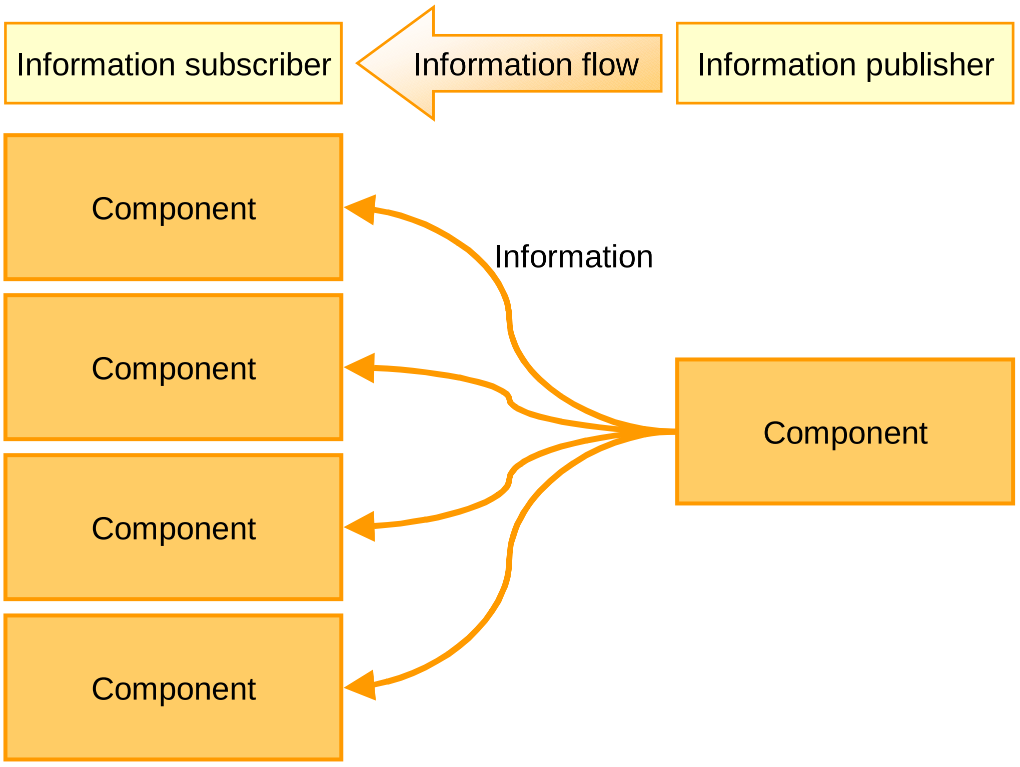

Figure 5 Information publishing

There are different possibilities to distribute information. The first possibility is that components, which are interested in specified information, get this directly from the information publisher. This can be done by polling or notifying the subscriber, when the information is changed in the publisher. The subscriber has to look for available publishers by using a specified service-lookup. This approach creates various problems. If several publishers with different information are loaded in the system the result is a highly networked system that is not scalable.

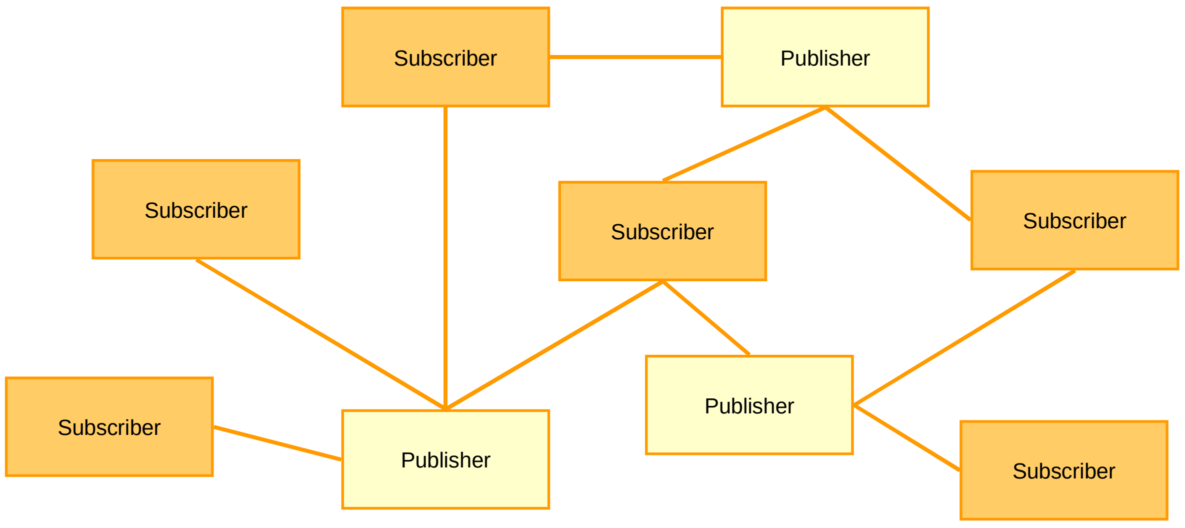

Figure 6 A highly networked system

The second problem will appear, if new publishers are loaded into the system. Older subscribers do not know anything about them and cannot understand the offered information, although it is possibly nearly the same as that of an older component. This problem is discussed in chapter 5.2.3.

Another possibility is to distribute information by multicast or broadcast mechanisms. In this sort of publishing every component gets all the information. This means a high amount of network traffic and calculating time to identify the information.

A third possibility is to build a sort of marketplace. Publishers offer their information in a central location, where consumers (subscribers) can get it. To avoid network traffic by polling algorithms, notification is used instead. This means that publishers send their new or changed information to the marketplace. Then the marketplace informs those subscribers, which are interested in this specified information. To integrate this mechanism, an additional component (ObjectSpace) has to be developed.

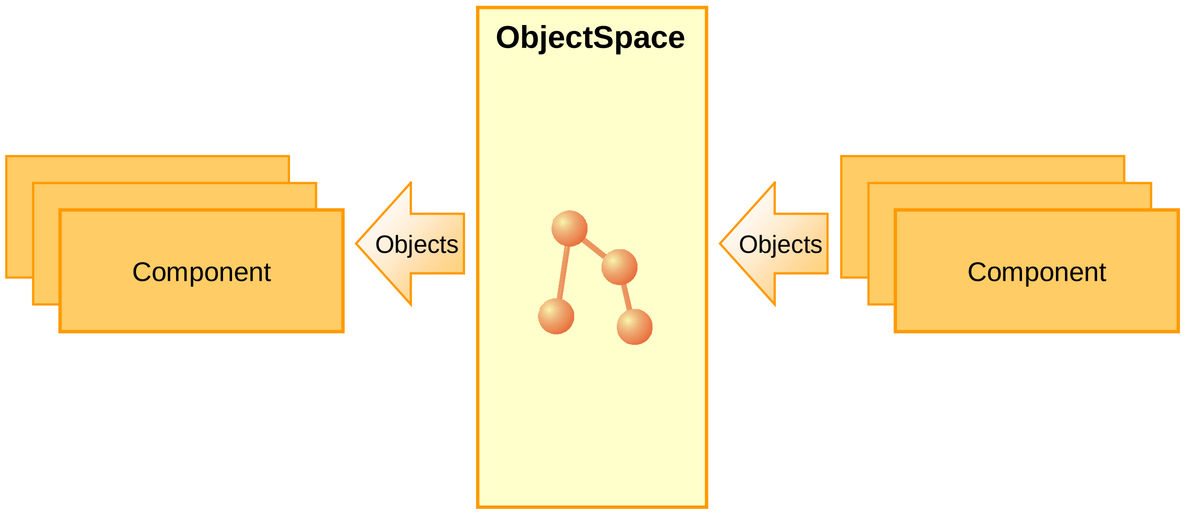

Figure 7 The ObjectSpace as a sort of market place

5.2.3. Identifying information

One possibility can be to identify information on account of its content. So for example, if specified words can be found in the information, we can suspect which sort of information we have. This approach could lead to failures because not all information is uniquely identifiable by its content. E.g. a mail contains a location. Is it the current location of the vehicle, the location where the driver should go or only part of an address? A better approach is to identify the information by using a sort of abstraction. So for example, if we get a SMS we don't have to know what SMS means. It is sufficient to know that it is a sort of message, which contains a sender, a recipient and the message itself. It could include additional information, but this is only interesting for someone, who really knows what SMS means. The object-oriented approach reaches this by specialisation respectively generalisation. The following diagram shows a hierarchy of information objects.

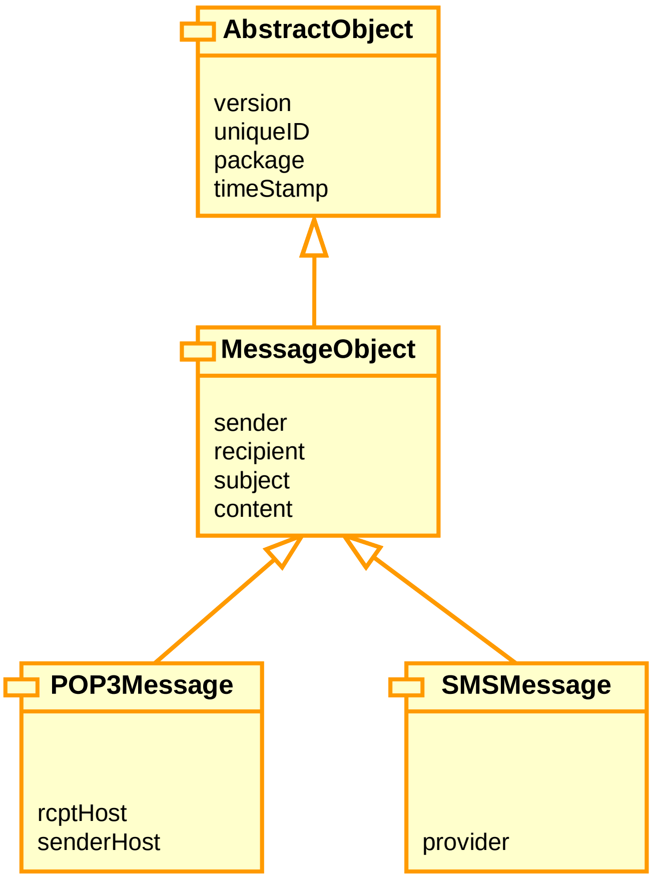

Figure 8 An object-oriented approach to identify information objects

The root object is `AbstractObject`, which contains information, needed to organise the objects. `SMSMessage` is derived from `MessageObject`. So it contains all information, needed by a message. The additional field provider is only relevant for special components or the administration. By this approach, a component, which is interested in messages, can also use `POP3Message`, `SMSMessage` or newer mail services currently not known. The only problem is that all information objects have to be well documented so that newer objects are derived from existing objects if possible. Chapter 5.3.4 shows how information objects are defined.

5.2.4. Use-Cases

The following diagram shows the use-cases of ObjectSpace.

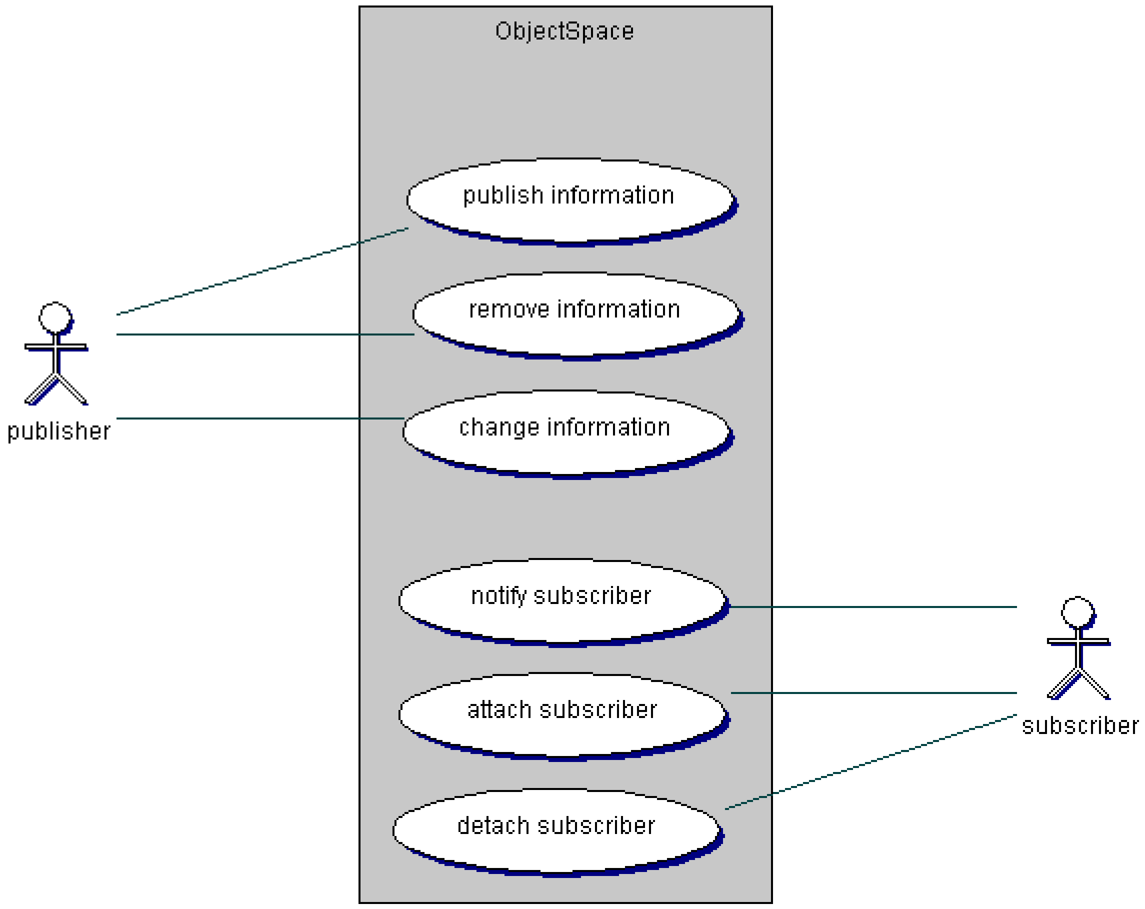

Figure 9 Use-Cases of ObjectSpace

Publish information
This is the essential use-cases of ObjectSpace. Information passed by other components should be published within the system. As shown above, the publishing is done by notifying all attached subscribers, which are interested in this particular bit of information.

