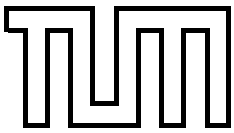
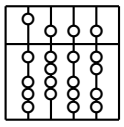


Technische Universität München
Fakultät für Informatik

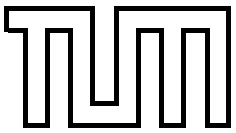


Diplomarbeit

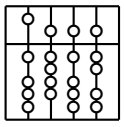
Data Management for Augmented Reality Applications

ARCHIE: Augmented Reality Collaborative Home Improvement Environment

Marcus Tönnis



Technische Universität München
Fakultät für Informatik



Diplomarbeit

Data Management for Augmented Reality Applications

ARCHIE: Augmented Reality Collaborative Home Improvement Environment

Marcus Tönnis

Aufgabenstellerin:	Prof. Gudrun Klinker, Ph.D.
Betreuer:	Dipl.-Inf. Martin Bauer
Abgabedatum:	15. Juli 2003

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

München, den 15. Juli 2003 Marcus Tönnis

Zusammenfassung

Erweiterte Realität (Augmented Reality, AR) ist eine neue Technologie, die versucht, reale und virtuelle Umgebungen zu kombinieren.
Gemeinhin werden Brillen mit eingebauten Computerdisplays benutzt um eine visuelle Erweiterung der Umgebung des Benutzers zu erreichen. In das Display der Brillen, die Head Mounted Displays genannt werden, können virtuelle Objekte projiziert werden, die für den Benutzer ortsfest erscheinen.

Das am Lehrstuhl für Angewandte Softwaretechnik der Technischen Universität München angesiedelte Projekt DWARF versucht, Methoden des Software Engineering zu benutzen, um durch wiederverwendbare Komponenten die prototypische Implementierung neuer Komponenten zu beschleunigen.

DWARF besteht aus einer Sammlung von Softwarediensten, die auf mobiler verteilter Hardware agieren und über drahtlose oder fest verbundene Netzwerke miteinander kommunizieren können. Diese Kommunikation erlaubt, personalisierte Komponenten mit eingebetteten Diensten am Körper mit sich zu führen, während Dienste des Umfeldes intelligente Umgebungen bereitstellen können. Die Dienste erkennen sich gegenseitig und können dynamisch kooperieren, um gewünschte Funktionalität zu erreichen, die für Augmented Reality Anwendungen gewünscht ist.

Ein kritisches Problem der Erweiterten Relität ist das große Datenaufkommen, das in einem verteilten System verwaltet werden muß. Diese Daten müssen zuverlässig an die Dienste zugestellt werden, die dem Benutzer den Zugriff ermöglichen. Somit müssen diese Daten im gesamten System in einem konsistenten Zustand gehalten werden.

Für Teilbereiche der Verwaltung dieser Daten können Datenbanken verwendet werden um Persistenz und effizientes Management zu gewährleisten.

Diese Diplomarbeit beschäftigt sich mit Datenmanagement in verteilten Augmented Reality Systemen. In dieser Diplomarbeit wird ein neuer Ansatz vorgestellt, der Benutzern transparenten Zugriff auf alle benötigten Daten liefert. Das erstellte Design wurde prototypisch implementiert und in der ARCHIE Anwendung getestet. Bei ARCHIE handelt es sich um ein DWARF Projekt, das Architekten durch Erweiterte Relität bei ihrer Entwicklungsarbeit unterstützen soll.
Nach weiteren Entwicklungsarbeiten könnte dadurch eine generell benutzbare Anzahl an Diensten geschaffen werden, die zu neuen Anwendungsgebieten der Erweiterten Realität führen können.

Abstract

Augmented Reality is a new technology that combines real and virtual environments.
In general glasses with attached displays are used to produce visual augmentations of the user's environment. These so-called head mounted displays are able to project virtual objects that are registered with the real world of the user.

The DWARF project is conducted at the Chair for Applied Software Engineering of the Technische Universität München and tries to use methods from software engineering to reuse components for faster prototypical implementation of new components.

DWARF consists of a set of software services that act on mobile distributed hardware and can communicate via wireless or wired networks. This communication allows to carry personalized mobile devices with embedded services, while intelligent environments provide location based services. The services discover each other and dynamically cooperate to provide desired functionality, which is required for Augmented Reality applications.

A critical issue of Augmented Reality is the large quantity of data, which must be managed in a distributed system. This data must be reliably delivered to services that provide user access on that data. The handled data must be in a consistent state allover the system.

Database systems can be used for parts of the management of data to guarantee persistence and efficient handling.

This thesis deals with data management for distributed Augmented Reality systems. The main contribution of this thesis is a novel approach of dynamic services, that give the user transparent access to all necessary data. The developed design has prototypically been implemented and tested within the ARCHIE application. ARCHIE states a project to support architectural modeling by Augmented Reality.
By further development a set of general usable services can be realized that allows to use Augmented Reality in new areas.

Preface

Purpose of this Document

This thesis was written as a Diplomarbeit, which is adequate to a Master's thesis, at Technische Universistät München's chair for Applied Software Engineering.
During September 2002 and May 2003 five other Computer Science master students and I designed, developed and presented new components for our research project DWARF and also developed a new application called ARCHIEon top of the framework that demonstrates the new included features.
Within this thesis I would like to explain the issues behind my work, show relations to other research projects, develop and document the DWARF specific implementation and finally show its future implications.

Target Audience

The thesis in hand addresses various different audiences. This section shall provide an overview of interesting parts.

Augmented Reality Researchers and other Computer Scientists should read chapters 2 to get an overview about our research project in Munich, DWARF. Chapter 3, 6 and 7 illustrate the issues and supplied solutions for data management in mobile Augmented Reality systems. Chapter 9 shows how these components can be used.

Future Developers might read chapter 3 to compare their research topics and chapters 5, 6, 7 and 8 to see how to extend the new components. Chapter 9 illustrates how to build applications with DWARFand chapter 10 provides ideas for future work.

General Readers might not be familiar with Augmented Realityand therefore should read chapter 1 for a general description of the problem domain. It introduces the concepts of Augmented Reality. Chapter 2 illustrate our research platform DWARFin an understandable way, even if some more detailed phrases are included.

The DWARF Team might already be familiar with chapter 1 and 2 which describe the concepts of Augmented Reality and DWARF. Chapter 3 might provide further information about data management issues in the Augmented Realitydomain. Chapters 5, 7 and 8 guide through the development process of new data managing relevant services. Finally chapter 9 might be interesting for reusing the new components in future applications and chapter 10 could provide further ideas on research topics.

Timeline

The thesis documents various phases I was involved in the extension of the DWARF framework.
Chapter 1 gives a brief introduction to the field of Augmented Realityand reflects the time I needed to get familiar with that research area: July 2002 until September 2002
Chapter 2 documents my first experiences with DWARFand introduces the demonstration application we planned in the time between July 2002 and November 2002.
Chapters 3, 5, 6, 7 and 8 reflect the main focus of my work, developing and building the data managing components necessary for a framework like DWARF. This activities filled most of the time between December 2002 and May 2003.
Chapters 4 and 5 explain the exploration of the world of computer science for related work and technologies. I performed these activities continously all over the time.
The last chapters conclude the thesis with information about reusing the produced components in new applications and discussions of the projects results inclusive topics for future work.

Electronic Version

If you are a developer and use DWARFfor support on your topic and you are currently reading a paper version of this thesis, be informed that also a electronic version in Portable Document Format (PDF) exists. This one includes cross references for all contents, figures and citations.

Acknowledgments

This thesis would never have been possible to be written without the restless efforts on many people.
First, I wish to thank the members of my ARCHIE team - Felix Löw, Otmar Hilliges, Chris Kulas, Johannes Wöhler and Bernhard Zaun and our advisors, Martin Bauer, Asa MacWilliams, Christian Sandor and Martin Wagner.
Second I would like to thank Gudrun Klinker, Bernd Brügge and Thomas Reicher who opened up an interesting research field, Augmented Reality in intelligent environments.
Special thanks go out to my friends, especially Bernhard, who had to endure my peculiarities over this long time.
Finally I would like to thank my family, who always helped me to keep my feet on the ground, even during some of the harder times.

Garching, July 2003

Marcus Tönnis (marcus@toennis.de)

Overview

Contents

List of Figures

1 Introduction

Introduction to Augmented Reality, the DWARF Framework and the ARCHIE Project. Scope and Structure of this Thesis

---

In this chapter I introduce the general background of my thesis, even to readers who are not yet familiar with the concepts of Augmented Reality.
The goal of my thesis was to develop components for managing data in Augmented Reality environments. These components, called services have been built in context of the DWARF framework, a framework for mobile Augmented Reality systems. The issues that lead to this thesis and a brief overview about the named DWARF framework are explained in the following sections.

1.1 Motivation

Since the first development of computers in the 20th century we have become familiar with exponential growth in this technology. Moore's law that states, that every 18 months computing power doubles at a constant price, still applies today. This enabled many companies to provide additional peripheral devices to computers such as graphical output devices and engines to handle acoustic input and output. The more powerful computers became, the more applications have been developed to use the newest functionality. A good example are architectural ones that facilitate their users to perform their plannings efficiently on computers.
Also network technology that arose in the 60ies of the last century became mature. The Internet as we know it today provides marvelous amounts of information and services to maintain and access them. Every user of the Internet can get access to information about topics of interest, people and products.
As wireless networks arose in the last decade of the last century, computers became versatile by always keeping network connection available. Everyone knows the rapid increasing amount of mobile telephones that provide more functionality than just talking to other people. By just typing some buttons, any user can get information about flight departures or traffic jams.

But even if new possibilities arose by the new technologies, there are still some limitations.

Abstraction of the User Interface The increasing amount of functionality is directly associated with a need for users to acclimatize themselves to the interfaces, that technical products provide. Even if great advantages have been done in the last years, users of the new technologies still have to make abstractions in their normal behavior.
Remembering the architectural applications announced earlier, architects know about that issue. For architects in their work, interaction with computers is in common done by mice, kinds of touchpads and keyboards. Everyone who ever tried to use buttons and sliders to move around a three dimensional object like a planned building, that is visible on his screen, knows about the problems.
Wouldn't it be easier to just move the head around a direct visible three dimensional representation on a normal workdesk, as we know it when we look at a sculpture from every perspective?

Combining Things and Associated Information Based on the assumption that real people or things are often associated with additional information, one can see that this information is often located anywhere, but not directly besides the object is corresponds.
Imaging, being on holiday in another city, one can see a sculpture. How to get information about the producer and the intention he had when building it?
Modern wireless communication technologies can supply this information to the tourist, but there is still the problem, that just the existence of such communication does not provide any possibilities for people to access them. This can be solved by a mobile device, the tourist is carrying with him. This one can access and visualize the available data, if the tourists current environment is equipped with computers providing services that supply this data. We call such environments *ubiquitous computing* environments. The tourist would finally be fully pleased, if the information about the sculpture would not appear on his mobile devices' display, but directly besides the sculpture. We call this enriching of the real world *Augmented Reality*.

1.2 What is Augmented Reality?

At the research group at Technische Universität München, we have tried to bring information and things together in combination with a intuitive user interaction. For this, we applied software engineering principles to the problem domain of Augmented Reality to create a reusable testbed for creating Augmented Reality applications.

Augmented Reality For this the new technology of Augmented Reality (AR) was used. This section is intended to give a short introduction into the problem domain of Augmented Reality.
Many readers will have heard about Virtual Reality (VR). In VR systems, the user is completely surrounded by a virtual environment. As this scene is displayed in a kind, so that the real world around the user is invisible, the environment of the user must, at least for minimizing risk of injury, be restricted to a certain area. Common used VR applications include flight simulators to train pilots or sophisticated computer games.

The main essence is that users are taken out of the real world and are put in a virtual world. Augmented Reality aims at leaving the user in the real world and only to *augment* his environment with virtual elements. Such elements may be visible objects, but also sound can be such an element. In the rest of my thesis, I restrict the meaning of augmented elements to virtual, but visible objects.
A specification of the core properties of Augmented Reality systems can be found by Azuma [26] who stated them in 1997. He defines:

Augmented Reality are systems that have the following three characteristics:

1. *They combine reality and virtuality*
2. *They are interactive in real time*
3. *and are registered in 3-D*

Although this is a very wide definition, it covers the main tasks that must be realized by any Augmented Reality system. Over the last years, as Augmented Reality systems became practicable, most research groups focused on visual augmentation. This can be seen in the proceedings of workshops and symposiums on Augmented Reality: IWAR 1999 [103], ISAR 2000 [104], ISAR 2001 [105] and ISMAR 2002 [106].

A fine example for Augmented Reality can be seen in figure 1.1.

Figure 1.1: An example of visual Augmented Reality: A person is augmented to the real world. Note the shadows! (Courtesy of Loria [11])

Visual Augmentation In at least the near future, there will be no possibility to augment virtual objects directly into the real world. This would require for instance electromagnetic fields holding some kind of haze, that can build the shape of the object to be augmented.
Currently, visual augmentation works in two ways:

Video see-through Images of the real world are recorded by a camera from the user's point of view. Some analysis is performed on the received images and some parts of the video image are changed to provide the additional virtual objects. The resulting video stream is displayed in a special kind of glasses, called *head mounted display* (HMD), that positions two small screens directly in front of the wearing person.

Optical see-through As with video see-through Augmented Reality, the user of an optical see-through Augmented Reality system must wear glasses, but in contrast to the glasses of video see-through, these glasses are transparent and only add virtual information at specific regions to the real world directly seen by the user. He directly sees his environment. To use this way, it is necessary to determine the position of the user. By the user's position, visual augmentation are computed, that fit for his current position.

1.3 The DWARF Framework and the ARCHIE Project

Now, as we know about Augmented Reality, it is useful to introduce the DWARF research group. At the chair for Applied Software Engineering at Technische Universität München a framework for mobile Augmented Reality applications is developed since 2000, called DWARF. For more details on the DWARF framework, please refer chapter 2.
In the time between it's initial version and July 2002 several applications have been built on top of the framework. Each one of this applications provided new reusable components to the framework, but still there are some parts missing.
For that reason the ARCHIE project was initiated in summer of 2002, that again should provide new reusable components to the framework. For more details on the ARCHIE project, please again refer chapter 2.

1.4 Scope of this Thesis

This thesis deals with adding data management functionality to the DWARF framework. Until the beginning of this thesis, the DWARF system did not provide any concepts for a well structured storage of all data needed for Augmented Reality applications in Ubiquitous Computing environments. But Augmented Reality systems have to deal with a lot of data. Virtual scenes must be rendered to output devices as well as applications must be configured to users and environmental settings.
Up to now, every application realized on the framework had to deal with that topic for itself. The central parts of the applications provided no reusable components for further ones, because virtual scenes were placed inside the applications and configuration parameters were

also set there.
In my thesis, I added components to DWARF framework that now allow application scenes that are configurable as well as objects to get manipulated inside scenes in various ways. Also configuration topics of framework services are dealt with.

1.5 Structure of the Document

This thesis has two kinds of structure. One gives all chapters in order of their appearance while the other explains our DWARF framework in different views on the system.

1.5.1 Outline of the Thesis

I first I discuss the DWARF and ARCHIE projects in order in chapter two, to set the necessary background information for explaining the requirements for data management in Augmented Reality in chapter three. This chapter also collects the requirements for ARCHIE as a application on top of the framework.
To find a solution fulfilling all requirements, I describe interesting work related of other research groups to my thesis' subject in chapter four. The others groups approaches are discussed in regard to our DWARF framework.
As DWARF applications are inherently distributed, I inspect current technology in storing data in distributed systems.
Now, as a lot of information is collected, in chapter six, I provide a brief overview about DWARF services that are adjacent to data management components and therefore were reused in the development of the ARCHIE application.
With the knowledge about the DWARF services I give a detailed overview about the design of the new services for data management. Also the design of some more services which were necessary to develop for either providing the required functionality of the planned ARCHIE scenarios or for testing purposes is explained in chapter seven.
Chapter eight focuses on the detailed object design and implementation of the services and gives information about the state of implementation.
In advance chapter nine describes how new applications can be built by use of the new services and what modifications in configurations have to be done.
Finally chapter ten concludes my thesis with results of the work. Learned lessons are explained and future work is announced.

1.5.2 Different Points of View

Through this thesis I also introduce three different views onto the DWARF framework.
The user's point of view explains the DWARF framework as recognized by a common user. This one is introduced in chapter two.
Both other views are in contrast to the first, as they focus on technical details which are invisible to the user. The developer's view concentrates on extending the framework by new

components and is introduced in chapter 7. Finally the application architects point of view focuses on the development of new applications by configuring parts of the framework for new requirements. This one is mentioned in chapter 9.

2 DWARF and ARCHIE

Introduction of a Research Platform for Configurable Distributed Wearable Augmented Reality Applications

This chapter was written in cooperation with Christian Kulas and Bernhard Zaun. It provides a general overview about Augmented Reality projects and frameworks, in particular DWARF, the Augmented Reality approach of Technische Universität München. After viewing the guidelines that lead to the DWARF framework it's current state of development is outlined.
At the end of this chapter the ARCHIE project is introduced as a group project of several SEPs[1] and diploma theses. The completion of the ARCHIE project provides new functionality to DWARF thereby making it more mature.

2.1 Augmented Reality: A Stake-holders Point of View

The original intent in the development of computers was to provide support to people whose work was too difficult or too lengthy to solve manually, like large mathematical equations. New technologies arose as computers gained speed and more peripherals were connected to them. But the basic intention remained the same. Computers are supportive tools.
The increasing spread of computer networks in the last decade of the 20th century allows the distribution of services allocated to specific tasks. For example, rendering of 3D scenes is a resource intensive procedure which can be separated to another hardware, while a second machine can handle necessary remaining tasks of an application. The distribution of dedicated services to various platforms can get used in the Augmented Reality domain, because applications using this discipline have to aggregate various areas of computer science, where each may require a lot of computation.

Using Augmented Reality to support people can happen in many different ways. But for the discipline of Augmented Reality two classes of computational assistances can be identified. On the one hand, there are independent tasks that can be supported by Augmented Reality, while on the other hand, the diversion of computers through the environment provides resources for Ubiquitous Computing. Both classes are described and in advance a combination of both is explained.

[1] System Entwicklungs Projekt - a project every computer science student at TUM has to absolve

2.1.1 Independent Tasks

Closely focused on a task, users may perform task-centered activities like maintenance or navigation [28]. To realize applications of this kind, developers can rely on paper based guidelines like maintenance guides or city maps. These guides can get formalized in state machines executed by taskflow engines [82]. The Augmented Reality application leads the user through the task step by step.
Because of the runtime environment being known in advance, applications are comparatively easy to realize. Due to their nature these applications provide no flexibility to the users. Only the specified task can be realized usually only in the location specific to the application.

2.1.2 Ubiquitous Computing

Another aspect influencing Augmented Reality is Ubiquitous Computing [112]. Many possibly dedicated computers are placed in the users' environment, offering various services. These services can be of any kind, let it be telephoning, printing, viewing movies, or even ordering pizza. The support by the computer hereby should be invisible and intuitive to the user.
No predetermined taskflow is specified for these systems. However they are only useful if users have a clear idea on how to perform atomic actions or even whole processes so they might sometimes require assistance.
The provided services are sometimes fixed in one geographic location and don't support automatic reconfiguration corresponding to the environment.[112]

2.1.3 Intelligent Environments

The combination of task-centered Augmented Reality and Ubiquitous Computing can result in Augmented Reality-ready intelligent environments. The aggregation of both aspects supplies services provided by the environment. Seamless interaction between these services and a mobile Augmented Reality system give each user a way to dynamically perform tasks as described in section 2.1.1.
For example, as the user enters or leaves a room, his Augmented Reality system recognizes context changes and informs the user about new services. Options could be offered via aural or visual channels. A head mounted display (HMD) can display information of tasks available from the current location. Also the HMD can be utilized to visualize the chosen applications' user interface by rendering e.g. virtual 3D scenes. If a corresponding tracking service is available the user can leverage this to get an accurately aligned view matching the current perspective.
Systems using such intelligent Augmented Reality-enabled environments are powerful, as they can accommodate not only predetermined taskflows but also spontaneous desires of the users.

2.2 Related Work

At the current time there are several research projects on Augmented Reality all over the world. The resulting software architecture of the systems differ wildly ([103], [104]), but two general directions can still be seen.

In the first one prototypes are built by research groups which often result in task-centered systems for e.g. basic tracking concepts or car development concepts [106]. Usually they are highly specialized and monolithic. Many of these systems provide small demonstration setups for particular tasks. Even though the realized tasks essentially have a similar focus in other systems, the reusability of their technology is quite difficult.

Other projects focus on middleware technology covering central Augmented Reality tasks and by this provide frameworks for applications. Although the concepts of software engineering [34] have been known for some time, they have not been widely applied in the Augmented Reality problem domain. But there are some projects tackling this issue.

The Computer Graphics and User Interface Lab of Columbia University has assembled different types of Augmented Reality applications [43] from a common basis. Their work focuses on providing a common distributed graphics library as a framework for reusable software components.

Mixed Reality (MR) Systems Laboratory of Canon Inc. developed a basic frame for mixed reality applications [106]. Their set includes HMDs and a software development toolkit for building MR/AR applications. The provided library supports common functions required in Augmented Reality applications, but still it cannot be spoken of a framework.

German Ministry of Education and Research founded the project ARVIKA [1] which is primarily designed as a Augmented Reality system for mobile use in industrial applications. The architecture is user centered, but relies on fixed workflows. It does provide a configurable access to offered features.
The industry, in particular a department of Volkswagen AG needing software engineering technologies, already used some basic ARVIKA systems for car crash simulations [106].

An example for a multidisciplinary research program is *UbiCom* (Ubiquitous Communications) from the Delft University of Technology. Their architecture combines mobile units with stationary computing servers and focuses on mobile multimedia communications and specialized mobile systems [65].

Another approach is lead by the *Studierstube* project at Vienna University of Technology. That group uses concepts of software architecture among other things, but only as far as to keep parts reusable for testing new user interface paradigms [90] or for reconfiguring the tracking subsystems [79].
This can however only partially be seen as a framework for multi-user and multiple

applications in Augmented Reality.

At last we will take a final view on projects about Ubiquitous Computing. Today several approaches and technology systems share the idea of providing services to users through a star-shaped architecture, such as Ninja [55] or GaiaOS [57], [83].
Extendible Augmented Reality frameworks should rely on a decentralized architecture instead of the architecture of the approaches of these projects. Although some of them providing service federation, they don't seem to offer context and configuration propagation.

2.3 DWARF

The Technische Universität München also has a research project on Augmented Reality, which is called DWARF. The name is an acronym representing the guidelines for the general system architecture. DWARF stands for **D**istributed **W**earable **A**ugmented **R**eality **F**ramework.

The DWARF infrastructure provides an extensible, flexible and modular software framework for reusable Augmented Reality relevant components.
The framework can be seen in four abstraction levels. Figure 2.1 shows that layers.
The bottom layer is the layer of dynamic peer to peer systems. It provides connectivity and communication mechanisms for processes.
On top of this layer, the solution domain resides, supplying general components for the domains of Augmented Reality, wearable and ubiquitous computing. Services for tracking and environmental context are located here.
The third layer is described by the application domain. Components reusing general tasks of the sublayer reside here.
The top layer is built by the concrete applications available for the users.

A suitable framework for the Augmented Reality domain has three aspects: the announced *services*, a connecting middleware and a common architecture providing a basic concept to enable applications [28]. This framework allows components to be reused between applications and to dynamically configure them. One could e.g. imagine that the same tracking system provides position and orientation of certain objects to different applications.

Services *Services* are dedicated components covering general Augmented Reality tasks. Each service provides certain abilities to the user or to other services. On the other hand they can rely on input from other services, suppling filtered, analyzed, and rebuild information to other components or users.

Middleware A distributed application dynamically matches, connects and configures services, so that these can communicate directly corresponding to their needs respectively their abilities. The amount of service and their combinations can be changed dynamically by the middleware at runtime if required.

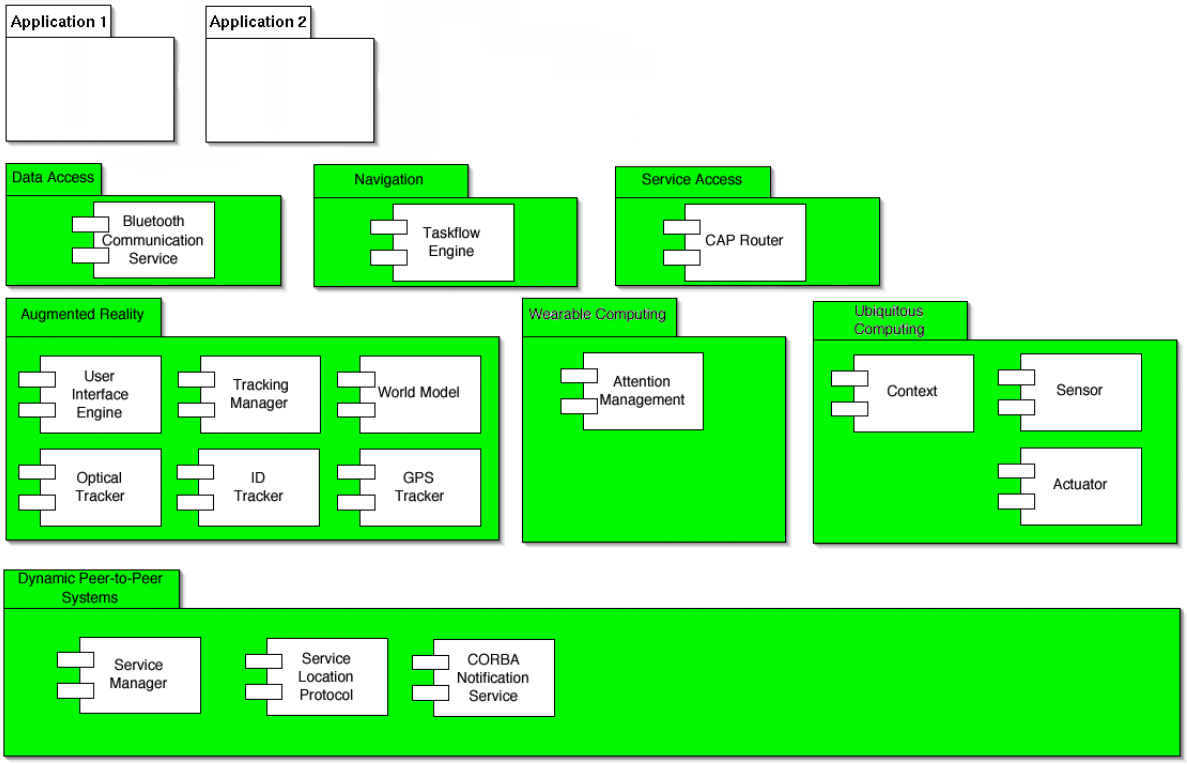


Figure 2.1: A layered architecture for the DWARF framework

Architecture A conceptual architecture describes the basic structure of Augmented Reality systems that can be built with it. To properly integrate different services with each other the respective developers have to agree on the roles of their services and on interfaces between them.

For our realization at TUM the main part of the framework consists of the service-manager and the service communication infrastructure. DWARF is designed as a distributed, and thereby wearable framework. Personalized software components can reside on different hardware components, even on mobile devices [28], enabling Ubiquitous Computing [112]. The *service-manager* handles all services and dynamically starts and stops required components on the corresponding hardware platforms.

The following section describes this context particularly.

2.3.1 Services

At DWARF applications each service can potentially run on it's own hardware device as an independent process. It is described in a *service description*. Additional information about service parameters is stored here using attributes. There could, for example be an attribute

`accuracy` for a tracking device, or a `location` attribute for a printer.

In DWARF those service descriptions are specified in conjunction with *needs* and *abilities*. These describe on a high level how services can connect. Looking at two connected services one has an ability and the corresponding partner has the matching need.

Each of these two has a *connector* specifying the communications protocol between them. Current protocols provide communications via event notifications, remote method calls (CORBA[2]) and shared memory. Two communicating services must have the matching communication protocols.

Needs and abilities also have a *type* which distinctly defines the corresponding interface. Thus for two matching services, one has a need with the same connector and the same type as the other services' ability has.
Hence types in needs and abilities can be used in various ways, e.g. a selection predicate is settable. The coding rules for these predicates follow the LDAP RFC 1558, 1960, 2054[16].

By the use of `minInstances` and `maxInstances` values for multiplicity are attributable to needs. If for example `minInstances` is set to "1" for a need of a service, this service will only be started properly when at least one corresponding ability of any other service is connected to it.

Figure 2.2 illustrates the description of two different services with a possible connection in easy readable XML-notation.

```
<service name="Tracker"
         startCommand="Tracker"
         startOnDemand="true" stopOnNoUse="true">
  <attribute name="location" value="GreatHall"/>
  <ability name="peoplesPositions" type="PoseData">
    <attribute name="accuracy" value="0.1"/>
    <connector protocol="PushSupplier"/>
  </ability>
</service>

<service name="Map">
  <need type="PoseData"
        predicate="(&amp;(location=GreatHall)(accuracy<1.0))"
        minInstances="1" maxInstances="10">
    <connector protocol="PushConsumer"/>
  </need>
</service>
```

Figure 2.2: Two simple connectable service descriptions

[2]Common Object Request Broker Architecture

2.3.2 Middleware

A *service manager* residing in each participating computer, is able to contribute it's service descriptions to a common pool. The service-managers internally check all possible connections between needs and abilities of all services and dynamically connect and start matching ones on demand.
As the middleware is the central part of the DWARF framework, it is partly well documented and more information would extend the topic of this thesis, the interested reader should reference [68] and [81].
Intra-service as well as internal service-manager communication take place via CORBA. Thus every service contains an interface to it.
The service-managers running on different computers find each other via SLP[3].

2.3.3 Architecture

A conceptual architecture defines the basic structure of Augmented Reality systems which can be constructed with it.
Thus it ensures that service developers agree on the roles of their own services within the system and on interfaces between them.

Figure 2.3 shows an example architecture for DWARF applications. It is separated into six packages. The distribution of services among the required subsystems of the general Augmented Reality architecture is shown, too.

Tracking The tracking subsystem is responsible for providing location information on real objects as positions in form data streams. This is the key feature of Augmented Reality applications, because the users location is required for the right alignment of virtual objects in his personal viewing device. An important issue is that the calculation of the location information must be done regularly in parallel to the other systems tasks.
Techniques used for tracking are video-based, use GPS or magnetic or inertial mechanisms. Often external trackers from the users environment provide corresponding information.

World Model The world model subsystem stores and provides all relevant data of real and virtual objects in regard to the users location and the performed applications. The model may be presented in various formats, but most often in reality augmented 3D scenes.
At runtime, a World Model managing component controls the access to the users current environmental model. This model is presented to the user.

Presentation This subsystem displays system output to the user. Besides 3D augmentation, components of this subsystem may also provide 2D status information or speech output.

[3] Service Location Protocol

This subsystem is one of the human-machine interfaces. It relies on tracking information about the user.

Control Subsystem The control subsystem gathers and processes any input the users make. It is to distinguish between actions of users and their movement. One can see any users input as actions consciously performed to control the system and it's applications.
Users input can be achieved by active gestures, voice or certain clicks on buttons. The input manager component is responsible for combinations of required input actions that may be needed by various applications.

