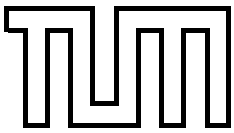
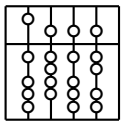


Technische Universität München
Fakultät für Informatik

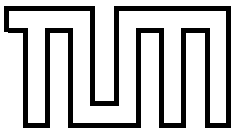


Systementwicklungsprojekt

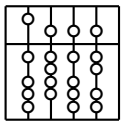
Filter Framework for DWARF

Distributed Wearable Augmented Reality Framework

Michael Schlegel



Technische Universität München
Fakultät für Informatik



Systementwicklungs Projekt

Filter Framework for DWARF

Distributed Wearable Augmented Reality Framework

Michael Schlegel

Aufgabenstellerin:	Prof. Gudrun Klinker, Ph.D.
Betreuer:	Dipl.-Inf. Marcus Tönnis
Abgabedatum:	5. September 2005

Ich versichere, daß ich diese Systementwicklungs Projekt selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 5. September 2005 Michael Schlegel

Zusammenfassung

Erweiterte Realität ist eine Technologie, die versucht eine virtuelle Realität und die Realität miteinander zu kombinieren. So können Informationen aus einem Rechensystem mit realen Objekten assoziiert werden.

Bei der Umsetzung solcher Systeme, treten bestimmte geometrische Transformationen gehäuft auf. Rotationen und Translationen sind Beispiele für solche Transformationen. Meist durchlaufen die Daten dabei ein ganze Kette von einfachen verständlichen Transformationen.

Die Idee ist nun solche Transformationen in Filter zusammenzufassen. Und die Filter zu Netzwerken zusammenzuschalten.

In diesem Systementwicklungs Projekt wird versucht ein möglichst offenes, wiederverwendbares Framework für solche Filternetzwerke zu erstellen.

Abstract

Augmented Realıty is a technology that tries to combine virtual reality and reality.

This requires a lot of geometric calculations. Such as rotation, translation and others. In most cases the data has to pass through a lot of calculations. The idea is to pool such transformations to filters and to combine those filters to a network of filters.

In this “Systementwicklungs Projekt” I try to develop a open and reusable framework for such filternetworks.

Contents

List of Figures

Listings

1 Introduction

In this chapter I give an basic overview of the DWARF project, the basic ideas of Augmented Realıty and what this project is about.

1.1 Augmented Reality

Augmented Realıty is a technique to combine the real world with computer generated data. This approach allows you to present information in a very convenient way. The user has a very natural interface to data from a computer model, because the information is right there, where the user expects it.

Example: In current cars the navigation system provides two ways of informing the driver about the route. On way is audio via voice, the other is a small display in or above the center console. With Augmented Realıty you can glue the arrow on the road, so the driver can concentrate on the really important thing, driving on the road.

1.2 DWARF

The DWARF [3] is an acronym for "Distributed Wearable Augmented Reality Framework". It is intended to allow the development of distributed AR applications by reusing configurable components. The distributed approach of DWARF allows to build AR applications by using more than one machine. With the chosen middleware CORBA it is even possible to use a heterogeneous infrastructure, due CORBA is not limited to a specific programming language or specific machine architecture.

DWARF consists of interdependent services, which expose their requirements, called Needs, and offers, called Abilities, with the help of service managers.[10]

1.3 Motivation

Augmented Realıty is a technology which deals with a lot of spatial calculations. There are coordinate transformations to be done, scaling of objects, translations of object to other places in space. Other non geometric calculations are for example the speed of an object, or the barycenter of two or more points. There are various kinds of such calculations.

These calculations are often concatenated. An example for this is a translation after a coordinate transformation. These chains of calculations happen quite often.

At the moment each application and service in DWARF implements its own algorithms. This is unnecessary and error-prone. It is better to have a well tested basic set of calculations. So it is obvious to make these calculations more general and find a way to connect these calculations to a network.

By implementing these calculations into several DWARF services we meet the distributed approach of the DWARF framework. The services can be distributed on many workstations and so even a small wearable device can handle big applications. Another advantage of implementing these algorithms into services is, that they don't have to be implemented in an other programming language.

This was not done in the past, because the was no common way to implement an algorithm in a service. The developer had to implement a DWARF service from scratch, which can be annoying.

1.4 Goals

The purpose of this project is to design and implement a framework, so new calculations can be easily implemented and reused by other projects. To reach this goal a plugin based architecture is chosen.

The second part of this project is the concrete implementation of a set of common filter plugin services, such as rotation, translation, speed calculation and other general basic functionality. The concept of designing filter as services was introduced to DWARF with the CAR project. [16] [12] [6]

As a spin-off this project discusses also how CORBA events can be handled in the DWARF environment, without lots of performance losses. CORBA events are needed for the information flow of the services

2 Requirements for the Design of Performant Networks

By the base decision of plugging the functionality into services by a PluginFramework , I explain in this chapter, what the plugin framework should be able to do. I discuss which features are worth to implement and which are not. Because performance is also an important issue, I will discuss how performance can be improved.

The PluginFramework should satisfy the following criteria:

generic The framework has to support all data types of the control flow, even types that will be built in. It should not be necessary to write a handler for a DWARF event type.

easy to use Developers shall not have to deal with large overhead in reading manuals, so that the functionality provided by this project has a higher acceptance rate for setting up applications. For easy setup of a new service an easy mapping between the service's description and the plugins functionality has to be found.

fast Plugins implemented with this framework, should be as fast as native java services.

2.1 Communication

Of course the plugins have to communicate with each other and with other services within the DWARF environment. The DWARF framework knows three different ways of distributing data. Either by direct method calls – object references – , by events or by shared memory.

Because the original intention of this plugin framework was to act as filter for continuous data like position data from a tracker subsystem, this work focuses on event typed communication. But there are more reasons to focus on events.

Interoperability Most services and applications in DWARF using events for information exchange. For example the ObjectCalibration service use events for sending the position of objects. It consumes the positions of the tracked marker and adds a static offset to the origin of the associated real object.

This event typed communication makes it easy to place a pluginservice between the objectcalibration and the existing application, without changing their interfaces. The only things, that have to be changed are the service descriptions.

Recording Events can easily be recorded and played back by other services. So a plugin-service can be debugged without having a complete tracking hardware at your fingertips.

Of course you could also record direct function calls, by introducing a proxy service. But the service for recording events exists already.

Asynchronous Using method invocation provides only communication in a synchronous manner. So a service must wait until the partner is finished processing.

By using the CORBA notification service, we get a asynchronous exchange of information.

The point-to-point communication architecture is another drawback of the method invocation, because a invocation is targeted for a specific partner. But in a Augmented Reality system more than one service wants to be aware of a objects position in space.

2.2 Networking

In most cases we don't want to use only one plugin, so we need techniques to connect them to a network. So basic plugin, such as rotation or translation plugins, can be combined to a more complex and powerful filter network.

The basic idea is just to reuse the DWARF middleware networking concept represented by the service manager. By packing filters in individual services we can define needs and abilities. And these need and abilities can be connected by the service manager.[10]

By adding attributes and predicates to the needs and abilities, we can construct a network of plugins.

Another reason to use more than one service is visualization. You can actually view service networks or subsets of them in the DIVE [13] application and see whether the connections are correct or not.

2.3 Performance

According to Azuma's definition [1] an Augmented Reality system should interact in real time with the user. So the PluginFramework has to comply with some requirements of performance.

2.3.1 Processes and Threads

In the DWARF environment each service runs in its own process. So sending data between two services costs at least one full context switch, which is expensive. See fig 2.1.

But using threads instead of processes does not solve this problem. To explain this, a closer view to the CORBA notification service is needed.