Remove information
Still published information can be removed from ObjectSpace. It is necessary that all components have the possibility to remove information objects and not only the component, which published the information. The user removes for example an email. Therefore the email will be removed from ObjectSpace by the component, which presents it to the user. When an information object is removed, all attached subscribers, which are interested in this information, are notified.

Change information
Information can be changed by all components. Subscribers, which are interested in this bit of information, are notified.

Notify subscriber
The notification is used to inform attached subscribers that the information structure is changed.

Attach subscriber
Components, which are interested in specified information, have to attach themselves to ObjectSpace. Components that are interested in several information types can attach themselves more than once.

Detach subscriber
Attached subscribers can be removed.

5.2.5. The persistence

Information, which is "thrown" into ObjectSpace, has to be stored, so that it is still available after the system is restarted. Of course not all the information has to be stored. ObjectSpace can also be seen as a sort of bus-system, which allows **asynchronous message passing**, but this is not the main aim of ObjectSpace. To use this component for message passing means to build a bottleneck in the inter-component communication. ObjectSpace is built for distributing abstract defined information. Different points have to be considered when thinking about the persistence.

- The component has to be moveable. So how can the persistence be accessed if ObjectSpace uses a file on the computer on which it runs?
- Different types of storage-media should be usable.
- Scalability
- Failure-tolerance

The component should be moveable. The persistence cannot always be moved together with ObjectSpace because of its hard- or software dependency. So the persistence has to be accessible via remote. By this the component can be moved and reconnect to the persistence without the loss of data.

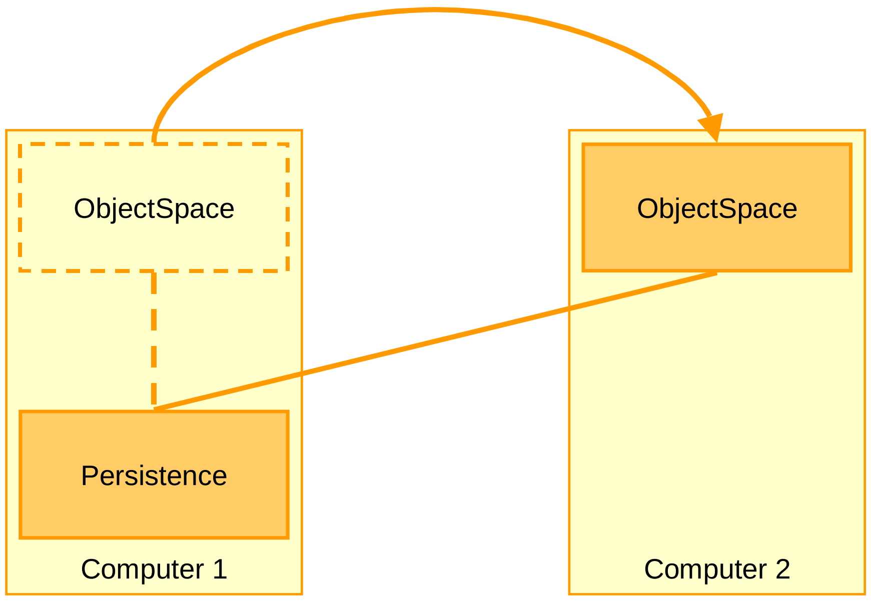

Figure 10 ObjectSpace is moved to another computer

The usage of different types of storage-media can be achieved by defining an interface for the persistence. ObjectSpace can only read or write information by using this interface. Hence it is transparent, which media is currently used. As seen before, the persistence has to be accessed via remote to guarantee the relocatability of ObjectSpace. This means it contains a remote interface that can be the same for every sort of persistence.

Scalability is more or less a problem of the data organisation in the persistence. For example if a relational database is used, the speed of accessing information can be increased by a well-defined selection of indexes. If a high amount of data is expected, the choice of persistence is very important. It makes no sense to select a media-card with 32 Megabytes storage-space, when many users with a high amount of emails use the system. This means the persistence builds a sort of interface to the real storage, with more or less functionality, depending on the media itself. If a server database is used, the functionality of the persistence contains only the translation of component requests to SQL statements for accessing the database. Hence the management of data is an important point.

Failure-tolerance can be achieved by using graceful degradation i.e. if the functionality of the persistence fails; one can switch to a lower level that guarantees that no data is lost, but that may only be accessible with lower performance or less functionality. This is called **multilevel persistence** in this dissertation.

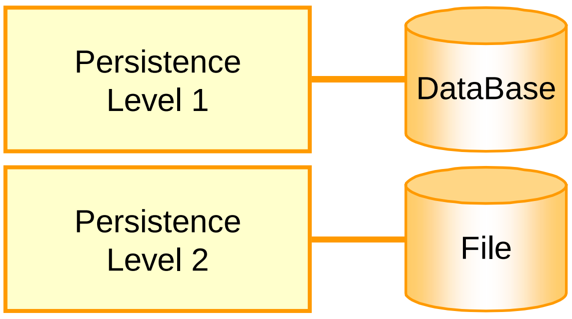

Figure 11 Multilevel persistence

ObjectSpace is implemented with two levels of persistence. The first level gives access to a server database (for the prototype of ObjectSpace MySQL server database is used). The second level encapsulates access to a file. Both levels implement the same interface, so that it is not visible for ObjectSpace, how the information is stored. If the first level fails, for example because the database server is switched off, the persistence is automatically switched over to the second level. When this happens a new job is created, which tries to access the first level. If it is available, the data of the second level will be synchronised and then the persistence will be switched back to the first level. While the second level is used, the data of the first level is not available, so it only guarantees that new data will not be lost.

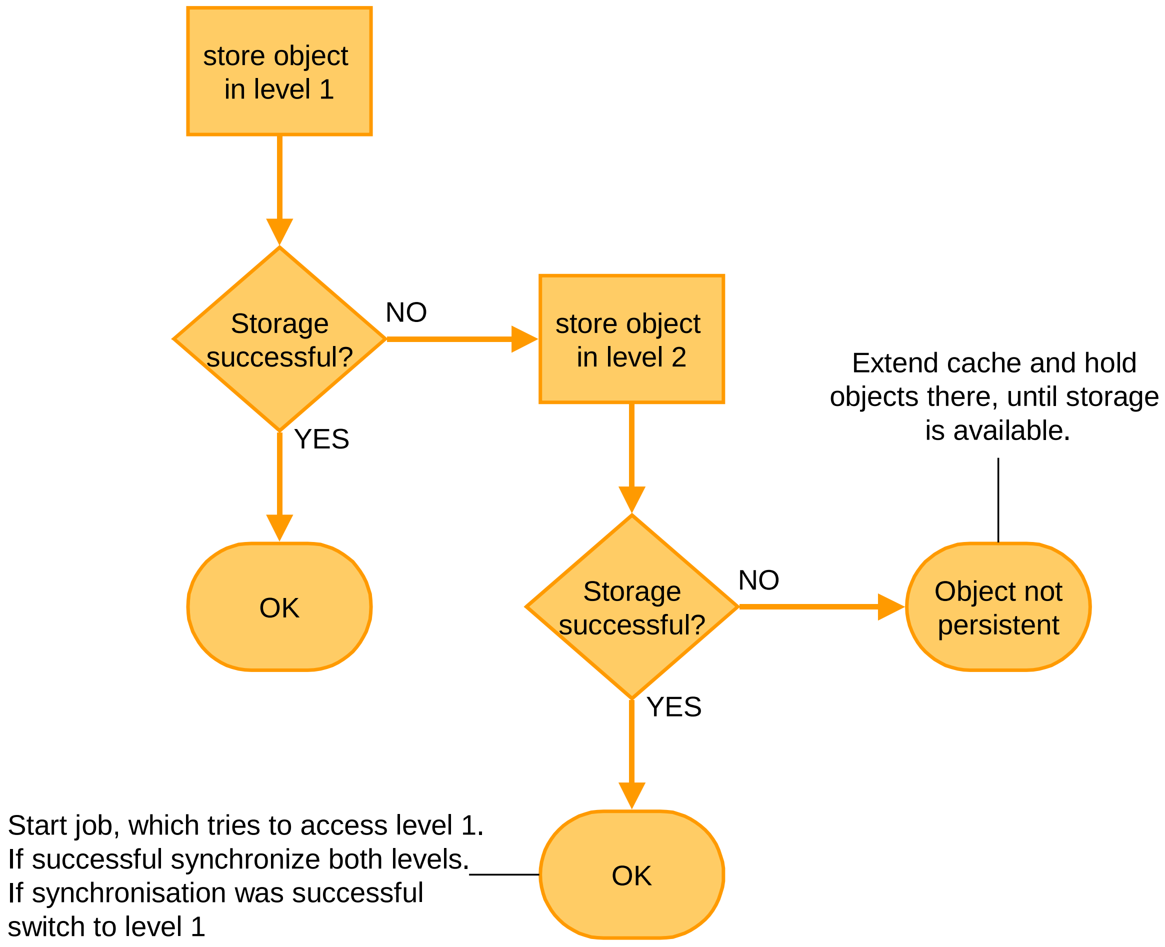
store object in level 1
Storage successful?
NO
YES
OK
store object in level 2
Extend cache and hold objects there, until storage is available.
Storage successful?
NO
YES
Object not persistent
OK
Start job, which tries to access level 1. If successful synchronize both levels. If synchronisation was successful switch to level 1

Figure 12 Flowchart of multilevel persistence

The persistence is encapsulated in a repository (see Figure 18 The architecture of ObjectSpace), which also contains a cache to increase performance. The repository manages the information objects. The cache can be set to a fixed maximum size (for example 100 objects or 2 megabytes). When the second level also fails, the size of the cache will be increased, so that newer data will not be lost. Of course this will only store data temporarily and is not a sort of persistence. The administrator of the system has to be informed, when the first level fails. For a better failure-tolerance, the two levels should run on different computers, so that even hardware failures can be handled.

5.2.6. Scalability and failure-tolerance by redundancy

Scalability is an important term. Although ObjectSpace will run in vehicles with a finite number of passengers, we also have to think about vehicles with a large number of users like airplanes or trains. In addition, COSIMA is not bound to the borders of one vehicle. To think about scalability means to have a look at the parts of ObjectSpace, which are influenced by this problem. The following diagram shows a possible solution.