Application Subsystem The application workflow resides in the application subsystem. Components of this section integrate logic, applications require for specific tasks. Abstract components off this subsystem are configurable by application specific code, configuration and content data. Hence components placed here provide functionality to the user, they are responsible for the bootstrapping of applications. An initial service describing the application must be started. This one expresses needs for other services and these finally assemble themselves through the service manager.

Context Subsystem Context information is handled by the context subsystem. This subsystem collects different types of context specific data and makes them available to other subsystems.
Context information may cover topics as user preferences, sensor data, time, tracking information, resource and domain knowledge or even information about the users current situation.

2.4 Extending the Space of Components

The DWARF framework is still under development. However multiple applications have been built by now with it.
Its initial version was verified by implementing an application for a guidance scenario called *Pathfinder* ([27, 68, 74, 82, 87, 110, 118]).
Following this, multiple projects (*STARS*,*Fata Morgana*, *FIXIT*, *TRAMP*, *PAARTI* and *SHEEP* [3]) increased the amount of services, provided by the framework.

2.4.1 Existing Services

Besides the classification to different levels in the last section, services can also be classified in four groups. One will see that some groups directly reference reusable framework components, others facilitate development while a group exists due to the development process of the framework. That group arose because the interests in the previous projects remained on the development of other components and demonstrative setups for applications, these components were use in.

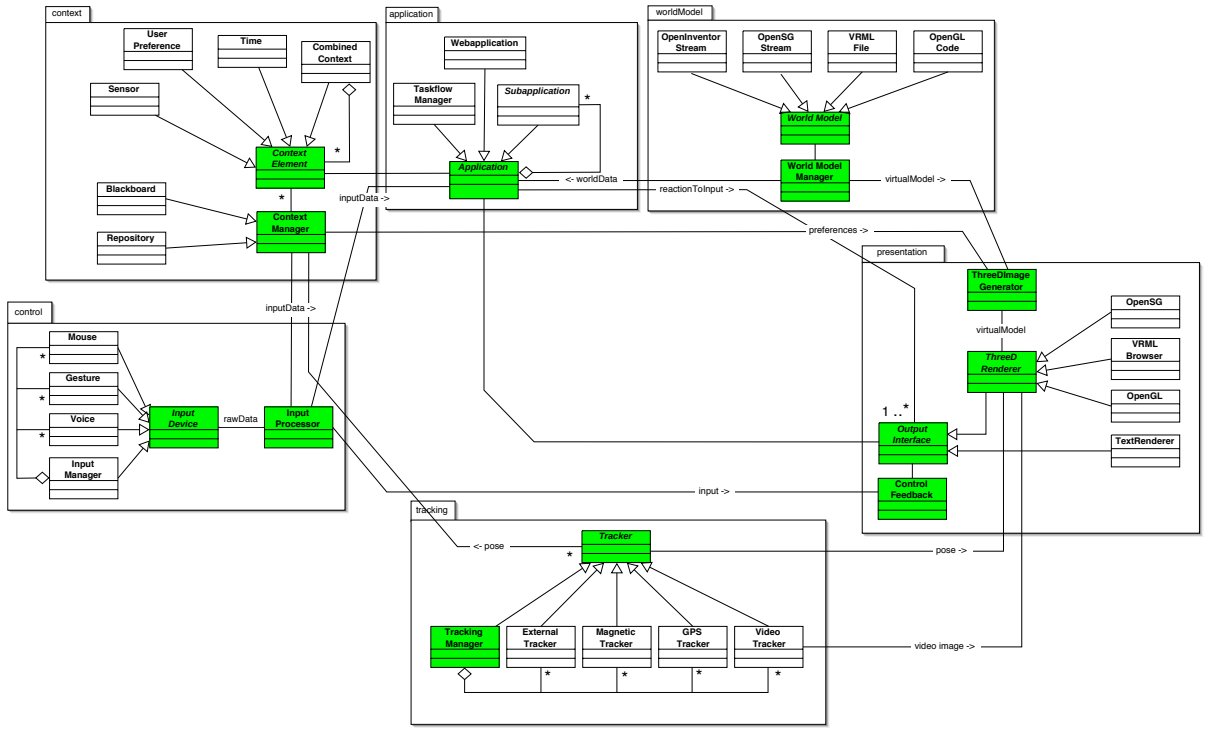


Figure 2.3: General DWARF architecture

Application Specific and Conceptual During the initial development stage of a new framework, not all requirements can be met. Thus to write a full-blown application some static services might be required to make up for missing framework functionality. With a perfect, stable, and feature complete framework, services like that will be obsolete.
Conceptual services demonstrate basic functionality and allow developers new to the framework to quickly familiarize themselves with it by looking at the code.

Testing A testing service assists the developer in debugging other services by simulating not yet fully implemented partner services by providing fake data. A tracking service could for example be tested with a black box test [34], made up by a test service which simply receives and displays the tracking data.

Task-focused Task-focused services are usually low-level services. They provide information on dedicated tasks. They can be configured in the range of their activity. Trackers for instance, provide position/orientation information on certain objects. Which objects to track can be specified, but the tracker always will track objects and provide their location information.
In DWARFs current state an ART[4] system is such a task-focused service as well as for example a sound-player service.

[4] Advanced Real-time Tracking - a commercial optical tracking system

Generic Components configurable to handle various interfaces and data-types. In contrast to task-focused components, these services provide a specific basic functionality on workflow activities that is configurable to required tasks. Their interfaces are adjustable to receive and provide task required data-types.
In this group the framework provides the User Interface Controller.

A goal of the development of a reusable framework for mobile Augmented Reality applications is to decrease the amount of components classifiable to the application specific area and to increase the amount in the other classes supplying configurable services.
Actual and new applications are made up by a combination of a set of configured generic services which require task-focused services during runtime. Current existing applications also rely on application specific components and new ones will, too. But their amount is intended to decrease in the future.

2.4.2 A Requirements Generating Project

Ongoing development in university projects produced a usable framework with various components. But the DWARF framework is still far from being complete.

New properly calibrated input and output devices were needed. Also a model for configuration and data persistence was required. Systems acting in intelligent environments also need components enabling the user to select services and providing Augmented Reality views on 3D-objects. The mobility aspect shall receive a contribution in form of a user-attached context aware service.
Finally, as research projects often result in unusable systems, human-computer interaction components should be evaluated for usability. So that the accustomization of these components to unskilled users takes not a great burden to them.
The infrastructure also required major improvements, to provide new generic middleware features like shared memory connections, template-services, and dynamic configuration bypassing[5].

These technologies extend the framework making it more suitable for future projects. During development of the mentioned project, new questions arose which make further improvements on the components possible.

While developing reusable framework components for a distributed dynamic system, the requirements for possible real applications should be kept in mind. The resulting architecture should be flexible enough to not only allow the development of the proposed new application but also support the development of completely different DWARF applications. Both approaches keep up continuous requirements engineering.
During the applications development process in the concrete project the requirements for

[5]Though a documenting paper about DWARF middleware is being written, these features are documented on the developers board only at the time of this writing. Additionally the feature implementation is still in an experimental phase

the framework components get validated.

A new team project was planned in July 2002 to provide a number of missing components. The aggregation of multiple student study papers into team projects proved to be very rewarding for all participating members in the history of DWARF. Members learned about team-work, building large systems and could practically experience their skills. Also the production process is more interesting than in a solo project, because a system is built, on which others rely and that will be reused by other projects.
Keeping this in mind, the ARCHIE project was founded in a workshop in Konstein in the summer of 2002. In combination with the current requirements for the framework, suitable scenarios were found to give application specific requirements for the production of the new components.

2.5 ARCHIE

This section introduces the ARCHIE project. The name is an acronym for **A**ugmented **R**eality **C**ollaborative **H**ome **I**mprovement **E**nvironment.

ARCHIE is an interdisciplinary project between the Universität Stuttgart represented by Manja Kurzak, a student graduating in architecture [64] and a team consisting of seven members from the Technische Universität München. Manja Kurzak provided information about design processes and the planning of the construction of e.g. public or private buildings. Her information is summarized in the following problem statement section.

This project provided a starting point for requirements elicitation on the different single projects of all team members. It was intended to use ARCHIE in a prototypical implementation to give a proof of the requested components and their underlying concepts. To build an application with full functionality for architectural requirements was not a main goal.

The realized concepts have been shown to stakeholders like real-world architects in a live demonstration of multiple scenarios. In new areas like Augmented Reality new requirements are often generated by the client when the capabilities of the technology get apparent. To prove the flexibility and extendibility of the new DWARF components the chosen scenarios are partly independent.

2.5.1 Problem Statement

A *Problem statement* is a brief description of the problem the resulting system should address[34].

Usually a number of people with different interests are involved in the development process of a building.

The *potential buyer* has to mandate an *architectural office* to initiate the building process because the process is too complex to handle for himself. A *mediator* is responsible to represent the interest of the later building owners towards the architect. The architect assigns work to specialized persons such as for example, *technical engineers* for designing plans of the wiring system.
Although the later building owner is the contact person for the architects office, he is only one of the many stakeholders interested in the building. Furthermore *landscape architects* have to approve the geographic placement of the new building in the landscape. Last but not least a *building company* has to be involved as well.

The architectural process itself is divided into multiple activities which are usually handled in an incremental and iterative way, as some specialized work goes to extra technical engineers, who propose solutions, but again have to reflect with the architects office.
After taking steps of finding and approximating the outer form of the building fitting in it's later environment, the rudimentary form is enhanced to a concrete model by adding inner walls, stairs and minor parts. This is followed by adding supportive elements to the model like water- and energy-connection, air-conditions, etc. As mentioned, this work always has to be reflected to the architect, because problems might occur during the integration of the different building components. For example the layout of pipes might interfere with the layout of lighting fixtures or other wiring. Spatial representation enhances pointing out problematic situations.
When the plans are nearly finished, the builder needs the possibility to evaluate the plan feasibility and if in the end all issues are resolved, all necessary plans can be generated and the builder can start his work.
In addition to that the building owner always needs view access to the model during the design phase. He wants to see his building in it's environment. End users should be given the option to give feedback during the design phase too. So, the architect's office receives feedback from many participants about their plans.

There are some entry points for Augmented Reality. The ARCHIE project delivers proof of concept for these aspects.
The benefits of the old style architectural design with paper, scissors and glue allows direct spatial impressions, while modern computer modeling does not provide these feature. Augmented Reality can bring these back to computer modeling.
Since preliminary, cardboard box models can not get scaled to real size and virtual models reside on a fixed screen, public evaluations are difficult to handle. Via abstraction of position and orientation independent sliders could be used to modify views. But 3D steering of virtual viewpoints is not as intuitive as just turning a viewer's head. Adding tangible objects as cameras provide familiar access to evaluation features.
User interactions with modern architectural tools require practice. So intuitive input devices would be useful.
Also in place of inspection of building plans and models would be a great benefit for all participating persons.

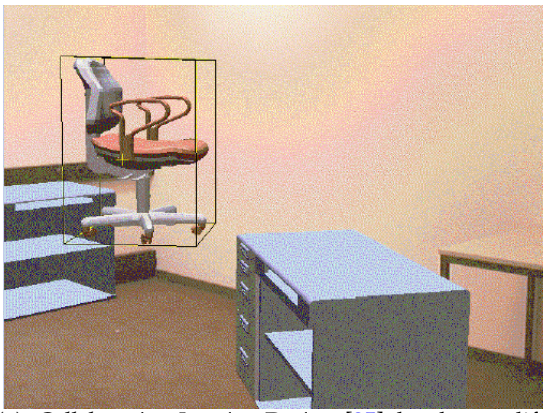
2.5.2 Related Work

This section lists research projects of other groups working in the Augmented Reality as well as in collaborative systems domain. Although focus remains on architectural tasks, the introduced projects aim on consulting our team in ideas and concepts transformable to the ARCHIE project.

The international project *Spacedesign* [44] resulted in a comprehensive approach using task-specific configurations to support design workflows from concepts to mock-up evaluation and review. The implementation allows free form curves and surfaces. Visualization is done by semi-transparent see through glasses which augment the 3D-scene. The approach allows cooperation and collaboration of different experts.
This project focuses on a stationary setup, useful on the task of car development, while ARCHIE should also be usable as a mobile setup.

A single system combining mobile computing and collaborative work [80] has been built by *Studierstube* [90]. The system, assembled from off-the-shelf components allows users to experience a shared space, while there are still synchronization requirements for late-joining partners. The result is far from a reusable context aware framework since it does not provide any kind of location awareness.

The *Collaborative Design System* [25] developed by Tuceryan et al. at the ECRC[6] demonstrates the interactive collaboration of several interior designer. The system combines the use of a heterogeneous database system of graphical models, an Augmented Reality system, and the distribution of 3D graphics events over a computer network. As shown in figure 2.4, users can consult with colleagues at remote sites who are running the same system.



(a) *Collaborative Interior Design* [25] local user lifts chair while a remote user moves the desk

(b) *An Application for Architecture* [102] overlapping a real office room with a CAD model

Figure 2.4: Screenshots from related projects

[6] European Computer-Industry Research Center

An architectural application [102] developed by Tripathy allows the user to view an architectural model within an Augmented Reality environment. The object model can be imported from any CAD[7] application, but it can not be modified. Though several additional information of real world objects can be seen, like the wiring of the lighting, or maintenance instructions for changing the bulb. Within the maintenance record the user even can see when the bulb was last replaced.

Webster et al. developed an application for construction, inspection, and renovation. The idea of supporting architectural tasks with Augmented Reality is not new but has already spawned testbed projects such as "Architectural Anatomy" [111]. Architectural Anatomy leverages Augmented Reality by giving the user a x-ray vision which might e.g. enable maintenance workers to avoid hidden features such as buried infrastructure, electrical wiring, and structural elements as they e.g. drill holes into walls.
The prototype application tracks the user with an ultrasonic tracking system and uses a head-mounted display for monocular augmented graphics. The project aims at building systems that improve both the efficiency and the quality of building construction, maintenance, and renovation.

2.5.3 Scenarios

A *Scenario* is a concrete, focused, informal description of a single feature of a system. It is seen from the viewpoint of a single user [34].
This section describes the Augmented Reality relevant scenarios of the ARCHIE system. The following list does not describe a full architectural system, because the ARCHIE project is only intended to be a baseline for the development of reconfigurable DWARF services.

Scenario: **Selecting Current Task**

Actor instances: `Alice:User`

Flow of Events:

1. Alice enters her laboratory which contains the necessary environment for the ARCHIE application such as a *ARTtrack 1* tracking system and a running *service manager* on at least one computer. Furthermore she is wearing a backpack with several objects mounted on it. There is for example a laptop that provides the viewing service for her HMD. She also holds an *iPaq* in her hand.
2. As her *iPaq* attains the range of the wireless ARCHIE-LAN, it's *service manager* connects to the one running in the laboratory, and they exchange their service information to each other. Now the *selector service* knows the available applications, and the menu pictured in figure (2.5) is displayed on the iPaq.
3. By selecting either entry and confirming, Alice can start the desired task.

[7] Computer Aided Design

Figure 2.5: The ARCHIE selection menu displayed on the *iPaq*

Scenario: **Calibrating the Devices**

Actor instances: `Bridget:User`

Flow of Events:

1. When she starts the calibration method with her iPaq she also needs to have the 3DOF pointing device in her hand.
2. Bridget can now see the current virtual 3D scene not calibrated on the 2D image plane. In addition to that the calibration scene appears superimposed in her HMD, too. And she is asked to align the peak of the 3D pointing device with the corresponding 2D image calibration point. Once Bridget aligned the points properly, she confirms the measurement by touching her touch pad glove.
3. As the calibration method needs at last six measuring points to calculate the desired projection parameters, Bridget will be asked to repeat the last step for several times.
4. After confirming the last calibration measurement the newly calculated calibration parameters will be transmitted to the viewing component.
5. Now her HMD is newly calibrated and can augment her reality. So the tracked real objects can be overlayed by corresponding virtual objects in front of her.

As the working environment is calibrated and ready for use, two architects want to perform a collaborative task: They want to develop the outer shape of a new building.

Figure 2.6: HMD calibration with a pointing device

Scenario: **Modeling and Form Finding**

Actor instances: `Charlotte, Alice:User`

Flow of Events:

1. Alice and Charlotte start the ARCHIE modeling application and their HMD viewing services.
2. As the system is initialized, both see the environment of the later building site.
3. Alice takes a tangible object, moves it besides another already existing building and creates a new virtual wall object by pressing the create button on her input device.
4. Charlotte takes the tangible object, moves it to the virtual wall's position and picks up the virtual wall by pressing the select button on her input device.
5. Charlotte chooses the new position of the wall and releases it from the tangible object.
6. Both continue their work and build an approximate outer shape of a new building.

Scenario: **Mobility - Location Awareness**

Actor instances: `Dick:User`

Flow of Events:

1. Dick starts the location-awareness subsystem based on the ARToolkit [61], running on his laptop attached to his backpack. In addition to the former setup an iBot camera is mounted on his shoulder to deliver video images.
2. The system is configured with the current room information. The information consists of the current room and the outgoing transitions (doors) to other rooms. A pie-menu giving information about the current ser-

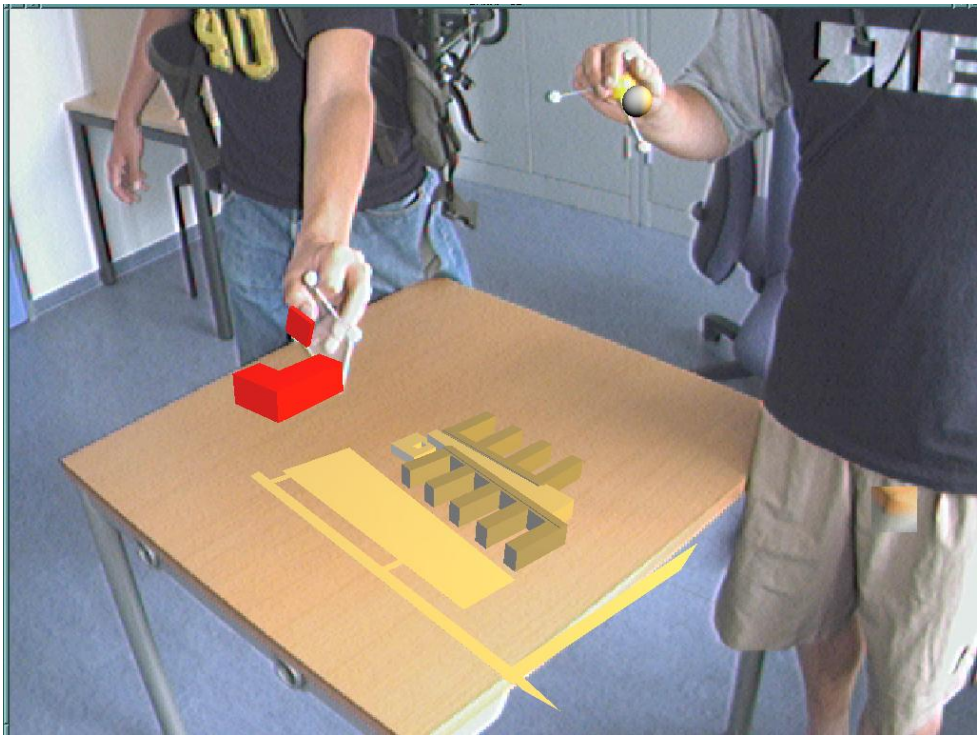


Figure 2.7: Modeling and Form Finding

vices of the room appears on the laptop.

3. Dick exits the room while detecting an ARToolkit marker attached to the door with his iBot camera. The system changes it's configuration according to the new state (new room). The old information is dropped and Dick has a new set of services available now which can be used in the current environment dynamically. The pie menu is updated.
4. Dick is able to exit and enter rooms and is enabled to use the corresponding services and room environment.

After different other Augmented Reality supported tasks are done, a group of later building, end users visit the architect's office, getting introduced to the plans of the architects office.

Scenario: **Presentation**

Actor instances: `Alice:User`

Flow of Events:

1. Alice starts the system, but instead of the previous used HMD, now a video beamer view is started, providing scenes as seen from a tangible camera object.
2. Alice takes this camera and moves it around the virtual model of the planned building.
3. The public can get a spatial understanding of the proposed building

Figure 2.8: Hardware setup for location awareness

which is displayed on the beamer screen. The model shown in figure 2.9 is rendered in anaglyphic red-cyan 3D. For a realistic 3D view the visitors need to wear the corresponding red-cyan glasses.

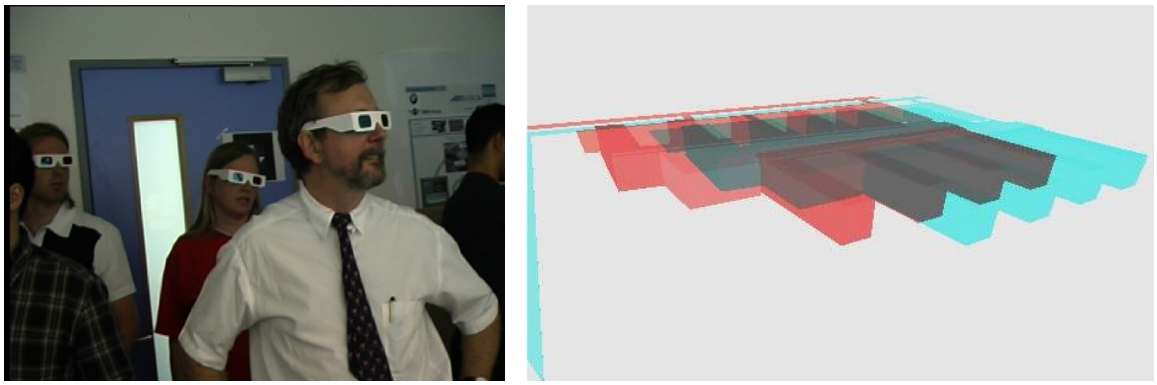


Figure 2.9: Presentation of a planned building to the audience

Scenario: **User Interaction Evaluation**

Actor instances: `Felicia:User, Gabriel:Evaluation Monitor`

Flow of Events:

1. Gabriel starts the ARCHIE application and configures it for a usability evaluation task.
2. He sets up the logging system and initializes it with the study and task name Felicia will be performing for an unambiguous log file.

3. After briefing Felicia appropriately, she is asked to perform a number of tasks which are monitored.
4. The logging system is automatically taking task completion times while incrementing the task counter too, so Gabriel can fully concentrate on Felicia's reactions to the system.
5. While Felicia is performing her tasks, Gabriel observes a number of real-time, updating charts which visualize her performance by e.g. applying standard statistical functions.
6. During the course of the study, Gabriel is always fully aware of what Felicia is seeing in her augmented display by looking at a special screen which duplicates Felicia's HMD view.
7. Felicia is debriefed after she has completed post test questionnaires handed out by Gabriel.

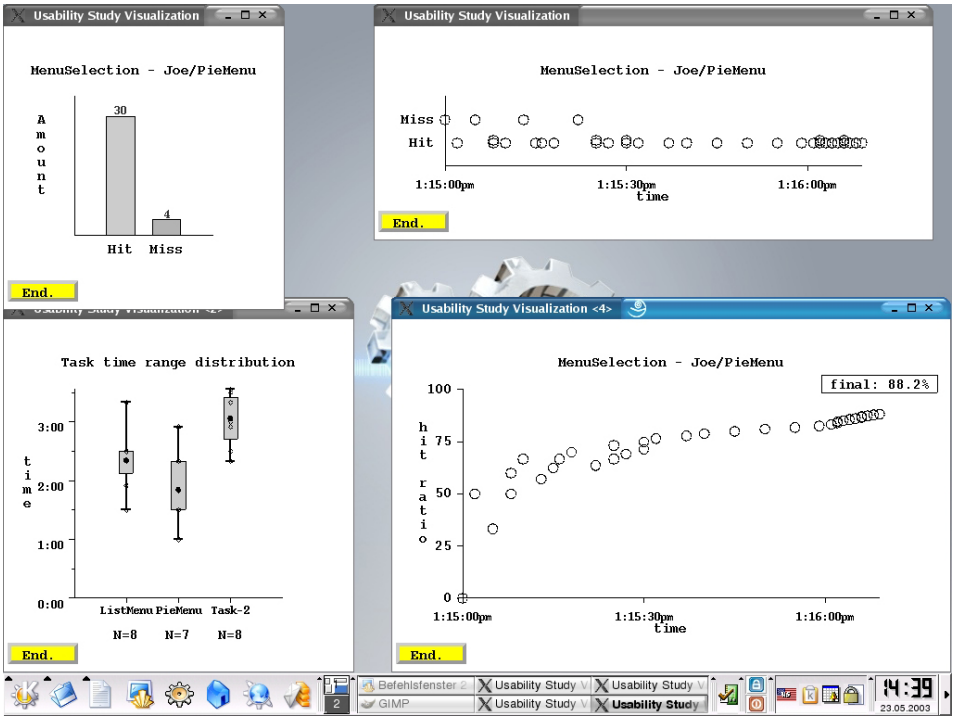


Figure 2.10: Live visualization of user performance in usability study

2.5.4 Requirements

This section describes the requirements our team elicited during the startup phase of the ARCHIE project. The methodology described in [34] was used to refine them step by step.

2.5.4.1 Functional Requirements

Functional Requirements describe interactions between a system and it's environment [34]. They are independent from the implementation, but can get realized almost directly in code. This section declares the *Functional Requirements* for the ARCHIE system.

Modeling process

Helping Wizard Architectural applications have many wide ranging functions. Desktop versions provide menus, toolbars and context menus. Entering the Augmented Reality domain will deprecate some functions by intuitive interaction, but still an option for selection of available functions is necessary.

Moving and Placing Objects For a good spatial understanding of virtual 3D-building models, these should be movable in an intuitive way. Rotating and moving functions should resemble the real world physics.

Interrupted Work The application must preserve the modeling states when the users switches back and forth between different applications.

Collaborative work

Shared Viewpoints Shared viewpoints are useful for other passive participants to watch the work of the designer. The system has to support the reproduction of a single view on multiple terminals. For a larger audience a videobeamer view would be even more useful.

Personal Views During the development process one architect might chose one certain submodel for editing which is then locked to his usage until he publishes the edited submodel again for his colleagues to see. So the changes can be be propagated to all participating viewpoints for consistency.

2.5.4.2 Nonfunctional Requirements

In contrast to the *functional requirements*, the *nonfunctional requirements* describe the user-visible aspects of a system [34]. These may not directly relate to the system's functional behavior. *Nonfunctional requirements* can be seen as overall requirements a system must fulfill. They influence multiple locations allover the later realization. In decomposition they can be seen as *functional requirements*, but this view would be hard to understand for the client during requirements elicitation.
This section reveals the *nonfunctional requirements* of the ARCHIE project that lead to the design goals of general DWARF framework components.

Augmentation

Correct in Place Alignment of Virtual Objects In order to have an Augmented Reality viewing device in architectural context, the viewing display must be calibrated so that virtual graphics are rendered in the position and orientation corresponding to the real world.

Three Dimensional Augmentation Spatial impressions of the outer shape of constructions such as buildings require three dimensional views on the virtual model.

Real-time It is a basic principle of Augmented Reality applications that the user can not distinguish between real and virtual objects. To uphold this illusion the view must update rendered objects at a high speed and accuracy so no significant differences can be seen compared to real objects in behavior. Therefore the tracking subsystem should provide adequate precision, too.

Convenient Wearable Devices Because the development of complex buildings is a long duration process, devices must be comfortable to wear and use. System output and input devices such as HMDs and gloves should be attached to the user in such a way as to minimize the loss of freedom of mobility.

Ubiquity

Mobility There could be more than one Augmented Reality enabled office in an architectural bureau, so the user's Augmented Reality device should support mobility by wearability and provide the execution of the application wherever possible without requiring additional setup steps. This encourages better collaboration between participants.

Application Selection Dependent on Location Often there are different projects in a company, available only at certain locations. So there should be a dynamic selection of applications and tasks depending on the user's current context.

Robustness In addition to omnipresence of computers the system must handle input data gracefully from possibly very large amount of users simultaneously. The system has to differentiate input devices by users to avoid ambiguous data streams.

The development for framework components also yield requirements.

Providing Service Functionality The services provided to the framework should fit in one of the architectural layers specified in the DWARF explaining section.

Dynamic Application Configuration Framework components may rely on context information. These services must be configurable via interfaces as described in [70].

Quality of Service Changes done by a user in a collaborative session must propagate to views of the colleagues. All views within the system participating on the same application need consistency. This aspect also applies to data not directly handled in a view, like internal service configuration.

2.5.5 System Design

An overview over the architecture of ARCHIE is shown in figure (2.11).

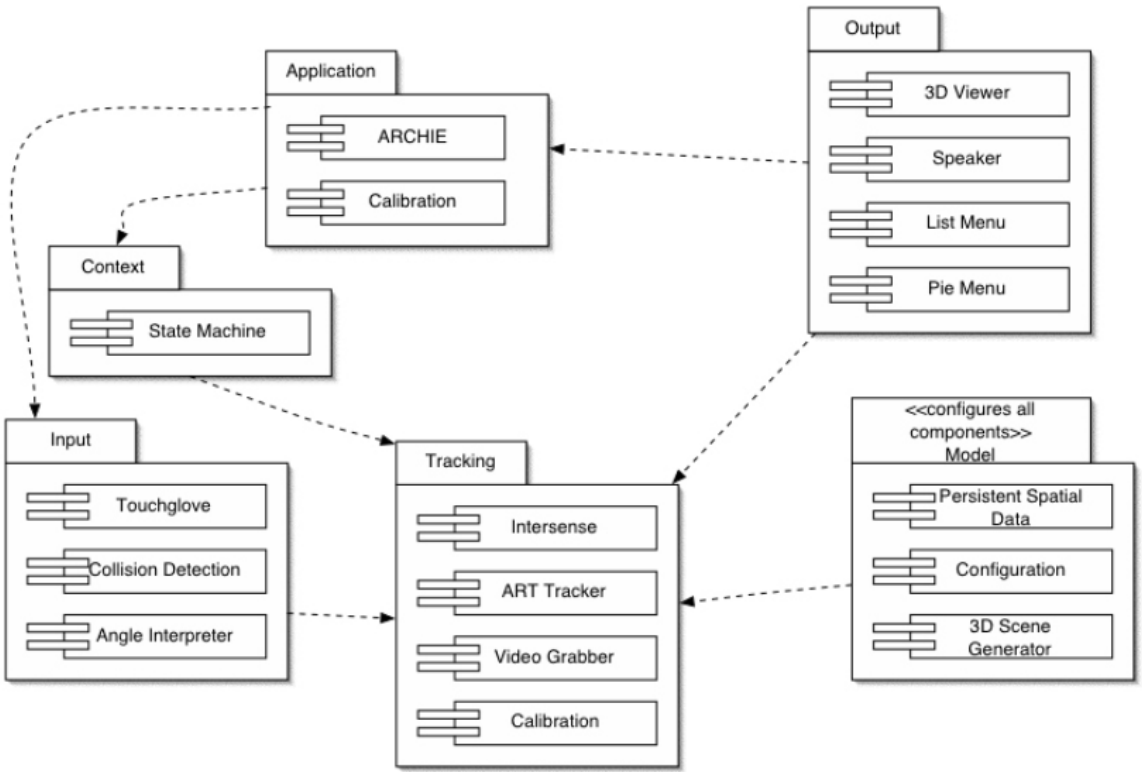


Figure 2.11: ARCHIE architecture

2.5.6 Focused Tasks

A usable framework has emerged from DWARF, but it is yet far from being complete, if one can speak of completeness on such a wide ranged research topic at all. So the implementation of new components focused on the individual research topics of our team members. This may result in some application specific components, but hopefully they will

get generalized in later DWARF projects.