When using a asynchronous event driven communication model, a CORBA service called "notification service" is used.

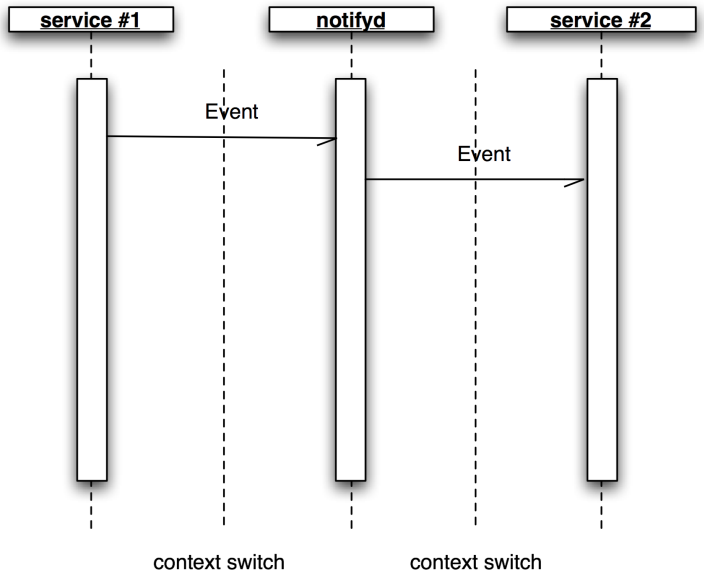


Figure 2.1: Context switches

2.3.2 Java Reflection

Due reasons of generality, this project will heavily use Java's reflection package. This could be a performance issue.

In the initialization of the plugins the pluginservice tries to access information about the plugin. This is done by Java's reflection capabilities. Indeed, this costs a lot of CPU power but is has to be done only once and only at startup.

Because the information the plugin service has about the plugins is collected via reflection, method invocations on the plugins have also to be done with reflection. The costs of these runtime reflection calls have to be estimated, therefore I will explicitly discuss performance issues on this topic in chapter 6.1.2.

3 Implementation

In this chapter I explain how the PluginFramework has been implemented. I describe the interfaces and the classes in detail and how they work together.

The implementation splits up into three parts.

PluginService The biggest and most important one is the *PluginService*. This service runs the communication with the servicemanager, sets up the need receiver and ability sender. For each plugin, exactly one PluginService is needed.

The PluginFramework differentiates between the plugin service and the plugins. A plugin is a concrete implantation of an algorithm. The PluginService manages the connection to the DWARF middleware. So the PluginService is needed to run the plugins, ant it is not appearing as the PluginService in DWARF but as a service of the plugin.

Each PluginService can only manage one plugin at once. If more than one plugin is needed, more than one PluginService has to be started.

LoadService The LoadService class is introduced due performance issues. It allows you to run several PluginServices in one process, so no expensive context switching has to be done. This becomes an issue, if you want to build a network of small generic plugins, and events have go through many plugins.2.1.

PluginLauncher The PluginLauncher is a helper service, which allows you to start java services via the LoadService class. The PluginLauncher provides a convenient way to start more than one plugin in the same process. This increases performance, because lesser context switches are needed.

The communication between the plugins has been realized with events. I choose this way, because it is the common way in the DWARF framework for continuous data streams, as for instance Poses are. An other advantage is that the user does not has to care where the plugins are located. They can be in the same process space or on another machine. And the flow of events can be easily debugged with DIVE .

The plugins are described by dwarf service descriptions. Figure 3.1 gives an example for such a description. As you can see, there is one service attribute which indicates which plugin should be used.

```
<service name="scaleSheep" startOnDemand="true"
```

```
    stopOnNoUse="false" isTemplate="false">

  <attribute name="PluginClass" value="ScaleFilter"/>

  <ability type="PoseData" name="Scaled">
        <attribute name="Factor" value="0.5"/>
     <connector protocol="PushSupplier"/>
  </ability>

  <need name="Position" type="PoseData" minInstances="1" maxInstances="1"
     >
        <connector protocol="PushConsumer"/>
  </need>
</service>
```

Listing 3.1: An example for a plugin description

3.1 Classes

The complete static class model of the PluginFramework can be seen in figure 3.1. I describe the classes in the order the are instantiated while a new plugin is started.

The PluginService has to know about the service it should start. The name of that service must be unique and is given by an option in the commandline.

3.1.1 PluginLauncher

Among many others, plugins should have two features. They should run in the same process, and they should start on demand.

But how can the servicemanager start a service in a specific process? The solution is the SvcLoadPOA interface. A class implementing this interface can be registered at the servicemanager as a callback for a service. When the servicemanager wants to start this service, this job is delegated to the callback object.

The purpose of the PluginLauncher is to register the callback object of all plugins to be started in the same process space at the servicemanager.

The PluginLauncher service itself has in normal case no needs and abilities. It is just introduced to explain the functionality or to start plugins for older applications. It can also be integrated in other services by use of the `ServiceLoad` class.

In sample service description 3.2 you can see how to tell the PluginLaucher which services should be started in the same process. In this case the service “ping” and “pong” are started by the PluginLauncher on demand of the servicemanager.

```
  <service name="PluginLauncher" startOnDemand="true" stopOnNoUse="false"
      isTemplate="false" startCommand="PluginLauncher.jar"
     startAutomatically="false">
```

```
  <attribute name="Start" value="ping,pong"/>

</service>
```

Listing 3.2: An example for a PluginLauncher.xml

3.1.2 PluginService

For each plugin defined in the PluginLauncher service, a new PluginService object will be created. This PluginServices are DWARF services that need their own servicedescriptions with needs and abilities.

```
      <attribute name="PluginClass" value="TranslateFilter"/>


      <ability type="PoseData" name="Translated">
              <connector protocol="PushSupplier"/>
      </ability>

      <need name="Position" type="PoseData" minInstances="1"
      maxInstances="1">
      <connector protocol="PushConsumer"/>
      </need>
```

Listing 3.3: An example for attributes describing a plugin

The `PluginClass` attribute defines which plugin class will be loaded. The `PluginClass` can be either a absolute path name or relative to `de.tum.in.dwarf.PluginService.-plugins`.

In the example 3.3 the plugin should have the name "TranslateFilter.class" and should be located in package `de.tum.in.dwarf.PluginService.plugins`.

3.1.3 PluginFactory

The PluginFactory[5] actually sets up the plugin, it is called for each need and ability found by the PluginService. It is implementing the singleton pattern [5] so there is only one PluginFactory, even more than one plugin is started in the same process.

It holds information about which plugins have already been instantiated and which needs and abilities. If necessary, the factory creates a new instance. The need or ability registration request from the PluginService is delegated to the plugin. The plugin then returns the corresponding handler to the factory and the factory gives it back to the PluginService.

3.1.4 Plugin

The Plugin class is an abstract template class. All plugins must inherit from this class because the management for the plugins is done here. The plugin class has all informations about the event sender, the event receiver and of course about the plugin.

A plugin get its information by implementing need callbacks methods starting with `setNeedData`. The Plugin class is looking via the java reflection API for methods starting with a constant prefix "`setNeedData`". The plugin class creates instances of receiver and sender handler, so this work is hided from the plugin user.

The plugin class implements the `AttributesChangedPOA` interface and calls the `attributesHaveChanged(Attributes attributes)` method, if attributes are changing. By overwriting this method, the developer of a plugin gets informed about the changing attributes.

3.1.5 ContextSwitch

To provide the ContextSwitch [17] ability the C++ `DefaultContextSwitch.cpp` class has been ported to Java. The PluginFramework is creating a new `DefaultContextSwitch` object for each ContextSwitch need found in the service description.