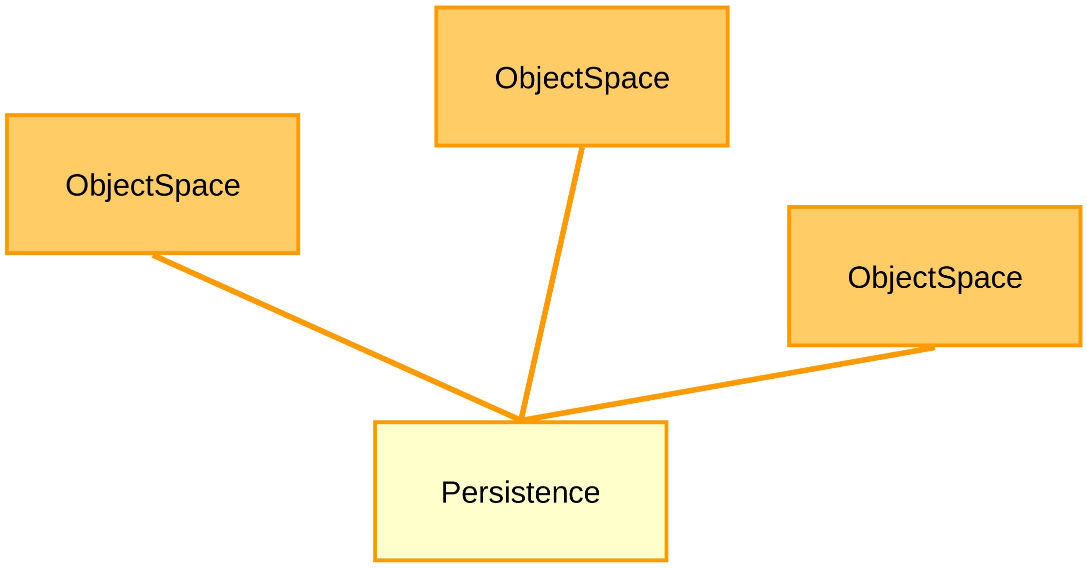

Figure 13 Distribution of ObjectSpace to increase scalability and failure-tolerance

ObjectSpace is instantiated more than once in the system. Subscribers and publisher prefer to use the ObjectSpace which is local to them, or which should be used on account of load balancing. The following discussion will show, whether this solution is practical. The following points have to be considered:

- Information passing
- Repository
- Filter algorithm
- Notification

During the information passing the publisher sends an information object to ObjectSpace. This means that it runs over the network. It is not possible to avoid this. In addition, the object has to be sent to the persistence and all other instances of ObjectSpace over the network to inform the attached subscribers. The `Repository` has to be central and has the same functionality as with one instance of ObjectSpace. To handle the high amount of information objects, the persistence has to be selected regarding the management of information objects. The main CPU-time is used by the filter, which verifies the information object. By creating multiple instances of ObjectSpace, the calculation is distributed over different computers. To notify all subscribers, the information objects have to be passed to all instances. Although the object is sent over the network to every instance, only those subscribers, which are interested in this information, are informed. The benefit of this is shown in the following example:

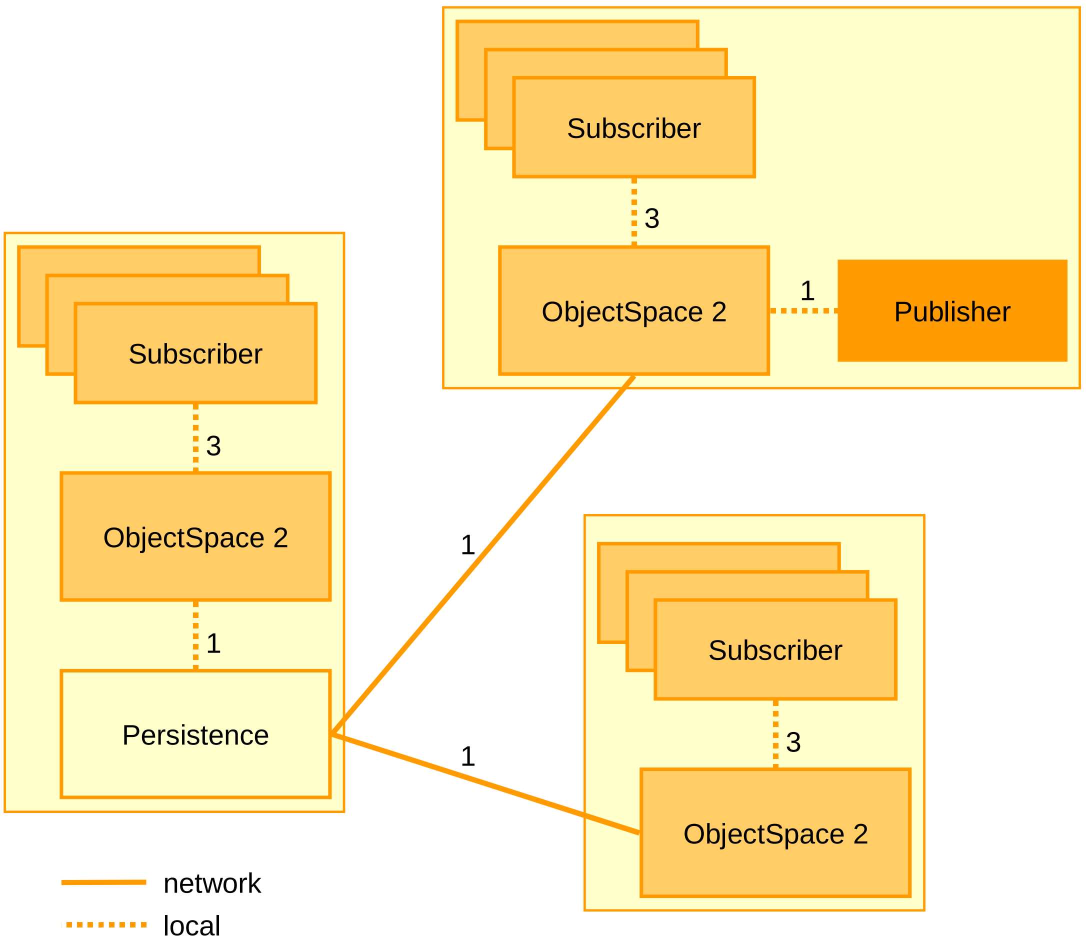

Figure 14 Information passing with multiple instances of ObjectSpace

The publisher transfers its information to the ObjectSpace local to it. The information object is sent to the persistence, from where it is passed to all instances of ObjectSpace. All subscribers are notified over the local connection. Therefore the information object is only sent two times over the network. If all subscribers are attached to one central instance of ObjectSpace the object has to be passed seven times over the network.

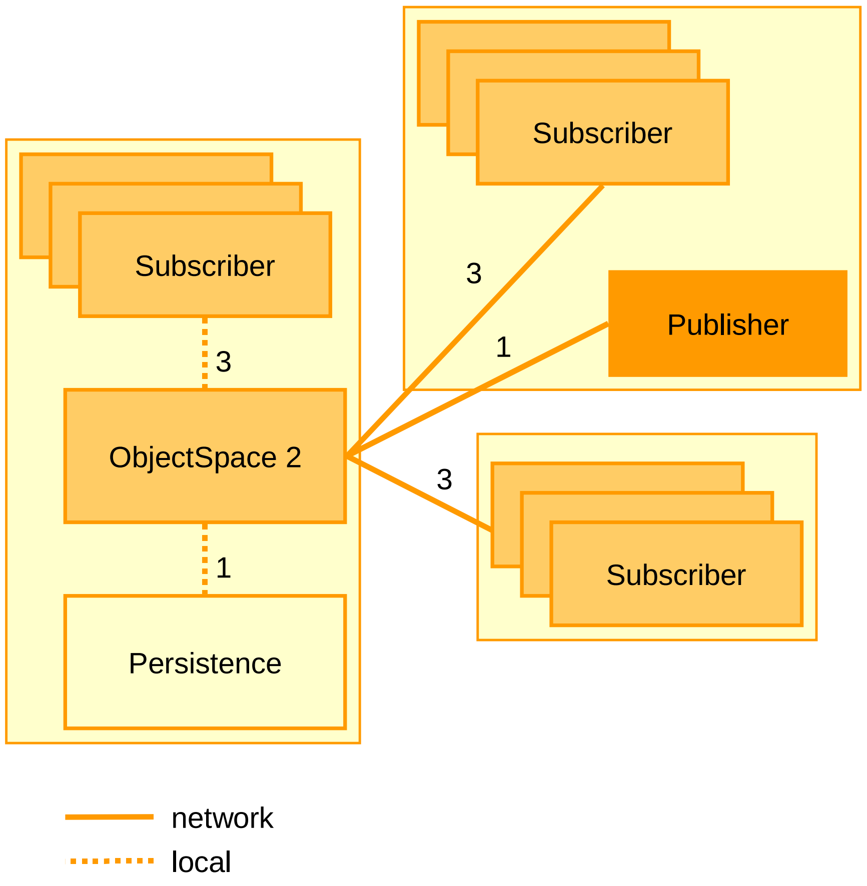

Figure 15 Information passing with one instance of ObjectSpace

However there are also disadvantages when starting multiple instances of ObjectSpace. If the hardware-architecture contains one server and several very small client computers, it could be more useful to start only one instance on account of the calculation time needed by the filtering mechanism. Therefore the hardware architecture is an important point to be considered by the load balancing and consequently the distribution of ObjectSpace.

Starting ObjectSpace more than once is not only a benefit for load balancing. It will also increase failure-tolerance. By an additional mechanism it would be possible to synchronise all instances. Only the `ListenerInfo` objects have to be synchronised because the information objects are available via one central access. So if a new subscriber is attached to ObjectSpace, its information is passed to all instances. Of course it is notified by the local instance. If one instance fails one of the other instances can take the job over. For this all instances have to be connected in a way, so that they can inform each other if a new subscriber is attached and to see, whether one instance is answering no longer. This is a typical solution for failure-tolerance by redundancy.

Another way to improve failure-tolerance could be to build ObjectSpace in a master-slave mode. Therefore two instances of ObjectSpace are started. One builds the master the other the slave. The master is the currently used ObjectSpace. If it fails, the slave is set to the master and a new ObjectSpace is instantiated on another computer to build the new slave. Like in the previous approach, the instances have to be connected for synchronisation and monitoring. This can be done in COSIMA, because the components can demand the loading of a component on a specified host by the Configuration Management.

5.2.7. Load-Balancing

In simple publisher-subscriber architectures, all information is passed to the subscriber over the network, on which it is verified. By this even uninteresting information be transferred. By installing filter directly in the server, only information objects, which really contain interesting data are passed to the subscriber. This is an optimisation of load balancing regarding network traffic. As shown in chapter 5.2.6, the creation of multiple instances of ObjectSpace on several computers could also be used to optimise load balancing. This can distribute the filter algorithms over several CPUs. In addition, this approach can reduce network traffic during the notification of subscribers.

In addition, the location of the persistence is an important point. The location mainly depends on the hardware. The demand on working and persistent storage defines where the persistent has to be started.

5.2.8. Attaching subscribers to ObjectSpace

The following collaboration diagram shows the dependency and interaction of ObjectSpace with subscriber and publisher. The collaboration diagram shows the messages from the point of view of components.

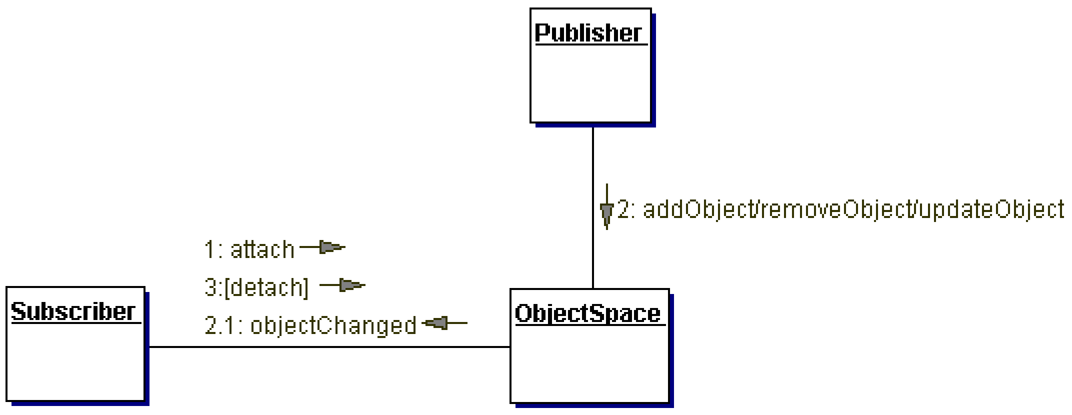

Figure 16   Messages between Publisher, Subscriber and ObjectSpace

Subscribers have to register (attach) themselves. During the registration they pass their own reference and a filter, which contains two essential parts: first the type of information objects the subscriber is interested in and second an operation, which verifies the content of information objects. Filters will be explained in chapter 5.3.1.1.

5.2.9. The distribution of information via ObjectSpace

The following diagram shows the flow of information from the publisher to the subscriber. All parts shown in this diagram are explained in detail in chapter 5.3.