Since students make up the workforce, who require a precise subject assignment, approximate tasks where given to the majority of our group members from the beginning. So the framework was extended with the realization of narrow topics.
These two restrictions result in the following implementation selection:

Input Device A SEP that enables a touchpad mounted on a glove in DWARF.

Output Device Another SEP provides a 3D rendering service which adapts to different output hardware.

Middleware Improvements Though the service-manager offers a variety of communication and connection interfaces, there is still a need for authentication of identifiable components. This is a SEP, too.

Location Awareness The last SEP within the team improves on optical feature tracking making location awareness possible.

Adjustment of Viewing Devices A diploma thesis that supplies calibration and configuration to different visual output devices on demand.

Managing the Users and Application Context Another thesis adding components to the framework for management of data of different types as 3D scenes and component configuration [101].
The proposed components are intended to hold information about real and virtual objects as well as to hold information about user context specific DWARF service configuration data.

Usability Evaluation of Human Computer Interfaces in Ubiquitous Computing The third thesis in our group provides a usability evaluation framework which can be leveraged to evaluate human computer interfaces within the Augmented Reality system to come to e.g. new Augmented Reality design guidelines. This includes, among others, components to take user performance measurements in DWARF, and visualizing this data appropriately to support the usability engineer monitoring a participant for usability study purposes.

To complete this list we also want to mention the work of our architectural client, who provided architectural requirements. This work has a direct focus on Augmented Reality. The thesis topic contains visualization of various kinds of radiation as thermal flow and refraction. A bargain DWARF might hopefully provide in the future.

There are other components required for ARCHIE which do not fit in one of the above listed categories. These will get generalized hopefully in new projects.

Although the second part of this chapter focused on the ARCHIE projects, one will see the development of reusable components for the DWARF framework in all cases of our team members in the following chapters.

3 Data Management in Distributed Systems

Many Factors have Influence the Design of Suitable Components

Now, as the general problems for the ARCHIE application are defined, the focus within this chapter concentrates on the requirements for Augmented Reality applications in the Augmented Reality. The main interest lies on incidental application data necessary to initialize them on startup and to provide task specific data during runtime.

Because DWARF is designed as a distributed wearable framework, this chapter gives a brief overview about distributed systems and data management in them. Subsequently it evolves the requirements for Augmented Reality in regard to our DWARF framework and enlightens the current state of the DWARF implementation. Afterwards this chapter gives a brief overview about the system dependent requirements of the ARCHIE application.

The activities performed to write this chapter follow the requirements engineering with its processes as described in [34]. I followed the structure explained, to assure a well formed methodology that allows other developers to extend and reuse the components for data management.
Because this chapter concentrates on the requirements for Augmented Reality, the parts containing scenarios and use cases of the ARCHIE application have been shortened, while tabular descriptions are detailed in the appendix.

3.1 Overview

The collection of requirements starts with a general overview about technologies available for use in Augmented Reality and their usability within.
Applications of this domain deal with data about real and virtual objects. Virtual objects must be rendered in the right alignment to their real environment. So rendering information of virtual objects is required to superimpose the real scene.
Various applications require extra application specific data. Some objects must be rendered in different style or other components require not renderable information about objects.
Because Augmented Reality systems either need personalized components as HMDs and must keep wearable and computations currently require a lot of computing power, application components can run on different hardware platforms. To follow the paradigm of ubiquitous computing, components of the environment must provide ad hoc services for dynamic use.

So it is useful to take a look into distributed systems and on data management in them.

3.1.1 Distributed Systems

Distributed system can be modeled in two ways: As client-server architecture or as peer to peer networks. In the first approach the clients know where to find the corresponding server by its address, while in the second, one client only knows his next neighbors.
Alike different processes residing on the same hardware platform, distributed services can communicate by remote method calls or by notification events. Hence the message transport does not only influence the systems hardware, distributed applications are slower in the overall workflow than local ones, because all transported messages have to get encapsulated via network protocols.

3.1.2 Data Management

Data Management is often a part of an application. Reusable components for applications or common tasks are often software that enable organization, storage, retrieval, and manipulation of data.
There exist two operational areas for such components: Persistent data handling and runtime data handling. Both approaches facilitate access to data, but runtime systems use hardware memory to deposit their data while persistent systems store their data on physical devices like harddisks.
Common approaches to persistent systems are file systems and databases. File systems allow archiving of self structured data in a directory hierarchy on a persistent device while database systems extend that by the type of data. They require typed data organized in configurable data structures.

3.1.3 Data Management for Distributed Systems

Approaches for data management in distributed systems are widely used in computer networks where many people use different computers. This section evaluates different architectures currently used in the computational domain and lists the advantages and disadvantages for the use in Augmented Reality.
The following sections shall only provide a brief overview about data management technologies to introduce the issues of that topic. Chapter 5 focuses more detailed on that topic.

3.1.3.1 File systems

Personal files of the user are stored on a network file system server. Every time a system is started on a specific machine, the shared directories are mounted to the local file system hierarchy. By this every user has fully transparent access the server's data via network, as

if the files were stored locally. Widely used implementations for that are NFS[1] [88], AFS[2], secure and SMB. NFS provides simple file access while AFS adds security and reduces netload by local caching of accessed files. Samba is a implementation for Windows systems. It simulates a Windows server for central data storage.

Advantages File systems are useful for the storage of unstructured or very differently structured data. Because the file abstraction is relatively low level, it enables applications to perform a variety of size and speed optimizations.

Disadvantages Systems using a file system loose efficiency when searching for specific data, because the file intern data structure can vary between different applications. Files, however, require the applications to take care of many issues, such as concurrent access and loss of data in case of system crashes.
The use of file systems for an Augmented Reality framework is surely not sufficient, because a large amount of handled data is structured and typed. Objects, real and virtual, have positions and orientations. Additionally, not all data is required anytime in every application on top of the framework.

3.1.3.2 Tuple-Spaces

Data can be handled in tuples, where each tuple holds ordered multi-sets of values and their types [38, 77]. Tuples can be stored in Tuple Spaces also known as blackboard or repository architectures. Such persistent data can get retrieved by so called templates with almost the same structure as tuples. Tuple Spaces provide associative access on tuples: A reading operation returns a tuple matching to a template.
Specifications and implementations such as Linda[52] and JavaSpaces[24], ObjectSpace, and T-Spaces[117] often extend tuples from multi-sets to trees, so that object-orientation is reflected.

Advantages Tuples can store information about objects of any kind. Simple typed values as positions, orientations, owners and colors can be put there.
Arbitrary searching is convenient to handle in tuple spaces by providing structure templates containing specific actual fields and formal range descriptions.

Disadvantages Although tuple spaces provide communications for distributed systems, there are centralized server holding the data. That architecture produces relatively heavy-weight communication and does not facilitate real-time applications.
The usage of different spaces in heterogenous environments is not easy to handle, because matching strategies are solely partially specified.

[1] Network File System
[2] Andrew File System

3.1.3.3 Databases

A database is a collection of data that is organized so that its contents can easily be accessed, managed and updated. The most prevalent type of databases are relational, tabular databases in which data is stored so that it can be reorganized and accessed various different ways. Also object-oriented databases are available that are congruent with the data defined in object classes and subclasses. Object-relational databases are a mix form allowing tables to extend other tables.
Some implementations also provide distribution of their data. Contents can be dispersed or replicated among different points in a network.
Relational databases contain aggregations of data records which can get modified with a specified query language, SQL[3].

Advantages Above all, relational databases provide sophisticated interaction specifications with optimized data handling algorithms so that even complex queries can be performed in acceptable time.
DDLs[4] provide generic runtime addition of new data types while prepared statements give a common interface for queries on data.

Disadvantages Though distributed database are available, database management systems in common try to optimize databases for efficient data access. This fact often decreases efficiency for distribution because it produces an overhead for computation of new accessing structures.

3.2 Data Management in Augmented Reality

Now a brief overview about the technological environment for distributed data management is given and gets connected to the requirements of mobile Augmented Reality. Further more the state of development of the DWARF framework as it was before this thesis started is described. One can see problems former Augmented Reality applications had in data management.

3.2.1 Distributed Data in Ubiquitous Computing Environments

Let us discuss the problems arising when data is handled in Ubiquitous Computing Environments. Data must be distributed heavily, so that services can provide them to users. Intelligent environments provide services to users. The availability of tasks to perform on the existence of services must be known to the user. Either he knows how to select a specific task or his personal HMD displays augmented information on possible tasks. Hence the viewing device and any additional Augmented Reality device are personalized to the user,

[3] Structured Query Language
[4] Data Definition Language

these can allow the user to set personal configuration.
The same holds true for running tasks, which also extend the user's reality and can also be configurable to users.
Both areas are of interest on data management, since the services building the application can be distributed over various hardware platforms.

Augmented Reality Augmented Reality systems superimpose virtual objects to the user's reality. Renderers for viewing devices require a lot of information about virtual objects. Object properties such as position, shape and material must be available for rendering devices to display all necessary objects. Other components working on these properties must be able to modify them. Also user interaction can modify these values.

Personalized User Environments The users environment must provide a configurable individual user context. This context stores personal configurations and provides task relevant information.

3.2.2 DWARF

As explained in chapter 2, DWARF is our research platform on Augmented Reality. It provides a distributed network of services with a middleware, the *servicemanager*. For a complete description of the capabilities of the middleware, see [68].
The initial version of DWARF supplied the *Pathfinder* application. In advance several further applications (*STARS, Fata Morgana, FIXIT, PAARTI and SHEEP*) were built on this framework. Each of these projects supplied new services extending the framework, but there was no concept for persistence and dynamic application configuration. This can be illustrated on the last demonstration setup, before the ARCHIE project.

3.2.2.1 SHEEP

SHEEP [69] is an acronym for *SHared Environment Entertainment Pasture*. This application was built as demonstration setup for the ISMAR 2002[106] conference on Mixed and Augmented Reality. The intention of this demonstration was to show the distribution and wearability of the DWARF framework. Also multimodal and multi-user interaction of the running application were a topic of interest.

The SHEEP presentation shows a playground with a green natural landscape and some sheep walking over the green grass. This scene can either be viewed in a vertical 2D projection on a table or in a 3D projection by use of a notebook. These notebooks are tracked, so that the viewing service can determine the notebooks location and render the 3D virtual scene in right alignment. The user gets a nearly realistic view from his viewpoint onto the virtual sheep scene.

Changes on the state of the application can be done via tangible real objects, tracked by the ART infrared-tracking system [5] and by speech input. Users are able to create new sheep in the landscape and a special user in the role of a *god* can change the color of sheep.

Figure 3.1: A "screen shot" from the SHEEP application

3.2.2.2 Problems

As SHEEP focused on multimodal user interaction, the green landscape and all sheep were static scenes residing on every hardware platform with a viewing service. When the application is started, the initial scene is loaded from a file and the virtual scene is initialized. Although the renderers of the scenes change the virtual sheep's positions as they move, a current state of the application can not be saved. Therefore no continuation is possible after application shutdown.
If another person is joining the application with his notebook, the scene initialized from the stored file does contain all sheep, because every sheep registers itself on the viewer of the new notebook. But if there were previous color changes on some sheep, these are not available in the new joined view. All sheep remain white because a sheep does not known about its color. This information is only known to the viewers who were running at the time the god changed the color.
Due to wide distributed redundant data about sheep over the majority of services, a consistent data modification is an issue that needs to be solved. Inconsistency and persistence of data are central issues in DWARF.

[5] ART - Advanced Realtime Tracking, Herrsching

3.3 ARCHIE

After the current state of DWARF was explained, the issues of the ARCHIE project are discussed with focus on data management.

Modern architectural applications provide a wide range of templates for objects to be created and have even more functions manipulating them. Small cottages as well as large buildings are realizable with applications of these type.
A wide variety of building and environment templates can be added to a already planned model. Virtual objects created from templates, such as default sized boxes, have properties describing their appearance and shape. These properties reveal from the type of the virtual object and must be modifiable in order to place any virtual thing to another location or to modify color and other appearance factors.
This will allow application users to build models of buildings for shape finding and extension to more detailed plans.

Performing architectural applications users need a personal environment for their virtual models augmented to their worn viewing device. Architects often want to have additional evaluative approaches to existing models for a new building site. Extra models shall reside only in the user's environment until he publishes it to his offices colleagues.
In some cases, changes at a virtual model must be directly promoted to other viewers, if two people want to perform a collaborative work in the same area, like the same virtual room. In that case, the changes have to be released to the working partners on a concrete user action or, if preset, shall be updated automatically in the partners viewing device. That fact requires also to provide personal settings of the application. The system must provide user dedicated components with their personal settings.

Because planning new buildings is a long duration process and often other models are reflected for inspiration, modeled buildings must have a persistent store and mechanisms for including other models. Also parts of models must be importable or at least viewable. If only a single room is needed, only the buildings outer shape, the access path to the room and the rooms virtual objects itself needs to be shown in the architect's view.

Using applications in the architectural domain also needs knowledge of the real world. The system must be able to provide structures of the environment of a building site, so that the planned building can get appreciated to its later environmental context.

3.4 Requirements

The last sections identified the general tasks necessary for data management in distributed Augmented Reality systems. This section aggregates the requirements necessary to design components for data management. Requirements are functions a system must have (functional requirements) or are a user visible constraint on the system (nonfunctional

requirements).[34]

3.4.1 Functional Requirements

3.4.1.1 Real and Virtual Objects

Service components have to store information about real and virtual objects, that may be important for the user's interaction with the specific application. Some requirements attribute data to these objects.

Creating objects Virtual objects must be creatable on specified positions with a specified type. Dedicated tangible objects revealing location and orientation are used for this, a type selection mechanism has to provide options to create objects of various types.

Relocating Objects Service components can send information about position or orientation changes which can attribute to stored objects. Components managing object data must be able to register for such events. Every change must be processed by the registering service and, if necessary, that information must be stored.

Relative Positions Every object has an associated position and orientation. To facilitate the use in various applications, object storing components have to provide means to compute these transformations relative to an arbitrary other object.

Changing Properties Stored objects are highly variable. To handle this variability in general framework components, it must be possible to store an arbitrary amount of arbitrary properties associated to each object. If an object requires some changes, for example another color or material, its properties must be associatable with service components providing fitting values.

Different Appearance The same object can, but does not have to, have different appearances in different views. Different templates for 3D rendering must be associatable to virtual stored objects.

Searching Specific tasks require the user to use information or items from his environment. A component holding information about the real world and augmented virtual scenes must provide interfaces to access information about required objects.

Grouping Virtual Components Items may belong to other things, so a grouping mechanism is required combining several objects to handle them as a whole object.

User Interface The components storing the object information shall have no direct interaction with users. Only software interfaces that are used by other components shall provide functionality.

3.4.1.2 Consistency

Object properties can be changed dynamically. As several applications may run at the same time, objects can be modified from various services of the framework. So, the storing components need to allow consistent multi-threaded access to objects and their properties.
After every change on objects, all services working on these data must receive a notification about the details of the change. The update mechanism must be efficient, because several services can act on different subsets of data.
This is very important for viewing devices, because these rely on a large amount of data to render virtual objects.

3.4.1.3 Privacy

Privacy is an important issue of multi users systems. Each user requires a dedicated personal private space for his working environment. This space must be able to hold parts of the users virtual environment. Users need to be able to modify the privacy state of objects.
Applications may provide personalized settings to users. That data must remain in the users private environment and only be accessible to him. Changes on private data shall not be distributed to other system components.

3.4.1.4 Persistence

As several applications can run at the same time and on different tasks, data managing components shall only handle objects that are in use for the specific task. All data not required in applications must be stored in a persistent environment.
This is necessary to keep hardware resources available for use in additional tasks and to preserve data in cases of component failures.

3.4.1.5 Configuration

The previous sections focused on applications. Data in applications contribute to real and virtual objects, their appearance and behavior.
But also components, maybe services, building the applications, deal with data. Different users in various roles can work in varying applications, but require different behavior of the applications to perform their specific tasks. So environment services require dynamic configuration in accordance with the user's context. Also users can modify configurations of applications to react in different ways.

3.4.2 Nonfunctional Requirements

Not directly related to functionality of a planned application like ARCHIE, are the non-functional requirements. They result from the green-field elicitation on the problem of data managed in distributed Augmented Reality frameworks.

Wearability DWARF is intended to run on wearable computing devices. This fact requires the data management components to ensure low memory and computation consumption beyond that for handling the stored data.

Adaptability on low Communication Overhead Because DWARF is a mobile framework, capacity of communication channels can become wide ranging. The system must provide mechanisms for redistribution of components with intensive communication between them.

Maintaining Inventory of Templates A reusable framework, where components can get reconfigured to new applications depends on a mechanism to author templates for new virtual objects.

Performance The data storing services are central components for applications. Many services access them simultanously and most of them are constrained by hard real-time requirements. Therefore it is necessary to ensure low response time, even on continuous data modifications.
Examples are viewing devices like HMDs. In order to get a direct feedback on actions, the user performed on objects, a visual feedback with low latency must be given.

Sufficient Performance for Augmented Reality Even if the data managing services work performant, rendering of motions can be slow. This is evolved by significant processing time on computation and on data flow in the distributed system. Therefore, the main topic is that transformation changes of virtual objects require promotions in low latency.

Fault Tolerance If e.g. a hardware component of the system crashes the last state before the crash must be restored immediately in another replacement service.

Multiplicity Services which are not exclusive to a resource may be reused, even if another instance is already running on the same hardware platform.

3.5 Scenarios

The last chapter gave scenarios for the ARCHIE project in general, while this section gives more detailed scenarios on the problems concerning to this thesis. The following scenarios

describe the parts of modeling and presenting virtual building models in a textual style. Tabular scenarios listing flows of events are released to the appendix.

The task of finding a outer form of a planned building is initialized by an architect starting the ARCHIE application in a development office.
The system displays the proper environment of the building site. Tangible objects in conjunction with the architect's system interaction device allow him to create virtual objects at the real objects positions.
Overlapping virtual with real items and a select action allows the user to move objects. Placing objects happens on a corresponding deselect action.
As the development activity continues, a colleague joins the architect's work. He also starts the ARCHIE application and the current state is initialized in his personal view. After the first architect publishes his changes from his private space, all model updates are visible to the colleague.
During the form finding activity some unnecessary virtual objects have been created, that needs to be deleted. The architect's colleague selects these virtual objects by overlapping them with a real object and a button click on his input device deletes them.
After the plannings are done, the model can get presented to building stakeholders. A video beamer projects a 3D scene that shows the whole building from the viewpoint of a tangible object representing a virtual camera.

3.6 Use Cases

The use cases are in detail in the appendix A.2, just as the scenarios. Here I will only give a graphical overview about the use cases names and their accessibility to a actor.

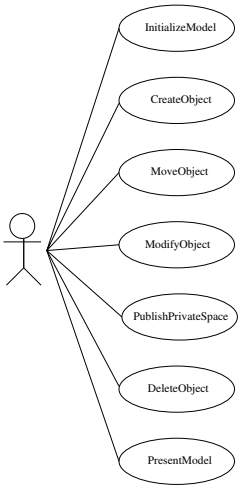


Figure 3.2: UseCases

4 Related Work

Other research groups also quest data management in regard to Augmented Reality, Mobile and Ubiquitous Computing

---

The requirements for the DWARF components in mind, it is useful to take a look into other research projects. This chapter illustrates the approaches of other groups to Augmented Reality.
The gained knowledge is a benefit for the architecture of services in regard to graphical and behavioral data handling. The introduced projects are explained in focus on data handling for graphical distributed systems and on common handled data.
I listed advantages and disadvantages for each point of interest of these projects. These appoint to the matching on the general requirements in Augmented Reality as explained in the last chapter.

4.1 Studierstube

An approach to Augmented Reality is lead by the *Studierstube* [90] project at Vienna University of Technology. This group's central architecture relies on a distributed scene model that can be viewed with HMDs and beamers. Manipulation by user interaction is also enabled.

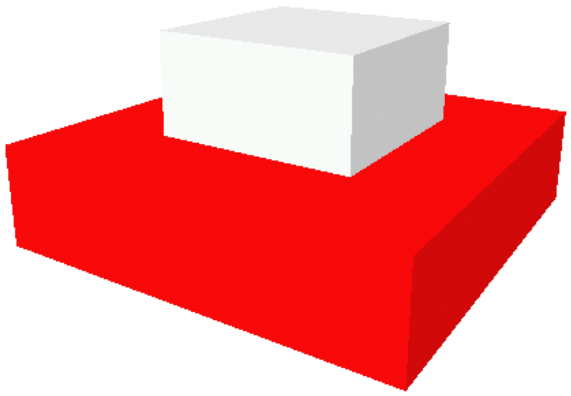
Their approach uses a 3D rendering library *openInventor* in C++ that runs on top of OpenGL. This guarantees portability to different platforms and hardware accelerated performance. 3D scenes are described by the OpenInventor [18] file specification [113]. This specification includes the VRML specification as a real subset.

A viewable scene can contain hierarchical compositions of actions on virtual objects in a so called scene graph. Such a scene-graph is an object-oriented structure reflecting the semantic relationships of graphical objects in the scene. It is composed of *nodes* which are classes of the implementation of the specification. For instance, a stepped pyramid can be modeled in OpenInventor as seen in the textual tree in figure 4.1. This is just a simple example, while OpenInventor offers a large variety of classes for many purposes, including management components for event handling and searching [71].
Each of these nodes is composed of fields that store attribute data for a particular node. A graphical tree is constructed from group nodes that store links to their children.
Scenes are rendered by traversing the graph and executing each nodes rendering function.

```
#Inventor V2.0 ascii

Separator {
   Transform {
      translation 0 0 1
      rotation 0 0 1 0
      scaleFactor 1 1 0.5
   }
   Cube {}
}
Separator {
   Transform {
      translation 0 0 0
      rotation 0 0 1 0
      scaleFactor 2 2 0.5
   }
   Material {
     diffuseColor 1 0 0
   }
   Cube {}
}
```

(a) source code



(b) rendered screenshot

Figure 4.1: A pyramid with two steps in OpenIventor

4.1.1 Data Structure

The architecture of *Studierstube* relies on the object-oriented structure of the scene-graph. This graph is not only a tree, but a directed acyclic graph holding all data necessary to render virtual objects as well as application specific data. The rendering mechanism itself relies on a tree, while application specific functionality also may require additional links to already referenced nodes.
This structure stores all application dependent information. This are the named renderable virtual objects as well as component configurations. These configurations are handled by the *Studierstube* extension that is implemented inside the scene-graph object structure.

Advantages The scene-graph data structure directly contains all data necessary for rendering. As it is directly accessible by the rendering engine without data access to remote machines, efficient rendering is possible.
The special kind of data structure, a directed acyclic graph, is able to hold all information on virtual objects, including their alignments relative to each other.

Disadvantages The use of the DAG provides mechanisms to store information about objects as their positions and shape. Also context information are representable by data types in nodes. But these increase the execution time of the rendering engine, where this

information in not directly required. Before user specific information is rendered, often other services require access to the DAGs data.
As we require access from other services to the data, some separation were necessary.

4.1.2 Replication

The *Studierstube* research group extended an implementation of the OpenInventor specification to a distributed shared scene graph [91] with handling similar to distributed shared memory.
Because descriptions of virtual objects require a large amount of data, the *Studierstube* group uses copies of the scene graph on every participating rendering engine. This enables interactive frame rates that are necessary for Augmented Reality applications.
To guarantee consistency between all scene graphs, a synchronization protocol has been specified. Modifications to scene graph objects fields have a fixed size and can be encoded in also fixed size messages. These can be propagated efficiently over the network to the other participants. Alterations to the scene-graphs structure use special messages for the creation and deletion of single nodes. Often not only one node is created or removed, but a whole sub-graph. To increase the efficiency of operations of this kind, the sub-graph is parsed from a common file or a URL.

All described changes to a scene-graph are distributed via multicasting (UDP[1] with negative acknowledgments) to all participating renderers.
The renderers residing on different machines include the changes to their private data structure and directly draw scenes to attached output devices from their independent viewpoint.

A distinction on modifications has to be seen. Discrete modifications are propagated directly while continuous changes to nodes are distributed as recently as the stream terminates. For instance, rescaling a virtual object to double size occurs immediately, while choosing a new color from a color space is done when the final color is chosen.

Advantages The update mechanism is able to change the graphs structure as well as the nodes internal data. By this it is adaptable for further use.

Disadvantages Replication is always directly performed on every modified object. When continuous changes are done to an object, the network system as well as the data managing components have to deal with a lot of load. No caching mechanism is available to decrease that load.

[1] User Datagram Protocol

4.1.3 Distribution

Application specific computations may be triggered by user events need not be distributed to every participating host. Instead *Studierstube* determines a master host, called a sequencer, for every application [92]. That one is responsible for performing all execution of incoming events in application code.
The computed updates within the scene-graph structure are then replicated via the consistency protocol of the distributed OpenInventor scene-graph to the slaves.

Advantages One host performing all application specific tasks minimizes the load of the slaves which can concentrate on 3D rendering.
Also pure master-slave setups for public demonstrations are possible.

Disadvantages The sequencer is equivalent to an application server. In ubiquitous computing environments, applications can be an accumulation of services. A large overhead would reveal, if whole applications had to be started for specific tasks. There is a coarse granularity.

4.1.4 Local Variations

A large set of possible applications requires not to distribute the scene-graph as a whole, but to provide a local space for each user. This is illustrated in figure 4.2, adapted from [90]. That can be useful in various ways:
First of all, there may be individual contents per user. The *Studierstube* project implementation provides mechanisms to choose objects to share with other users and their views or to keep them local.
Second, the team of Vienna university provides a mechanism to give objects different representations in different views. For instance, a teacher may see solutions to problems, while students may not and just see the questions. This is also useful for highlighting selected objects to the user who is going to perform changes on that objects.

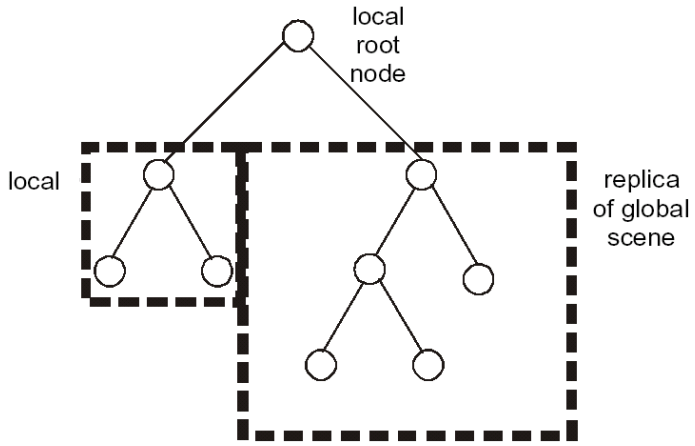


Figure 4.2: Local variations allow to customize the behavior for each user

Advantages Private space is handled in an elegant way.

Disadvantages Besides the 3D renderable data, application node specific data can be hold locally. If no consistency mechanism exists, local parts of an application can become inconsistent with others.

4.1.5 Persistence

The *Studierstube* group uses the file system for persistent data storage. Nodes, hierarchical scenes and even the applications are specified in the OpenInventor file format[113].
On creation of sub-graphs, files are read directly or via an URL. The read string is parsed and the object graph is built. Finally, the objects are added to the specified parent node of the scene-graph.
Storage is handled by serializing the scene-graph to the file-format and writing that to a file for later retrieval.

Advantages This form of persistence is easy to implement because OpenInventor implementations provide mechanisms for standard classes. Extending classes only must also provide the inherited interface and then can get serialized, too.

Disadvantages If only partial data is required of a scene or an application, there are two possibilities. One is to rebuild the whole scene-graph in memory and to search in the instances, while the other approach requires additional parsing tools for the file system.
Further problems of file systems have already been illustrated in section 3.1.3.1.

4.2 Nexus

The University of Stuttgart has a Sonderforschungsbereich on environmental models for mobile context aware systems called Nexus.
The concept of the Nexus project aims at the development of a generic platform that serves as a basis for location aware applications and supports mobile users with handheld computer devices. The intention is to facilitate access to information which is useful at the current user's location.

Although the project is at it's first greater steps, a general scenario has been realized for supporting a traveler arriving at Stuttgart. Navigation to selected waypoints as well as WWW based information on companies and products in near range of the users are the main aspects of the introduced scenario.

The focus of Nexus is not directly Augmented Reality, but relies in mobile and ubiquitous computing. Features provided by Augmented Reality are only used for information display on virtual 'Litfasssäulen' (VLIT, advertising columns). These columns have a fixed location and can provide personal information to users in the near distance. In addition to that columns, navigation arrows and information can be displayed in a user personal HMD or on his handheld.
Because of the mobility and the context awareness, a model of the real world resides in the Nexus system that is used for computation of necessary information. This model can also get augmented to the user's view to give further information on certain objects.

The general architecture of Nexus is organized in three layers [76].
The top layer contains the user's personal client devices on which the applications run. Similar as in the WWW, applications play the role of Web browsers. These communicate via wireless communication with the middle layer.
That middle layer is the federation layer which contains the so called Nexus nodes. This second layer federates the different data sources of the bottom layer and provides a unified view of all information including that of the real and augmented virtual world to the applications.
Finally the bottom layer consists of various servers, that store the data of the Nexus system. Compared again to the WWW, these play the role of HTTP servers. That servers either handle static spatial data of the environment or handle continuous changing location information of mobile objects.
A more detailed explanation about all parts is given in the next sections, while figure 4.3 illustrates the arrangement of the layers.

The architecture of Nexus can also be described as seen in figure 4.4. Details are also explained in the following sections.

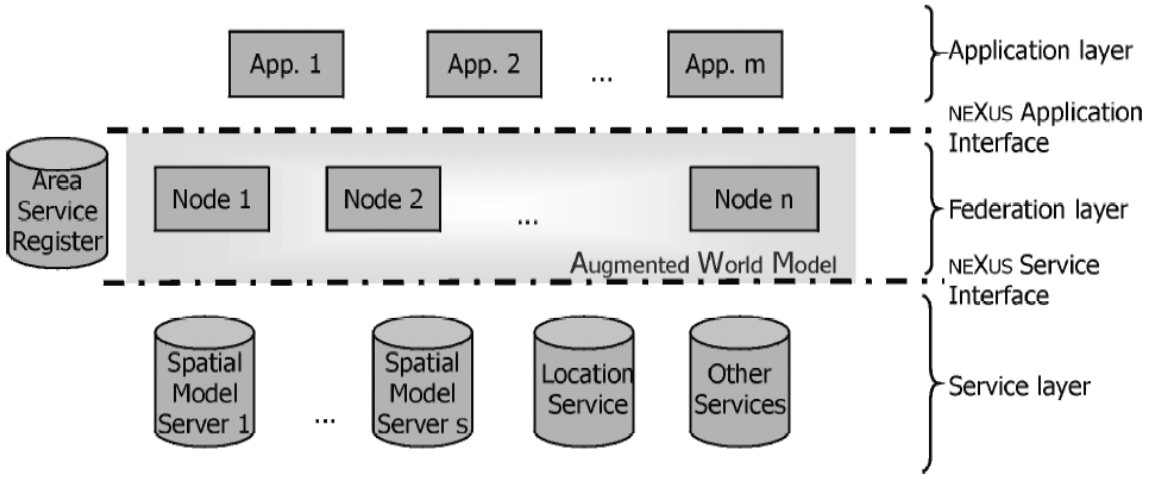


Figure 4.3: The Layers of the Nexus platform

4.2.1 Data Management

The main task of the Nexus infrastructure consists of the management of spatial models that represent the real world as well as additional virtual objects [109]. Because of large amounts of data on real and virtual objects of the Augmented World (AW) as a whole, area-dependent distributed autonomous Spatial Model Servers (SMS) store all static information of objects belonging to a certain environmental area, called Augmented Areas in correlation to the AW. [50, 107]
These Spatial Model Servers are realized by object-relational databases. Currently used DBMS are Oracle [14], IBMs DB2 [7], and Informix [9].