3.2 Testing

Testing software is a fundamental issue in software engineering. But testing services in the DWARF environment can be a painful job. You have to start the service manager and the corresponding services. Perhaps you must write new test services to test your service.

In the java environment the `junit` framework [7] has become a very common tool for testing software[2]. So a interface for testing with the junit framework is built-in in the plugin framework.

To avoid the complex CORBA and DWARF middleware a new class named MockEventSender 3.1 had to be introduced [9][4]. This class implements the same interface as the RealEventSender but it stores the events in a local FIFO queue. So the test suit is able to gather that events with a special getter method. With this class it is possible to test a plugin without starting the whole DWARF middleware. Within the test suite you can send and receive events to and from the plugin. You can also change the attributes. So the whole functionality can be tested easily.

The testing feature has be switched on with the `switchOnTesting()` method of the plugin superclass. Changing the object, which should be tested, is not the common way, but an other solution would become to complex. A JUnit test not touching the base classes would require a of the `PluginService` class and

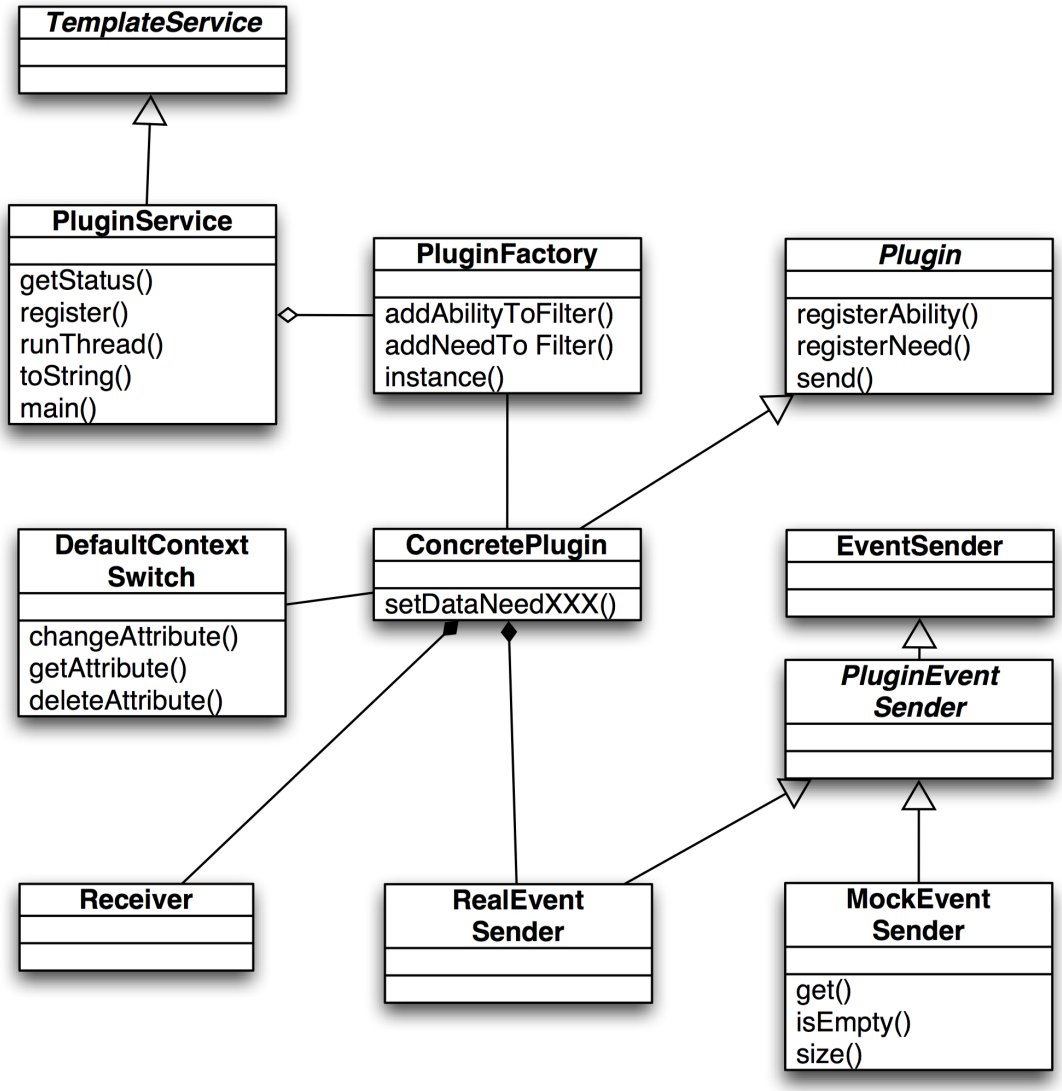


Figure 3.1: PluginService

4 Filters and Networks

In this chapter I introduce a few basic filter services. I divide them into two sections. The geometric filters are about basic geometric transformations. The statistic filters calculates statistical values. Some of these filters are base on a work of Gerhard Reithmayr. [15]

4.1 Filters

Here are implementations of filters using the PluginFramework .

4.1.1 Geometric Filters

4.1.1.1 Rotation

The rotation filter plugin rotates the Position `PoseData` around a given quaternion. This quaternion can either be given with attributes `QX`, `QX`, `QZ`, `QW` or by the orientation part of Quaternion `PoseData` need.

If using the attributes for configuration, you have to define all of them. Otherwise the correct quaternion is used.

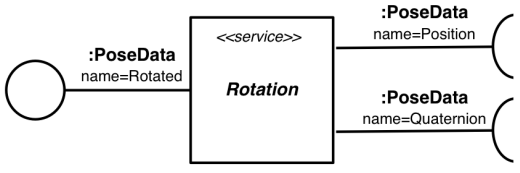


Figure 4.1: Rotation Service

After rotation, a point $p = (0, x, y,)^T$ is then given by $p' = qpq^{-1}$.[14]

```
<attribute name="PluginClass" value="RotationFilter"/>
<attribute name="QX" value="0.0"/>
<attribute name="QY" value="1.0"/>
```

```
  <attribute name="QZ" value="0.0"/>
  <attribute name="QW" value="0.0"/>
```

Listing 4.1: An example for attributes of the rotationfilter service

4.1.1.2 Translation

The Translationfilter gets PoseData (x) and translates this position with the given translation vector(t). So we have the following map $f : \mathbb{R}^3 \to \mathbb{R}^3$ with $f : x \mapsto x + t$

The service can be configured by the service description. The following attributes are possible `TranslationX`, `TranslationY`, `TranslationZ`. These attributes are ignored unless all three have been set to a correct float value. The other way to configure the service is using event. By sending a PoseData to the service position of this event is used for the new translation vector.

If no translation vector is defined the zero vector $(0, 0, 0)^t$ is assumed. The service just forwards the incoming data.

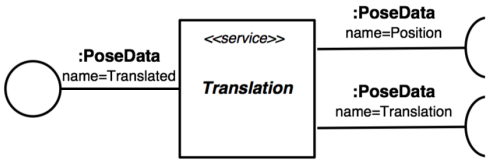


Figure 4.2: Translation Service

```
<service name="translation" startOnDemand="true" stopOnNoUse="false"
isTemplate="false">

  <attribute name="PluginClass" value="TranslationFilter"/>
  <attribute name="TranslationX" value="0.5"/>
  <attribute name="TranslationY" value="1.2"/>
  <attribute name="TranslationZ" value="0.1"/>

  <ability type="PoseData" name="Translated">

  <attribute name="ID" value="translating"/>
    <connector protocol="PushSupplier"/>
  </ability>

  <need name="Position" type="PoseData" minInstances="1"
  maxInstances="1" predicate="(ID=translation)">
    <connector protocol="PushConsumer"/>
  </need>
```

```
  <need name="Translation" type="PoseData" minInstances="1"
  maxInstances="1" predicate="(ID=translator)">
    <connector protocol="PushConsumer"/>
  </need>

</service>
```

Listing 4.2: An example for a translation filter service description

4.1.1.3 RangeFilter

The RangeFilter is a high and low-band filter for incoming position data. It can restrict PoseData to a certain space.