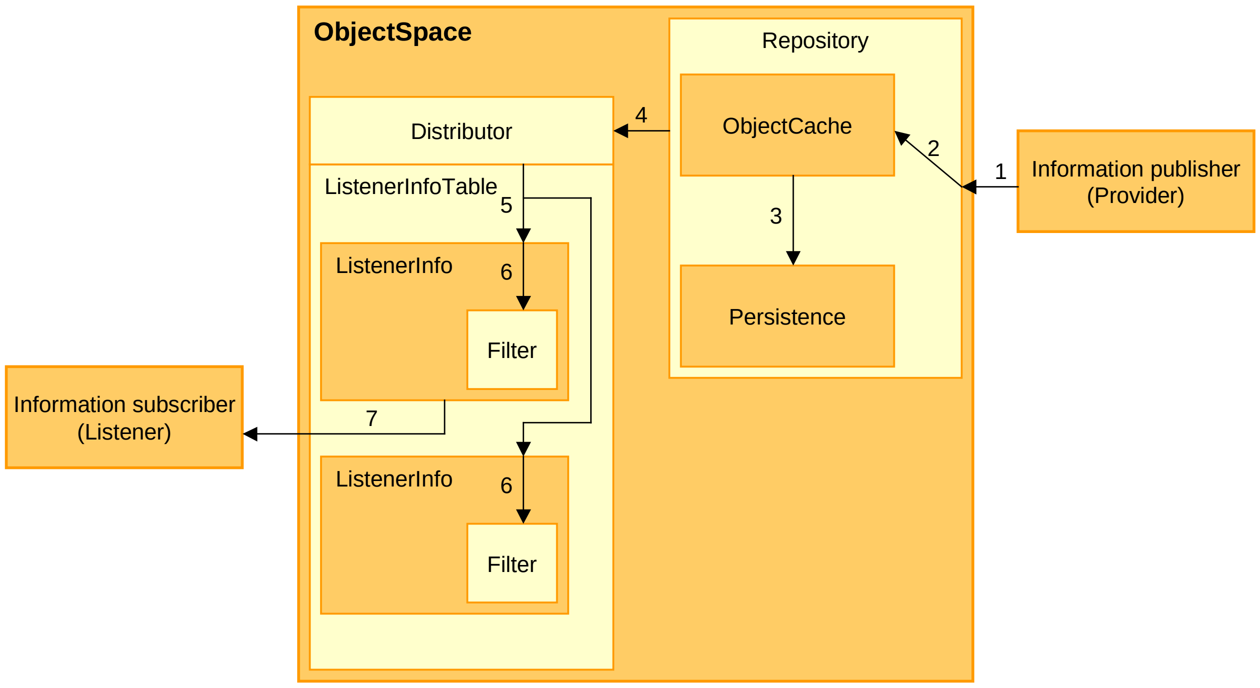

Figure 17 The flow of information objects

1. In the first step the publisher sends its information to the ObjectSpace. This service offered by ObjectSpace can be used via remote, so the publisher does not have to run on the same computer as the ObjectSpace. The information is passed directly to the Repository.

2. In the Repository the information is passed to the cache. The cache should increase the performance and decrease the need of memory. It is organised write-through, so that new information is automatically stored in the persistence.

3. The information is sent to the persistence, where it is stored permanently.

4. After the information is stored in the Repository, it is then passed to the Distributor, which is part of the `ListenerInfoTable`. The `ListenerInfoTable` contains `ListenerInfo` objects. These objects contain information about the registered subscribers. Subscribers are also called listeners, based on the event handling integrated in Java [Goll99]. There is one `ListenerInfo` object for every subscriber. The subscribers, which are interested in specified information, have to register themselves before. During the registration they pass their own reference to enable notification, a filter object which contains an algorithm to identify the information and the type of information the subscriber is interested in. This information is stored in a new instance of `ListenerInfo`.

5. The information is passed to the `ListenerInfo` objects. Only those `ListenerInfo` objects are notified, which are interested in this type of information.

6. The information is sent to the filter that is installed by the Subscriber to verify the content.

7. In the example above only the first filter returns true, which means that the information contains the needed values. The subscriber is notified and the information is passed to it. The reference, which the subscriber passed to ObjectSpace during registration, is a remote reference, which means that the subscribers don’t have to run on the same machine as the ObjectSpace.

5.2.10. Notification

The notification is based on the event handling implemented in Java [Goll99]. This means if an information object is changed or a new object is added to ObjectSpace the subscriber is automatically notified. For this the subscriber (listener) has to attach itself to the ObjectSpace and pass its reference for the callback. The notification can be made in two ways. The first is the classical way, defined by the publisher-subscriber pattern. The subscriber is only notified about the change and can then fetch the changed object from ObjectSpace. At the second way, the object is transferred to the subscriber together with the notification. This sort of notification reduces network traffic. The first way brings a benefit under certain circumstances. If for example RMI is used for the communication, one will get a high-coupled architecture. Both, the subscriber and ObjectSpace, are client and server, which means that they both need the remote interface from each other. ObjectSpace is built language independent. So it is not important to uncouple the components. Hence the second way is used.

5.2.11. Language independence

Language independence means that the interface of ObjectSpace should be usable by components that are implemented in any programming language. Therefore the interface cannot contain the transfer of language dependent objects, for example instantiated Java classes, which are then passed to a component written in C++. Hence all transferred data must be definable in a format that can be handled in any programming language. For complex data structures, which are normally held in an object, XML (Extended Markup Language) can be used to define them. All other data can be transferred as atoms (e.g. Integer, Byte array etc.).

From the analysis we get an architecture, which will be described in the next chapter.

5.3. The architecture of ObjectSpace

This chapter shows the architecture of ObjectSpace. The architecture is based on the publisher-subscriber-pattern (observer-pattern). Components that want to publish information do this by ‘throwing’ information-objects into the ObjectSpace. These components are called publishers. The ObjectSpace will then inform all components (subscribers), which are interested in this specified information. For many information-objects it could be necessary to store them for a longer time. For example ‘email’ could be information, which should also be available after the system is restarted. So persistence, preferably a server-database is used to store these information-objects.

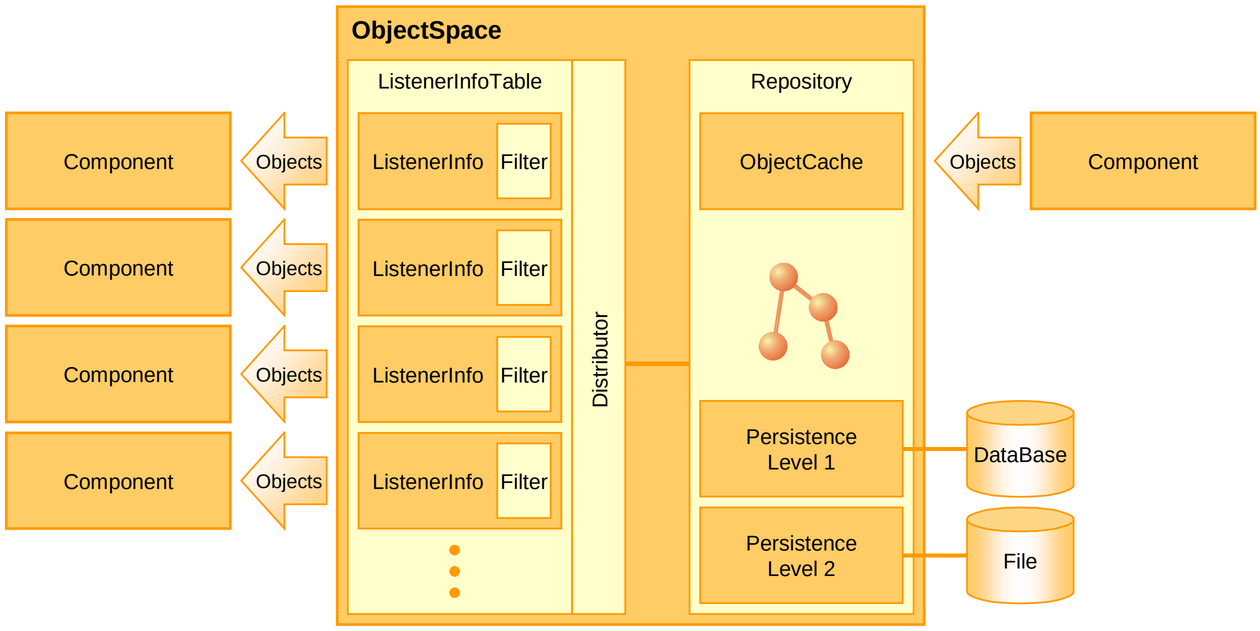

Figure 18 The architecture of ObjectSpace

Although the persistence is drawn as part of ObjectSpace, it is an additional component, which is controlled and accessed via remote. It is encapsulated in the repository and by this transparent for other parts of ObjectSpace.

5.3.1. ObjectSpace

The following chapters will explain the elements of ObjectSpace.

5.3.1.1. ListenerInfoTable

This table manages all `ListenerInfo` objects. A `ListenerInfo` object is created when a new Listener (subscriber) is attached to ObjectSpace.

ListenerInfo

A `ListenerInfo` object contains all information, passed from the subscriber during registration. Therefore it contains the remote reference, the protocol used to notify the subscriber[1] and the installed filter. The `ListenerInfoTable` informs all `ListenerInfo` objects by passing the changed or new information object. The `ListenerInfo` object will then use the filter to verify whether its subscriber can use the information object. If the filter-algorithm returns true, the attached subscriber is informed. So the `ListenerInfo` object can be seen as a sort of proxy, representative for the subscriber.

Filter

As shown above, filters are installed by the subscribers to verify whether an information object is interesting. A filter contains the type of information object and an algorithm, which allows verifying the information object's content. For example a subscriber is interested in emails, which contain n appointment. This means the filter contains the object type (e.g. `cosima.common.Email`) and the algorithm which will examine whether the information object contains the tag `<appointment>`. Filters are used to optimise load balancing by reducing network traffic. ObjectSpace can be implemented in any programming language. So filters cannot be defined in normal objects. They have to be defined in a platform and language independent manner. XML (Extended Markup Language) [Beh00] was selected to declare such filters. In the following this declaration will be explained. DTD (Document Type Definition) is used to define, how a filter has to be written in XML. It can be seen as the class declaration of an XML object. The following DTD shows that a Filter can contain boolean operators (AND, OR) to combine operations and the operations themselves. Every operation is a equation containing number or string operators.

```
<!ELEMENT Filter (AND? | OR? | Operation?)?>
<!ATTLIST Filter
     ObjectName CDATA #REQUIRED
>
<!ELEMENT AND (AND*, OR*, Operation*)*>
<!ELEMENT OR (AND*, OR*, Operation*)*>
<!ELEMENT Operation EMPTY>
<!ATTLIST Operation
     DocumentValue CDATA #REQUIRED
     Operator (     CONTAINS | STARTSWITH | ENDSWITH |
                    EQUALSTRING | NOTEQUALSTRING | LESSTHAN |
                    GREATERTHAN | LESSOREQUAL |
                    GREATEROREQUAL | EQUAL | NOTEQUAL)
                    #REQUIRED
     CompareValue CDATA #REQUIRED>
```

The following example shows a filter declared with XML.

```
<Filter ObjectName="cosima.common.Climate">
     <OR>
```

[1] The protocol is needed by the ListenerInfo object to notify its subscriber. ObjectSpace can use different protocols for inter-component communication. So the remote reference could be for example a RMI remote reference, a CORBA reference, or simply an IP-address and the port for building a connection.

```
         <AND>
             <Operation  DocumentValue="Temperature"
                         Operator="GREATERTHAN"
                         CompareValue="20"/>
             <Operation  DocumentValue="Temperature"
                         Operator="LESSTHAN"
                         CompareValue="30"/>
         </AND>
         <Operation   DocumentValue="Temperature"
                      Operator="EQUAL"
                      CompareValue="0"/>
     </OR>
</Filter>
```

This filter defines that the subscriber is interested in information object of the type `Climate` in the package `cosima.common`. The subscriber will be informed if the field Temperature is greater than 20 and less than 30 or equal to 0. How this filter works will be shown by the next pseudo-code example:

```
t = cosima.common.Climate.Temperature;

If ((t > 20 && t < 30) || t == 0)
     return true;
else
     return false;
```

As a result the subscriber is only informed if the information object is of the right type (like shown in chapter 5.2.3 derived objects are also accepted) and contains the required values. More details about the creation of filter objects are shown in chapter 7.2.

5.3.1.2. Repository

The repository contains two essential parts: the cache and the persistence. It manages the storage of information objects.

Cache
The cache is used to increase performance by holding a configurable number of objects in memory for faster access. It is organised write-through, so that new objects are automatically passed to the persistence before they are sent to the subscribers. The cache can be configured in two ways. The first is to determine the number of objects. This is useful for the performance, but eventually not for optimising the usage of memory. Hence a second way can be selected, where the maximum size of memory usage can be determined. The decision which way is used depends on the kind of information objects (size, amount etc.) and the used hardware. The cache replaces objects, which are not used for a longer time by recently used objects. This approach is called least-recently-used. Therefore the objects have to get a timestamp for every request, which is done by the cache itself. When an information object is requested, the repository at first passes the request to the cache. If this fails the object is fetched from the persistence.

Persistence
As shown in chapter 5.2.5, the persistence is built as a multilevel storage. It was also shown that the persistence has to be a standalone component, so that ObjectSpace reaches the requirement to be moveable. Hence it also allows the ObjectSpace to be instantiated more than in the system to consider load balancing and scalability. The ObjectSpace needs an additional configuration from which it can get the location (address) of the persistence. ObjectSpace has to register itself at the persistence, so that it is possible to synchronise all instances of ObjectSpace.

5.3.2. Publisher (Information Service Provider)

Publishers, which offer information to the system, are passing their information in form of objects to ObjectSpace. So a publisher creates an information object by using the predefined XML structure. The object is then transferred to ObjectSpace, where it is stored and sent to the subscribers. The publisher has no knowledge about the subscribers. How a publisher can be implemented is shown in chapter 0.