Advantages Object-relational databases serve very good for storage of data on any kind of objects, allow efficient searching and manipulating of object relevant data. In addition, the distribution of the handled data over a variable amount of model servers is necessary to balance the load of that servers.

Disadvantages The limitation of the announced SMS to certain areas works well for fixed located objects, but often it is necessary to relocate objects of the AW for different applications. Imagining the ARCHIE application, different architects may work on the same model at different offices, maybe in different cities. So absolute positions can not get declared to a corresponding SMS.

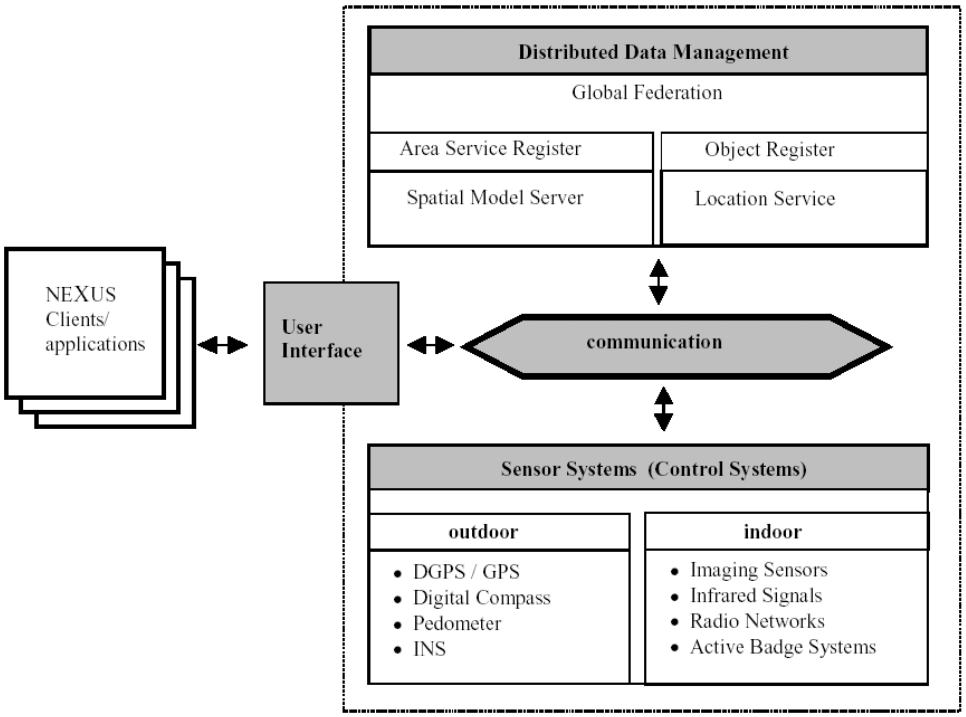


Figure 4.4: The Architecture of the Nexus platform

4.2.2 Data Structure

The data stored in the SMS must keep usable for different kinds of applications. Therefore Nexus provides a standardized class schema for data handling. A top level object is a *Nexus Object*. Derived classes of that are among others a *Nexus Data Object* covering sensor-, event- and also spatial object-classes. The last is the base class for every real or virtual object, about which information can be stored in the Nexus system.
The given structure can easily be extended to support further applications of the Nexus system.

Advantages A standardized data structure is useful for handling data. The already defined structure is able to handle common required tasks as navigation and object information.

Disadvantages When new applications requiring new data are added to the Nexus system, the class structure needs to be extended by new classes or fields within that classes. This does not aim to a reconfigurable framework as DWARF is designed to. A more generic approach would cover this aspect, where new data can dynamically get configured to the data structures.

4.2.3 Federation and Consistency

To provide a transparent data access upon these distributed SMS, a global federation system encapsulates the model servers and provides access to the Augmented World. The Augmented World is produced by the *Area Service Register* of the second Nexus layer. This register aggregates all SMS that register themselves.
Information retrieval is handled by a specified XML-query language, the *Augmented World Querying Language (AWQL)*. The global federation system decomposes the queries to area or object relevant needs and hands the subqueries to the corresponding registers. In case of area concerning queries, the area service register determines the relevant servers for static model information and sends the subqueries to the appropriate servers. These give the resulting data back to the global federation. In case of queries appointing to dynamic objects, an object register does the same for it's subqueries, determines the corresponding location services (these are explained more detailed in section 4.2.5) and also brings back the results to the global federation (see figure 4.4). The global federation aggregates the results and supplies the result to the Nexus clients.
If different bottom layer servers may have inconsistent data on the same virtual object, the mediating nodes of the federation layer are responsible to provide a consistent view on that objects.

Advantages The responsibilities for consistency rely in a dedicated mediating component.

Disadvantages Using XML as query language produces a significant runtime overhead for parsing the queries. The encapsulation of the base servers for data behind their registers and them again behind the global federation increases communication between components and by this decreases the processing time for requests. This is unprofitable for real time systems as DWARF is designed to.
In addition, if SMS really provide inconsistent data, there may anyway be problems with encountering which one is right for the specific task. So it would be more useful to ensure that only one data server holds the data of a specific object.

4.2.4 Geographic Information Systems

Data access from the SMS can be handled via SQL (SQL99). But the specification of SQL supports spatial queries just in a limited breadth. To cover the gap of spatial queries and to provide functionality to the user's personal devices, geographic information systems for shortest paths and spatial object selection are assembled in the Nexus platform. Currently the commercial GIS system ArcView [4] is used in Nexus.

Advantages GIS are useful to provide functionality for certain applications as, for instance navigation paths or for the determination of the nearest objects against a persons location.

4.2.5 Mobile Objects

Equally to static model data, dynamic data on objects, as mobile units are (e.g. the system users itself), is maintained by a object register that aggregates all location services. While all other information about mobile objects is stored in the SMS, their location can be requested from that services. The separation of mobile objects like persons was made, because databases or geographic information systems are not well suited for altering continuous position updates. [108]
The management of that location service is fully transparent encapsulated by the global federation service. This one queries required data by determining the corresponding location services via its object register..

Advantages It is useful not to write every position or orientation change to the persistent database, so a runtime location service is a helpful approach to this problemacy.

5 Survey of Data Management Technologies

Data Management Technology that is currently available can facility the Design and the Development of Efficient Components for the Framework

---

The last chapter illustrated different approaches to data management in distributed systems in the problem domain of Augmented Reality. These approaches used different persistence models as well as different consistency mechanisms.
These approaches in mind, this chapter will provide a survey about suitable technologies for data management in regard to the DWARF framework.

First, evaluation criteria for the later discussion of usability in framework components are collected. In advance major technologies for data management are evaluated and discussed: File systems, Databasesi, XML-databases and Tuple Spaces

5.1 Evaluation Criteria

This section briefly lists the evaluation criteria I established to get useful background knowledge for the later design of the services and for use in them to decrease implementation time and to get efficient implementations for the discovered issues.

Fast Data Retrieval To serve the real time constraint that lies upon Augmented Reality systems, access to data must be fast among other things.

Concurrent Access Within a distributed system, there may be more than one service, that needs information about the handled virtual objects or about application specific data at the same time. Several services must be able to access data concurrently.

Access at fine Levels of Detail Reusable framework components may often require just a few informations. For that reason, access to data must be able in fine grained dimensions. This is also necessary to minimize network load.

Complex Queries Various associations between objects may exist in Augmented Reality systems. So complex queries must be supported to range over various data sets.

Large Amount of Data Augmented Reality applications have to deal with large amounts of data about the virtual world. If information of the real world is also aggregated, the amount of data is even larger. Data managing systems must be able to deal these sizes.

Handling Associations As users may define relations between objects, these must be realizable down to the data managing systems to provide efficient handling.

Open Source The DWARF project is intended to be open source. It relies under the GNU General public license [5]. So all products associated with the project also should be under the GPL or at least be open source in any kind.

Common Query Interface The storage system used should have a query interface that is shared with others in that area. This ensures that the storage system is still later easy to replace if other, more performant or usable ones are developed.
In best case, a standardized Query specification would be very comfortable.

5.2 File systems

Files are storage abstractions provided by operating systems. Applications store their data as a sequence of bytes and define how and when data should be retrieved. As the structure is relative low level, applications can perform a variety of speed optimizations on reordering the file structure. But the applications themselves have to take care of concurrent access and loss of data on system crashes.

Discussion File systems are in common better used when unstructured or widely different structured data must be handled. Data in Augmented Reality systems is, in case of object definitions well structured. Thus explains why the use of file systems for an Augmented Reality framework is surely not sufficient.

5.2.1 Flat Files

Storing data in flat files delegates all responsibilities for data organization and access to the software system that uses them.

Discussion Even if large amounts of data can be receive very fast from a file, so do files not facilitate their use in distributed environments, because management of the structure lies by the application, even if access to files can be provided fully transparent by distributed file systems.

5.2.2 XML Structures Files

XML [22] structured files provide a ordered hierarchy of well defined entities of data sets. Elements are pre- and suffixed within the file by tags that describe their type. Standardized software implementation exist to build memory structures of such files.

Discussion XML based systems loose efficiency when parsing the XML-files, because the notation to store them produces a large overhead in size.
Even if fine grained access is provided by a standardized query language named XMLQuery[23], no concurrent access to the persistent state of files is maintained. Associations must also be maintained by self defined references.

5.2.3 Conclusion

File systems work well for manually maintained service descriptions, but they do not provide the features of concurrent access.

5.3 Relational Databases

Relational databases provide an abstraction of data that is higher than in flat files. Data is stored in tables that comply with a predefined type called *database schema*. Each column in a table represents an attribute and each row represents a data item as a tuple of attribute values. Several tuples in different tables are commonly used to represent the properties of an individual object.
Databases of this kind are very useful for large amounts of data. Mapping complex object structures to a relational database can be challenging.
Relational databases support accessors by standardized query languages as SQL[20] and in most cases provide functionality to increase the efficiency of search operations by indexing data sets.

Discussion Relational databases provide a mature technology. In SQL they provide a well structured standardized query interface and they can efficiently deal large amount of data.

5.3.1 MySQL

MySQL [12] is an open source database that relies under the GNU general public license. It supports cross platform support for a wide amount of hard- and software including Linux and Windows as well as MacOS X and Solaris. Also transactions are supported to guarantee an always consistent state of the stored data. Beginning with version 4.0, MySQL will support full text indexing which provides higher throughput when searching in the data. At

last, future versions of MySQL will support general geographic information system features as well as subqueries for complex queries.

Discussion MySQL seems to be a suitable candidate for data management in Augmented Reality systems, because it is among other things under the GNU GPL which is advantaged for the use in the GNU GPL based DWARF framework. In addition many required features are supported or will be supported in near future.

5.3.2 SapDB

SapDB [17] is also a open source database under the GNU GPL, but supports fewer platforms than MySQL, hence the major ones are covered. The features are nearly the same as in MySQL, except that no support for geographic information systems either exists nor is planned.

Discussion SapDB has nearly the same features as MySQL, except for the introduction of GIS functionality.
Not directly related, but useful to know is the fact that there have only been minimal efforts on documenting this database system.
It is said that SapDB scales better than MySQL on concurrent access.

5.3.3 Conclusion

Relational databases appear very good to match the needs of Augmented Reality systems to manage relevant data.

5.4 Object-Relational Databases

Object-relational databases provide the same interfaces and almost the same functionality as relational databases. In advance, they extend relational databases by the concept of inheritance of tables.

Discussion The concept of inheritance is interesting for the use in Augmented Realitysystems. Because every object has a position and an identifier, but other properties may vary, inheritance structures would fit well to the issue of storing data about virtual and real environments.

5.4.1 PostgresSQL

PostgreSQL [15] is released under the BSD license and provides also nearly the same set of features as MySQL, except full text indexing.
PostgresSQL also supports object-relational database behavior.

Discussion The fact of indexing tables would be a great opportunity, but unfortunately, PostgreSQL does not support this feature. By this, PostgreSQL is not as suitable as MySQL for the requirements of a storage system for Augmented Reality systems.
The fact of object-relationality powers PostgresSQL, as at least a large set of objects in an Augmented Reality system have positions and orientations and many others definitely can also get inherited.

5.4.2 Conclusion

Object-relational databases like PostgresSQL can very good be used to facilitate data management in distributed Augmented Reality environments. They can deal with almost all evaluation criteria.

5.5 Object-Oriented Databases

Object-oriented databases provide services which are similar to a relational database, Unlike relational databases, object-oriented databases store their data as objects and associations. Schema definitions are possible. Databases of this kind reduce the need to translate between objects and data storage entities. In addition to providing a higher level of abstraction OO-databases provide developers with inheritance and abstract data types. Databases of this kind are usually slower than relational databases for typical queries.

Discussion As object-oriented databases in common are slower than relational databases, this minimizes the advantage of easy storage of objects, that need no adaption to a table structure as of relational databases.

5.5.1 Goods

GOODS [6] is an object-oriented distributed database management system that uses an active client model. The multi-threaded database server is language and application independent. The client application interface to the database provides transparent persistency for common programming languages.

Discussion A distributed object-oriented database would perfectly match to a variety of the issues that exist for distributed Augmented Reality systems. But Goods scales badly on increasing amount of clients. And does not provide too many features that would be required to access data efficiently.

5.5.2 DB4O

db4o [2] is an object-oriented database which is entirely written in Java. It provides automatic management of the database schema. Objects can easily get pushed to the database without requiring changes to application code to make it storable. Comes in conjunction with the S.O.D.A. [19] querying interface.

Discussion Because it is written in Java, this one will not be the fastest. But also the common interface

5.5.3 Conclusion

Although Object-Oriented databases exist, the most of them are written in Java, which acknowledges that these are not suitable for the relevant performance in query response time.
In advance, most of all object-oriented databases have various different query languages they define. So replacing the database after development would be much more complex than by standardized systems.

5.6 XML-Databases

A short view should be taken in databases that can store XML-documents.

Discussion XML is, as seen in section 5.2.2, not to suitable for use in Augmented Reality. This gets verified when using XML-structures as output of databases. This in addi tion produces higher load on network.

5.6.1 Native XML Databases

Native XML databases fall into two broad categories:

Text-based Storage Store the entire document in text form and provide some sort of database functionality in accessing the document. A simple strategy for this might store the document as a BLOB in a relational database or as a file in a file system and provide XML-aware indexes over the document. A more sophisticated strategy might store the document in a custom, optimized data store with indexes, transaction support, and thers.

Model-based Storage Store a binary model of the document (such as the Dynamic Object Model (DOM) or a variant) in an existing or custom data store. For example, this might map the DOM to relational tables such as Elements, Attributes, Entities or store the DOM in pre-parsed form in a data store written specifically for this task.

Discussion There are two major differences between the two strategies.
First, text-based storage can exactly round-trip the document, down to such trivialities as whether single or double quotes surround attribute values. Model-based storage can only round-trip documents at the level of the underlying document model. This should be adequate for most applications but applications with special needs in this area should check to see exactly what the model supports.
The second major difference is speed. Text-based storage obviously has the advantage in returning entire documents or fragments in text form. Model-based storage probably has the advantage in combining fragments from different documents, although this does depend on factors such as document size, parsing speed (for text-based storage), and retrieval speed (for model-based storage). Whether it is faster to return an entire document as a DOM tree or SAX events probably depends on the individual database, again with parsing speed competing against retrieval speed.

5.6.2 XML-enabled Databases

Database systems, extended by an output interface transffering their own data structures to XML-documents, are called XML-enabled. XML-enabled storage uses schema-specific structures that must be mapped to the XML document at design time.

Discussion XML-enabled are implementations of the adapter pattern and provide an interface for reuse of relational and other databases. Otherwise there are no differences to native XML databases. The upper layer is often handled by standardized interfaces.

5.6.3 Conclusion

XML is, as seen in section 5.2.2, XML-structures are not to useful for use in Augmented Reality because of they resource consuming structure.
But XML-databases in common have less query interfaces and specifications than object-oriented databases.
They could somewhen fill the gap between the relational databases with their issues in building the database schema for object structures on the one hand and the unstandardized query structures of object-oriented databases.

So we should keep these databases in mind, because in the near future, such systems can be useful for user defined input on objects. Such input can often be handled easily in XML-documents that work also fine for developers when configuring some services.

5.7 Tuple Spaces

Tuple Spaces are also known as blackboard or repository architectures. Tuple Spaces handle their data in ordered multi-sets that are ordered in tuples.

5.7.1 Linda

Linda [10] was originally developed to be a global communication buffer for parallel processing systems. Data can be stored for reuse and continually refined. Linda's tuple spaces are comprised by three main principles: anonymous communication, universal associative addressing, and persistent data.

Discussion It looks like Linda is no more maintained.

5.7.2 JavaSpace

Although a working implementation is not yet available, Sun plans for JavaSpace servers to handle concurrent access and be able to store and retrieve data atomically. JavaSpaces provides a mechanism for storing a group of related objects and retrieving them based on a value-matching lookup for specified fields. This allows a JavaSpaces server to be used to store and retrieve objects on a remote system.
The JavaSpaces architecture also supports a simple transaction mechanism that allows multi-operation and/or multi-space updates to complete atomically.

Discussion Even if it is designed to serve distributed systems, the tuple accessing methods stores entries that they understand only by type and the serialized form of each field. There are no general queries in JavaSpaces technology, only 'exact match' or 'dont care' for a given field.

5.7.3 TSpaces

IBM's TSpaces[8] is a Java-based distributed-object architecture including a development platform, processing environment, and addressing mechanism. It's tuple spaces are based on the tuple spaces of the Linda prototype.
It has a built-in database that provides data integrity, transaction support, indexing support, and a simple query language (described as "much simpler than SQL, but better than the

overly restrictive ”formal” tuple queries offered by Linda[1]). But TSpaces can also be used as a global persistent communication buffer

Discussion There are two planned implementations of the tuple database. The memory-resident database uses simple file persistence for tuples and indexes. The heavy-weight solution uses the commercial IBM DB2 database. Both facts work against a use in a research Augmented Realityproject.

5.7.4 Conclusion

The Tuple Spaces technology is designed for simple persistent storage, such as storing a objects properties for look up by the it's id. This fits well for that kind of storage of virtual objects but does not support complex querying referencing various needs. It is useful to provide a wide interface for querying stored data for use in many services that may be developed in the future.

[1] http://www.almaden.ibm.com/cs/TSpaces/html/UserGuide.html

6 Overview of related DWARF Service Technologies

Since the first DWARF application various reusable service have been developed

Now it is on time to get a deeper view into the DWARF framework. The services that are developed have to access services of the framework and will be accessed from other services. This chapter gives a brief overview about adjacent services of the data managing ones and shortly illustrates what interfaces these have. This knowledge is helpful when later developing the system design of the components with focus on data handling.

The chapter is divided in two parts. The first documents services that existed before the ARCHIE project begun, while the second part informs about the services that have been developed in regard to the ARCHIE project.

6.1 Services Existing before ARCHIE

Several services and subsystems existed before the ARCHIE project was initiated. Not all services need to be listed, just that subset that communicates with the new components are relevant. In detail this are the tracking services of the tracking subsystem and the User Interface Controller.

6.1.1 Tracking

The services of the tracking subsystem provide information about real objects positions and orientations against a predefined origin.
To transmit data of that kind, a data type called *PoseData* has been introduced to the framework.
The PoseData or pose is a data type that aggregates an objects name, type, its position, orientation and a accuracy field. Finally, a timestamp is available, that gives information, about the time, a PoseData event was constructed. Figure 6.1 shows the PoseData's fields in UML notation.

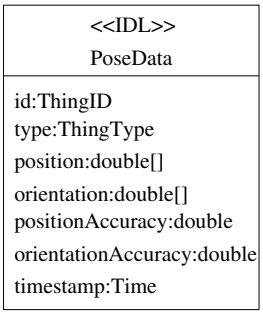


Figure 6.1: UML-diagram: The PoseData data type

The tracking subsystem will be used in the ARCHIE application to provide exactly the described PoseData to inform about real objects movements.

6.1.2 User Interface Controller

The *User Interface Controller* (UIC) is the service that handles any kind of discrete user input. It is realized by a configurable petri-net. Incoming events are set to places that are named in the incoming events *EventName* field and if a transition can switch, it does.
Transitions can use the events that have been placed on incoming places. Figure 6.2 shows the simplest petri-net, the UIC can handle and that is applicable for use in DWARF. What happens here is the following:

1. On startup always a place named 'start' must exist and this one is initialized with a token.
2. If a event is sent to a event consuming interface of the UIC and if this event carries information that it should be placed on the TestStringSenderJavaEvent place, the transition can switch.
3. On switching, the 'sendEventTransition' transition can get access to the UIC's event sending interfaces. If any events are constructed, these can be sent out.
4. After the transition is finished, the start place is set with a token again. That can be seen by the double arrows connection between the 'start' place and the 'sendEventTransition'.

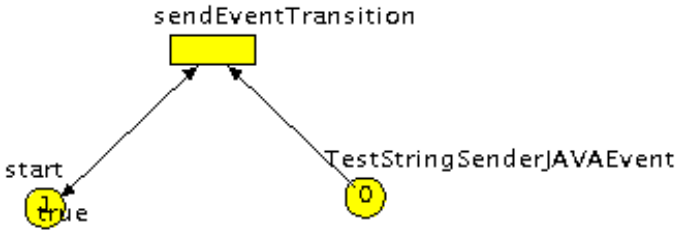


Figure 6.2: Petri-Net: The simplest petri-net that the UIC can handle

It is obviously that the UIC can easily be used to model task flows for user interaction. The UIC will be used in the ARCHIE system to handle user input and by this to control the workflow of the whole scenario.

6.2 A Service built in Context of ARCHIE

This section documents only one service that is of interest for the ARCHIE application. But this is the most relevant one.

6.2.1 The Viewer

Viewing devices like HMD and video beamer are attached to a computer that builds the hardware platform for the *Viewer* [58]. The Viewer is a DWARF service which does right in (real) place rendering of virtual objects.

Such a viewer needs a scene, that describes the objects shape and appearance to display. A common used structure is the scene graph. The rendering component provides a root node, an object, where a ordered set of children can be added. Such children can be basic shapes like boxes or spheres, but also transformations like translations, rotations and scalings are kinds of children. These children can recursively include other basic types or compositions of them.
The scene graph of the DWARF Viewer is the same as of the Studierstube project 4.1, so the example from this chapter can be referenced again on page 43.

The Viewer is intended to be a stateless service. Although it has a intern scene graph, this one can currently only modified by a relative flat interface that accepts object description in the OpenInventor file format [113].
To address this fact, the generation of the scene has to be handled by the data model managing components. This must be considered in the system design.

Another interesting fact is that the Viewer directly can handle PoseData input on selected objects. If a real object that provides PoseData via the tracking system is connected to a virtual object, it's position is aligned as the real one is moved. This has to be considered when realizing the data service that handles moving of virtual objects.

7 System Design

Design Decisions for Services that handle Data or provide further Functionality

Contained within this chapter is the system design of the data management services as well as the design of the ARCHIE application.
First of all, I introduce the important Design Goals I kept in mind while developing the new DWARF services. This is followed by a subsystem decomposition, a description about hard- and software mapping and a section about achieving persistence for the handled data. In advance, security, access and software control issues are explained for the new components. The collection is concluded by a section about software control.
Finally I describe the functionality provided by each DWARF service in detail, and show how these interact with the rest of the system.

7.1 Design Goals

Most of the collected goals reveal from the non functional requirements listed in chapter 3. But some have still been developed on standard criteria of software engineering and on criteria as described in [75].
The main design goals have been divided in the sections of Performance, Maintenance, Dependability and Usability.

7.1.1 Performance

This design goals include speed and space requirements that impose the system.

Response Time Creation, modification and deletion of virtual objects and their properties must be fast.

Low Overhead The data managing components shall consume few memory and computing resources, so that they either can run on portable computers or do not decrease speed of other services running on the same hardware platform.

Low Latency The latency of communications of all kinds should be as fast as possible to guarantee at least weak real time criteria.

Efficient Throughput As mainly 3D renderable scenes require a lot of data, data distribution must be efficient. The main focus must rely on reading data access. It is more often necessary to get some information, than to modify some values.

7.1.2 Dependability

The goals of dependability reveal the need for a system with minimal system problems and crashes and their consequences.

Reliability The services must perform their tasks as specified.

Robustness All services should work reliable even when multiple other services use them and produce high load.

Availability The components should run stable.

Fault Tolerance The services must handle network errors gracefully and still provide usability.

7.1.3 Maintenance

Maintenance criteria appoint to system changes and configuration after deployment.

Scalability The service components should scale to different configurations as well as to many network hosts and high amount of data.

Extensibility The services design must allow later extensions, so that new functionality can be added easily.

Modifiability The new developed services must be able to be modified for multiple applications and their corresponding needs.

Controllability Configured application as well as virtual objects and other data needs to be controllable during run-time as well as at the time, no part of that data is used.

7.1.4 Usability

Goals of this section deals with possible end-users of applications built with by use of the new services.

Presentability The performed development work on the services must be presentable in a demonstrative application.

Reusability The services interfaces must be reusable in different kinds of applications as well as for dynamic configuration of components.

7.1.5 Trade-Offs

Because of different design goals some trade-offs were necessary and due to the ARCHIE application, some other trade-offs were necessary.

Functionality vs. Development Costs and Presentability Some functionality as dynamic property modification, different appearances of virtual objects and template maintenance are specified in the system design, but have not yet been implemented. None of these features is inherently difficult, but would have required more implementation time, that would have been missing on other services that were required for the presentation of the ARCHIE application.

Performance vs. Low Overhead As graphical applications deal with a lot of data about rendering information, all required data was decided to store as near as possible (in the distributed system) at the 3D rendering engine to keep performance, even if consistency communication gained a larger overhead.

Security vs. Development Cost Security issues were not investigated, although they are required for data access in the private space, on service level and in the persistence layer. On the one hand, there was another student project dealing with security issues and on the other hand the aspect of security would have increased the design and development time.

7.2 Overview - The Developers Point of View

In addition to the users or stake-holders point of view that was introduced in chapter 2, this section brings up another point of view, the developers point of view.
The developers view concentrates on the development of modules for a framework. A module consists of hard- and software that provides a certain functionality to the user or to other modules. The software of a module consists of services that are connectable to others by use of specified connectors and information about the handled data types. Restrictions to connections are enabled by attributes and corresponding predicates. These parameters collected in a service description for each service allow the service manager to connect and start services if and only if all needs are satisfiable.
Developers who are going to provide new services to the DWARF framework have to specify static service descriptions for each new service and new data-types and interfaces to the framework. They can even reuse already specified interfaces in new service components.

Modeling Services The granularity of services matches the rank of the subsystem decomposition. So it is useful to introduce the notation style the DWARFresearch group uses at Technische Universität München.
The general structure relies on UML[1] [86, 85] in its new specification of the UML 2.0 standard [21]. Services are drawn as classes with their name and a stereotype announcing them as services. Abilities are modeled as arcs with their handled data type aside, while needs are modeled as circles.
Planned connectivities between services are modeled by directed dependencies. Therefore, connected services are shown with continuous lines.
Figure 7.1 gives an example service including its needs and abilities. A collisions detecting service is shown that requires PoseData of objects and that will send events about collisions on that objects.

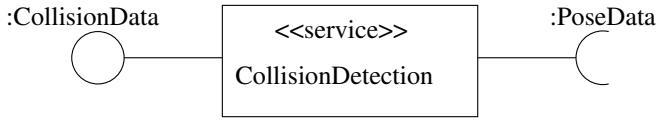


Figure 7.1: An example for DWARF services: A Collision Detection Service with a need for PoseData and a ability that provides data on collisions

Multiplicity of Services Before decomposing the system, a informative note must be taken. When a service is named, it does not mean, that just one service does really exist within the whole framework. Services can either reside on different hardware platforms or can, if configured so, be started in multiple instances.
So, naming a service appeals to its functionality, it provides to its environment.

7.3 Subsystem Decomposition

This section is divided in three parts. The first one concentrates on the new data managing services. The second provides the general ARCHIE system service decomposition in regard to the required application services. Finally the third concentrates on testing purposes of framework services.

7.3.1 Data Managing Components

In chapter 3 two major aspects were found. Management of world related data as virtual and real objects and dependencies between them. On the other hand, dynamic service configuration is an also relevant topic. Both aspects appeal to disjunctive solutions within

[1] Unified Modeling Language

DWARF services.
Because data is stored in a database and because object manipulation may happen very often, model managing components must be separated to both requirements.
By this we see three important tasks for data handling services: A scalable model, a service providing required data from a database and a service responsible for dynamic configuration.

ModelServer A service belonging to this area describes the *adapter pattern* [51] for the aggregated database and provide object information to the system. Also data manipulation on the level of objects and their associations is possible in a generic way.

Model While the ModelServer just provides object data and locking mechanisms for manipulation access, the model service is responsible for manipulating objects data. It provides a usable interface for other services and is scalable to various context factors.
This includes consistency management, but also direct access to viewing devices.

Configuration Services require an initial access point to configure themselves on startup and to manage their configuration dynamically. The Configuration service maintains runtime configurations of active and inactive services.
Services of this kind also reflect the adapter pattern

7.3.1.1 Design Rationale

There are several reasons for the division in three subsystems.

Two points of interest rely on object information: Consistency and Persistence. The declaration of a ModelServer service for persistence is already announced. But also a service for just providing consistency keeping functionality could have been introduced. By this, the Model service could just provide functionality to modify objects data, while another service maintains data exchange between all participating Models. But functionality of this kind is directly included in the Model service because most of the data is directly referenced for rendering to the users Augmented Reality device. So short communication paths are necessary to provide a high update frequency in all rendering services.

The separation of persistence providing services to a Model and a Configuration service had further reasons for introduction. A Model storing all information about the virtual environment could easily contain further object description about services and user preferences. But on the one hand, there are a lot of services that do not require information about an application specific virtual world, but only require user or application relevant data. So, a Model service would provide functionality that were used very seldom and, in addition data of this kind needs no consistency mechanisms because people are unique in their existence. On the other hand the distinct separation into two services supplies a Model service that can completely focus on the management of virtual objects and on related data

as, for instance relative positioning against other objects.

The use of the adapter pattern for encapsulating the used databases provides space for future modifications in the used persistent data structures as well as complete replacements of the databases through more suitable ones.

7.3.1.2 ModelServer

The ModelServer provides an initial interface for data access on any kind of objects. This services provides persistent data management and storage to the DWARF framework. Once another service is connected to this service, it can have a private session for manipulation of the configured virtual environment.
The functionality, the ModelServer provides, is covered by the following three paragraphs.