The service can be configured either with event or more detailed with attributes. In the default state the service filters nothing.

To define a lower bound the attribute `maxPosition` has to be set to a comma separated list of float values.

```
  <attribute name="minPosition" value="1.0,-2.0,-"/>
  <attribute name="maxPosition" value="10.0,-,10.0"/>
```

Listing 4.3: Example attributes for RangeFilter

In figure 4.3 the `minPosition` is set to `1.0,-2.0,-`. So every PoseData with x coordinate lower than 1.0 and y coordinate lower than -2.0 will not be forwarded. Any value which is not a float value, like the dash `-`, switches the filtering for this coordinate off. So the y values would not be examined. Setting the attribute to `-,-,-` would switch the filtering off at all.

A lower bound can also be configured by sending a event to the MinPosition need. Once the event arrived, the filtering is started and can not be switched off with events. Another disadvantage using events, is that all coordinates are used.

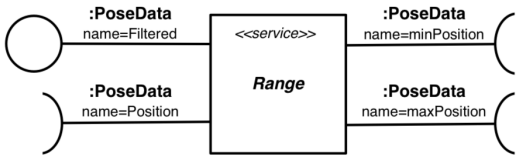


Figure 4.3: RangeFilter Service

4.1.1.4 Distance

This service expects two PoseData. Every time a event is incoming, the distance of the last is sent in an `InputDataAnalogUnlimited` event. The service has no special attributes for configuration.

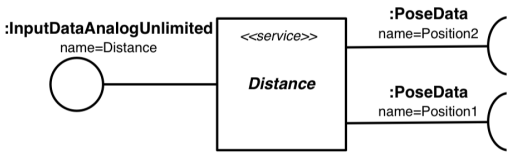


Figure 4.4: Distance Service

4.1.2 Statistic Filters

4.1.2.1 SpeedFilter

This service calculates the speed from the incoming `PoseData`s. Therefor it uses the fields timestamp and position. Each time a new `PoseData` arrives, the service compares the incoming and the last event and sends the speed value as a `InputDataAnalogUnlimited` event and the direction of the moving target as `PoseData` event. The direction is a 3-dim vector containing the object's movement between the last known positions. This is stored in the `position` field of the `PoseData` datatype.

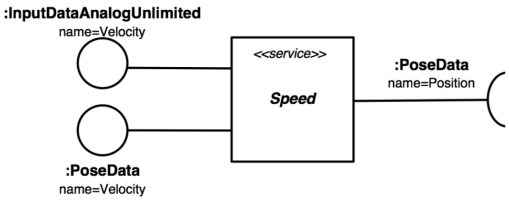


Figure 4.5: Speed Service

It would be nice to have a datatype holding all the necessary informations about velocity, like direction and the speed value. But the discussion about the design of that datatype is not yet finished, so I can only make a proposal for such a datatype in section 6.4.5.

4.1.2.2 ThresholdFilter

The ThresholdFilter evaluates changes of the actual and the last event. There can be a lower and higher bound for angle and distance change.

The service can be configured with the following attributes: `maxDistance`, `minDistance`, `minAngle`, `maxAngle`

The values of the attributes `minAngle` and `maxAngle` have to be given in radian ($0 \leq \alpha_{min}, \alpha_{max} \leq 2\pi$).

For example the attribute `<attribute name="minDistance" value="5.0"/>` only lets `PoseData` events pass, which have a distance of more 5 units to the last known `PoseData`.

```
<attribute name="minDistance" value=""/>
<attribute name="maxDistance" value=""/>
<attribute name="minAngle" value=""/>
<attribute name="maxAngle" value=""/>
```

Listing 4.4: Possible ThresholdFilter attributes

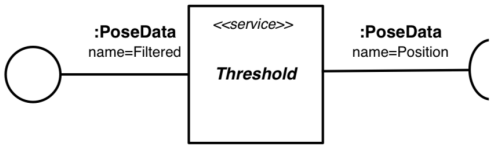


Figure 4.6: Threshold Service

4.1.2.3 Confidence

The ConfidenceFilter Plugin filters incoming events based on their confidence value.

It works either in high or low pass mode and has a configurable threshold value. In high pass mode it only passes events that have a confidence equal or greater than the threshold value, and vice versa in low pass mode.

It has the following attributes. The `Confidence` attribute contains the threshold float value from 0 to 1. The `Mode` attribute can be set to either `high` or `low` to denote the kind of filter mode.

```
<attribute name="Confidence" value=""/>
<attribute name="Mode" value=""/>
```

Listing 4.5: Possible ConfidenceFilter attributes

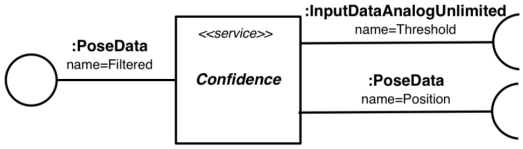


Figure 4.7: Confidence Service

4.1.2.4 WeightedAverage

Often incoming position data has a high deviation, but you don't want your object jump around in space. This is a domain for the WeightedAverage Filter. It can average the position as well as the orientation data. Therefor it consists of a FIFO queue a_i for $i \in \{1, \ldots, n\}$ and a few weights w_i for $i \in \{1, \ldots, n\}$. The incoming `PoseData` is stored in the FIFO, which size is defined by the amount of weights. The position is calculated with the formula 4.1.

$$r = \frac{1}{n} \sum_{i=1}^{n} w_i a_i \tag{4.1}$$

The average orientation is computed as the normalized weighted average of the orientations in the queue in log space. That is, they are transformed to 3D vectors inside the unit hemisphere and averaged in this linear space. The resulting vector is transformed back to a unit quaternion.

Note that the sum of the weights should be equal to 1, otherwise the data is scaled. Also it makes no sense to set the weights to negative values. But perhaps somebody finds it useful, so the service does not check these two conditions.

$$\sum_{i=1}^{n} w_i = 1 \tag{4.2}$$

$$w_i > 0 \qquad \forall i \in \{1, \ldots, n\} \tag{4.3}$$

The service can be configured with two attributes. The `Type` attributes can have the values `all`, `position`, `orientation` or `none`. When set to `all` the position as well as the orientation is averaged. If set to `position` only the positions is averaged. Analog for the `orientation` value. The value of the `Weights` attribute must be a comma separated list of float values.

```
<attribute name="Type" value="all"/>
<attribute name="Weights" value="0.2,0.3,0.5"/>
```

Listing 4.6: Possible WeightedAverageFilter attributes

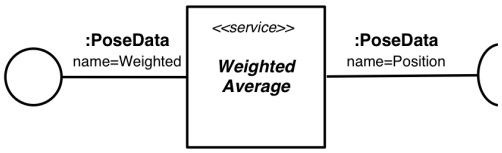


Figure 4.8: WeightedAverage Service

4.2 Example

The fig. 4.9 shows a network of DWARF services. The TOC and the BubbscherBalkenFilter service were implemented using the PluginFramework .