5.3.3. Subscriber (Information Consumer)

Subscribers are interested in specified information. Therefore they have to register themselves at the ObjectSpace and pass additional information like shown in previous chapters. This concludes a filter, which has to be created by the subscriber in the predefined XML form. The subscriber is automatically notified if interesting information objects are added or changed. The notification is based on the event handling of Java. How a subscriber has to be implemented is shown in chapter 7.2.

5.3.4. Information objects

As shown in chapter 5.3.1.1, objects, which are passed to ObjectSpace, have to be platform and language independent. Therefore the information objects are defined in XML. The following DTD contains the definition of an information object. Every object contains attributes like name, package and version. The root object (`AbstractObject`) contains an additional attribute `UniqueID` that is used to manage the object in ObjectSpace. It can be seen as the primary key. Three essential types of data can be attached to every object: numbers, strings and binary data. This data can be seen as the fields of the object. More details about how an information object can be defined is shown in chapter 0.

```
<!ELEMENT RootObject
     (DerivedObject? | Number* | String* | Data*)*>
<!ATTLIST RootObject
     Name CDATA #FIXED "AbstractObject"
     UniqueID CDATA #REQUIRED
     Version CDATA #REQUIRED
     Package CDATA #FIXED "cosima.common.objectspace"
>
<!ELEMENT DerivedObject
```

```
     (DerivedObject? | Number* | String* | Data*)*>
<!ATTLIST DerivedObject
     Name CDATA #REQUIRED
     Version CDATA #REQUIRED
     Package CDATA #REQUIRED
>
<!ELEMENT Number (#PCDATA)>
<!ATTLIST Number
     Name CDATA #REQUIRED
>
<!ELEMENT String (#PCDATA)>
<!ATTLIST String
     Name CDATA #REQUIRED
>
<!ELEMENT Data (#PCDATA)>
<!ATTLIST Data
     Name CDATA #REQUIRED
>
```

The following example shows a message object, defined in XML. The XML definition can be seen as the instance of an information object.

```
<RootObject UniqueID="f01a345b386e9dc0" Version="1.0.0">
     <DerivedObject      Name="Message"
                         Version="1.0.0"
                         Package="cosima.common.message">
          <String Name="Sender">sender@test.com</String>
          <String Name="Recipient">rcpt@test.com</String>
          <Data Name="Subject">
               <![CDATA[This is the subject]]></Data>
          <Data Name="Content">
               <![CDATA[This is the message]]></Data>
     </DerivedObject>
</RootObject>
```

To build a real object oriented hierarchy [Boo94], the data (fields) of the object have to be parsed bottom-up. If a field is requested that is defined more than once, the field of the lowest child is given back. By this it is possible to override fields in derived objects. With ObjectSpace additional classes are developed, which allows the user of ObjectSpace the creation and usage of information objects in an easy manner.

5.3.5. Synchronisation

Synchronisation during the information transfer between publisher and subscriber can be created in two manners. The publishing process could be synchronous. By this the publisher transfers information to ObjectSpace and waits until all subscribers are notified. The other way is to transfer the data to ObjectSpace and return immediately. This is called asynchronous. Both approaches are possible. The synchronous approach could be a better solution if ObjectSpace is used as a sort of bus-system, which allows multicast-method calls. This blocks the caller until the method is called in all subscribers. So the information objects are used as a sort of method parameter. This is not the main goal of ObjectSpace. It is used for publishing abstract described information as a sort of market place. The idea is that a publisher is not interested in whether or how many subscribers want to get its data. So the asynchronous way is preferred.

The publisher transfers its information to ObjectSpace and returns immediately. The information is passed through cache and repository to the attached subscribers. It is also possible that subscribers are notified, which attach themselves after the information is published. While the information is passed through ObjectSpace, other publishers can transfer their information. This means that ObjectSpace can be used in parallel. This problem is discussed in the next chapter.

5.3.6. Mutual Exclusion

There are two essential areas where mutual exclusion [Coul99] has to be considered in the design of ObjectSpace. The `ListenerInfoTable` in which attached subscribers are stored, is used from different processes at the same time. While attaching or detaching subscribers the table is used to get the references from attached subscribers for the notification. The second area is the transfer of information. Several publishers could pass their information objects at the same time. As a result the `ListenerInfoTable`, the cache and the repository have to be designed according to this problem. The solution is mutual exclusion, which means that critical regions are protected and will not be used by several processes in parallel. Using semaphores can do this. As shown above, Java will be used to implement ObjectSpace. So there are two essential possibilities to solve the problem. The first is to use a monitor to protect the methods. This can be done by using the keyword `synchronized` with every critical method. Another possibility is to protect the tables themselves. The collection-classes of Java offer the possibility to create collection-objects with the helper-class `Collections`. [Mue01] This class contains methods to create a collection-class, where all methods are `synchronized`. This is an easy way, but it also means that multiple readers cannot use the table at the same time. Another point is that this way can easily cause deadlocks.

5.4. Design

This chapter shows the design of ObjectSpace. It describes classes and interfaces. All diagrams are based on UML 1.3.

5.4.1. AbstractObject

The design of `AbstractObject` is based on DOM (Document Object Model). `AbstractObject` is only defined by an interface that declares which functionality has to be offered by the implementation. As shown in the analysis, information objects are language and platform independently defined using XML: So the following classes form a sort of Helper, which is used to create and change information objects

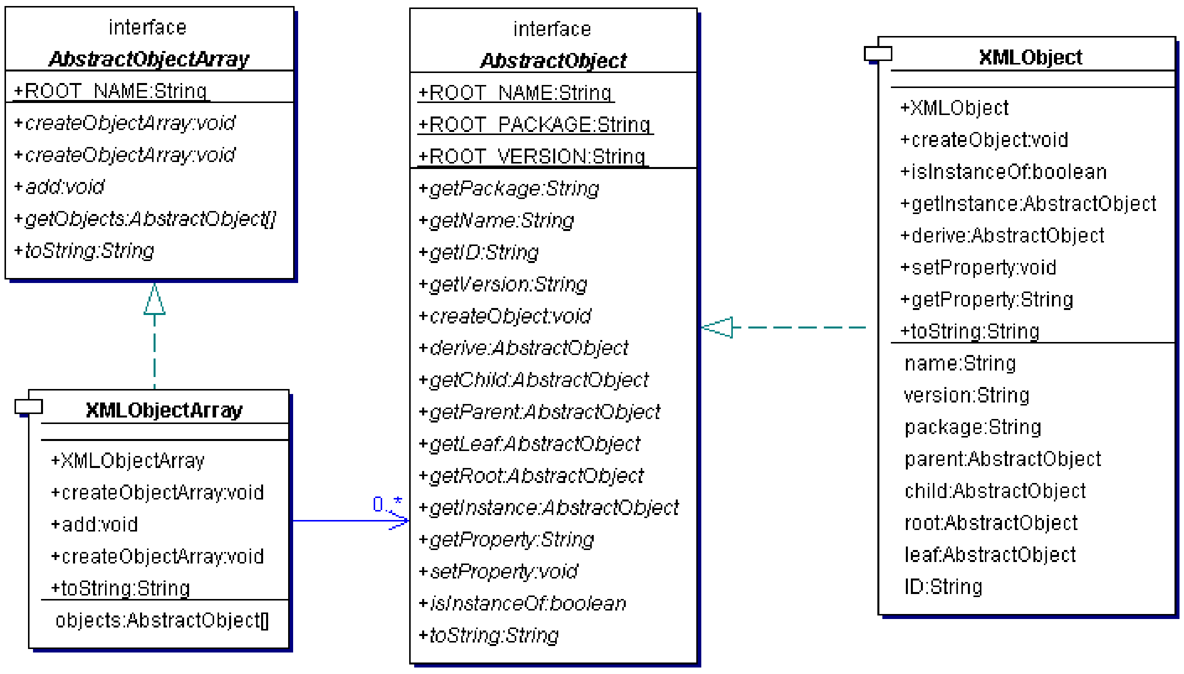

Figure 19 AbstractObject

`AbstractObjectArray` can be used to form arrays of information objects. It represents an XML string, in which all the information objects are listed.

`XMLObject` and `XMLObjectArray` constitute the implemented objects. Interfaces are Java specific and are not defined in C++. In other languages, these interfaces can be created by declaring fully abstract classes. As ObjectSpace only uses strings to manage information objects, the classes above are not implicitly necessary. It is also possible to create the XML string directly and pass it to ObjectSpace. The following diagram shows, how the message object of chapter 5.3.4 can be created.

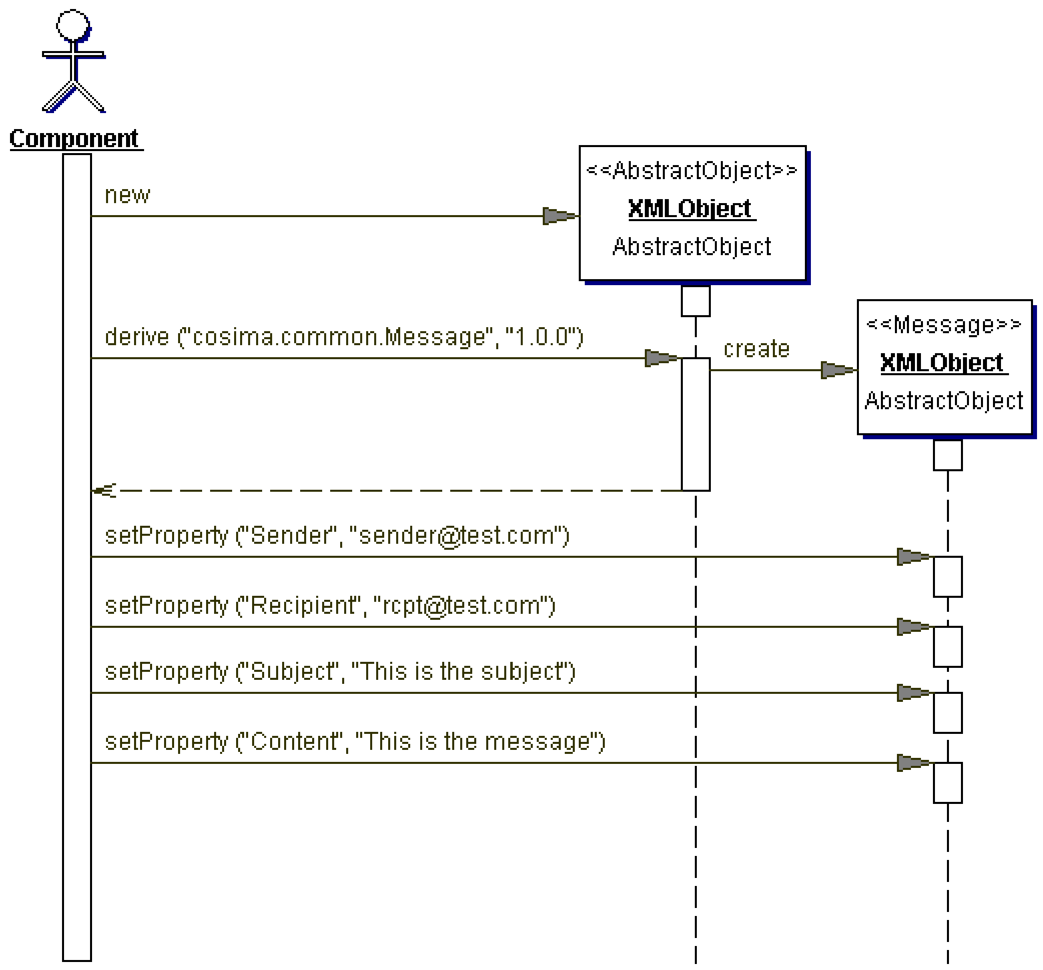
Component
<<AbstractObject>>
XMLObject
AbstractObject
<<Message>>
XMLObject
AbstractObject
new
derive ("cosima.common.Message", "1.0.0")
create
setProperty ("Sender", "sender@test.com")
setProperty ("Recipient", "rcpt@test.com")
setProperty ("Subject", "This is the subject")
setProperty ("Content", "This is the message")

Figure 20 Creation of a message object

If the method `derive` is called, a new object (Message) is created, which is derived from `AbstractObject`. The method `setProperty` is used to change the properties (field values) of the object.
ObjectSpace is implemented platform and language independently. So it is not possible to pass objects of different languages between the components. Hence it is necessary to transfer atoms (Integer, Byte, Byte-arrays…). Although an information object is a complex object hierarchy, it is possible to map the objects in a XML string. The class collection of Figure 19 is a small framework to create and edit information objects easily. The method `toString` returns the string representation of an information object which can then be passed to ObjectSpace. The format of the XML string is shown in chapter 5.3.4. It is also possible to create an `AbstractObject` from a given XML string, if it meets the determined DTD.

5.4.2. Repository

The Repository manages the information objects. It is part of ObjectSpace.