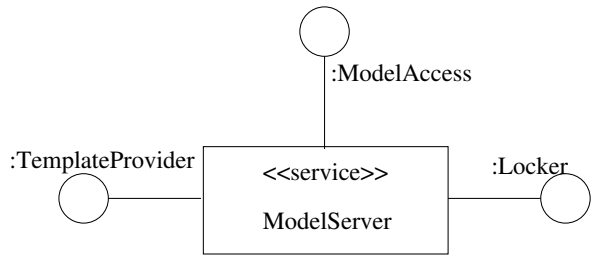


Figure 7.2: UML diagram of the ModelServer showing three abilities

Environment Manipulation Two alternative usages of virtual objects are possible. One takes a manipulate scene and provides this to the user. The user can interact with objects off his environment and their state must be kept even if the application is terminated. The other possible need for virtual objects is just temporary and requires no further persistence.
For the first kind of interaction, the ModelServer provides a ModelAccess, where objects can be created, objects properties can be set and objects can even be removed.
For temporal objects, a TemplateProvider provides configurable templates for use in other services.

Locking Each ModelServer is responsible for a set of objects. Each object exists only once in a ModelServer. To guarantee a unique write access to such an object, a Locker can achieve locks on objects. Additional services can still get read access on that object, but are not allowed to modify any data.
Because it is optional to get a lock on an object, the full range of flexibility is kept. Other services can still manipulate their local copy and are able to provide personalized scenes to the user.

Persistence Each object must have a unique identifier by which it can be found in every data structure. In case of the creation of a new object, a VirtualIdManager coordinates its ads. That identifiers must be stored in the database.
Access to the used database is provided by the concrete adapter for DataAccess .

Design Rationale Other services can receive private copies of parts of the stored objects and their dependencies between each other. Scenarios which will not be written back are realizable as well as short-termed user environment augmentation. Types of augmentation can be, for instance, virtual extended pointing devices or pure informative objects that can be modified in private, but need no persistence.
On the other hand, application or task specific objects can be stored and if required, can be locked. This locking mechanism makes it easier to keep track of consistency of Model services with the persistent data model. This persistent model could be manipulated, even if the Models themselves keep consistent.

7.3.1.3 Model

The Model service is the access point for other services that want to alter objects properties. This service reflects a dedicated data model and provides interfaces for selecting and manipulating any object whose properties are stored.
The following paragraphs illustrate the main topics of this service.

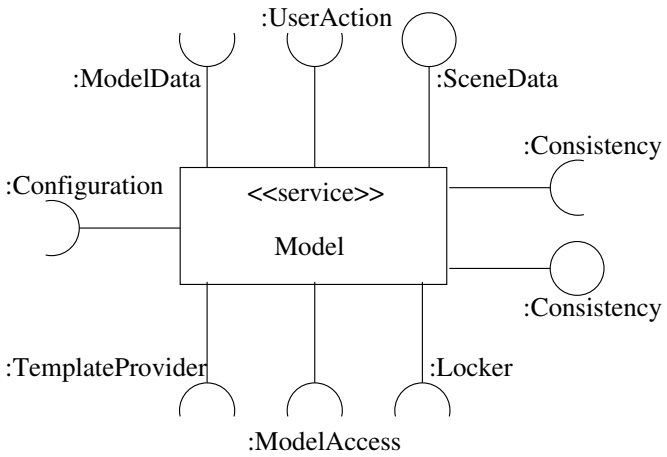


Figure 7.3: UML diagram of the Model service showing all needs and abilities

Data Modifying Interfaces As for discrete actions on a model a so called ModelData event is used, a ModelData Handler is responsible for receiving and treating them. Actions handled by this interface include basic features as object creation ad deletion as well as more advanced ones for publishing modified data to the ModelServer and other Models. The publishing to other Models results in visibility in other Viewers that are connected to these

Models.
Because position and orientation changes may occur continuous on different objects, Model services are responsible for accepting current PoseData information on virtual objects. This work is performed by the PoseDataHandler.
Modifications on other objects properties are modeled by a InputDataHandler that accepts input data generated by other services which are most often controlled by users.
The selection of objects to modify is handled by a UserActionHandler that also can aquire locks on objects.

Caching Unless no concrete publish action is performed, changes on objects states are cached within the ModelService. This task is fulfilled by the PublishCache. This component stores all newly created and modified objects as well as that objects which are deleted from a model.
The cache has a configurable so called dirty bit which allows publishing of objects every time, a change in any field appears.
Caching appears also at all connections to the ModelServer, where information about new objects is available.

Consistency Because more than one Model service can run and hold information about the same set of virtual objects, a consistency mechanism connects all Models with the same configuration to each other. If a Model publishes its changes on some objects, these are combined in one update objects which is propagated to all other connected services.

Generating Scene Graphs A SceneFactory traverses the abstract DAG of the model and generates a scene graph for use in the Viewer components. The generated scene-graph can be sent to the Viewer by use of the SceneData data type.
As the SceneFactory works on object templates, also other hierarchical structures can be created for further use in other services. For this, only the corresponding templates must be authored to the ModelServer.

Design Rationale The separation of object selection and object manipulation is useful in that sense, that also the Viewer can directly react on selection events. This facilitates just in time moving of virtual objects, as the viewer can bypass the longer communication paths via the Model. This is only helpful in pose changes, but not in changes of other object properties, because in that case, new scene graph information has to be generated.
The PublishCache was introduced to keep dataflow and processing time low, if object properties are modified by continuous streams. Until the modification is not finished and published, no changes will be propagated to other services except the direct connected Viewers. A case this appears is for instance a user performed color selection. The users TouchPadGloveService supplies continuous data of varying input data that is mapped to the objects color.
The configuration of the PublishCache is possible to two states. Changes can be published directly as they occur or in a set on a concrete publish action. Realizing both ways allows consistent states above all participating Models as well as just in time rendering on all

connected Viewers. This is useful for small applications with few environmental changes, while the concrete publish action is better used in larger setups with multiple partners and an intensive amount of changes.
Caching of object templates as well as of objects properties minimizes time overhead for inter service communication. Requesting data of both kinds requires the ModelServer and uses CORBA for communication. Especially, if the ModelServer resides on another host, the requiting time could be much longer than necessary for a real time demand.
DWARF as a whole is intended to act in real time, network communication have to use shortest possible paths. So another service creating acceptable data for 3D renderers would unnecessarily lengthen communication time between services. Because of this fact, the model component includes a subsection for the production of renderable data of virtual scenes.
In general, the scene factory produces 3D renderable scene graphs for the Viewer, it can also produce any other hierarchical structures for other services. This could be useful for further services which could rely on aggregated sets of information about virtual objects.

7.3.1.4 Configuration

Configuration may include users personal settings as well as information on users roles, performed applications or acting locations. The Configuration service provides a unique interface for services to configure themselves on startup and provides an interface for dynamic changes on any settings.
Configuration is possible on any composition of environmental attributes of framework components.

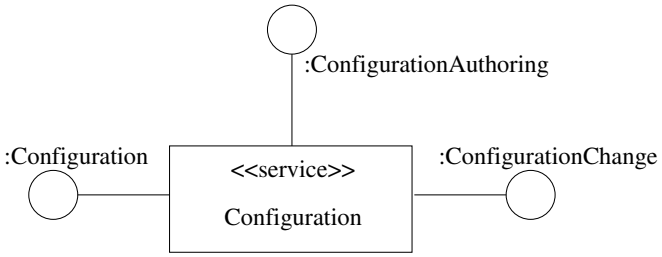


Figure 7.4: UML diagram of the Configuration service showing all abilities

Services Namespaces This service provides key-value pairs of configurable properties. For each service that can be configured to any environmental attribute the Configuration provides a ServiceHandler that offers a private namespace for the configuration values. This namespace can be authored by the corresponding service directly.

Authoring Authoring of new setups of services is facilitated via a ConfigurationAuthoring interface that enables the preparation of any kind of namespaces.

Persistence All data is directly received and written to a persistence layer.

Design Rationale Separate namespaces are necessary to provide reusability of services. By this, services can request the same properties even if they are configured to a new application.
The authoring interface was built absolutely free, because at the current time of development, many new services appeared in the framework and it was unclear which kind of attributes they would require until the final application was finished. See section 7.6 for further details on access control.
Finally, the configuration data is directly maintained in the persistence layer. This approach was taken, because modifications do not occur as often as virtual objects are altered. So, the focus of real time does not directly come into account for configuration.

7.3.2 ARCHIE System Decomposition

To build a demonstrative application, there is a need for further services. This section explains these services, which do in most cases not directly belong to data management in its characteristics, but that were required for the demonstration setup or for paradigmatic purposes.
By this, this section provides the general ARCHIE architecture. For this, already available services of the DWARF framework are reused as they were introduced in chapter 6.

7.3.2.1 Discretizing Continuous Streams

An already established service of DWARF is the User Interface Controller (see section 6.1.2). This service is intended to receive only discrete events. For instance these are user actions or position or even collision events of real or virtual objects.
There may be more services in future that require discrete events. For that reason a configurable generic service must be established in the framework. This service must be able to handle any type of events.

Figure 7.5 gives a schematic overview about the setup of services related to the Discretizer service within the ARCHIE application. The included services work as follows.

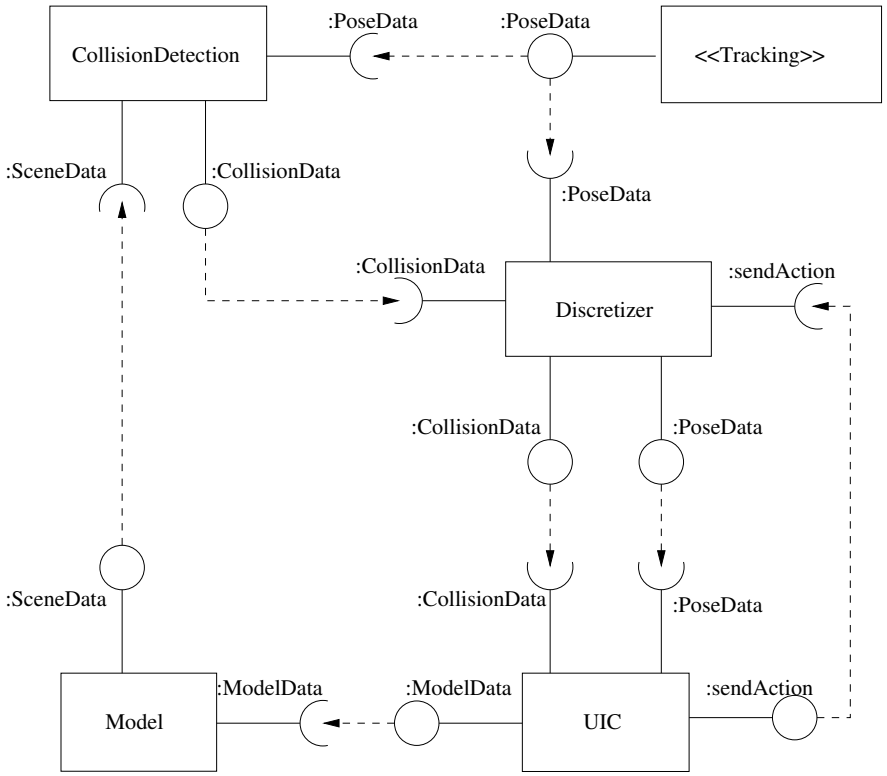


Figure 7.5: Schematic UML diagram showing the input side of the UIC and related services

Tracking This stereotyped icon reflects all services providing position and orientation information on a given set of real objects. The DWARF framework provides the already announced data type for this information, *PoseData*.

CollisionDetection Collisions are events that occur, if a real object touches a virtual object. A CollisionDetection service must have information about the virtual world as far as it is necessary for the concrete application. For this reason, it is connected to the Model which supplies SceneData that describes the virtual environment of the performed application.
In case of a collision between objects, this service sends *CollisionData* events. These events contain information about the participating objects and the position, where this event occurred.
Although a CollisionDetection is available from the SHEEP setup, it is not applicable for use in ARCHIE, so a new application specific service is required here.

Model The Model service provides all information of the virtual world to the CollisionDetection service. Here the first need for a data holding component can be seen.

Discretizer Because the UIC shall only receive discrete events, but the tracking and collision detecting service provide continuous events, a discretizing service is required that supplies this data.
This service is configurable to various data types of events and provides an interface to request the corresponding current event.

Design Rationale A service with the task of discretizing event streams is useful to keep network communication traffic low and to provide events to services which are only interested in specific events.

7.3.2.2 Detecting Collisions

Although the DWARF framework provides a CollisionDetection, this one is not reusable for the ARCHIE application. The existing service takes PoseData of real objects as well as of virtual objects. Figure 7.1 illustrates the needs and abilities of that CollisionDetection service. Because models of virtual scenes can contain a large amount of virtual objects, it would not be useful to add a PoseData sending ability to the Model service. That would overload the network communication as well as it would consume too much computation power of the platform the Model is running.
For this reason a new CollisionDetection named *PatternCollisionDetection* is introduced to the framework. This one analyses scene-graphs for the virtual environment and takes PoseData of real objects.

Design Rationale The separation of Model and CollisionDetection enables applications to initialize the CollisionDetection once on startup with a static scene. This is suitable if users are not allowed to modify object information and if no later communication is required.
The decision to develop this service as an application specific one has its reasons in being far distant to the thesis topic.
A suitable realization of a

7.3.2.3 ARCHIE Modeling Scenario Service Overview

Now, as all new services are described, their arrangement for the ARCHIE modeling scenario is illustrated in this section.
Figure 7.6 gives a general overview about the participating services. Hence the modeling scenario is dynamic with a varying amount of users it is impossible to give a static overview about the system structure. So, the figure illustrates the connectivities for one user and his dedicated services. But also the general required ones are shown.

Environmental Data Getting started with the Model, it provides a building site environment which is used to configure the other services that require environmental data. Hence only one Model is running, there is no need for consistency, so the consistency needs and abilities are not modeled. The Model itself receives its data from a ModelServer which is

able to provide this data., thus has the required templates and object information.
After startup, the PatternCollisionDetection and the Viewer are initialized with the environmental scene. So, the user is able to see the building site.

Creating Objects The tracking services provide pose information about location and orientation of controlled real objects. One is responsible for insertion of new ones.
As on startup also the users touchpad is configured with the required set of system interaction controls. The user can type a button for creating an object and the UIC that receives this event requests the current pose of the real object for object creation. The UIC then sends a creation ModelData event to the Model. This one contains the pose of the new object. The Model creates the SceneData and sends it to the connected SceneData receivers.

Moving Objects By typing a select button, the TouchPad service sends a boolean select event on which the UIC requests a collision event from the Discretizer. This event is only available, if a collision is produced by the PatternCollisionDetection.
After receiving this event, the UIC sends a UserAction event. The PoseDataHandler of the Model and that of the Viewer are connected to the tracking service which supplies the PoseData of the corresponding real object. The Model needs this data for storing the current objects location while the Viewer directly uses this data for as fast as possible rendering of the moving virtual object.

Deleting Objects Alike moving deleting is handled, but instead of the move button, the delete button must be pressed and a ModelData event for deletion is sent out. The Model receives this one, removes the objects and sends a delete information in a SceneData event to the Viewer.

Manipulating other Objects Properties By use of another select event, the UIC sends a ModelAction to the Viewer and the object is highlighted. The Model receives this event to and offers a need for the objects input data. Meanwhile the TouchPad is reconfigured to provide that input data for the objects properties. This is not shown in the figure, because it only happens dynamically.

Storing Back Objects Hence this is a demonstrative scenario, all built objects are stored back to the ModelServer on a concrete button click of the TouchPad. The UIC receives this one an delegates a ModelData to the Model which performs this action.

Design Rationale Even if data is hold redundant, this design allows a minimal flow of events throughout the whole system, but still provides a real time approach, because rendering is always directly connected to the tracking system.

7.3.3 Testing Services

For that purpose the DISTARB service was introduced. This service is able to be dynamically reconfigured to user selected connectors, types and directions. It is able to do remote CORBA method calls as well as it can send events.
Developers start the DISTARB service, configure it to the interface they want to test and start their service or services. DISTARB automatically matches the selected interface and opens the corresponding graphical interface.

Method Calls Users can select methods to call and execute them. Required parameters are requested and handed to the propagated message call.

Events Developers can configure the fields of the event to be sent and can select additional data types to be created and inserted into the event. Required fields are also requested.

Design Rationale Developing new services for the DWARF framework requires black-box as well as integration tests. If during development not all services are fulfilled in the required order, further testing tools are be suitable for that kind of tests. The developer can test his services functionality while other developers can concentrate on finishing their services instead of postponing this for testing the interfaces.

7.4 Hardware Software Mapping

This section describes required hardware and third-party software components that are used by the data managing services.

7.4.1 Hardware

Hardware components have to provide several features to facilitate efficient workflows for applications.

7.4.1.1 Fast Inter-Service Communication

As the DWARF system is intended to run at least in parts on mobile devices, the user dedicated interaction services should be able to run on small computers as PDAs or notebooks. This is most relevant for the Viewer component.
Because the Model in some cases has to deliver large amounts of data in a continuous row, its communication path should be as short as possible. Best it should run directly on the same hardware module as the Viewer.
Otherwise all services can run on every platform that provides the general communication

subsystems of the DWARF middleware.

7.4.1.2 Computation Power

The Model service can also run on environmental hardware, that in normal cases provides more computation power and memory.

7.4.2 Third Party Software

Software Engineering remembers us to reuse already existing software products. The area of data management has the sector of persistence that can be facilitated by third party software.

7.4.2.1 Persistence

As MySQL was directly available at the chair, this free software product was used, because it stands under the GNU General Public License (GPL). The GPL is required, because the DWARF framework itself also uses this license.

Rationale This open source database suits well for small, as for large systems, because extensive reuse of code within the software and a minimalistic approach to producing functionally-rich features provides a database management system that is efficient in speed, compactness, stability and ease of deployment. The unique separation of the core server from the storage engine makes it possible to run with strict transaction control or with ultra-fast transactionless disk access, which are appropriate for our situation.
Upcoming versions of MySQL will include support for a subset of the SQL92 with geometry types environment proposed by the Open GIS Consortium. This allows efficient storing and manipulating spatial data, including geographic data. Which could be interesting in future work.

7.4.2.2 Consistency

Therefore there might be additional products for keeping consistency between memory data-structures in distributed systems, related products could be included to the service components which cover that issue.
In the current version there is a designed consistency interface between the services which suit well for the demonstrative approach.

7.5 Persistent Data Management

This section illustrates the kind of storage of data that is handled by the corresponding DWARF services.
In other words these are the data for configuring DWARF services and any kind of information about the users environment, virtual objects.

The database tables are illustrated by entity-relationship (ER) diagrams [62]. Further details are shows in tables revealing concrete declarations of used data fields.

7.5.1 Service Configuration

Current state of the development of the DWARF framework revealed four attributes that describe the context of services: *application*, *role*, *room* and *user*. Services can use them to get configured to a contextual situation which may require additional configuration inside that service.

These attributes must be passed through to the persistent storage. Services who want to store their configuration need that attributes to declare their namespace to the Configuration service. Because properties of different services could overlap in their names and by this in their semantics. An additional identifier was introduced, the *configurationKey*. This one separates the namespaces to the granularity of services, because every service that requires internal configuration has to declare this field. Figure 7.7 shows an entity-relationship diagram of this database structure while table 7.5.1 declares all fields of the configuration database table.

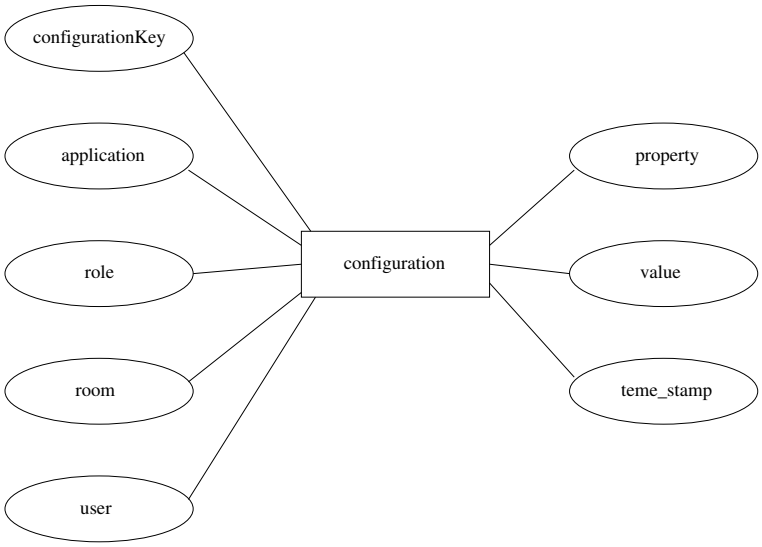


Figure 7.7: ER-diagram showing the fields of the configuration table

Table *configuration*:

Field	Type	Null	Key	Default
configurationKey	varchar(255)	NO	MUL	
application	varchar(255)	YES	MUL	NULL
role	varchar(255)	YES	MUL	NULL
room	varchar(255)	YES	MUL	NULL
user	varchar(255)	YES	MUL	NULL
property	varchar(255)			
value	blob	YES		NULL
time_stamp	timestamp(14)	YES		NULL

The *Type* column indicates the fields type. The *Null* column indicates fields if this field may not have a value (YES) or must have one (NO), because not every service must be configurable to all contextual attributes. The *Key* column declares that every field with a *MUL* index can have multiple equal values. The *Default* column should be clear.

Rationale This simple structure suits well for all required configuration data, even if a blob is used for the data. This allows to store even large XML-strings.
This structure is easily extendable by new attributes that could be declared in the future to describe new contextual properties.

7.5.2 Database Model of the Environment

The modeling of requires some more tables as the database scheme of the service configuration.

7.5.2.1 Identifying Objects

Each virtual object has a unique existence and by this a unique identifier. A table storing this data is the *objects* table. It contains a field for the objects identifier (*virtualObjectId*) and its *type*. Every object has a pose revealing its location and orientation relative against its parent. See appendix B. All relevant data is stored in the *objects* table;

Table *objects*:

Field	Type	Null	Key	Default
virtualObjectId	varchar(255)	NO	PRI	
type	varchar(255)	NO		
x	double	NO		0
y	double	NO		0
z	double	NO		0
rx	double	NO		0
ry	double	NO		0
rz	double	NO		0
rw	double	NO		0

7.5.2.2 Binding further Information to Objects

A requirement for data management was that further information must be bindable to objects. For data of that kind, the *object_properties* table is responsible. The *virtualObjectId* indicates the object, the property belongs to, while *property* gives the properties name. The value is stored in the *value* column.

Table *object_properties*:

Field	Type	Null	Key	Default
virtualObjectId	varchar(255)	NO		
property	varchar(255)	NO		
value	varchar(255)	NO		

7.5.2.3 Relations between Objects

Remembering the pyramid 4.1 out of chapter 4, it could be stored as two objects, the red lower part and the white upper one. If done so, the model database would contain three objects. Two would describe the red and the white box, while the third would compose both to a object. Compositions of this kind would be stored in the *object_relations* table.
The fields *parent* and *child* reflect *virtualObjectId*s and the *type* column gives information about the type of relation between two objects. In general, relations between objects are transformations of translations and rotations, wherefore a default value set this to a *transform* value. But also other types can be specified.
This table enhances the flat object model to a DAG when only using the direction from *parent* to *child*.

Table *object_relations*:

Field	Type	Null	Key	Default
parent	varchar(255)	NO	PRI	transform
child	varchar(255)	NO		
type	varchar(255)	NO		

7.5.2.4 Generating Usable Representations of Objects

Although it is imaginable that services directly deal with raw object information like the Model service does, but more often services require understandable formatted data. For instance the Viewer requires its data in the OpenInventor file format [113]. For representations of this kind, templates must be stored. These templates are patterns where the objects properties can get inserted by their property name.
Each template has a *type* that must be unique and can be used as primary key. The *pattern* column is realized as a Binary Large Object, because templates can have a large size.

Table *templates*:

Field	Type	Null	Key	Default
type	varchar(255)	NO	PRI	
pattern	blob	NO		

7.5.2.5 Providing Default Initializers

Because objects must get initialized when a new object is going to be created in an application, the system must provide default values The structure is that of the *object_properties* table, but instead of the virtualObjectId, just the patterns type is declared. By this it is possible to retrieve all properties for a specific template.

Table *default_properties*:

Field	Type	Null	Key	Default
pattern_type	varchar(255)	NO		
property	varchar(255)	NO		
default_value	varchar(255)	NO		

7.5.2.6 Rationale

The described fully features the current design of the services as they are realized. But also future refactoring to generic data types by DDLs for use in object-relational databases can be easy done by some little SQL queries.

7.6 AccessControl and Security

As already described on page 65 the issues of access control and security aspects have not been declared to a design goal. But questions of this area should be considered in future extensions of the data managing components.
In fact, the current revision of the data handling DWARF services do not implement any security mechanisms.

Design Rationale It is well known that it is a bad idea to add functionality for security aspects after design and development of new software components.
However, security aspects for the DWARF framework as a whole should be investigated in more detail. If a concept for security is developed, this could get extended down to the databases. By this, common databases functionality of access rights could get used.

7.7 Global Software Control

This section explains the design of the control flow in the new services. It describes what issues supervised for the services to run efficient and fast without blocking themselves.

Multithreading By the use of CORBA in the middleware, all services have to deal with multiple threads. Each service has its own control flow and can access various other services. So, every service has to supervise the threads accessing it from other services.

WorkerThreads Services who access more than one other service for a specific operation or services performing a lot of time consuming collinear executable work must deal with threads to provide efficient operation time to the system. On the other hand, services can be accessed by multiple services. Therefore, these also have to deal with simultaneous access. This performance goal is reached by WorkerThreads and a ThreadPool. See [93] for a general description about this topic.

Avoiding Timeouts The servicemanagers control methods for services require a short response time. To avoid unexpected behavior, at least that methods have to delegate operations to other threads as far as they are not required for return values.

Avoiding Deadlocks To avoid deadlocks, services that are accessed should update status information inside and handle the access requests separately in worker threads. This pattern of synchronization should be used intensively in the new services.

Asynchronous events vs. Synchronous Method Calls To address the design goals of performance and reliability, the use of method calls and events must be distributed carefully over the services interfaces.
Asynchronous events provide faster execution because notifications are sent out and the sending method has no significant overhead. But events can be lost and are by this not too useful for reliable communication.
On the other hand synchronous method calls are executed whenever a reliable connection is established, but have a longer execution time, because the calling method has to wait for the return values.
To keep consistency, the use of events is not suitable. This is addressed in the design of the services.

7.8 Boundary Conditions

This section explains special conditions, the services have to handle on startup, on shutdown and in case of exceptions.

7.8.1 Startup

Startup conditions appoint to the Configuration and the Model Service. The ModelServer itself required no additional startup handling.

Configuring Services Most of the services of the DWARF framework can be started on demand by the mediating servicemanager. An extra role plays the Configuration service. This one has to provide configuration for services depending on their contextual attributes. For this, the service has to run before other services are startable, because the abilities for configuring other services cannot be stored in static service descriptions. If the service is

started, the database is queried for available configurations and appropriate configuration abilities are generated.

Services for the Virtual Environment The Models startup is as follows: Once it is initialized with its configuring service description and all required services are connected, it requests its intern configuration from the Configuration service. This configuration contains information about which objects are to handle by the Model service. These objects are retrieved from the ModelServer and aggregated to a SceneData event. That event is propagated to the CollisionDetection and the connected Viewers.

7.8.2 Shutdown

On shutdown the information about objects the are contained in the current users scene must be written to the service configuration depending on the contextual attributes. In advance all data cached in the Model service which is not published must be written to the persistent storage.

7.8.3 Exceptions and Errors

System or Communication Failure If a host or module breaks down, the servicemanager hopefully finds another host, that is able to start a new service of the same kind and to configure it to the previous contextual attributes. After a short timeout, the new service is set up and connected to the services requiring the needs and abilities of the lost service.

Incorrect Events As the middleware also provides a generic events distribution mechanism, there could always be services in the framework that may supply events whose structure does not fit in the services event consuming interface. Wrong events have to be filtered and a log message must be generated, so that service developers can use this information to find out what is the source of that events.

7.9 Subsystem Functionalities

In accordance to the subsystem decomposition in section 7.3 on page 68, this section gives a more detailed overview about the interfaces of the services and the handled data types. See appendix C for the IDL definitions source code.
In addition to the data managing services also the services I has to develop in regard to the ARCHIE application are described.

For convenience, the class diagrams within this section reveal additional dash-lined boxes. The contained text gives information about the connector protocols [68, 70] of the corresponding CORBA interface. The servicemanager uses that connectors to find fitting

needs, respectively abilities.

I start this section with the description of the Configuration, because in advance some data types are also used for describing virtual objects.

7.9.1 Configuring Services

Services can be configured with properties. It depends on the services how they use these properties. A general data structure to provide this data was introduced by the *StringProperty*. As in most cases more than one property is required to configure a service, these can be composed into a sequence of StringProperties which is named *Properties*. Both classes are modeled via IDL structs. Therefore they can be distributed via CORBA to other services. Figure 7.8 illustrates the structure in an UML class diagram.

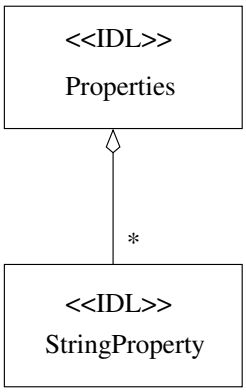


Figure 7.8: UML-diagram: Composition of StringProperties to Properties

7.9.1.1 Configuration Interface

On connecting a service to the Configuration a session [68] per service is created and the services namespace is opened, the service can get and set its properties.
The Configuration service acts passive here, so the service can decide on his own, when to request which data.

getProperty returns the value of a specified StringProperty.

setProperty sets the value of a also specified StringProperty.

getProperties returns all StringProperties composed in a Properties instance.

setProperties sets all StringProperties given in a Properties instance.

A workflow for getting a property look like follows:

1. By connecting a configuration requesting service to the Configuration service, a *ServiceHandler* is created and its Configuration Interface is handed to the other service.
2. After the services are connected, one calls the *getProperty* method on his instance.
3. The *ServiceHandler* constructs the database query and delegates it to the *DataAccess* by calling *executeQuery*.
4. The returning value is returned to the calling service.

7.9.1.2 External Authoring of Services

Services properties can be authored from the service itself via the Configuration Interface, but also from other services. Here also a session is created per service going to do some authoring. This interface also allows to generate new namespaces for new configurations of services.
Besides its large functionality, the interfaces declaration is very simple and works equally to the workflow description of the Configuration interface, except that instead of the ServiceHandler, the *ConfigurationAuthoring* class perform the work.

setProperty allows to set StringProperties keys and values to specified contextual attributes.

7.9.1.3 Propagating Configuration Changes

As the Configuration Interface only provides a passive interface, services that require dynamic configuration can offer a need for the *ConfigurationChange* ability. The same session is returned to the requesting service that already provided the Configuration Interface or a new one is created if no other exists. If so, each service will receive an event if at least a property changed. This event contains the corresponding StringProperty or Properties and is sent by a *ChangeEventSender*.
The workflow is in accordance with that of the Configuration Interface.

Figure 7.9 shows the Configuration service with its abilities and the classes handling them.