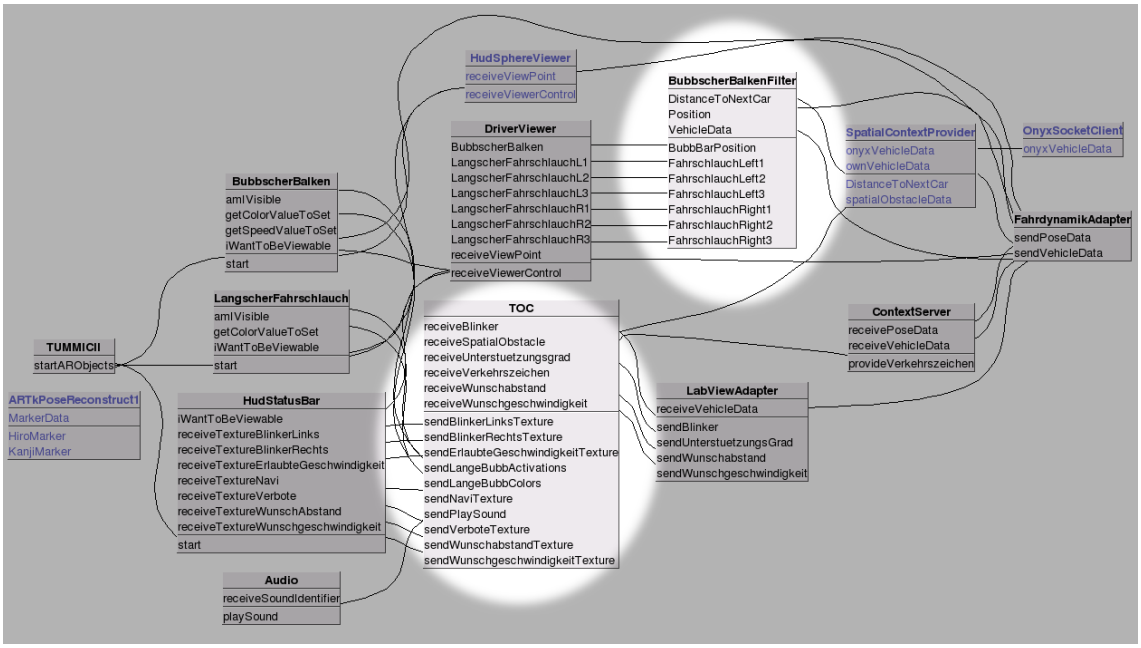


Figure 4.9: Usage of two plugins in a complex network

5 Howto

This chapter gives an detailed description how to develop, test and use new plugins for this plugin framework.

5.1 Write a Plugin

The first thing do to, is to decide where the new plugin should reside. The default location is the package `de.tum.in.dwarf.PluginService.plugins`. If you decide for an other location, you have to determine this in the plugin servicedescription with a full qualified classname. A convention for depositing plugins should be used. Base spatial transformations should reside in this package, while e.g. 3Dto2D transformation, which are often required for UI visualization could be placed in a package `de.in.dwarf.PluginService.plugins.ui`. This should enhance the reusability of plugins.

5.1.1 Plugin Class

This class contains the implementation of the algorithm and the communication. This class has to be declared in the service description as seen in listing 5.6. Of course the implementation can be split up to several classes, but the communication methods must reside in the class declared.

5.1.1.1 Input

For each need attribute of your service, you have to implement a `public void` method starting with `setNeedData` followed by the need name. This method must have only one argument of that type, which is defined in the need description. Listing 5.1 defines a need for `PoseData` with the name `Ping`. The related Java code 5.2 implements a method named `setNeedDataPing` with one argument of type `PoseData`.

```
<need type="PoseData" name="Ping" minInstances="1" maxInstances="1">
    <connector protocol="PushConsumer"/>
</need>
```

Listing 5.1: Need Description

```
public void setNeedDataPing(PoseData pingPoseData){}
```

Listing 5.2: Java code for a need

5.1.1.2 Output

To send events out of plugins, you have to use the **`void`** `sendData(String abilityName, IDLEntity object)` method of the plugin superclass. The value of the variable `abilityName` has to be the name of the ability defined in the ability description. The argument named `object` has to contain the data, which should be sent out. It has to be of the type `IDLEntity` which is the declared by every predefined CORBA struct.

In listing 5.4 you can see, how to send events for an ability defined in service description 5.3.

```
<ability type="PoseData" name="Position">
   <connector protocol="PushSupplier"/>
</ability>
```

Listing 5.3: Ability Description

```
  Pose pingPose = new Pose();
  sendData("Ping", pingPose.getIDLEntity());
```

Listing 5.4: Java code for sending events

You do not have to write your own handler, even not for event types defined by yourself. This work is done by the dwarfs Java common package.

5.1.1.3 AttributesHaveChanged

If you want to be aware of changing attributes, you can overwrite the method **`public void`** `attributesHaveChanged(Attributes attributes)`, whose implementation is empty.

The plugin superclass calls this methods each time the attributes of the service description are changing. Even at the initialization of the service this method is called. So this is a convenient way to gather the attributes, set in the service description.

5.1.1.4 ContextSwitch

To use the ContextSwitch [17] facilities only a new need for ContextSwitch has to be defined in the service description. The attributes needed to configure the ContextSwitch need are described in [17].

5.1.2 Service Description

The service descriptions of plugins do not differ much to other service descriptions.

The start command has to be set to `PluginService.jar -Dservicename=` followed by the servicename, as you can see in fig.5.5. Notice that the servicename in the startup attribute has to be identical to the servcename attribute.

```
<service name="howto2" startAutomatically="false" startOnDemand="true"
   stopOnNoUse="false" startCommand="PluginService.jar␣-DserviceName=
   howto2">
```

Listing 5.5: Startup Command

Furthermore you have to specify which plugin class should be used for that service. This is done with an service attribute named `PluginClass`. If the class is located in the default plugin package `de.tum.in.dwarf.PluginService.plugins` you just have to give the classname 5.6. But if your plugin is somewhere else you have to give the full qualified classname 5.7.

```
    <attribute name="PluginClass" value="PingPlugin"/>
```

Listing 5.6: PluginClass Attribute default location

```
    <attribute name="PluginClass" value="my.special.place.PingPlugin"/>
```

Listing 5.7: PluginClass Attribute special location

5.2 Use Plugins

The plugin is written, now we want to see in action.

5.2.1 Start Plugins

There two different ways to use plugins. One way is to start all plugins in the same process. The other more simple way runs all plugins in different processes.

In more than one process Running plugin services in different processes is easy. Just run `java -jar PluginService -DserviceName=yourServiceName` for each service. If you have a proper service description, the service should start up in its own process.

In one process If you want to start more than one plugin service in one process, you have to use the PluginLauncher service. In listing 3.2 you can see an example service description for the PluginLauncher. The attribute `<attribute name="Start" value="ping,pong"/>` means that the plugin services with the servicename "ping" and "pong" should be started by the PluginLauncher in the same process.

Notice that the PluginLauncher has to run before the Plugins are started, because the PluginLauncher registers the special startup routine at the servicemanager. Otherwise the servicemanager will start a Java virtualmachine for each plugin and the plugins run in different processes.

5.2.2 Configure Services

Services should be configurable, to reach a wider field of work.

Attributes Configuration can be done with attributes in the servicedescription. By editing the servicedescription the service can be customized. These attributes can also be changed at the runtime. If the plugin implements the `attributesHaveChanged` method, the plugin respond to that change.

Events An other way for some configuration is to send the new configuration data via events. This method requires of course an own need for each data type and a own `setNeedData` method. But it has the advantage, that it can be configured by an other service in a convenient way.

5.3 Test Plugins

Testing services in dwarf can be annoying, because you have to run a servicemanager, DIVE , your test environment and of course your test service. This a lot work and costs a lot of time. A more convenient way to test java code is the `junit` framework. More information about the junit framework and detailed documentation can be found on the project webpage[7].

5.3.1 Write a Test Case

To test plugins without starting dwarfs middleware, the `MockEventSender`, `MockAbilityDescription` and the `MockAttribute` classes have been introduced.