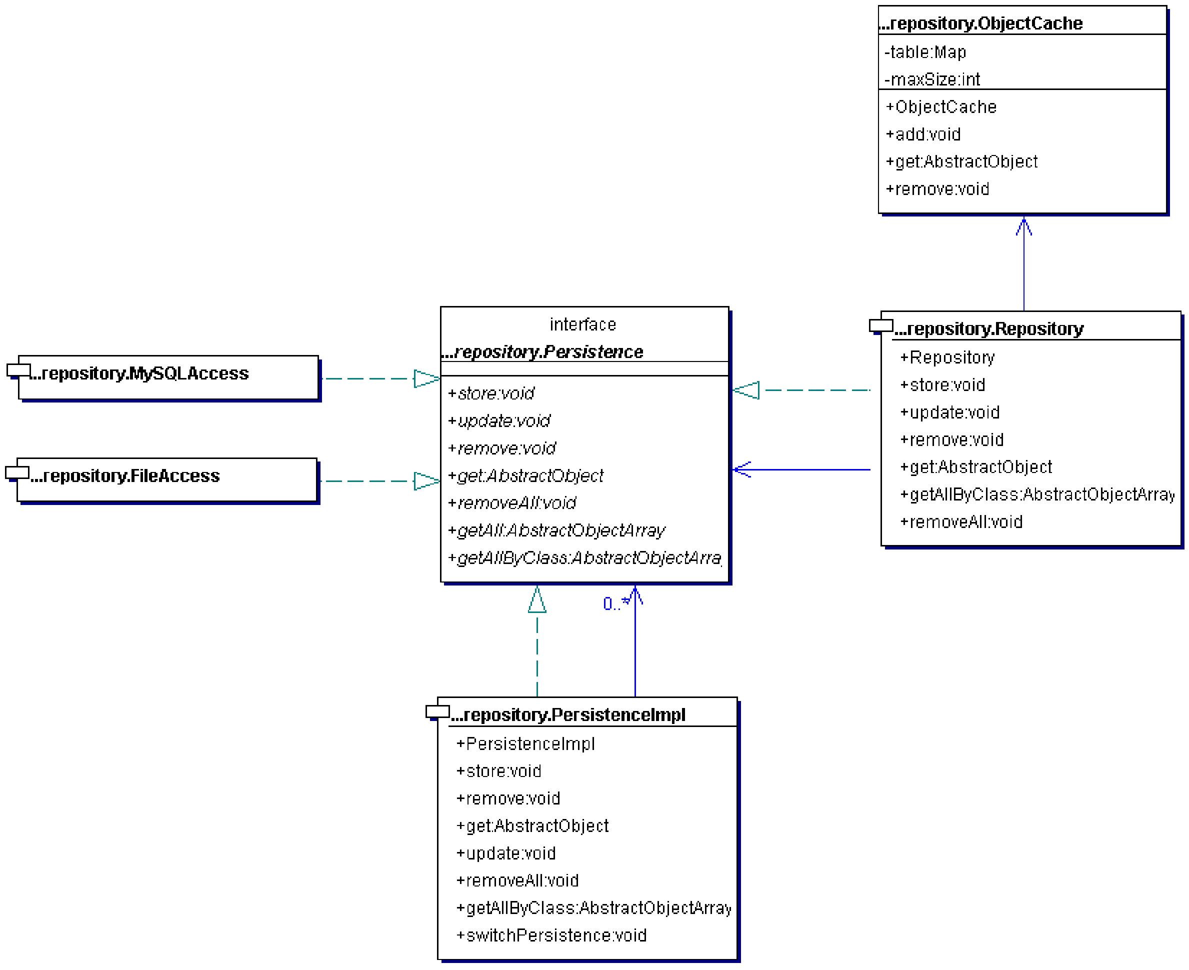

Figure 21 Repository

The repository contains two essential parts. The class Repository associates the cache and the persistence. As shown in the analysis, the persistence is built as a multi-level persistence. Additionally, the analysis shows that the persistence has to be a component itself. By this it is possible to move ObjectSpace during runtime or build a multi-instance architecture. The repository keeps an instance of `PersistenceImpl`, which manages both levels of persistence. So all types of storage objects implements the remote interface `Persistence`, which offers remote access to store and read information objects. By implementing the Factory-pattern [Gamm99] in `PersistenceImpl` it is possible to instantiate Persistence objects dynamically.

5.4.3. Filter

Filters are used by subscribers to filter information objects, i.e. to select whether a specified information object should be transferred to its subscriber. Filters are used to optimise load balancing. The filter-algorithm can be installed in the ObjectSpace during the registration of subscribers. Like information objects, filters are defined in XML to guarantee platform and language independence.

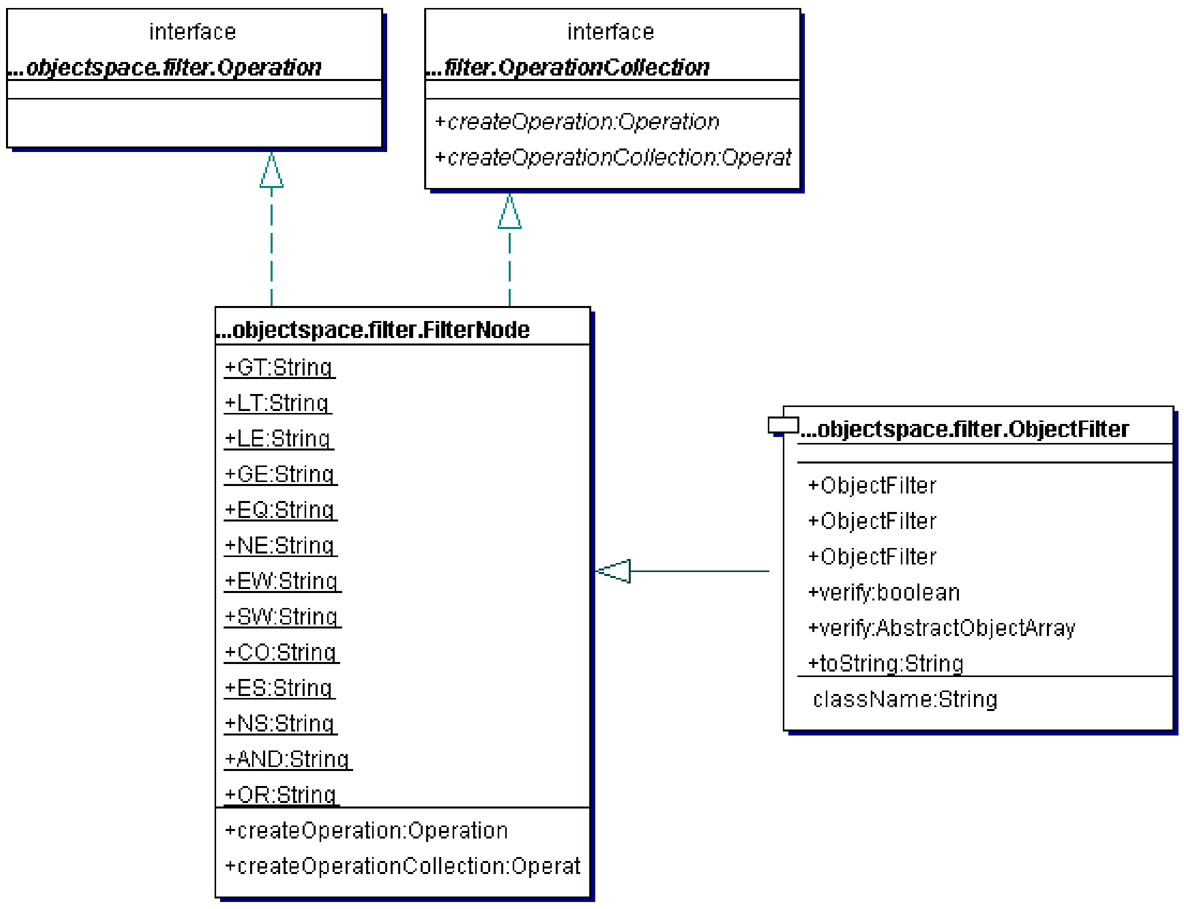
interface
...objectspace.filter.Operation
interface
...filter.OperationCollection
+createOperation:Operation
+createOperationCollection:Operat
...objectspace.filter.FilterNode
+GT:String
+LT:String
+LE:String
+GE:String
+EQ:String
+NE:String
+EW:String
+SW:String
+CO:String
+ES:String
+NS:String
+AND:String
+OR:String
+createOperation:Operation
+createOperationCollection:Operat
...objectspace.filter.ObjectFilter
+ObjectFilter
+ObjectFilter
+ObjectFilter
+verify:boolean
+verify:AbstractObjectArray
+toString:String
className:String

Figure 22 Filter

Figure 22 shows the class-diagram of a small framework, which helps to create and edit filter objects. The string representation of filters is shown in chapter 5.3.1.1. The design of Filter is based on DOM. Every node in the filter is accessible by an interface. Every node in the filter can be an operation or a collection of operations, which are combined with a boolean operation. The following sequence-diagram shows the creation of the example filter of chapter 5.3.1.1.

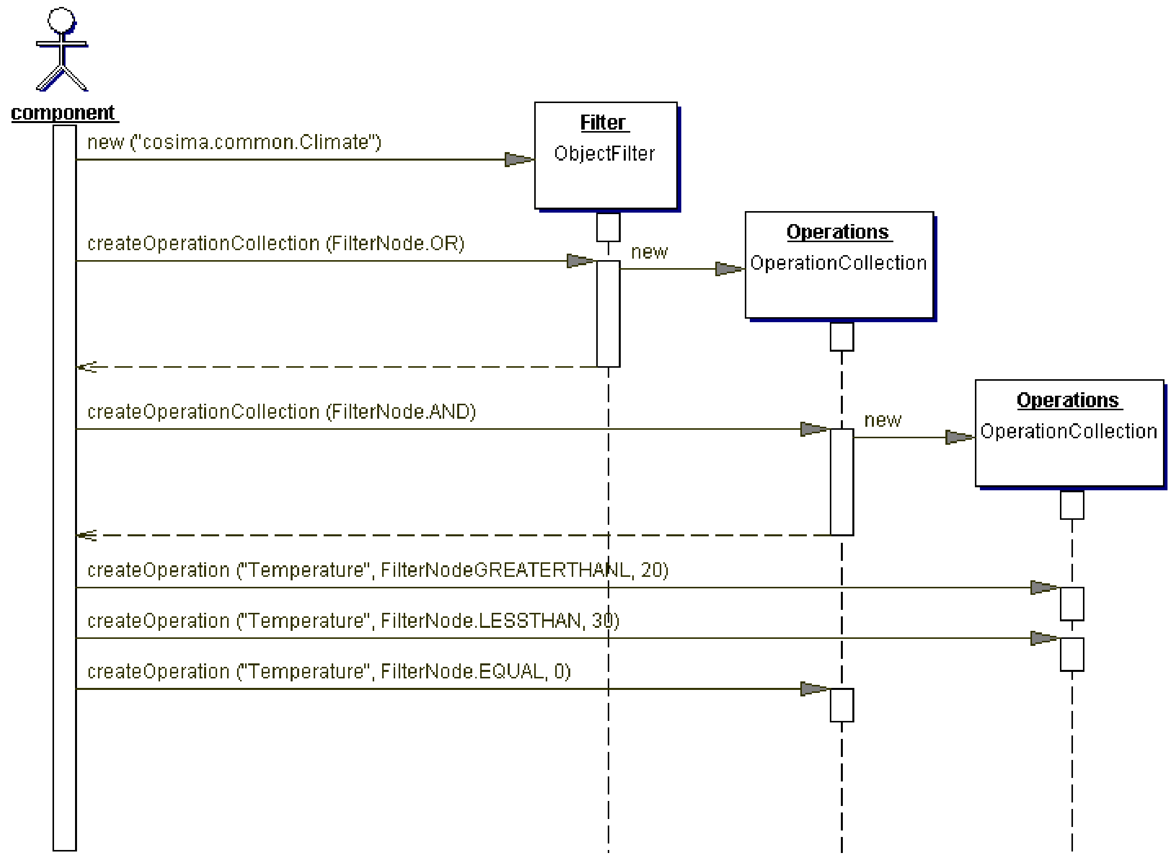
component
Filter
ObjectFilter
Operations
OperationCollection
Operations
OperationCollection
new ("cosima.common.Climate")
createOperationCollection (FilterNode.OR)
new
createOperationCollection (FilterNode.AND)
new
createOperation ("Temperature", FilterNodeGREATERTHANL, 20)
createOperation ("Temperature", FilterNode.LESSTHAN, 30)
createOperation ("Temperature", FilterNode.EQUAL, 0)

Figure 23   The creation of a filter object

5.4.4. ObjectSpace

Figure 24 shows the classes of the whole implementation. `ObjectSpaceImpl` builds the implementation of ObjectSpace. This class implements the remote interface ObjectSpace, which is used by publishers and subscribers to communicate with ObjectSpace.