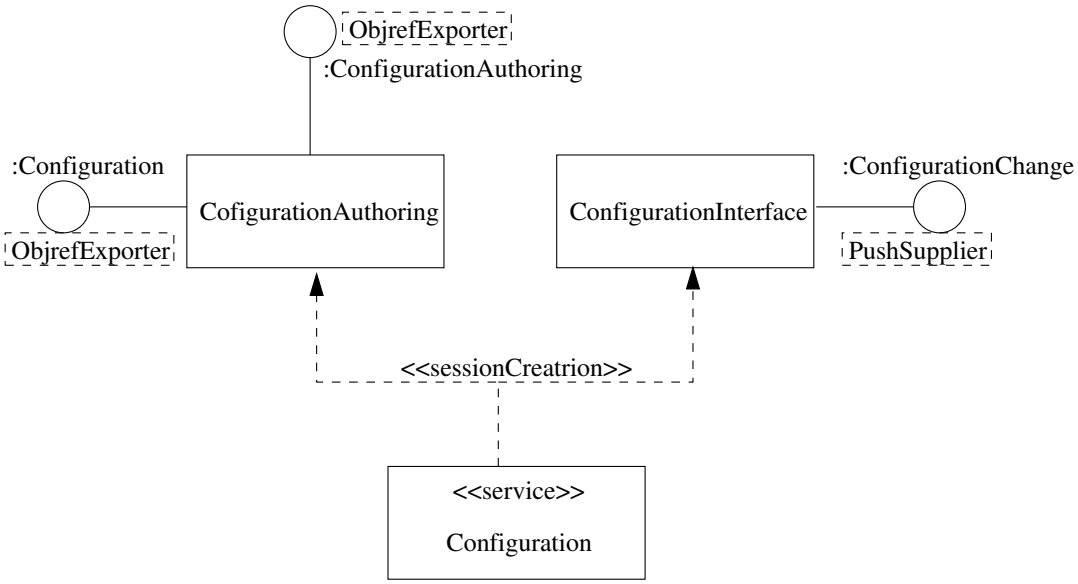


Figure 7.9: UML-diagram: The Configuration services interface with its connectors

7.9.2 Handling and Managing Object Information

A virtual object is a object that does not exist. Therefore no class *VirtualObject* does exist. Objects are described by the *ObjectProperties* class which can be seen in figure 7.10. For handling of a larger amount of ObjectProperties, these can be aggregated in a *ObjectPropertiesSeq* class.

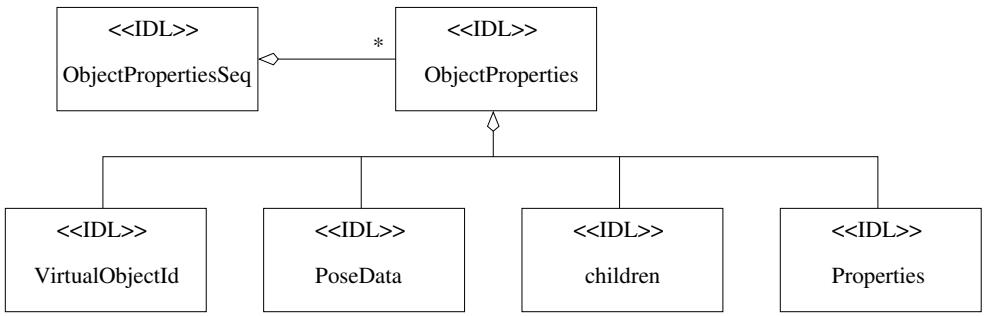


Figure 7.10: UML-diagram: The ObjectProperties IDL with aggregated IDLs

7.9.2.1 ModelAccess Interface

Above all, the ModelServer has to deal with information about virtual objects. The data has to be provided by an abstract interface. This interface is described by the *ModelAccess* interface. It allows the management of objects themselves, while the management of relations between objects is maintained in the ObjectProperties themselves. The interface is as follows:

createObject creates and returns a new object by its type in the persistent data layer.

getObjectProperties returns the objects properties by a given VirtualObjectId.

getDefaultProperties provides default values for a given type for object initialization.

setObjectProperties writes a given ObjectProperties back to the persistent layer if it was locked before.

deleteObject removes a specified object.

Object creation works as follows:

1. When *createObject* is called on the ModelAccess, this one calls *reserveVirtualObjectId* on the VirtualIdManager.
2. The VirtualIdManager reserves to next free identifier in the database and returns this as a VirtualObjectId.
3. The ModelAccess places the type of the object in the `objects` table.
4. Finally it queries the default ObjectProperties for the specified type, places the VirtualObjectId and returns them.

7.9.2.2 Gaining Write Access for Persistence

Services can use ObjectProperties in any kind, they want, but can only store back information if they have a lock on that object. Functionality of this kind is is provided by the *Locker* interface:

lockObject locks a specified object, if it is not already locked by another service.

unlockObject releases a lock.

The cases of locking and unlocking perform as follows:

1. If *lockObject* is called, the Locker checks it's set of locked objects, if this one is already locked and returns true or false. The Locker may also return an AlreadyLocked exception. This depends on it's configuration.

1. If *unlockObject* is called, the Locker releases the lock on the object. If no was set, nothing happens.

7.9.2.3 Creating Object Representations

Object information can be used in various ways. The most relevant one is 3D rendering by the Viewer. For this, a template mechanism has been introduced by the *TemplateProvider* whose interface is as follows:

getTemplate returns a template for a specified type.

The ModelServer class declares the service interface that can open up sessions for every new service that requires any interface. It includes the session interface [68]. If the service-manager requires a new session, new instances of the corresponding classes are created and returned.
Figure 7.11 illustrates the described class structure.

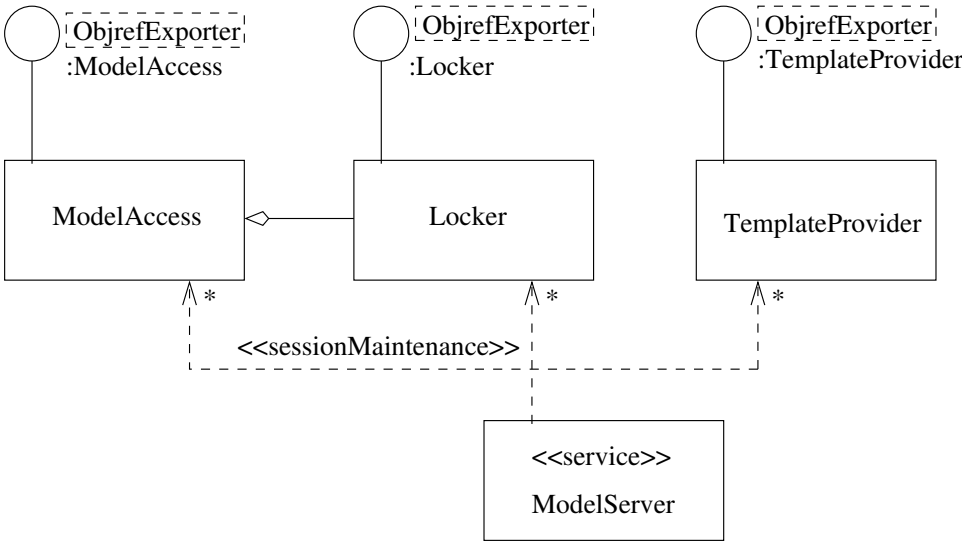


Figure 7.11: UML-diagram: The ModelServer's arrangement of classes

7.9.3 Providing Object Access and Scenes

To provide access to virtual objects, the Model deals with three different data structures. First, these data types are explained and then, by the gained knowledge about the data types the interfaces of the Model service are explained. See appendix D for the IDL specifications.

ModelData The ModelData is used to send information to the Model about actions to perform on the current data set. The actions are specified in an IDL enum structure [81, 13]. Objects can be created, deleted and published. Further actions will be to group and ungroup objects, and to edit objects properties. These are currently not listed, as they are not implemented. For further information about the state of the implementation see chapter 8.

To handle the specified actions, the Model service requires additional information. In case of object creation, the type and the pose is required. For deleting a object, its VirtualObjectId is used and publishing uses also the VirtualObjectId.

UserAction The UserAction is responsible for object selection and moving. Equally to the action in the ModelData, this one uses a enum of type UserActionType for selecting and deselecting objects. To connect a real to a virtual object, both identifiers must be given.

SceneData The SceneData is used to send scene information about the users virtual environment to the Viewer and the CollisionDetection. But by use of other templates, also other textual encoded 'scenes' can be sent to other services that might need information about the users world.

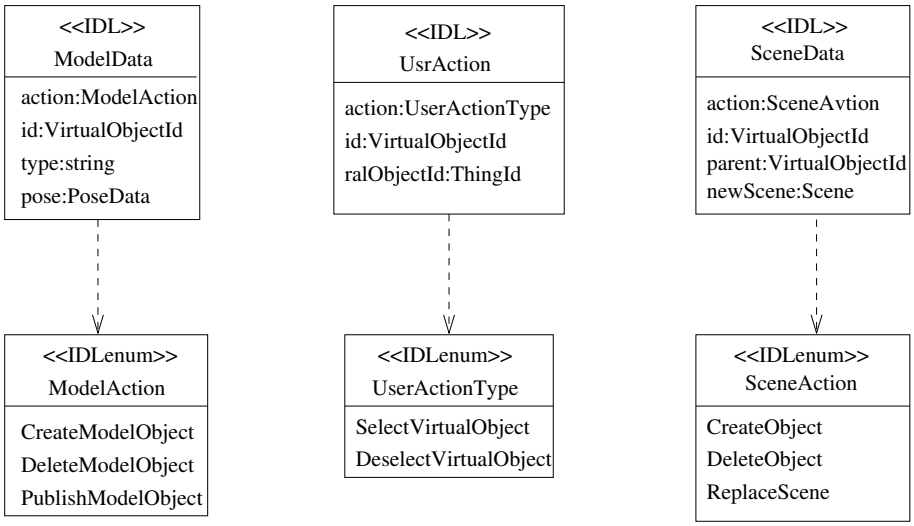


Figure 7.12: UML-diagram: The IDLs handled by the Model service

7.9.3.1 Getting Configured

The description of the Models interfaces follows the normal workflow behavior. Thus first of all the service configuration after startup is explained.

The *ConfigurationHandler* class realizes the adapter pattern for the Configuration Interface of the Configuration service. All methods provided there are implemented here, too, and simply call the remote methods.
But they also provide management of the service initialization. Before another class of the service can access data, the remote partner must be accessible. If so, the ConfigurationHandler reports to the Model. Figure 7.13 gives an overview in UML.

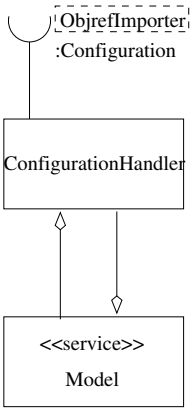


Figure 7.13: UML-diagram: The interface of the Model service to access its configuration

7.9.3.2 Accessing the ModelServer

When the Model is initialized by the servicemanager, also connections to the ModelServer are necessary. On startup for each of these three interfaces shown in figure 7.14 , exactly one session is created on the side of the ModelServer. The setting of the partner by the servicemanager and its handling is analog to that of the ConfigurationHandler. This is illustrated by the suffix 'Handler'. See figure 7.14 for a graphical description in UML.
All these connections are modeled by remote method calls to provide reliable communication paths between Model and ModelServer.

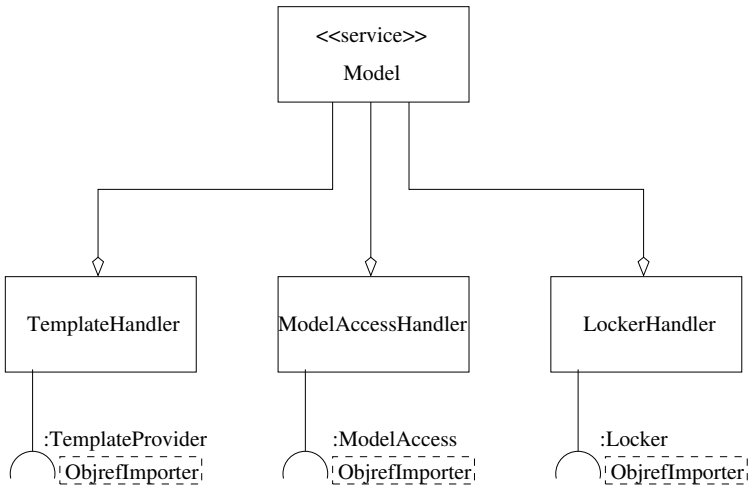


Figure 7.14: UML-diagram: The interface of the Model service to the ModelServer

7.9.3.3 Interacting with the Model

The reusable interface of the Model service consists of three event handling interfaces. Every one of them is realized again in an own class that maintains the CORBA communication. For every other service that is connected to one of this adapters, a session [68] is created. Taking a look at figure 7.15 reveals this information in UML.
An extra role plays the *SceneDataEventSender*. This service holds an event channel [68] that is set directly after the service is started.

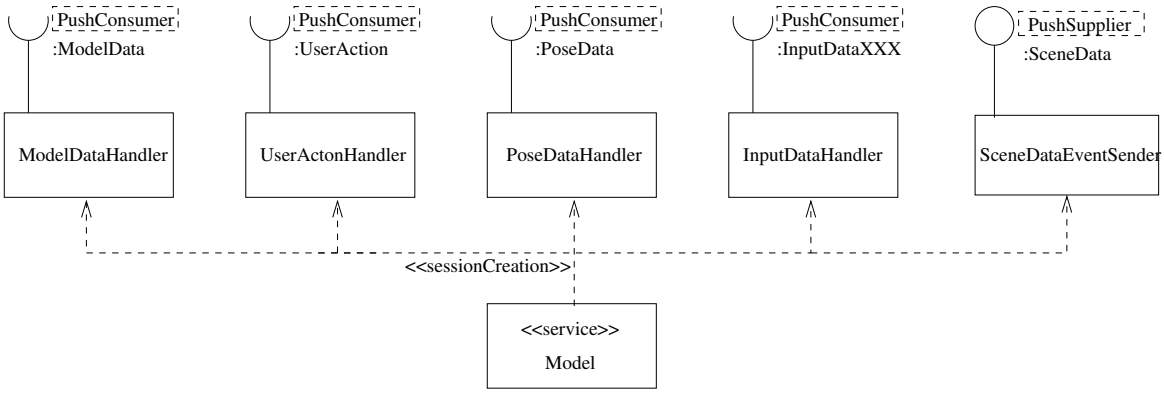


Figure 7.15: UML-diagram: The interface of the Model service to other services

Initialization of the Startup Scene Now, as the SceneDataEventSender is described, too, the workflow for the initialization of the startup scene can be modeled.

1. After the ModelAccessHandler, the ConfigHandler, the TemplateHandler and the SceneDataEventSender are initialized and have their partners, the Model that tracks all initializations calls *initializeStartupScene* on the SceneDataEventSender.
2. The SceneDataEventSender starts up a new thread that executes the method *initializeStartupScene_intern*.
3. This method requests the objects to display from the ConfigHandler by calling *getApplicationsVirtualObjectIds*
4. In advance, the the scene creation is delegated to the SceneFactory.
5. Finally, as the whole startup scene is constructed, it is sent out by a SceneDataEvent with action 'replace'. This is done by calling *send* on the SceneDataEventSender.

All relevant interfaces and classes for the management of virtual objects are explained now. Its about time to describe how other services can interact with the new services.

Intern Classes There are two more classes that are required for the Model to provide the functionality it is designed to. This is the *PublishCache* on the one hand and the *SceneFactory* on the other hand.
Every event handling class has a reference to the Model and by this can get an instance of both classes as well as the required instances of the classes communicating with the ModelServer. This is illustrated on the example of the ModelDataHandler in figure 7.16.

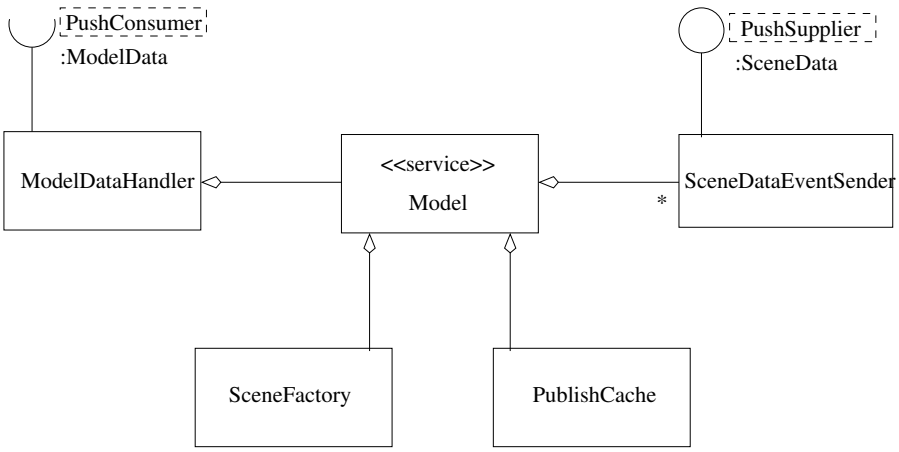


Figure 7.16: UML-diagram: The up to now unnamed classes of the Model service

Object Creation Figure 7.17 illustrates the performed operations in UML sequence diagram notation.
All other method calls that require objects from the Model class work in nearly the same manner.

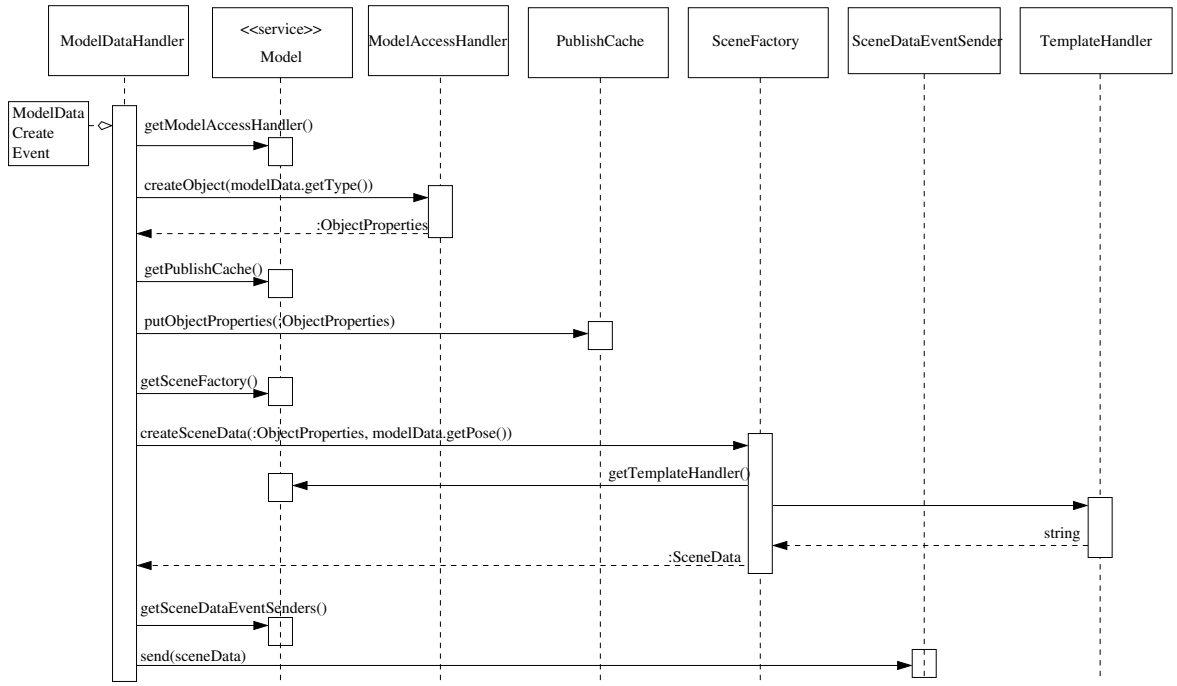


Figure 7.17: UML Sequence diagram: The Workflow in the Model when creating an object

Modifying an Objects Properties

1. The UserActionHandler receives a select event on a specified virtual object, but no real object is specified. It calls *lockObject* on the Locker that is received from the Model by calling *getLocker*.

2. The Locker requests *lockObject* on the ModelServer and receives a true or false to indicate a successful or not successful operation.

3. The UserActionHandler notifies the Model about the kind and amount of properties.

4. The Model generates a new need for the corresponding InputData types and sends a notification to the UIC to store back its need for InputData. Otherwise the UIC would receive wrong data and send uncontrolled events.

5. The InputDataHandler is connected to the users touchpad and gets the ObjectProperties of the selected object from the PublishCache by calling *getObjectProperties* of if not available from the ModelAccess by also calling *getObjectProperties*.

6. On touching defined sliders on the touchpad, the user can vary the values of the objects properties as far as the InputData types are applicable to the types. Setting textual information is not supported.

7. When the user finished and the InputDataHandler receives an 'finished' event, notifies the Model to resume its need for the InputData types and send the reactive notification to the UIC. Finally the UIC activates its need for the touchpad again.

7.9.3.4 Consistency Interface

updateObjectProperties updates given sets of ObjectProperties and deletes objects to be deleted on the interface that implements this method.

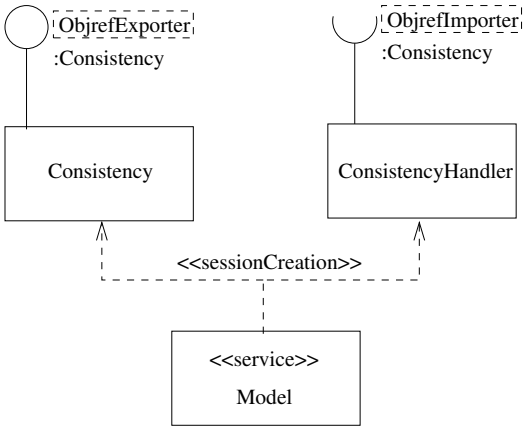


Figure 7.18: UML-diagram: The consistency interface of the Model service

Publishing all Modified Objects

1. A ModelData event is received by the ModelDataHandler, containing a publish action without a specified object.
2. The ModelDataHandler calls *flush* on the PublishCache
3. The PublishCache calls *handlePersistence* on itself.
4. All ObjectProperties are encapsulated in a ObjectPropertiesSeq object and are removed from the PublishCache.
5. The *setObjectProperties* method is called by the PublishCache on the ModelAccessHandler.
6. The ModelAccessHandler calls the remote method of the ModelAccess and this one directly writes them to the database, but only if they were locked. If so, the locks are released and a list of ObjectProperties VirtualObjectIds is returned.
7. Same is proceeded with a deleteList of VirtualObjectIds.
8. Now the PublishCache calls *handleConsistency* on itself.

9. By checking, if the VirtualObjectIds are listed in the returned lists from the ModelServer, the reduced ObjectPropertiesSeq are given as parameter on the calls to the ConsistencyHandler method *updateObjectProperties* as well as the list for deletions. This one calls the remote methods of the Consistency class as described above.

Receiving Consistency Updates In another service the updateObjectProperties method is called in case, consistency is required. The workflow is as follows:

1. As *updateObjectProperties* is called the handed lists are decoupled to the single ObjectProperties and the SceneFactory is delegated to create the corresponding scenes.
2. As the SceneData object return one after another, these are given as parameter to all SceneDataEventSenders methods *send* and by this are sent to other services.
3. Deletion is handled in the same manner.

7.9.4 Handling of Event Streams

Continuous event streams can be directed to the Discretizer. This service consumes streams by configured interfaces and provides a interface to request events of a specific type.
To deal with events, all interfaces the Discretizer provides consume or supply events.
Because for the ARCHIE scenario only pose and collision events are required, the interfaces of the service only have to deal with both kinds. For this, the service itself accepts events, stores them back in dedicated places for each type and can send one of them on request. Sending is handled by type dedicated event senders that inherit from a general EventSender (see figure 7.19).

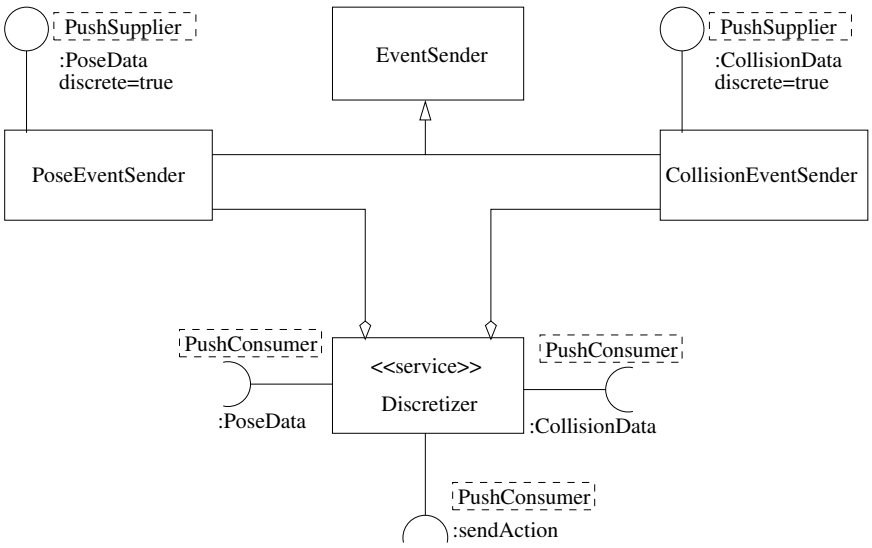


Figure 7.19: UML-diagram: The Discretizer service

To provide a generic configurable service that can deal any type of events, the dedicated classes can get released from the service and the EventSender can be used to send any kind of events.

7.9.5 Detecting Collisions between Objects

Collisions between real and virtual objects are found by the PatternCollisionDetection service.
On startup this service should receive a SceneData event that configures the initial users environment. This event is maintained by the *VirtualObjectEventHandler*. Thereafter additional events can arrive and manipulate the 'world'.
If real objects are tracked, their PoseData events can be received by the *RealObjectEventHandler*. For each event of this type, all virtual objects are checked for collisions at the pose events location and, if a collision occurs, a corresponding CollisionData event is sent by the *CollisionEventSender*.

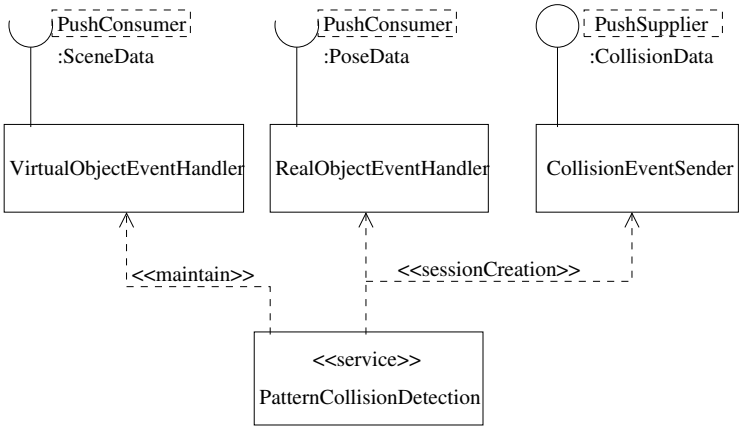


Figure 7.20: UML-diagram: The PatternCollisionDetection services interface

7.9.6 Handling the Users Input

As introduced in chapter 6, the UIC has a configurable interface. For the ARCHIE application its interface is configured as described in figure 7.21. There are no specific classes associated with the needs and abilities, because the handling of these classes is UIC intern.

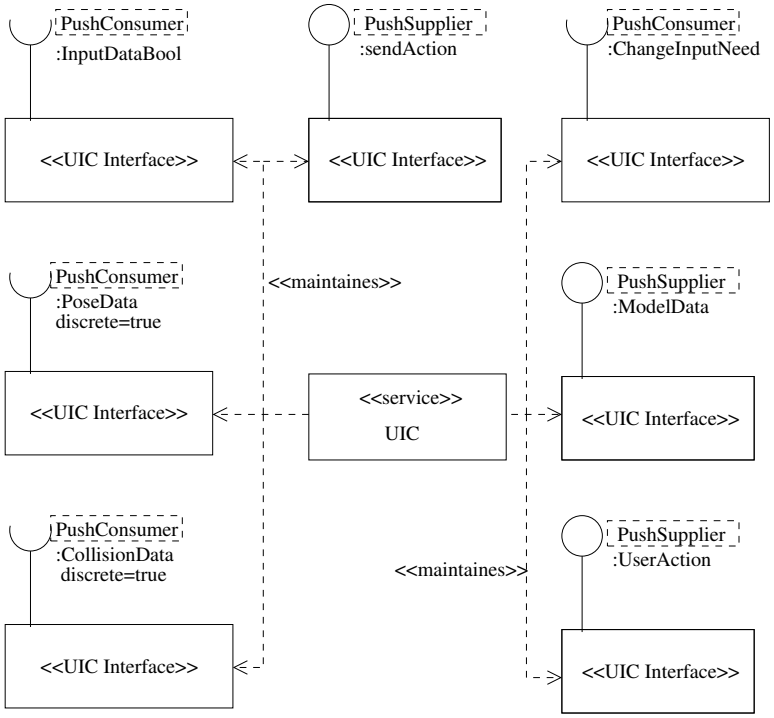


Figure 7.21: UML-diagram: The interfaces of the

Users actions arrive at the InputDataBool need. In correspondence to the petri-net places, InputDataBool events are set on these places. As the net does its transitions, the other needs and abilities come into use.
For object creation, the PoseData need is is required, while for deletion and selection the CollisionData need is used. Events will arrive, if the sendAction ability is used to inform the Discretizer to send the corresponding event.
The two abilities on the right side of the diagram are used to send events about object selection (UserAction) and object manipulation (ModelData). For more details about the UICs internal petri-net, chapter 8 may be referenced.

7.9.7 Testing Service Functionality

The DISTARB service itself does not provide any static interfaces. The service can be configured to any output required or the accept any classes also required. See chapter 8 for more details.