The `sendData` method of the MockEventSender stores the outgoing events in a FIFO queue. So the test case can get information about the sender. Therefor three methods have been implemented.

get() returns the first object in the sender queue and removes it

isEmpty() returns true if no objects are in the queue

size() returns the size of the sender queue

As seen in listing 5.8, you have to switch the plugin into testing mode, which can be done with the `switchOnTesting()` method. The `registerAbility` method of the plugin object returns a MockEventSender.

Because no xml servicedescription is read, you have also to setup the abilitiyDescription by yourself. Because we don't want to use the middleware, you have to use the `MockAbilityDescription` class. This can also be seen in listing 5.8.

```
    RotationFilter m_plugin;
    MockEventSender m_sender;
    Pose m_pose = null;

    protected void setUp() throws Exception {
        super.setUp();
        m_plugin = new RotationFilter();
        m_plugin.switchOnTesting();

        MockAbilityDescription ability = new MockAbilityDescription("
            Rotated","PoseData");
        m_sender = (MockEventSender) m_plugin.registerAbility(ability,
            null);

        m_pose = new Pose(new double[] {1.0, 2.0, 30.0});
    }
```

Listing 5.8: junit setUp code

In listing 5.9 a typical test routine is shown. Such code should reside in a junit TestCase class which is in the same package as the plugin and should be named equally to the plugin, but with "`Test`" as prefix.

```
 public void testNothingConfigured() throws WrongTypeException{

   assertTrue(m_sender.isEmpty());
   m_plugin.setNeedDataPosition((PoseData) m_pose.getIDLEntity());
   assertTrue(m_sender.size() == 1);

   PoseData result = (PoseData) m_sender.get();

   assertTrue(m_pose.equals(new Pose(result)));
}
```

Listing 5.9: junit test method

6 Conclusion

In this chapter I discuss the results of this project. I present also some performance tests of the framework and what features can be implemented in the future.

6.1 Performance

All performance tests in the following chapter have been done on a Celeron 2.94GHz Linux Debian Sarge machine with OpenORB 1.3.0.

6.1.1 Event online time

The interactivity with an Augmented Realıty system should happen in real time [1]. So the delay caused by data processing should be as small as possible. Because the data processing is done here in several services, the communication between these services becomes an issue.

To measure the time the events are traveling from an ability to a need, the ping and the pong services were written 6.1. The ping service writes the actual time in the event and sends it to the pong service. Immediately after receiving the event the pong service takes the system time and calculates the difference of the time stamps. This kind measurement work only, if both services running on the same machine.

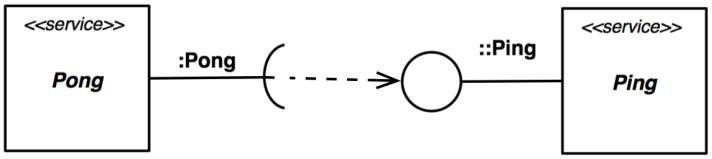


Figure 6.1: Ping and Pong Service

6.1.1.1 Test of Multithreaded Services

The results can be seen in Tab. 6.1. An interesting result is the significant lagging, if an other service is present. When using one process the ping time is 1.25 times slower and when using two processes even 1.42 times slower. Also the statistical spread increases dramatically. If

	average	median	standard deviation
One process without DIVE	2.64 ms	2 ms	2.29 ms
Two process without DIVE	2.76 ms	2 ms	2.33 ms
One process with DIVE	3.30 ms	2 ms	6.71 ms
Two process with DIVE	3.92 ms	2 ms	9.66 ms

Table 6.1: Table of ping times

using multiple processes the spread rises by a factor of 4.1 and if using a single process it rises by a factor of 2.9.

Single process without DIVE This test series measures the ping time, if both services running in the same process.

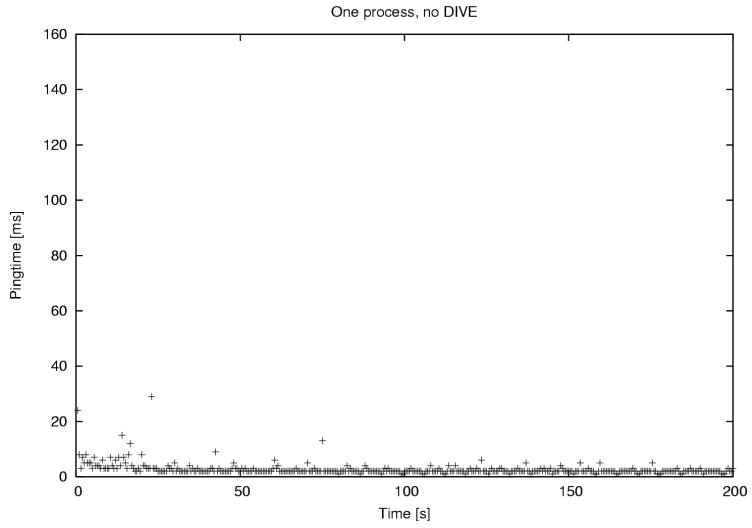


Figure 6.2: One process without DIVE

Single process with DIVE In this test both services are started in the same process. The DIVE service is also started.

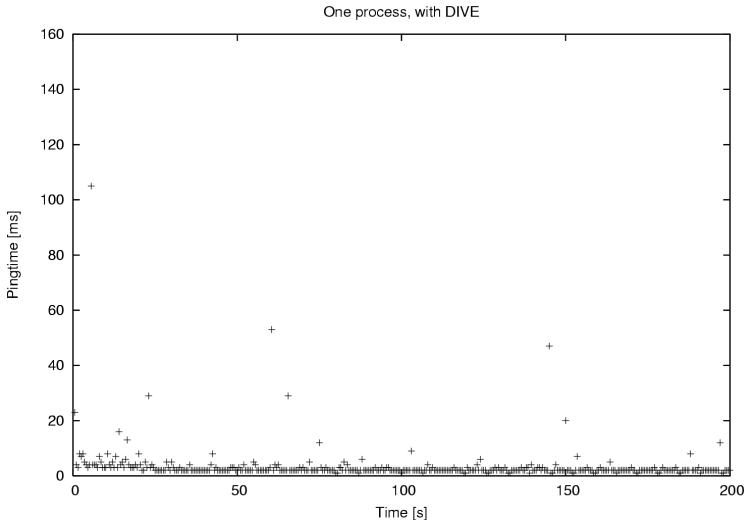


Figure 6.3: One process with DIVE

Two Processes without DIVE Now the ping and the pong services running in two separate processes.

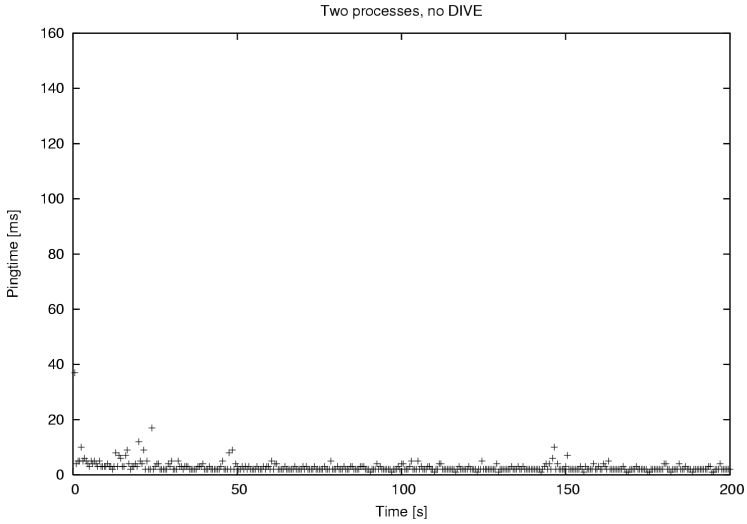


Figure 6.4: Two Processes without DIVE

Two Processes with DIVE The ping and the pong services running in different processes and DIVE is started.