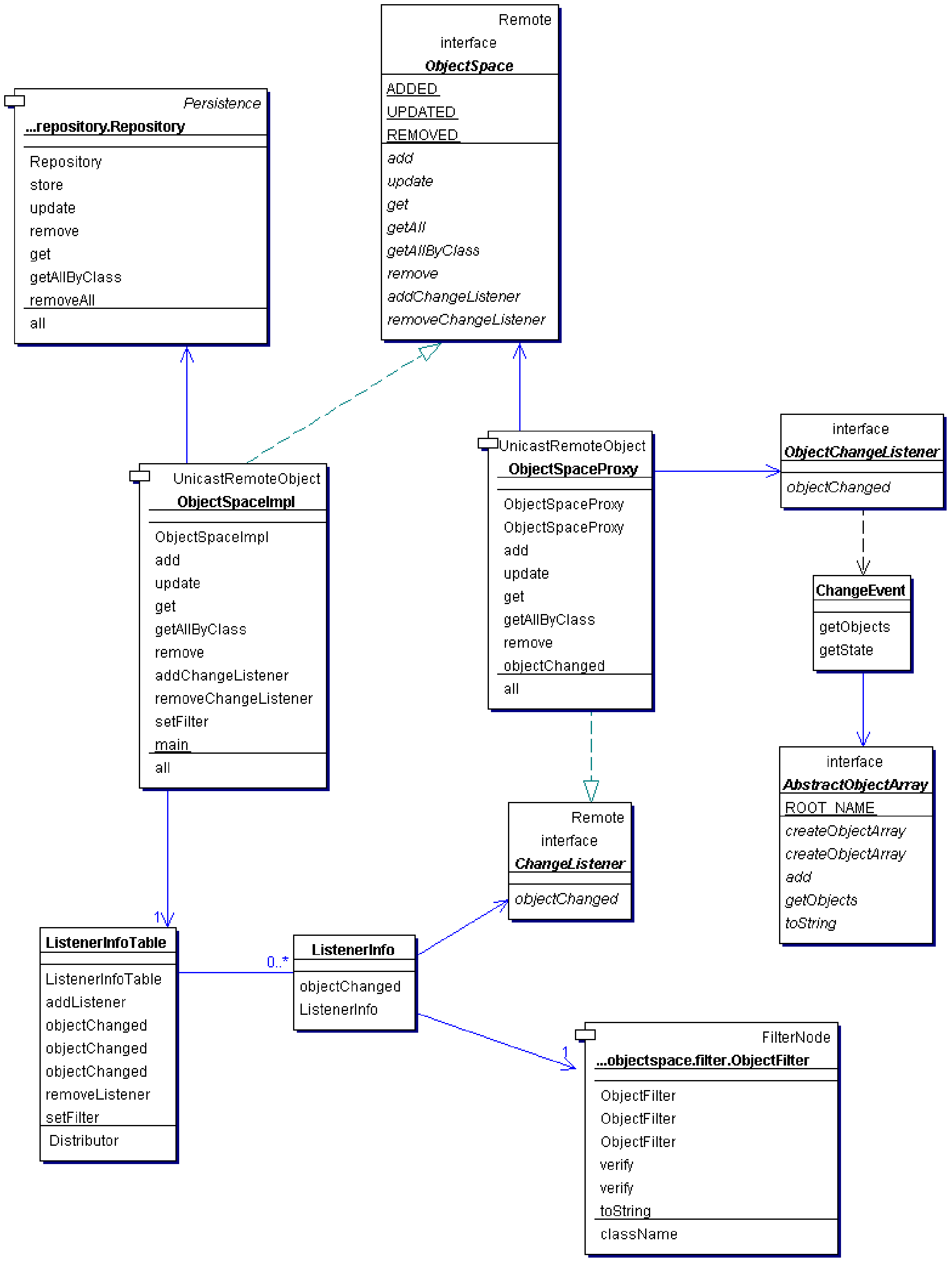

Figure 24 ObjectSpace

`ObjectSpaceImpl` associates one instance of the Repository, which manages the persistence and the cache. The `ListenerInfoTable` contains the `ListenerInfo` objects, which encapsulate the attached listeners (subscribers) and their installed filters. The method `objectChanged` is called in the subscriber by the `ListenerInfo` object. The subscriber has to implement the interface

`ChangeListener` so that it can be informed that objects are added, updated or removed. The `ObjectSpaceProxy` is built to encapsulate the communication with ObjectSpace, so that it will be easier to implement subscribers or publishers. The following diagram shows how this proxy simplifies the development.

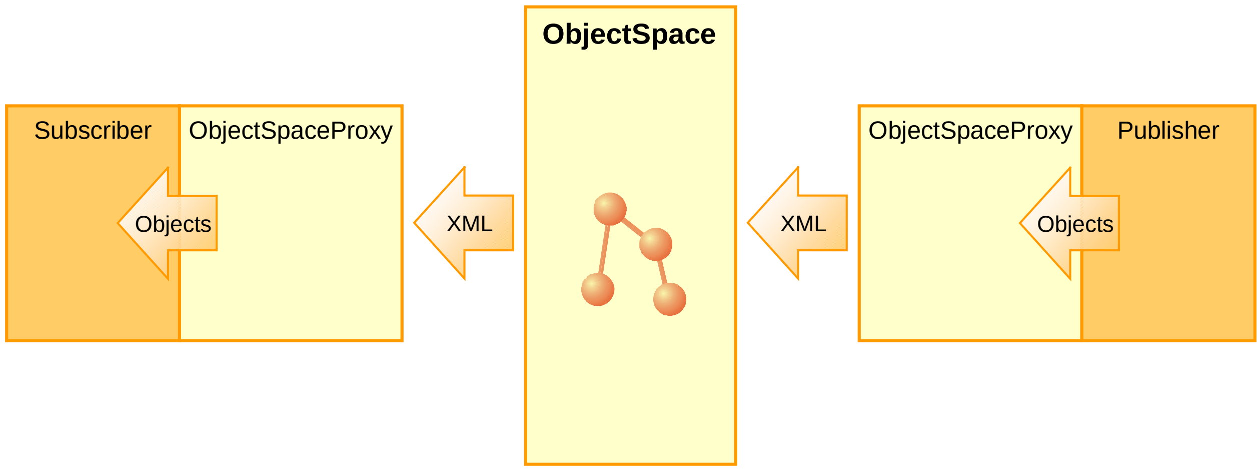

Figure 25 Using proxies with ObjectSpace

The publisher instantiates an `ObjectSpaceProxy` that will automatically create a connection to ObjectSpace. The publisher can now use the `AbstractObject` framework, without knowing that XML is used to publish these information objects. The information object is converted in the proxy to a XML string, which is then sent to ObjectSpace. The subscriber also instantiates a proxy and gives his filter to it during the initialisation. The information object, which is sent from ObjectSpace to the proxy in a XML string, is converted from the proxy to an `AbstractObject`. The subscriber has to implement the interface `ObjectChangeListener`, so that the communication with ObjectSpace is transparent for these components. To guarantee language and platform independence, the proxies have to be implemented in the languages used to build the components. So the proxy hides the communication and the transfer of XML instances. The proxy itself has to implement the interface `ChangeLister`, which contains the method `objectChanged`. The difference between the interfaces `ChangeListener` and `ObjectChangeListener` is the parameter of the method objectChanged. In the `ChangeListener` interface the method gets a string, which represents the information object. In the `ObjectChangeListener` it gets a `ChangeEvent` object, which contains the changed objects. This is based on the event handling implemented in Java. The `ChangeListener` Interface is needed for language independence and can be replaced by any sort of communication. The following diagram shows the registration of subscribers and the transfer of information objects using proxies.

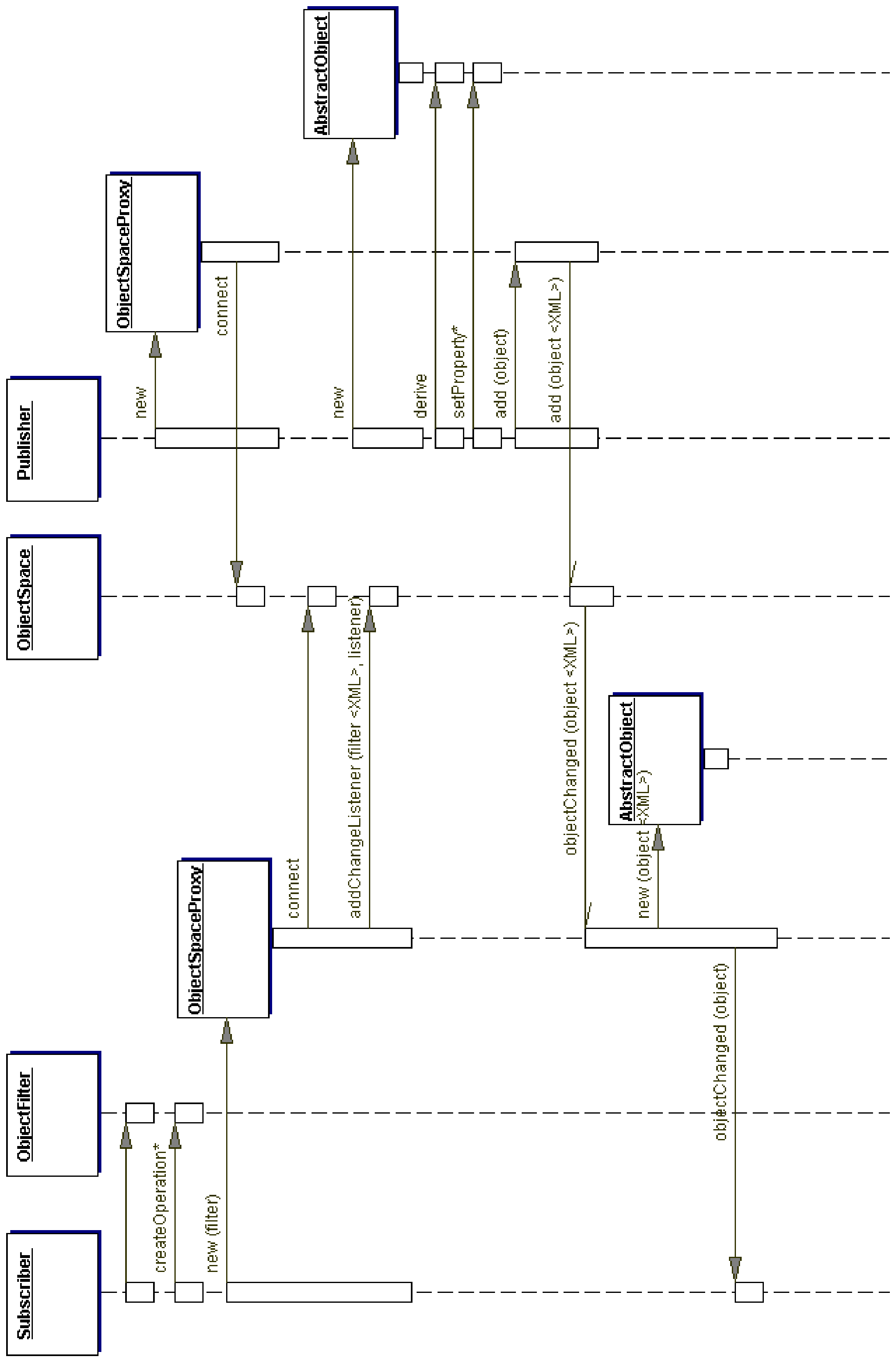

Figure 26 Registration of subscriber and publishing of information with proxies

How subscribers and publishers are created is shown in chapter 0.

5.5. Test

To test ObjectSpace, two essential tests are made. During the white-box test we look at the implementation. This test is made during the implementation. ObjectSpace is made of different small components, like cache, repository etc. These components can be tested independently by building test-units.

The black-box test can be made by using the service interface of ObjectSpace. The service interface is a remote interface, which can be used by other components. So test components are built for the functionality-test and the load-test. During the load-test different aspects are tested:

How does the component react, if a bulk of information objects is transferred?

Will there be any problems if a large number of information objects are stored in the repository?

Will there be problems with mutual exclusion if several components transfer information objects in parallel?

The last point is difficult to test, because it is more or less a hazard if the system fails during parallel usage. The longer the parallel test is made, the more the probability increases that the system is resistant against failures.