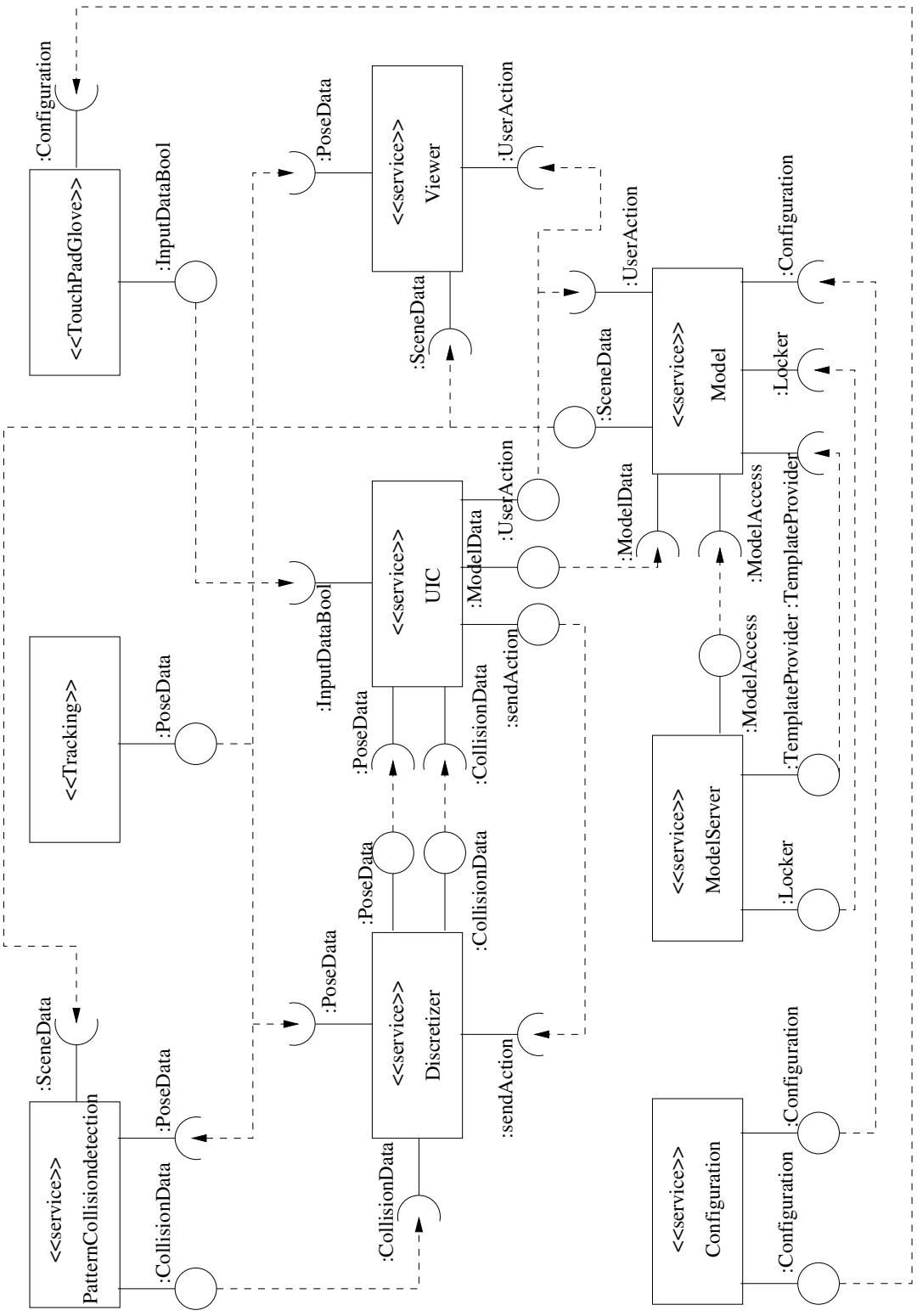


Figure 7.6: Schematic UML diagram revealing the general connectivities between the services participating in the ARCHIE modeling scenario

8 Implementation of the DWARF Services

"Think horizontally - implement vertically." **(Book of Douglass, Law 69)**

In contrast to the last chapter which described the general structure of the services that are developed for the DWARF framework, this chapter is more technical, as it documents the first implementation of the new components.
This chapter is of interest for such people who are going to extend the services further.

First I give some general statements on the implementation. In advance, a section is dedicated to all services that were realized in context of my thesis and the ARCHIE scenario. For every service a overview about service intern classes is given, if required, the state of implementation is explained and information about implementation specific details provide a deeper view into the services structure.

8.1 General Statements

Before I iterate over all services I realized during the implementation phase, some general announcements must be made.

Depositing Classes Each service is placed in an own directory of the DWARF directory hierarchy. Even if some services, for instance the Model and the ModelServer could have been placed in the same file system folder, it appeared useful to separate each one. By this separation, each service, particularly the Model service, is able to compile independent of any others that may require some maybe not installed libraries. So, the Model service can be started on every Machine, even if no ModelServer can be compiled on that hardware.

Naming Convention Classes that implement any interfaces are not allowed to have the same name as the interfaces. Is fact appears on the CORBA IDL interfaces. Because services that want to communicate with others by method calls have to narrow [13, 81] on the IDL interface, the IDLs are named by the general names. Classes implementing these interfaces are named with a suffix `Impl`. This indicates that they implement the particular interface.
For instance, the class implementing the *ModelAccess* interface is named `ModelAccessImpl`.

Debugging During development and for status information, logging and debugging information is necessary. For that reason I used the DEBUGSTREAM library in C++ and log4j in Java. Both are well known logging APIs and provide useful functionalities.

8.2 ModelServer

This section portraits all steps and information required to get a deeper overview about the services intern structure.

8.2.1 Object Design

In the phase of Object Design the gap between the system design and the implementation is closed by additional functionality and supporting classes.

8.2.1.1 Sessions

The ModelServer class declares the DWARF service interface with all corresponding methods. It also includes the session interface [68]. If the servicemanager requires a new session for a service, new instances of the corresponding classes are created and returned. The corresponding classes are the TemplateProvider, the

8.2.1.2 Supporting Classes

For the ModelServer there are two more classes which were briefly announced during system design, but were not integrated in the services interface documentations. This are the VirtualIdManager and DataAccess class. Figure 8.1 gives a complete overview about these classes and their aggregation in the services structure.

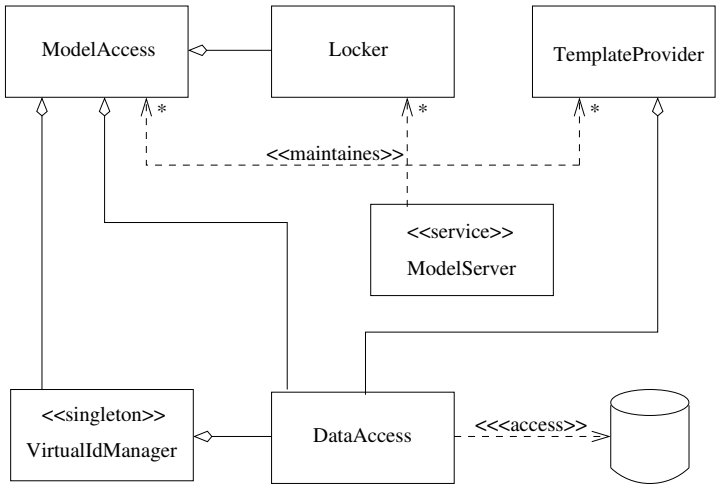


Figure 8.1: UML diagram of the ModelServer showing all classes

DataAccess As one can see, this class encapsulates the used database and manages the log-on to receive data. Its only functionality is to execute queries on the database and to return the result in kind of a generic tabled structure.
The DataAccess class encapsulates a bridge pattern for replaceable databases.
Classes that require access to the database are the ModelAccess, the TemplateProvider and the VirtualIdManager. Each one of them holds a reference which is set during object creation. That happens when a new session is opened to a Model service or another service that may in future access objects data directly.

VirtualIdManager As announced in section 7.3.1.2 on page 70, the VirtualIdManager manages the allocation of new virtual objects. Therefore it implements the singleton design pattern. Only this way, it is guaranteed that the virtual objects identifiers are unique.

8.2.2 Implementation

Programming Language The implementation is done in C++ [97] to guarantee fast execution of method calls.

Database For persistent object storage, the MySQL database was used in its version 3.23.40. I took this database, because it is open source under the GNU General Public license and works fine on our chairs development environment that mainly uses Linux as development platform.
Although MySQL currently does not support sub-queries which will be of interest for future data retrieval and also does not support geographic functionality, Both will be available in future versions.

But MySQL is implemented efficient and scales even under high load, so that even large and many Augmented Reality applications can concurrently deposit their data.

8.2.3 State of Implementation

The ModelServer is implemented except for the following points:

- The Locker is currently only realized as a bare skeleton. For the ARCHIE application no access control on objects was required.
- Because of the missing Locker, no consistency list for objects is generated by the ModelAccess if `setObjectProperties` is called.
- The VirtualIdManager does not yet realize the singleton design pattern. It is a normal instantiatable class.

8.3 Model

The Model service is the largest service I implemented for the DWARF framework. But already all classes were announced during system design, because all are required for specific operations.

8.3.1 Implementation

Programming Language The Model service is also realized in C++, because this service has to deal with a lot of data and therefore should by itself consume less memory space and also should run as fast as possible.

STL Because all data must be stored in memory, the Model services' classes make intensive use of the classes provided by the Standard Template Library (STL) for C++.

Adapters Some classes need to invoke remote methods or may send events. These do not directly call the corresponding method, but call a method with equal signature on a local class that encapsulate the remote classes. The local classes realize the adapter pattern and provide debug information.

8.3.2 State of Implementation

- Due to difficult interface specification with the Viewer, the complete handling of InputData of all kinds is missing. To provide this functionality, the InputDataHandler class and an event interface, including IDLs, to connect to the UIC must be realized.

- For the fact of manipulating objects data, the database schema needs a additional column to determine the kind of a objects field. This information must also be added to the ObjectProperties IDL.
- Grouping objects must still be realized. We postponed this during the preparation of the ARCHIE demonstration, because the tangible user interface (Touchpad) is not suitable for the required amount of buttons.
- Publishing of single objects is missing, currently all objects are published.
- Efficiency provided by use of workerthreads is currently missing, but can be added easily to the classes structures.

8.4 Configuration

8.4.1 Object Design

Generating Abilities Nearly all classes of the Configuration service are already explained. The AbilityGenerator is still missing.
This class is invoked on startup by the Configuration class, queries the database for available service configurations and generates the configuration abilities. Figure 8.2 describes the setup in UML.

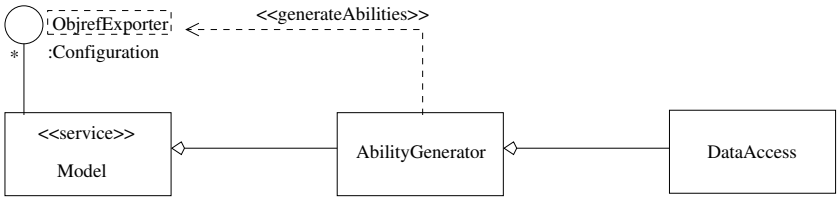


Figure 8.2: UML diagram showing the AbilityGenerator and adjacent classes

Handling Services One last note should be made to the ServiceHandler and the ConfigurationInterface.
The name ConfigurationInterface is useful for services to narrow [13, 81] on this name, so every developer know that he deals with the Configuration. But the implementation within the Configuration service handles other services as sessions and by this provides a ServiceHandler for each connected Service. Figure 8.3 illustrates the realize inheritance.

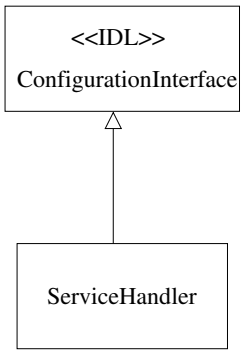


Figure 8.3: UML diagram showing the inheritance of the ServiceHandler

8.4.2 Implementation

Programming Language Also the Configuration services classes have been implemented in C++.

Database The database access is handled equally to the ModelServer. See paragraph 8.2.2 on page 104 for details.

8.4.3 State of Implementation

- The ConfigurationChange event interface is not yet implemented, because currently no dynamic configuration changes are required. Therefore the ChangeEventSender is also missing.

8.5 Discretizer

8.5.1 Object Design

The Discretizer stores the always newest event it receives from a stream. In some cases, a stream may provide events with longer dead-times between. To handle this, the Discretizer must know about a timeout, after which, no event must be available for a request. Therefore two more classes were realized to store events, their content and a timestamp. The timestamp is generated when the event is received. Figure 8.4 shows the structure of this classes. As seen, the PoseContent and the CollisionContent also store the events itself. These are used to minimize object creation time overhead.

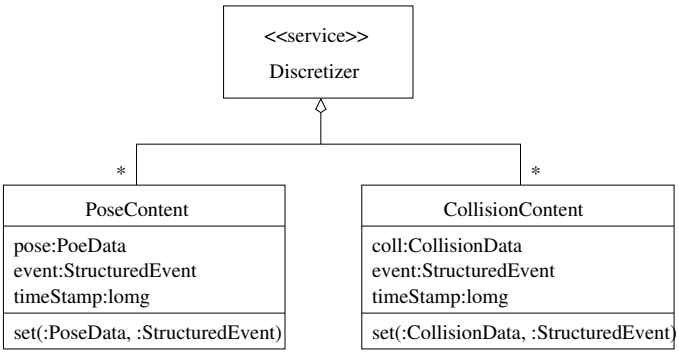


Figure 8.4: UML diagram showing the classes, the Discretizer uses to store events and their content

8.5.2 Implementation

Programming Language The Discretizer has been implemented in Java [40] for several reasons.
It does not have to deal with much computation on data except it receives a large amount of events. Also, not every event needs to be stored. Only the most current event does.
In addition, in future Java's feature of reflection will be used as illustrated in the next section.

8.5.3 State of Implementation

- The service is not yet generic as initially designed. To be generic, reflection must be used, because events may contain data that describes their affiliation in more detail. This will be easily done in Java if required.

8.6 PatternCollisionDetection

The PatternCollisionDetection is only used by the ARCHIE application and will hopefully in future be replaced by a more powerful one.
To provide a general CollisionDetection to the DWARF framework, an OpenInventor implementation should be used to maintain objects as well as their shapes.

8.6.1 Object Design

The PatternCollisionDetection receives events about virtual objects and therefore has to build up a virtual world. This world is maintained by the *World* class. The World stores each virtual object in a data structure named *Something* with their absolute position. All, also every Something is collected in a intern set of the World (see figure 8.5). In case a new event of a real object is received, all virtual ones are checked and in case of an collision, both

events data is aggregated in a CollisionData which is sent out in a new event.
Collisions are detected by the euclidic distance [33] of the two objects PoseDatas and by this does not maintain the objects chape.

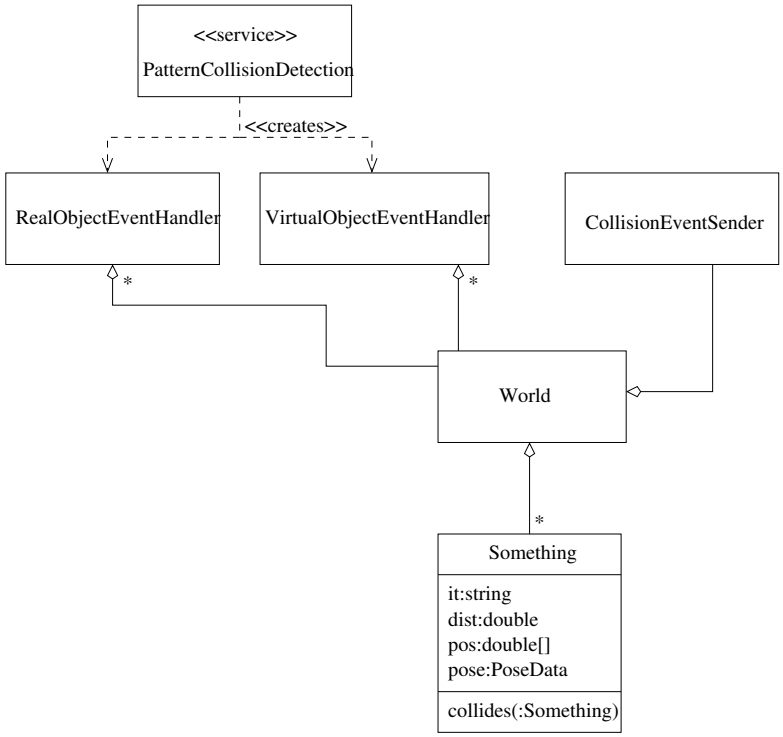


Figure 8.5: UML diagram showing the classes of the PatternCollisionDetection that are used to maintain the scene

8.6.2 Implementation

Programming Language The PatternCollisionDetection has been implemented in Java [40, 93].

Collections The PatternCollisionDetection makes use of the Java Collections API to work efficient on large amounts of data. [31]

Objects Sizes To associate a approximate size to an object, the *dist* attribute has been added to each Something. It is generated from the objects shapes scaleFactor. It is evaluated if the distance between two objects is created. Only if the computed distance is smaller than the dist value, a collision is generated.

Efficient Data Handling To minimize value retrieval, the objects position has been extracted from the PoseData, but is also stored to return the tracked objects complete pose in case of a collision. [53]

8.6.3 State of Implementation

- The PatternCollisionDetection is completely implemented as far as it only was realized for the ARCHIE demonstration.

8.7 DISTARB

The DISTARB service is intended for testing purposes. During the development of the ARCHIE application it came up that all services will be integratable in non deterministic order. To limit timeouts, a service was required that allows testing of services functionalities. This is the DISTARB service.

Figure 8.6 shows the services startup screen. Here the kind of connection, a need or a ability can be chosen. Also the kind of connector is selectable between PushSupplier and ObjrefImporter. After clicking 'activate', the new service description is propagated to the servicemanager.

Figure 8.7 illustrates to interface for method calls. Because the CORBA-stubs [13, 81] provide no direct information about the handled type of object before a narrow operation is performed, the user has to select the right object type to narrow to. After narrowing, the drop down list of the methods is filled and the user can select, which method to call. If any arguments are required, these are requested in order of appearance.

Finally for the handling of events another frame is opened up. Here the user can fill some fields. The DomainName is usually 'DWARF', while the TypeName reveals the type of the object within the RemainderOfBody. The EventName can be used by event receiving services to do additional handling. OptionalHeaderFields and FilterableEventBody are not used, but implemented to handle strings for extra purposes. Finally the RemainderOfBody can take any object. For strings, their content can directly be typed in the text field, while for other objects, the Combo-Box is used. Object instantiation is also graphically guided as on method calls.

8.7.1 Object Design

Within the DISTARB service the class named equally is responsible for Managing event sending and importing object references. It has a graphical user interface which again has a LayoutManager named RowLayout.
In case any objects are required the DISTARB class has to know abut types to handle. This

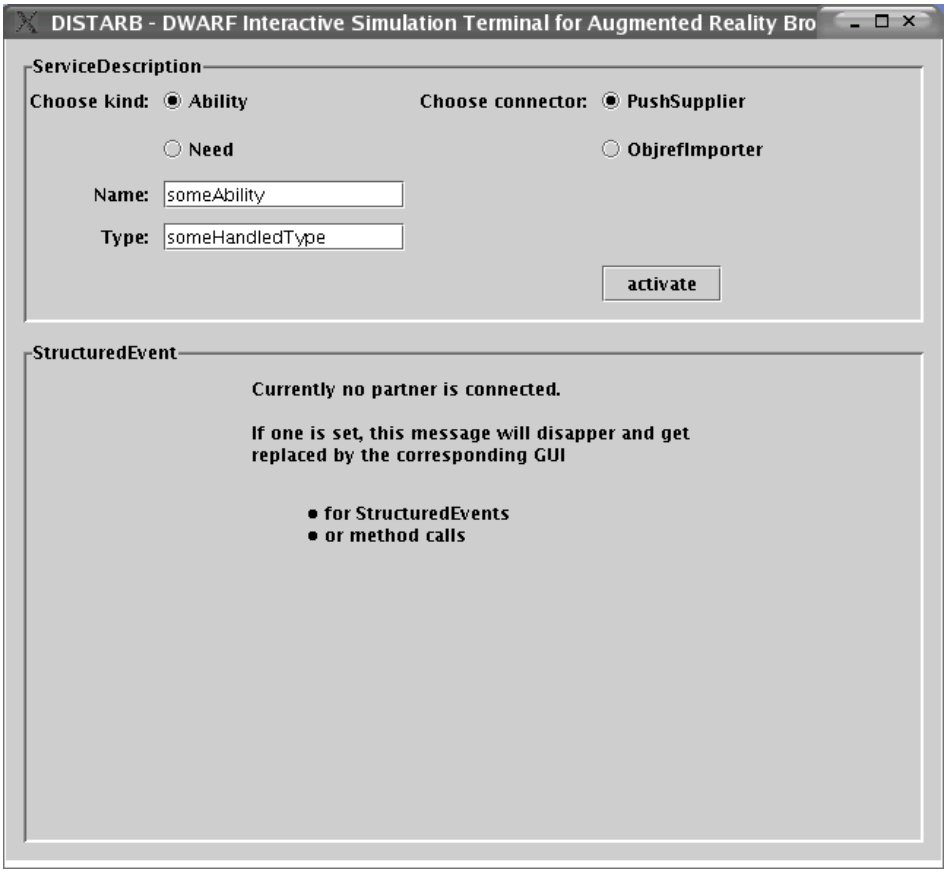


Figure 8.6: The startup screen of DISTARB - here, the services interfaces can be chosen

information is placed in two property files, *interfaces.properties* for classes which can be narrowed for method calls and *default_configuration.properties* for events to be sent.
Objects can be created by the *InstantiationManager* that guides through the required parameters and returns an ObjectReference.

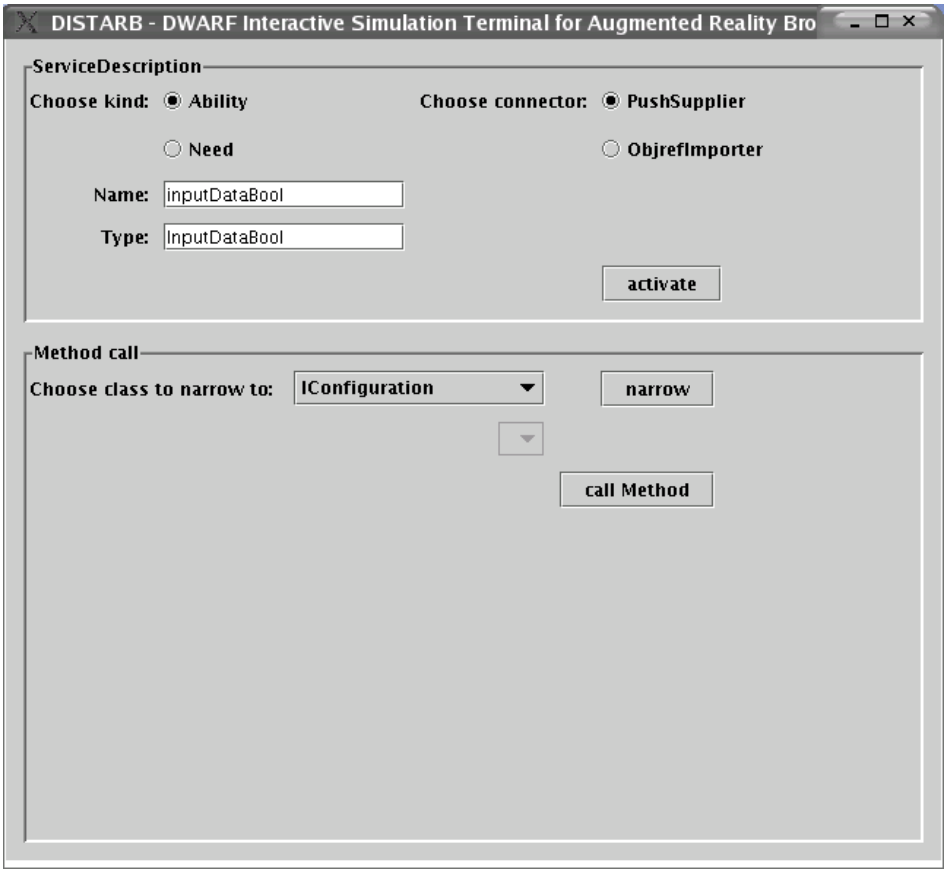


Figure 8.7: The DISTARB service with a connection to a service that allows method calls

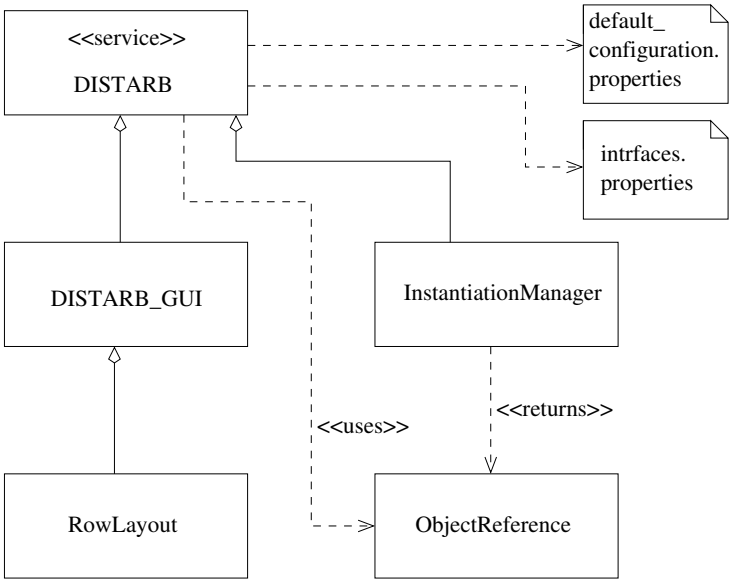


Figure 8.9: UML diagram showing the classes of the DISTARB service

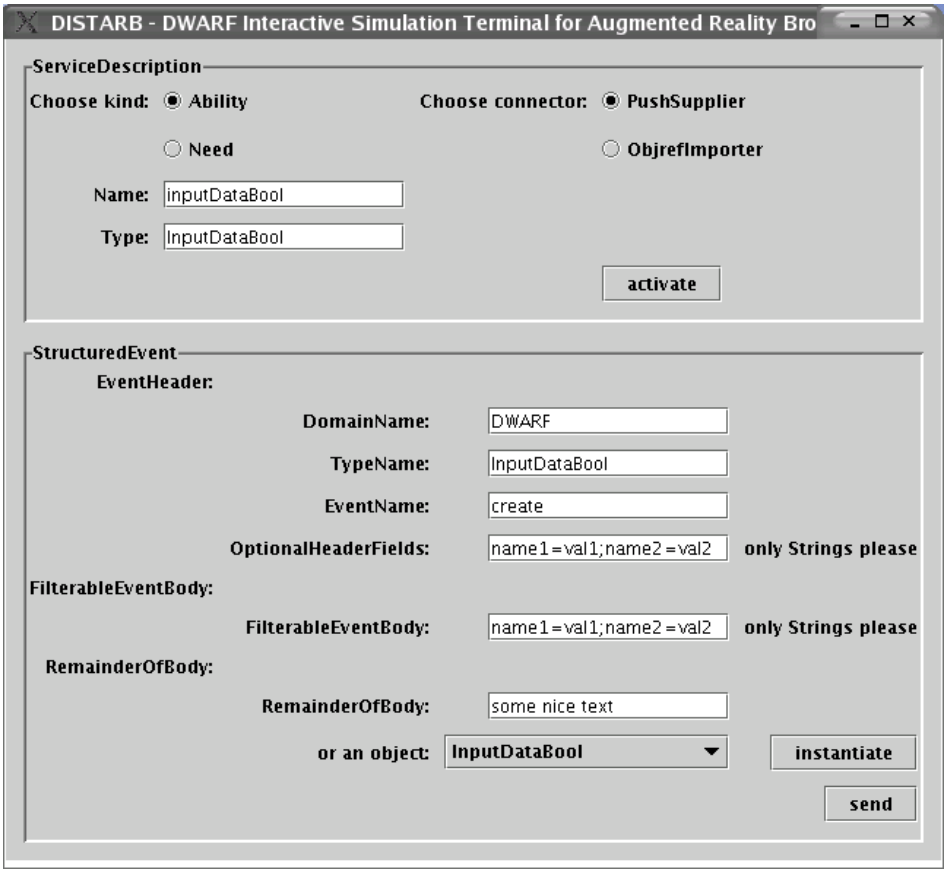


Figure 8.8: The DISTARB service with a connection to a service that allows sending of events

8.7.2 Implementation

Programming Language The DISTARB service is entirely written in Java because of Java's strength in reflection.

Graphical User Interface The frames have been built by use of Swing which is a part of the standard API of Java since SDK 1.2.0.

Reflection Reflection is a part of the standard API of Java since SDK 1.2.0.

8.7.3 State of Implementation

- Static fields are currently missing. This is required on events if their IDL contains enums.
- Additional tabs to simulate more than one service would also be useful.

- Support for service attributes would be useful. Currently, the accessed service must be adapted if it has a predicate in it's need.

8.8 User Interface Controller

The UIC has not been implemented, but has been configured in major parts by me. Special thanks here to Bernhard Zaun who did additional modifications when we realized in a time short before the demonstration, that the Touchpad, which we used for input, was to small to realize all required buttons. Thus he refactored the petri net to save at least one button. Here we had the problem, that additional functionality would not have had enough free space on the Touchpad, so, we didn't even realize this features. See [63, 115] for further details on that issue.

The ARCHIE configuration handles four input actions. The following list counts six elements. Two more are included for a better separation of the topics. Which both are encapsulated behind others is explained there.

create If a create event occurs, the *requestPose* transition sends an event to the Discretizer so send the pose of the object that is used for object creation. We called this one *virtualObjectCreator*. After the pose is requested, a token is placed on poseRequested. If the pose event arrives at the UIC, it is placed on discretePose. By this, the createVirtualObject transition switches, builds a ModelData create event and sends this out through the matching ModelData ability. The edge to the *selecting* place is explained in advance.

delete Object deletion work equally to creation, except that a collision is requested here from the Discretizer.

publish On a publish event, a ModelData event is created that contains a publish action.

selecting This is a well known structure to switch between two states in a petri-net. After startup, the *initTransition* places a token on the *isNotSelected* place. This allows the *selectTransition* to switch in case a *selecting* token is placed. The handling of deselection is equal, when the selecting place is set by a token again.

select Selection is no direct accessible action. It is encapsulated behind the selecting place. The workflow is equal to the delete operation. The only difference is that here a UserAction event is sent out that selects an object for moving.

deselect Deselection is also no direct accessible action. It is encapsulated behind the selecting place. The workflow is equal to the select operation.

Figure 8.10 shows the whole petri-net.

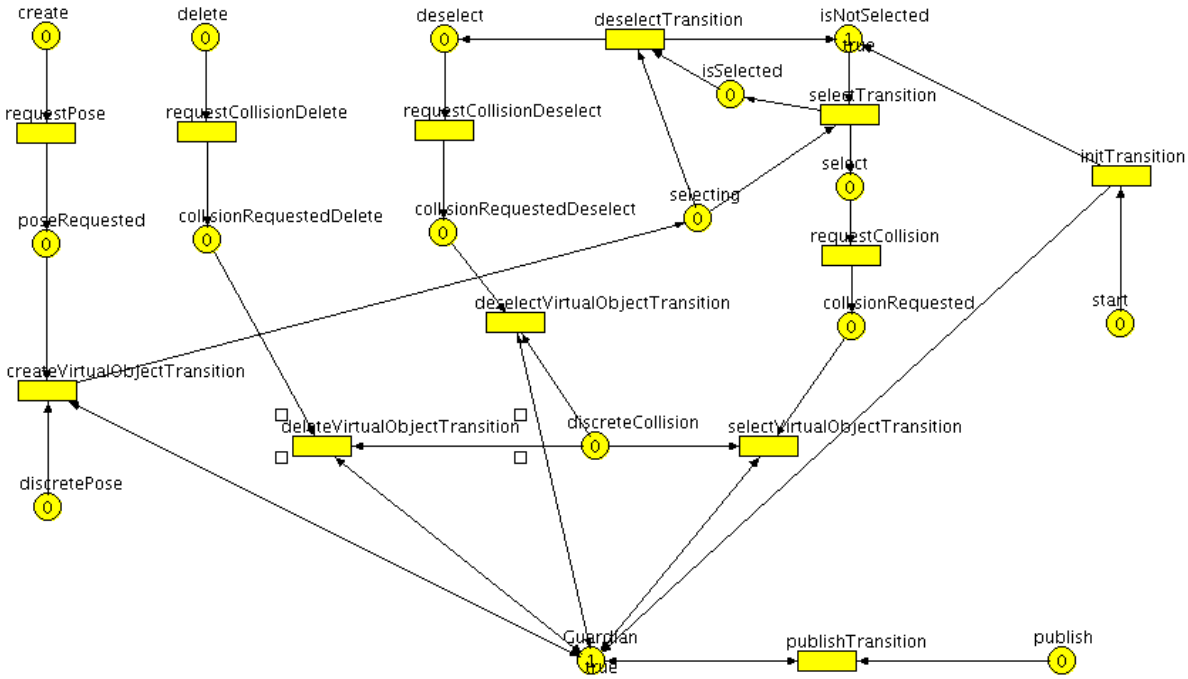


Figure 8.10: UML-diagram: The interfaces of the UIC

8.8.1 State of Implementation

- All described features work well and are fully implemented.
- Missing functionalities are: grouping, modifying virtual objects properties and publishing single objects.