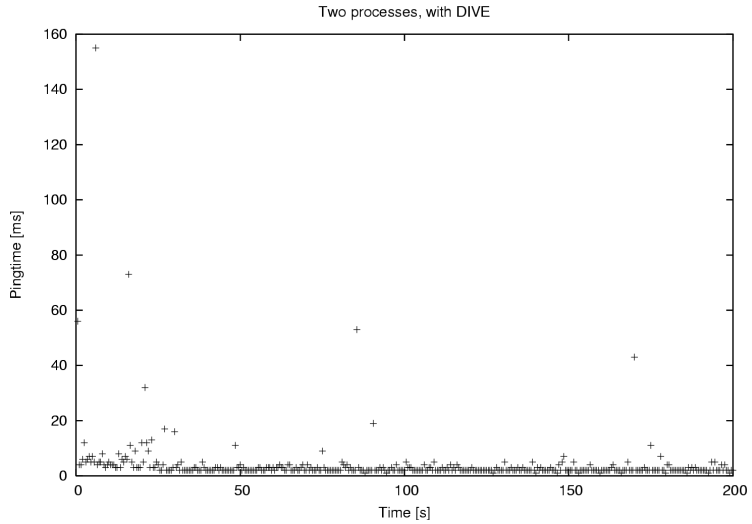


Figure 6.5: Two Processes with DIVE

6.1.1.2 Test at different frequencies

Because a data rate of two events per second is not very realistic in an Augmented Reality system, so tests with various data rates have been done. Because logging events on the console seems to slow down services, two test series was done. One with logging and the other without. The DWARF framework is using the Log4J framework [8] for the logging facilities.

```
m_log.info("Received the pong #" + pingID + ". time: " + diff+ "ms.");}
```

Listing 6.1: Example for logging with Log4j

	With Log4J		Without Log4J	
	Average	Std. Deviation	Average	Std. Deviation
2Hz	3.0175m	4.33027ms	3.08ms	4.4438ms
5Hz	3.025ms	4.37381ms	3.0725ms	4.62372ms
8Hz	3.0775ms	4.40781ms	3.265ms	4.21535ms
16Hz	3.085ms	2.96132ms	3.025ms	3.67608ms
31Hz	3.5925ms	5.77015ms	3.105ms	4.21526ms
63Hz	12.925ms	25.2517ms	3.2325ms	5.33419ms
125Hz	15.4325ms	25.9683ms	3.52ms	6.01719ms

Table 6.2: Table of ping times with various frequencies

As can seen in fig. 6.2 the average ping time is increasing dramatically (6.6), when using logging with at hight frequencies. Also the standard deviation is raising (6.7), when logging more than 31 messages per second.

But logging less than 31 messages per second does not affect the ping times at all.

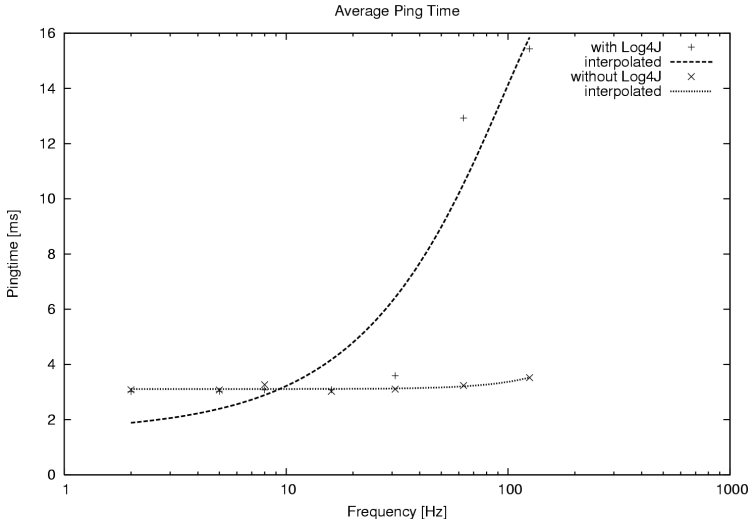


Figure 6.6: Average Ping Times at different frequencies

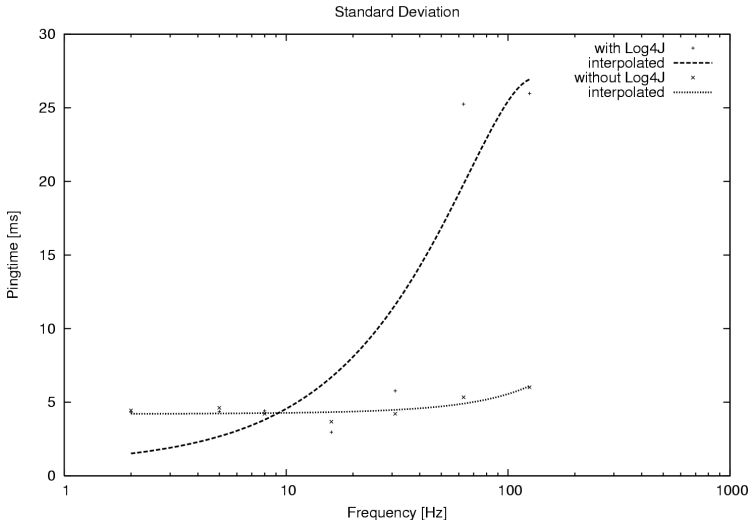


Figure 6.7: Standard Deviation at different frequencies

6.1.2 Java Reflection

Because the pluginframework uses a lot of Java's reflection capabilities, I did some performance tests. The test code calls a method 1000000 times by direct invocation. The same amount of calls were done with Java's method invocation method of the reflection package. The called method does nothing and returns immediately.

	Time per call	
Type of calling	`Java 1.4.2_05`	`Java 1.5.0_01`
$1 * 10^6$ direct calls	6ns	5ns
$1 * 10^6$ reflection calls	125ns	124ns

Table 6.3: Table of invocation times

Table 6.3 shows, that method calls with reflection are about 21 times slower than direct calls, regardless of which version of Java's virtualmachine in is used.

Method calls with reflection at runtime are only used if a event is received and the plugin superclass has to invoke the corresponding method in the plugin. So this happens not too often. In normal cases this happens not too often. In normal case about 30 up to 100 events per second coming in. So this small overhead cost only little CPU time and does not affect the ping times of events.

6.2 Discussion

As I started this project, the task was to develop a filter framework, which helps developing new filters for the DWARF framework. But during work the task changed towards a more general approach – the PluginFramework .

6.2.1 One Service – One Plugin

The first idea was to implement more than one filter in a service. This has not been implemented, because of at least three reasons.

1. Due we want connect more than one plugins to a network, we need to describe this network. With the previous approach it would be possible to have more than one plugin in a service. So it would be necessary to connect plugins within a service and plugins in different services. Connecting plugins in different services can be described with service descriptions as usual in DWARF [11]. But the description of connections between the plugins in the same service, would require a new language or a reimplementation of DWARF 's builtin functionality.

   By placing each plugin in a service of its own, the infrastructure of DWARF can be reused.

2. The namespace. In DWARF need and ability names have to be unique. But using more than one filter this could cause troubles, especially when using more than one plugin of the same type.

3. Using more than one plugin in the same service the service description becomes bigger and more complex. And in the visualization of that services in DIVE you could hardly differ between the several plugins. But splitting up into several services can also become an issue (see 6.4.2), but this seems to be the less evil.