During the testing of ObjectSpace, it could be seen that the handling of information objects needs the longest time. Information objects are passed as strings to ObjectSpace. There they are converted to `XMLObjects`, which means that the string has to be parsed. The next step is the verification of information objects with the help of filters. The filters are also stored as strings. So the filter has to be parsed and interpreted. The following diagram shows the critical sections:

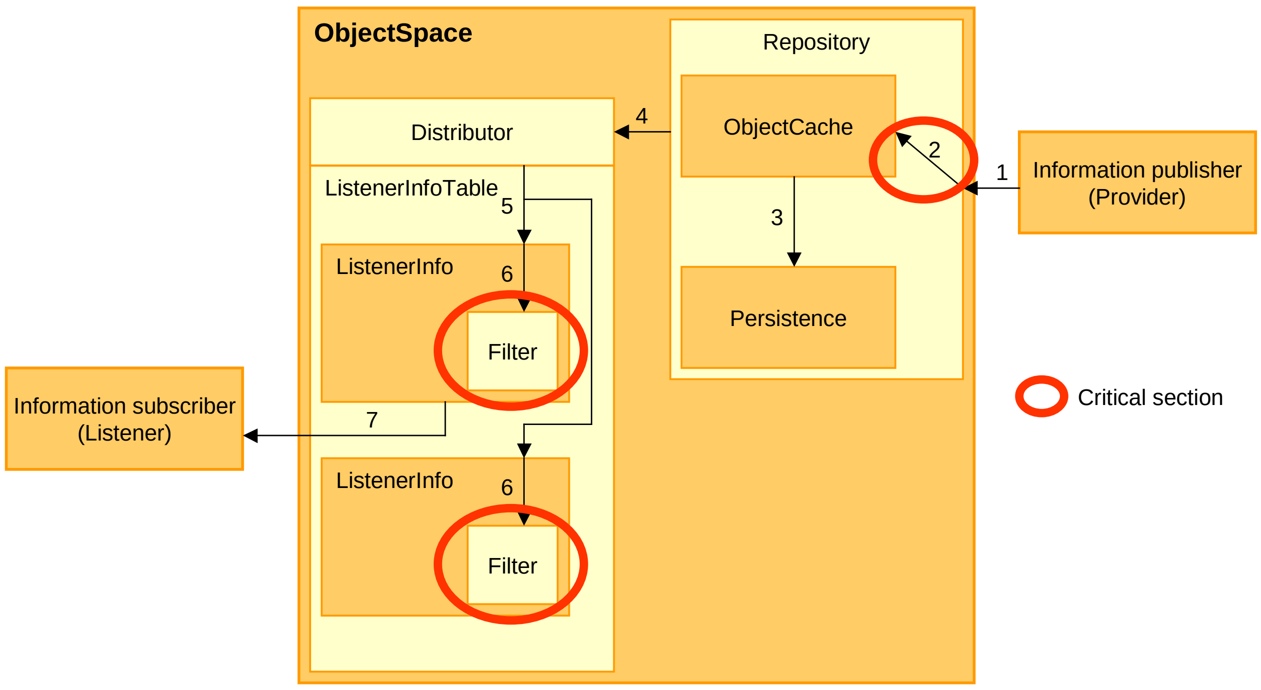

Figure 27 Critical section, which needs long processing time

As shown in the analysis (chapter 5.2.6) the filter-algorithms can be distributed by building multiple instances of ObjectSpace. By this, the ObjectSpace is scalable depending on the performance of the persistence.

6. Conclusion and Outlook

The project could be successfully finished. ObjectSpace is integrated in COSIMA and is currently used to publish messages in a sort of Universal Messaging System. ObjectSpace was implemented in Java. The communication interface was implemented in RMI. The main problem regarding design and implementation was the requirement to build a platform and programming language independent system. Similar systems as for example JavaSpaces from SUN cannot meet this requirement because it is Java specific and works exclusively with Java objects. The need of installing 'logic' dynamically in form of filters in ObjectSpace, which should be instantiable by any language implementation could only be achieved by defining a sort of programming language. XML was selected because this technology is a standard which allows easy usage on account of its high propagation.

In future versions, ObjectSpace will use CORBA instead of RMI for the communication with other components. As shown in the thesis other sorts of communication could be used without the need to change publishers or subscribers. The communication is encapsulated in proxies, which can be changed without any dependencies; therefore, also proprietary protocols can be used. The design of ObjectSpace was made in a manner so that it can be implemented in any programming language. Hence it is possible to create an implementation for example in C++, which can be faster, at least today. In a future version it also conceivable that DTDs are used during the publishing for a better identification of information objects. An additional point which was not considered in this project is security. Every publisher can offer information and every subscriber can read it. Probably in future versions this problem is considered and an extra security is implemented to protect the data against non-authorised access.

Another point is the persistence. The implemented version of ObjectSpace uses a relational database. Future versions could use object-oriented databases, which are based on XML. Currently several companies work on a standard for a new query language especially for this. This could be an optimal supplementation of ObjectSpace.

ObjectSpace could also be seen as a sort of software bus system. If we disregard the persistence, the information objects can be seen as descriptions of method calls or bus messages. This would allow multicast or broadcast remote calls. But this is not the aim of ObjectSpace. Software bus implementations for distributed systems have another architecture because the performance is a very important point and the main interest of the project is the publishing of information objects in a manner that it is identifiable by any component, which is interested in this sort of information.

7. Appendix

The following chapters will show how components are created that use ObjectSpace for information transfer. ObjectSpace is currently implemented in Java. The following chapter will describe the implementation in this programming language. The following chapters will not show how COSIMA components have to be implemented. The focus will be on these parts, which are required to use ObjectSpace. Several mechanisms, like the service lookup, are encapsulated in base classes delivered with COSIMA.

7.1. How to build an information publisher

This chapter will show how publishers can be created using the helper classes delivered with ObjectSpace.

Publishers pass information objects to ObjectSpace. The first step is to define the information object. A DTD can help to declare the content of the object, so that it can be used by other components. The following example contains the publishing of a message object, which is then be used by the subscriber as described in the next chapter.

7.1.1. Defining the message object

The object hierarchy is just defined by ObjectSpace (see chapter 5.3.4). Hence only the content of the message object has to be defined in a DTD for other developers. This will show which data is contained by the message object and how it can be derived. So the DTD is used to document the information object and can be seen as a sort of class definition. As shown above the object itself is defined in XML:

```
<RootObject UniqueID="f01a345b386e9dc0" Version="1.0.0">
     <DerivedObject      Name="Message"
                         Version="1.0.0"
                         Package="cosima.common.message">
          <String Name="Sender">sender@test.com</String>
          <String Name="Recipient">rcpt@test.com</String>
          <Data Name="Subject">
               <![CDATA[This is the subject]]></Data>
          <Data Name="Content">
               <![CDATA[This is the message]]></Data>
     </DerivedObject>
</RootObject>
```

ObjectSpace does not know anything about an information object hierarchy. By this, all base objects of an information object have to be transferred. Hence the DTD is needed to know, which fields have to added in the base objects. To create a XML string in a program is simple but not very predictable. Therefore, additional classes (AbstractObject framework) are delivered with ObjectSpace (see chapter

5.3.4). The following code shows how the framework can be used to create a message object.

```
AbstractObject object = new XMLObject ();
AbstractObject message = object.derive
               (“cosima.common.message.Message”, “1.0.0”);
message.setProperty (“Sender”, “sender@test.com”);
message.setProperty (“Recipient”, “rcpt@test.com”);
message.setProperty (“Subject”, “This is the subject”);
message.setProperty (“Content”, “This is the message”);
```

The object is now created and can be passed to ObjectSpace:

7.1.2. Publishing Information

To get a connection to ObjectSpace a proxy object has to be instantiated.

```
ObjectSpaceProxy proxy = new ObjectSpaceProxy ();
```

The proxy object automatically creates a connection to ObjectSpace and can then be used to pass the information.

```
proxy.add (message);
```

It is also possible to build a direct connection to ObjectSpace and transfer the information object in a XML string:

```
obectspace.add (message.toString ());
```

The next chapter will show how the transferred information is used by the subscribers.

7.2. How to build an information subscriber

This chapter will show how subscribers can be created using the helper classes delivered with ObjectSpace. There are two ways to implement a subscriber component. The first is to use a proxy object for communication with ObjectSpace, the second is to use ObjectSpace directly. This chapter will only describe the first way. The first step is to define a filter. The DTD, which describes how a filter has to be defined is shown in chapter 5.3.1.1. The following example shows a filter, which examines whether the received message contains an appointment by searching for the tags `<appointment>` and `</appointment>` in the content of the message.

```
<Filter ObjectName="cosima.common.message.Message">
    <AND>
        <Operation  DocumentValue="Content"
                    Operator="CONTAINS"
                    CompareValue="<appointment>"/>
        <Operation  DocumentValue="Content"
                    Operator="CONTAINS"
                    CompareValue="</appointment>"/>
    </AND>
</Filter>
```

The filter can also be created by using the filter framework (see chapter 5.3.1.1) delivered with ObjectSpace. The following code example shows how the filter above is created with this framework.

```
FilterNode filter = new ObjectFilter
("cosima.common.message.Message");
OpertionCollection collection =
          filter.createOperationCollection (FilterNode.AND);
collection.createOperation
          ("Content", FilterNode.CONTAINS, "<appointment>");
collection.createOperation
          ("Content", FilterNode.CONTAINS, "</appointment>");
```

The next step is to build the contact to ObjectSpace and register the subscriber what can be done by instantiating `ObjectSpaceProxy`. For that the subscriber has to implement the interface `ObjectChangeListener`, which allows the notification by ObjectSpace, if the information structure is changed.

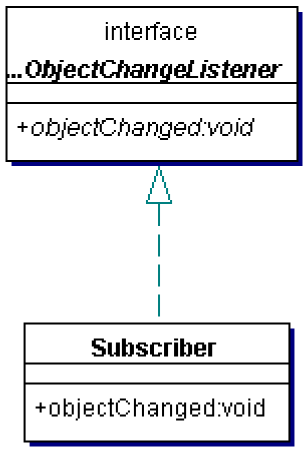

Figure 28 Implementing the ObjectChangeListener

The method `objectChanged` has to be implemented, which is called by the proxy, if objects are changed.

```
pubic void objectChanged (ObjectEvent event)
```

`ObjectEvent` contains a list of changed objects received from ObjectSpace. This approach is based on the event handling implemented in Java.

```
ObjectSpaceProxy proxy = new ObjectSpaceProxy
                                       (subscriber, filter);
```

The proxy automatically creates the connection to ObjectSpace and attaches the subscriber. When the information structure in ObjectSpace is changed, the changed objects are passed to the filter. If the objects are interesting for (in this example, if they contain an appointment), the objects are passed to the proxy, which encapsulates them in the event and by this to the subscriber by calling the method `objectChanged`. The `ObjectSpaceProxy` can always be used for both, the subscriber and the publisher. Hence it is possible that information objects are deleted by the subscriber which then changes its role to publisher.

7.3. Used Software to develop ObjectSpace

UML diagrams for analysis and design were made with Together Control Center 4.1 by TogetherSoft Corporation.

ObjectSpace is implemented with Java JDK 1.3 by SUN.

Tests were made with JUNIT.

8. Glossary

COSIMA	COmponent System Information and Management Architecture. Dynamic component based system, which allows components to be loaded, unloaded, updated and moved during runtime.
ObjectSpace	COSIMA component which allows asynchronous distribution of information objects. The information objects are defined in XML in an object-oriented way, so that the information can be identified by any component in the system.
RMI	Remote Method Invocation. Java specific implementation of a remote call. Allows the invocation of methods of objects in other virtual machines, also on other computers over the network.
CORBA	Common Object Request Broker Architecture. CORBA defines an architecture for communication between objects.
XML	Extensible Markup Language. XML is designed to describe data. It uses a DTD to formally describe the data.
DTD	Document Type Definition. The purpose of a DTD is to define the legal building blocks of an XML document.
DOM	Document ObjectModel. The DOM is a programming interface for HTML and XML documents. It defines the way a document can be accessed and manipulated
POP3	Post Office Protocol Version 3. Protocol, which is used from mail clients to get internet mails from a server.
SMS	Short Message Service. It is a feature available with some wireless phones that allow users to send and receive short messages with 160 letters.
Service interface	The service interface is part of COSIMA components. A component offers its service, which can be accessed by other components by using that interface.
UMS	Universal Messaging System. UMS is a flexible, modular system which allows the usage of different networks and services like POP3, SMS, FAX etc.

9. References

[Stuem99] Matthias Stuempfle, The COSIMA Primer, DaimlerChrysler AG, 1999

[Coul99] George Coulouris, Distributed Systems – Concepts and Design, Addison-Wesley, 1999

[Gamm99] Erich Gamma, Design Patterns, Addison-Wesley, 1999

[Scha99] Stephen R. Schach, Classical an Object-Oriented Software Engineering – With UML and Java, McGraw-Hill, 1999

[Boo94] Grady Booch, Object-Oriented Analysis and Design, Addison-Wesley, 1994

[Goll99] Joachim Goll, Java als erste Programmiersprache, Teubner, 1999

[Wil99] Gerhard Wilhelms, Java professionell, mitp, 1999

[Mue00] Frank Mueller, Component Development, DaimlerChrysler AG, 2000

[Mue01] Frank Mueller, Assignment: Patterns and Collection classes, Brunel University, 2000

[Mue02] Frank Mueller, Assignment: Sockets and RMI, Brunel University, 2000

[Beh00] Henning Behme, Stefan Mintert, XML in der Praxis, Addison-Wesley, 2000