6.2.2 General

The approach not to focus mainly on filters allows to develop more general services with this framework. Nearly each service which uses event communication can be implemented. For example a event recorder and/or player for any type of event.

6.2.3 Usability

It is very convenient to set up and test a service with the PluginFramework . Only a few lines of code are enough. Even a developer, which is not skilled in CORBA can implement a DWARF service and can concentrate on things that really matter.

6.3 Lessons learned

I learned a lot during this work. First of all I learned a lot of Java programming. I got used to write junit tests and got familiar with Java's reflection capabilities.

An other big lesson learned was developing with the great Eclipse IDE. And of course I learned much about CORBA, but I think I only see the tip of that huge iceberg.

6.4 Future work

Things can always be improved.

6.4.1 Inprocess autostart

Services in DWARF can be started on demand by the servicemanager. If the servicemanager finds a service, which is implemented in java, it starts a new Java virtual machine for this service. But as seen in figure 6.1 it is worth to start more than one service in the same process. This can be done by using the `LoadService` class, which implements the `SvcLoad` interface. The PluginLauncher service uses this class to register itself as a callback. So all service registered by the PluginLauncher are started by the PluginLauncher and running in the same process space.

This works fine, but it is a complex job to set it up. The best place to implement such inprocess starting, would be the servicemanager. There could be a new attribute in the service description e.g. `inProcess=''true''`. Then all services tagged with the "inProcess" attribute would be started in the same process by the servicemanager.

6.4.2 Huge networks

At the moment each plugin is a service of its own, so it can be visualized in DWARF . This advantage can turnover if the networks grows. This is not a problem of the filter framework but of DIVE , although the filter framework make the problem even bigger.

So it would be nice hiding the filter framework in DIVE or to replace it by a new symbol, e.g. a triangle with all outgoing and incoming connections.

6.4.3 Method Calls

Communication with plugins can only be done with events. This is enough for dealing with continuous data streams. To use the pluginframework beyond this purpose, it would be nice if the plugin framework could handle `ObjrefExporter` and `ObjrefImporter`. So the pluginservice could be used in a wider field of application.

6.4.4 Graphical User Interface

To create a network of pluginservices or of any other services, the description of the needs and abilities have to be augmented with predicates and attributes, to assure the right abilities are connected to the right needs. Some services can also be configured with attributes in the service description. All of this configuration has to be done with an editor or other even more sophisticated tools like eclipse XMLBuddy[18]. This can be error-prone.

It would be nice to have a GUI tool which allows to build and configure a network. This would be a convenient way to build such networks and a save way to avoid typos and other misconfiguration.

6.4.5 Velocity Datatype

The speed filter (4.1.2.1) uses two datatypes not in the intended way. The `InputDataAnalogUnlimited` datatyp was designed to configure other services and the `PoseData` was introduced to send positions of objects and not information direction.

So it would be nice to have a special datatype concerning velocity. Due the discussion about the content of `VelocityData`is not yet finished, I make a proposal, which can be seen in listing 6.2. This datatype can hold beside the plain velocity also the direction of the target object. The other entries are the same as in the PoseData datatype.

So it would be nice to have a special datatype concerning velocity. Due the discussion about the content of `VelocityData`is not yet finished, I make a proposal, which can be seen in listing 6.2.

This datatype can hold beside the plain velocity also the direction of the target object.

```
struct VelocityData {
  string source;       // describing the source id
  string target;       // describing the target id
  bool hasVelocity;    // true if a velocity is set
```

```
  bool hasDirection;   // true if there is direction
  double velocity;     // velocity
  double[3] direction; // direction of the moving object
  Time timeStamp;      // the time stamp of the event
  double confidence;   // the confidence level
  double timeError;    // error of the time
}
```

Listing 6.2: Definition of the VelocityData datatype

The direction vector $d = (x, y, z)^t$ should be normalized to the time of one second, so the 2-norm of that vector should be the same as the speed v.

$$\left\| \begin{pmatrix} x \\ y \\ z \end{pmatrix} \right\| = v$$

E.g. a target object moved the last 0.5s from point $a := (1, 2, 3)^t$ to point $b := (2, 3, 5)^t$. So the target moved the in direction $c := b - a = (1, 1, 2)^t$. And it is moving with the speed of $v = \frac{\|c\|}{t} = \frac{2m}{0.5s} = 4\frac{m}{s}$. Because the direction vector of the datatype `VelocityData` should have the length v, it has to be scaled to $d := \frac{c}{\|c\|}v$.

Of course it is a disadvantage to have redundant data, speed and the direction norm. But so the user can decide either to use only the speed or to use only the direction.

The other entries in the `VelocityData` type, like timeStamp or confidence, have analog meanings as in the `PoseData` datatype.

Bibliography

[1] R. AZUMA, *A Survey of Augmented Reality*, in Teleoperators and Virtual Environments, Vol. 6, Issue 4, 1997, pp. 335–385.

[2] K. BECK and E. GAMMA, *Test Infected – Programmers Love Writing Tests*. `http://members.pingnet.ch/gamma/junit.htm`.

[3] *DWARF Homepage*. `http://wwwbruegge.in.tum.de/DWARF/WebHome`.

[4] P. FRÖHLICH and J. LINK, *Kaffeeprobe JUnit - Entwickeln und Testen in Java*, iX, 3 (2001), p. 108.

[5] E. GAMMA, R. HELM, R. JOHNSON, and J. VLISSIDES, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison Wesley, 2004.

[6] M. GEIPEL, *Run-time Development and Configuration of Dynamic Service Networks*. `http://wwwbruegge.in.tum.de/pub/DWARF/SepGeipel/SEP.pdf`, June 2004.

[7] *JUnit Homepage*. `http://www.junit.org`.

[8] *Log4j*. `http://logging.apache.org/log4j/docs/`.

[9] T. MACKINNON, S. FREEMAN, and P. CRAIG, *Endo-Testing: Unit Testing with Mock Objects*, in Extreme Programming Examined, G. Succi and M. Marchesi, eds., Addison-Wesley, 2001. `http://www.connextra.com/aboutUs/mockobjects.pdf`.

[10] A. MACWILLIAMS, *DWARF – Using Ad-Hoc Services for Mobile Augmented Reality Systems*, Master's thesis, Technische Universität München, Institut für Informatik, Feb. 2001.

[11] A. MACWILLIAMS, T. REICHER, and B. BRÜGGE, *Decentralized Coordination of Distributed Interdependent Services*, in IEEE Distributed Systems Online – Middleware '03 Work in Progress Papers, Rio de Janeiro, Brazil, June 2003.

[12] V. NOVAK, C. SANDOR, and G. KLINKER, *An AR Workbench for Experimenting with Attentive User Interfaces*, in Proc. of IEEE and ACM International Symposium on Mixed and Augmented Reality, Arlington, VA, USA, Nov. 2004. To appear.

[13] D. PUSTKA, *Visualizing Distributed Systems of Dynamically Cooperating Services*. `wwwbruegge.in.tum.de/pub/DWARF/OberSeminar/SEPPustka.pdf`, mar 2003.

[14] *Mathworld*. `http://mathworld.wolfram.com`.

[15] G. REITHMAYR, *On Software Design for Augmented Reality*, PhD thesis, Technische Universität Wien, Institut 188 für Software Technologie und Interaktive Systeme, Mar. 2004.

[16] CAR *Project Homepage*. `http://wwwbruegge.in.tum.de/projects/lehrstuhl/twiki/bin/view/DWARF/ProjectBar`.

[17] M. WAGNER and G. KLINKER, *An Architecture for Distributed Spatial Configuration of Context Aware Applications*, in 2nd International Conference on Mobile and Ubiquitous Multimedia, Norrköping, Sweden, 2003.

[18] *XMLBuddy*. `http://xmlbuddy.com/`.