9 Reusability in new Applications

"Learning is like a sandstorm, you just see when everything settled." **(unknown)**

In this chapter I show what changes have to be done to configure the new services to new applications. In common this would be the application architects work. That is why I introduce the third and last view on the DWARF framework, the application architect's view.

Modifications to the frameworks components can be done in three different ways, depending on the requirements, the planned application has to provide. First, it may just be enough to add new templates to the ModelServer. Second, often the configuration of the Model service must be changed to add new virtual objects to the scene. These objects also have to be added before startup to the ModelServer. Finally, often it is required to modify the Model's service description to satisfy the requirements of the new application.

9.1 The Application Architects Point of View

The application architect is often called the application developer. But I prefer to use the first phrase to distinguish between the application architect and the module developer that was introduced in chapter 7.

The application architect is responsible for the design and configuration of Augmented Reality applications. His goals are to keep the costs of production low and to keep in delivery time for the application.
People configuring application on a framework have to deal with implementation aspects. They have to know which changes on service descriptions are allowed and which not. This depends on how these are handled by the service managing classes. In other words, these people arrange the services in new setups, so that they provide the desired functionality.
They also have to know how to change services configurations or how to add new ones. Properties have to be set in the Configuration service and often the applications UIC needs a new setup.
Finally they have to know how to add real as well as virtual objects to the corresponding components of the framework. Same appeals to templates of virtual objects.

The following section provide all relevant information required by application architects.

9.2 Extending Models and Templates

This section explains how to modify the repository of the ModelServer.

9.2.1 Adding Data to the Database

Unfortunately a graphical user interface for this is not available. For this topic, the Configuration service provides a little extra feature.

If the Configuration service is started with one commandline parameters, the method *handleCmdProps* is called. This method parses the file *../share/query.txt* that resides in the same directory as the service descriptions and reads the file specified as commandline parameter one. Within the query file, a SQL statement is given, that should be rewritten in accordance to the table definitions in section 7.5 and contain the `%1` as value. This will be replaced by the written file. In advance, the query is executed and the files content is written to the database that this Configuration service is able to access.
This approach allows to add large textual files as XML-documents much easier than it would be done by pasting these files directly into the commandline of a database access tool.

9.2.2 Adding Templates and Default Properties

To add Viewer templates to a database, one has to know a templates structure. A template is written in the OpenInventor file format, except that some values are replaced by placeholders. These placeholders must have the same name as the fields in the default_properties table. Figure 9.1 gives an example for a template of a red plane.
As one can see, nine placeholders are defined. Each one of them is mandatory, because the '$endl' defines a line break that is required after the initial comment and the others describe where to insert the PoseData on creation.

9.2.3 Direct Creation of Objects

Objects can also be directly added to the database, but I do not advice this, because one has to guarantee, that all values are set correctly. This could be complicated, if more complex objects are going to be added. Better do this inside the corresponding Augmented Reality application.
For that purpose an extra authoring application could easily be configured. By just replacing the application attribute in the Model service and all those services who get their attributes from them, the reconfigured ARCHIE application writes it's data to the new aggregation of contextual attributes. Backing up the original state wen finished does not delete the built model. Now, the new application can access this data as it were it's own.

```
#Inventor V2.0 ascii $endl

 Transform {
   translation $trans_x $trans_y $trans_z
   rotation $rot_x $rot_y $rot_z $rot_w
 }
 DEF $virtualObjectId Separator {
  Transform {
    scaleFactor 0.05 0.05 0.01
  }
  Material {
    diffuseColor 1.0 0.1 0.1
  }
  Coordinate3 {
     point      [ -1 -1  1,
                   1 -1  1,
                   1  1  1,
                  -1  1  1,
                  -1 -1 -1,
                   1 -1 -1,
                   1  1 -1,
                  -1  1 -1
                ]
  }
   IndexedFaceSet {
      coordIndex [ 0, 1, 2, -1, 0, 2, 3, -1,
                   5, 4, 6, -1, 6, 4, 7, -1,
                   1, 5, 2, -1, 2, 5, 6, -1,
                   4, 3, 7, -1, 3, 4, 0, -1,
                   3, 2, 7, -1, 6, 7, 2, -1,
                   0, 4, 1, -1, 1, 4, 5, -1]
  }
}
```

Figure 9.1: An example for a template - a red wall

9.3 Changes on Configuration

The Model service in its current implementation requests two properties from the Configuration service. Which entry is taken belongs to the aggregation of the contextual attributes of the Model service.

objects This field lists all virtual objects by their VirtualObjectIds. They a separated by a comma.

publishAlways Can be `true` or `false` and indicates if a change to a virtual object shall be distributed immediately or not.

9.4 Changes on Service Description of the Model Service

Finally the service description of the Model provides a wide playground for changes.

First of all any kinds of attributes and predicates can be added to restrict to connectivity of this service or to provide access for other services.
But the names of the needs and abilities may not be changed. They are required to determine the service intern handlers.

I will iterate over the service description of the Model as it run in the ARCHIE modeling scenario, because this one included all actually available needs and abilities. A note on predicates is useful. This service description is exactly the description that was used in the ARCHIE presentation. At this day I added the predicates for a specific hostname to ensure that the Model and adjacent services do really run on that machine. I did not remove them to illustrate the use of predicates.

The Service

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">

<service name="Model"
         startOnDemand="true"
         stopOnNoUse="true"
         isTemplate="true"
         startCommand="Model">
```

In the header, no changes should be done this enables the Model to be started if required and to shut down if no more required. Even multiple instances of a Model can run on the same hardware platform if required.

Need: Configuration

```
<need name="configuration"
      type="Configuration"
      predicate="(&amp;(hostname=atbruegge34)(configurationKey=Model))">

     <!--This attribute is added to the predicate automatically.-->
     <!--It is also given as attribute to the Configuration for -->
     <!-- opening a session.-->
     <attribute name="application" value="*"/>

     <!--This is given as attribute to Configuration for a session.-->
     <attribute name="configurationKey" value="Model"/>

     <connector protocol="ObjrefImporter"/>
</need>
```

The Configuration need enables the Model to receive it's personal setup. This need is mandatory, because the Model requires the Configuration to get information about the virtual objects it maintains.

Need: ModelData

```
<need name="directActions"
      type="ModelData"
      predicate="(hostname=atbruegge34)">
     <attribute name="application" value="*"/>
     <connector protocol="PushConsumer"/>
</need>
```

The need for ModelData is not required, as there can be Models that do not allow modifications on objects. For instance a pure presentation.

Need: UserAction

```
<need name="userActions"
      type="UserAction"
      predicate="(hostname=atbruegge34)">
     <attribute name="application" value="*"/>
     <connector protocol="PushConsumer"/>
</need>
```

Also UserActions are not required.

Need: TemplateProvider

```
<need name="objectTemplates"
      type="TemplateProvider"
      predicate="(hostname=atbruegge34)">
     <connector protocol="ObjrefImporter"/>
</need>
```

In contrast to the last two needs, this one is required, as it is needed for the initialization of objects for the connected Viewer.

Need: ModelAccess

```
<need name="accessToModel"
      type="ModelAccess"
      predicate="(hostname=atbruegge34)">
     <connector protocol="ObjrefImporter"/>
</need>
```

Also the access to the Model is necessary to get the data of all virtual objects.

Need: Consistency

```
<need name="provideUpdates"
      type="Consistency"
      minInstances="0"
      maxInstances="4"
      predicate="(hostname=atbruegge34)">
     <connector protocol="ObjrefImporter"/>
</need>
```

If just one Model is running, this need is unnecessary. As one can see consistency is handled by sessions.

Need: PoseData

```
<need name="virtualObjectCreator"
      type="PoseData"
      predicate="(&amp;(ThingId=virtualObjectCreator)(FilteredData=true))">
     <connector protocol="PushConsumer"/>
</need>
```

The need for PoseData is restricted to the filtered PoseData of of the virtualObjectCreator that was used to move virtual objects. This need should always only accept PoseData of one type and object, otherwise unpredictable positions and orientations could be set.
This need is not mandatory.

Abilities: SceneData

```
        <ability name="scenes"
                 type="SceneData">
                <attribute name="application" value="*"/>
                <attribute name="poseChanges" value="true"/>
                <connector protocol="PushSupplier"/>
        </ability>
        <ability name="scenesWithoutPoseChanges"
                 type="SceneData">
                <attribute name="application" value="*"/>
                <attribute name="poseChanges" value="false"/>
                <connector protocol="PushSupplier"/>
        </ability>
```

Two abilities for SceneData events exist. One for scene changes of any kind and one for all changes on objects except changes in the objects pose. As Viewers determine pose modifications on their own, they do not require the same information from the Model.
In the current implementation, a Model must have both SceneData abilities satisfied when a CollisionDetection is required. Only when consumers for both are set, the scenes are initialized.

Ability: Consistency

```
        <ability name="receiveUpdates"
                 type="Consistency">
                <connector protocol="ObjrefExporter"/>
        </ability>

</service>
```

The Consistency ability is the reverse end of the corresponding need and therefore also not required if no consistency is needed.

10 Conclusion

Useful Prototypes of all Services are realized, the Results can be discussed, but although there is still more Work to do

---

This chapter concludes my thesis. Now, after much information about technical details, let us take a step back and review the new DWARF components.
I will present results of the work in detail on the services and in a whole for the ARCHIE setup. In advance I will explain the most important lessons learned and finally I provide a section about possible future work in the design and in the implementation.

10.1 Results

10.1.1 Services for the Management of Virtual Objects

The first main result of this thesis' work are the services dealing with objects of the virtual environment.

New way of Managing the Users Environment The designed services describe a new approach to the management of virtual objects and their relations. I do not know how they will be accepted in the future for other demonstrative projects. The Model and the ModelServer can easily be bypassed by directly using the SceneData IDL to support the Viewer with information. But I believe that they have a good potential for larger applications.

Scalability on Contextual Attributes The design of the Model service allows to get scaled for various needs. In common the Model is designed to work as a private Model for one user, but it can also be configured to serve applications, dedicated locations or any aggregations of them.

Configurability to serve Multiple Applications To serve in intelligent environments, the Model must be able to serve more than one application at a time. This is designed and I look forward to test this in the future.

Interconnectivity to other Services The Model service can be configured to share multiple views as well as various other services. This allows various applications to deal with one Model if required.

10.1.2 Services Configuring other Services

The second main result of my thesis was the Configuration service.

Transparent Access to Configuration Data The Configuration service provides transparent access for any service. So no service has to bother from where to get it's configuration. Also services that want to get configured for the first time, just have to offer their need and will also get transparent access to a Configuration service that will serve this one in future.

Scalability of Configuration Ranges Any aggregations of contextual attributes, a service has can be used to get access to the service's configuration. Even if adding new attributes is not yet generic, this approach will hopefully serve well for the problem domain of Augmented Reality in distributed mobile environments.

10.1.3 Services Minimizing Network Load

The Discretizer is an nearly reusable component of the framework.

Adaptability to various Streams When refactored, the Discretizer will serve for any kind of streams. Because it does not require any additional libraries, it can run in general on any platform that provides the Java Virtual Machine. In future this service can easily be used to minimize network load, if only single events are required.

10.1.4 Services for Testing or Simulating other Services

The DISTARB service opens up a new dimension of testing and simulating framework components.

Testing Tools are Necessary Without a testing tool, that can simulate services that require peripheral hardware, many services could only be tested, if the corresponding partner service is available. Often this is not the fact, because not all developers are present at the same time.
By this fact, the DISTARB service could be used to simulate the Touchpad service, even at those times, the Touchpad itself was out of order.

Simulating Output of Chains of Services Often there are chains of services that provide data. For instance, the UIC of ARCHIE requires CollisionData, this is provided by the Collision detection, which relies on the tracking system and on the Model. To test a simple selection, whole chains of services have to be started, which is often time consuming, too.
By use of DISTARB, the UIC's output can easily be generated without starting whole subsystems.

10.1.5 Validation of all Services in ARCHIE

In addition of the realization of new services, we also built a demonstrative application, ARCHIE.

Demonstration System was Built Quickly This phrase belongs to my components. For the other services I like to refer to my colleagues works.
As the services were finished, I only had to do some configuration for the Model service in the Configuration service and had to add some templates to the ModelServer. The service descriptions had to be configured, which was also quick done.

The Scenario was Successfully Demonstrated The general scenario of the ARCHIE application was a complex one. For the parts, my components were participating, the services worked as desired, except that one Viewer was not connected during the modeling scenario. But this was not a bug, only a small configuration failure in the startup sequence and could have been fixed during runtime by simply shutting down the Model service. It would have restarted and got connected to all Viewers.
During the presentation scenario everything worked fine.

Performance Testing performance would not give reliable information about the designed services, because time saving thread handling for concurrent database access is also not realized as separate worker threads in the Model.
But the object relevant components reacted immediately and quickly on actions, even when it appeared to require more time during presentation. This was due to the UIC which required sometimes long time to switch through the petri-net.
The current implementation is usable for distributed Augmented Reality systems.

10.2 Lessons Learned

This section is dedicated to some lessons I learned in working with the DWARF framework and on writing this thesis.

10.2.1 Working with the Framework

Using Java for Services Sending events in Java requires all fields of all objects aggregated in the StructuredEvent classes RemainderOfBody to have non null values.

Remote Timeouts If a method that is called from the servicemanager on a local service takes to much time to perform it's work, the servicemanager assumes, that the service that does not return is dead and calls the appropriate disconnect method on all other services that were already connected to the 'dead' one.

10.2.2 Personal Learned Lessons

Programming Languages Never begin to learn a new programming language in your diploma thesis. You will never get enough vocabulary to produce really good quality of code.

My Java is better than my English I beg your pardon for this fact.

10.3 Future Work

This section describes several possible future projects for extending and expanding functionality of the data handling services of the DWARF framework.

10.3.1 Extensions to the Implementations

The current implementation of the services is not finished. It must be finished and can be extended in various ways.

Realization of Missing Implementations As listed in the chapter about the implementation, several tasks are still not resolved. These must be realized before any additional steps are performed.

Thread Handling To provide efficient management of runtime, the thread handling must at least be extended in the Model and the ModelServer.

Aggregation of Models data to one Viewer To deal with various Models on one Viewer, some changes are required or a SceneCompositor must be implemented. More than one Model may be useful, if a Model manages a general application specific area, while others serve as personal interaction components.

Extension of Event Interfaces The event interfaces of the Model should be realized as session, This would allow logging of user interaction and provide more efficient access for continuous streams, for instance when relocating objects.

Maintaining the Model's and the Configuration's Database A graphical tool to maintain the stored configuration properties, templates and objects of a scene is required to facilitate authoring of applications.

Reconfiguration of Input Devices When object's properties are going to be modified, the UIC must be switched off while this task is performed. Events and interfaces for this must be realized.

10.3.2 Extensions to the Design

The design of the services that manage the environmental model and the services context can be extended.

Support for other relations than Transformations The current design only maintains transformations of objects as translations and rotations. As revealed in the requirements engineering chapter, additional types of associations between objects should be controllable.

Refactor Interface to Viewer The SceneData interface works well for whole objects, but looses efficiency when modifying fields of objects. This interface between Model and Viewer should be refactored to be more fine grained and generic.

Support for Data that is not applicable to the InputData Types The Model service still lacks interfaces to modify object's textual properties. Although manipulation of properties of this kind is possible directly on the ModelServer, such functionality better fits to the Model service. A design for a new interface handling this issue is required.

Maintaining the Configuration As some services may also contains user accessible configuration, this data should also be maintainable by the user. Some kind of interaction definition is required here.

Relocation of Object's Properties As DWARF is designed to serve ubiquitous computing environments, users may access their settings and their virtual environment from various locations. Even if the Model component may run on relative near hardware platforms, persistent storage of data may require far distant communication. It were suitable to relocate these data if necessary.

Security As already mentioned, security is a serious issue. It is difficult to delimit the boundaries. But authentication of users is required in any way to prohibit illegal access to private data.

Support for Geographic Queries The functionality of geographic information systems is surely required for a lot of future applications. For that fact, this should be included in the design of the services or additional services should be included in the design.

10.3.3 Architecture extensions

Although the general architecture is DWARFitself, there's a small one for the separation into the separate handling services that cover two areas that are researched by data management. The first is the users context specific world, which is stored in the ModelServer. The second is the services' intern area that reflects the module developers area and contains service configurations in the Configuration service.
Extending the maintained amount of data managing areas to three, we could use XML-Content management systems to provide service authoring to the DWARF system. This approach can be realized by XML-databases (see section Conclusion on page 58) that cover the used databases as well as the directory of the servicemanager's directory, the service descriptions are stored.
By adding this area to the architecture we could easily manage service descriptions on multiple hosts.
At the moment this is well thought, but this could be a tremendous step into the management of services in a distributed environment.

10.3.4 Extensions to the ARCHIE Application

The ARCHIE application can be extended to serve more features. By this it would provide an additional feature demonstration setup for the DWARF framework.

Realize a Permanent Demonstration Setup The current configuration of the services of the ARCHIE scenario is only realized in the Configuration service for single users. A general demonstrative demo user with configurable configuration is required.

More Virtual Objects and a Selection Mechanism Placing red planes is fine, but modeling buildings is more comfortable when additional objects are available. To select the type of these objects, an extra creation workflow is required to add to the UIC.

A ARCHIE System Models
Scenarios and UseCases of the ARCHIE application

This section provides different views of the ARCHIE application focused on the data management components. These are outlined in scenarios and Use Cases.

A Scenario is a informal description of a single system feature. The features described are actually supported by the current implementation and have been tested in the final presentation in spring 2003. The scenarios focus on the ARCHIE application as they have been presented.

A.1 Scenarios

This section explains concrete sequences of interaction between actors and the system. These are the completely realized functionalities of the new DWARF components. For further information on functionality named in the requirements but not listed here, see the last chapters of the thesis about implementation, further work and the conclusion.

The scenarios announce a glove and viewing components. Both of them were realized during the ARCHIE project. Because this thesis focuses on data management, information about these two components may be referenced to the papers [58], [115].

Scenario: **Initializing the Application**

Actor instances: `Alois: User, view, tracking: Service`

Flow of Events:

1. Alois, an architect, wants to perform some work on a new project. He enters the room where his workdesk is located.
2. The system recognizes, that the ARCHIE application can be executed and displays a corresponding option on Alois wearable Augmented Reality device.
3. Alois takes his wearable Augmented Reality device and selects the appearing ARCHIE application.
4. The DWARF framework recognizes the action and starts a personal user environment.
5. While the system is starting, Alois selects his viewing device, places it on his head and selects this viewing device on his Augmented Reality device.

6. The system initializes the current state of the application in the HMD: A model of the building of the faculties for mathematics and computer science.
7. Hence Alois head is tracked, he always has a right aligned view onto the model.

Scenario: **Creating an Object**

Actor instances: `Alois: User, view, tracking: Service`

Flow of Events:
1. The viewer in the HMD displays a view onto the model, where the new Leibniz-Rechenzentrum[1] is planned. He takes the tangible object representing a creator object which is configured to build a wall and moves it to the intended position where the new outer wall of the later LRZ shall be.
2. He performs the create action via a button click on his worn glove.
3. The system recognizes the action, takes the position of the real tangible object and the selected type to insert on this position and builds a scene of a wall.
4. A new wall appears on the corresponding position of the tangible object in his personal view.

Scenario: **Moving a Wall**

Actor instances: `Alois: User, view, tracking: Service`

Flow of Events:
1. Alois realizes, he placed the wall on the wrong location.
2. He takes a tangible object and places it right in the virtual wall he wants to move.
3. Alois performs the select action on his glove.
4. The application user interaction controller restricts the access to the specified wall to Alois.
5. The wall is attached to the tangible object and can be moved by moving the real object.
6. Locating the wall on the right position, Alois performs th deselect action on his glove.
7. The system releases the ownership of the virtual wall.
8. The wall is positioned on the location of the real object.

Scenario: **A Technical Engineer joins the Modeling Work**

Actor instances: `Alois, Theodor: User, view, tracking: Service`

Flow of Events:
1. While Alois is working, Theodor wants to help his colleague, enters the room, selects the same application and the HMD.

[1] LRZ - The data computation center of the Bayerische Akademie der Wissenschaften, the Technische Universität München and the Ludwigs-Maximilian Universität München

2. The system initializes the current state of the planned model without the add ons of Alois.

Scenario: **Publishing Model Changes**

Actor instances: `Alois, Theodor: User, view, tracking: Service`

Flow of Events:
1. After some modeling work is done, Alois decides to contribute his changes on the model to his colleagues.
2. He presses the publish button on his glove.
3. The new walls are propagated to Theodors view. So Theodor is able to get an impression of Alois' ideas. All new created walls are now selectable to other users, too.

Scenario: **Deleting Walls**

Actor instances: `Theodor: User, view: Service, tracking: Service`

Flow of Events:
1. Theodor, a technical engineer, realizes, that some walls have been planned on a location, nothing is allowed to build.
2. He takes a tangible object for deletion, places it inside a virtual wall.
3. On the delete action, sent by the corresponding glove button, the wall disappears.
4. This action recognized by the system removes the object only from the users personal view, unless these changes are published.

Scenario: **Model Presentation to the Client**

Actor instances: `Building Stakeholders: User, view, tracking: Service`

Flow of Events:
1. After finishing a lot of work, Alois and Theodor leave the room, do some other activities (e.g. drink some coffee) and meet the later building owners and some end users.
2. All enter the room and Alois selects the ARCHIE application again. Additional to his HMD, he selects a video beamer view.
3. The system attaches this view to a virtual camera represented by a small tangible camera, contacts Alois private model and initializes the beamer view.
4. All guests can see a view onto the model from the cameras position.

A.2 Use Cases

As described in [34], Scenarios are instances of Use Cases. A Use Case generalizes all possible scenarios for a given piece of functionality. The following set of use cases have been

used during requirements analysis to develop initial models of the proposed DWARF components.

All use cases can be either be performed by any user or by other services. In case of ARCHIE only users are the only actor. But there may be parts of applications, where standalone services require visualization as virtual objects or want to change some properties of objects. So these could also be actors.

Use Case: **InitializeModel**
Initiated by: `User`
Communicates with: `view: Service`
Flow of Events:
1. *(Entry Condition)* The user enters a room whose environment is able to provide functionality for a specific application. The user selects the application on his personal digital assistant, where it appears as option.
2. On initialization, the model component checks which data to load.
3. After the applications scene information revealed, the model connects to a data server and retrieves the data of all virtual objects.
4. All information necessary to perform the application is cached locally.
5. The user decides on a specific viewing device.
6. *(Exit Condition)* When the connection to the view is set up by the service manager, the scene is built by initiating Use Case *CreateObject*, sent to the viewing device and displayed there.

Use Case: **CreateObject**
Initiated by: `User, service`
Communicates with: `view: Service`
Flow of Events:
1. *(Entry Condition)* A application is running and a input device is configured or a service comes to a state it requires visualization of some data.
2. An action is sent to the personal model containing information which kind of object to create and whose properties (material, color, ...).
3. The personal model requests a object description template which is accepted by the viewing component.
4. A connected data server provides this information and hands over.
5. The model builds a displayable scene and sends that to the users personal view, while the virtual objects properties reside in the users private space.
6. *(Exit Condition)* The new object appears in the users view.

Use Case: **MoveObject**

Initiated by: `User, service`

Communicates with: `view, CollisionDetection: Service`

Flow of Events:

1. *(Entry Condition)* During work in an application, some modifications on an object are necessary.
2. The user generates a collision between the virtual object and the real tangible selection device for moving.
3. The CollisionDetection supplies these collisions.
4. The user performs the select action.
5. The acting service attaches the real objects position and orientation information to the collided virtual object.
6. The user moves the real object, places and performs a deselect action.
7. Real and virtual object are decoupled and the new position is stored locally, private to the user.
8. *(Exit Condition)* No object is selected.

Use Case: **ModifyObject**

Initiated by: `User, service`

Communicates with: `view, CollisionDetection, glove: Service`

Flow of Events:

1. *(Entry Condition)* During work in an application, some modifications on an object are necessary.
2. The user generates a collision between the virtual object and the real tangible selection device for modifications.
3. The CollisionDetection supplies these collisions.
4. The user performs the select action.
5. The acting service attaches the gloves input data to the collided virtual objects corresponding property.
6. The user slides over a Touchpad on his glove, the property of the virtual object is changed and painted in the view.
7. The deselect action is performed.
8. Touchpad and virtual object are decoupled and the new property is stored private to the user.
9. *(Exit Condition)* No object is selected.

Use Case: **PublishPrivateSpace**

Initiated by: `User, service`

Communicates with: `Model: service`

Flow of Events:

1. *(Entry Condition)* While in an application, some modifications on virtual objects have been performed.
2. The model receives a publish action.
3. All changes are stored persistent.

4. The cached changes are propagated to the other models and by this to the other views.
5. *(Exit Condition)* The users private space is empty.

Use Case: **DeleteObject**
Initiated by: `User, service`
Communicates with: `view, CollisionDetection, glove: service`
Flow of Events:
1. *(Entry Condition)* While in an application, some virtual objects must be deleted.
2. The user moves the tangible object for deletion into the virtual object.
3. The CollisionDetection propagates collision information through the system.
4. After performing an explicit delete action, the colliding object is removed from the private space of the user.
5. *(Exit Condition)* The virtual object is removed from the private model and the view.

Use Case: **PresentModel**
Initiated by: `User`
Communicates with: `view: service`
Flow of Events:
1. *(Entry Condition)* A video beamer is attached to the system.
2. The user selects the viewer for his application.
3. The new view is connected to the users data model.
4. The use case InitializeModel is performed except application selection.
5. The views viewpoint is attached to the tangible camera.
6. *(Exit Condition)* The users without HMDs can see the virtual model on the beamer from the cameras point of view.

B Computing Objects Relative Positions

As each Virtual Object has it's own Coordinate System, Relative Positions must be Computed

---

This appendix describes how objects positions and orientation can be computed relative to another object.

Every virtual object can be included in a hierarchical tree data structure. Every node within this structure has is own coordinate system that gives information about the translation and rotation of this object against it's parent. Figure B.1 gives an example of an object with it's translation and rotations. This facilitates adding of objects to a scene.

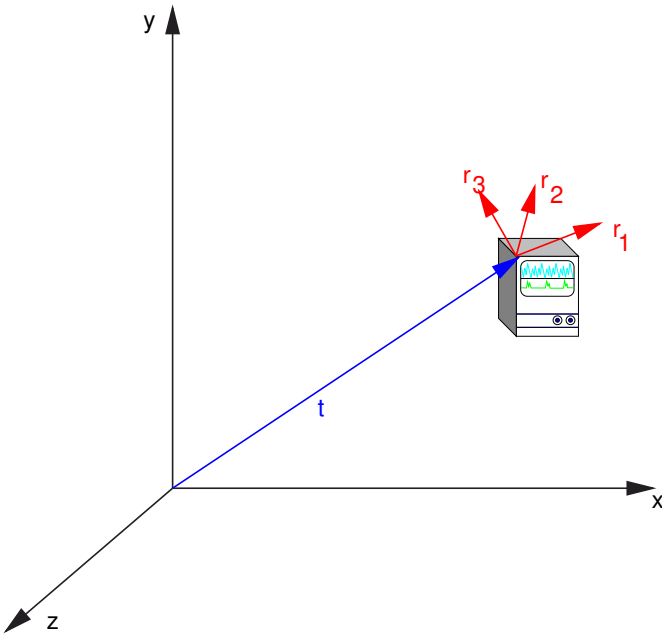


Figure B.1: General coordinate transformation (Courtesy of Wagner [110])

For instance, a table can be added to a room by just giving it's offset against the origin of the room. This values are easier accessible than first computing the offset against the

absolute origin of the world's coordinate system.

If a object's parent is exchanged, the relative pose has to be computed again. As every object has a associated PoseData, this contains all values required for this computation.

A object $T_O = (q_O, t_O)$ PoseData has a a three dimensional *translation* vector

$$t_O = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}; \tag{B.1}$$

and an four dimensional *rotational* vector

$$q_O = \begin{pmatrix} q_x \\ q_y \\ q_z \\ q_w \end{pmatrix}; \tag{B.2}$$

that represents a *Quaternion* [95].

If this object is going to be added as a sub-node to an object $T_K = (q_K, t_K)$, we also have the pose of this object and can compute the new relative pose of T_O, $T_N = (q_N, t_N)$ as follows:

$$norm = \frac{1}{q_x^2 + q_y^2 + q_z^2 + q_w^2}; \tag{B.3}$$

$$q_N = \begin{pmatrix} q_x * -norm \\ q_y * -norm \\ q_z * -norm \\ q_w * norm \end{pmatrix}; \tag{B.4}$$

$$t_N = q_N \cdot (t_O - t_K) \cdot q_N^{-1}; \tag{B.5}$$

C Interface Definitions of the Services

Service Developer Interfaces for Programming with the new Services

This appendix contains information which is only of interest for developers of services for the DWARF framework. It provides a reference to the services' CORBA interfaces.

C.1 Configuration

```
#ifndef __ICONFIGURATION_IDL
#define __ICONFIGURATION_IDL

#include <DWARF/Service.idl>

#pragma prefix "in.tum.de"

module DWARF {

  struct StringProperty {
    string key;
    string value;
  };

  typedef sequence<StringProperty> Properties;

  interface ConfigurationInterface {
    string getProperty (in string key);
    void setProperty (in string key, in string value);
    Properties getProperties();
    void setProperties (in Properties props);
  };

  interface ConfigurationAuthoring {
    // This will also generate a new namespace
    void setProperty(in string configurationKey,
     in string application,
     in string role,
     in string room,
     in string user,
     in string property,
     in string value);
  };

};

#endif //__ICONFIGURATION_IDL
```

C.2 ModelServer

```
#ifndef __MODELSERVER_IDL
#define __MODELSERVER_IDL

#pragma prefix "in.tum.de"

#include <DWARF/DwarfCommon.idl>
#include <DWARF/IConfiguration.idl>
#include <DWARF/PoseData.idl>

module DWARF {

  exception AlreadyLocked {
    string message;
  };

  interface Locker {
    boolean lockObject (in VirtualObjectId id) raises (AlreadyLocked);
    void unlockObject (in VirtualObjectId id);
  };

  struct ObjectProperties {
    VirtualObjectId id;
    string type;
    Properties props;
  };

  typedef sequence<ObjectProperties> ObjectPropertiesSeq;

  exception NotLocked {
    string message;
  };

  interface ModelAccess {
    ObjectProperties createObject (in string type);
    ObjectProperties getObjectProperties (in VirtualObjectId id);
    ObjectProperties getDefaultProperties (in string type);
    void setObjectProperties (in ObjectProperties properties) raises (NotLocked);
    void deleteObject (in VirtualObjectId id);
  };

  interface TemplateProvider {
    string getTemplate (in string type);
  };

};

#endif // __MODELSERVER_IDL
```

C.3 Model

```
#ifndef __IMODEL_IDL
#define __IMODEL_IDL
```

```
#include <DWARF/Service.idl>
#include <DWARF/ModelServer.idl>

#pragma prefix "in.tum.de"

module DWARF {

  interface Consistency {
    void updateObjectProperties (in ObjectPropertiesSeq propertiesSeq,
                                 in string deleteList);
  };

};

#endif //__IMODEL_IDL
```

C.4 DwarfCommon

This IDL file was added for convenience, because it contains the general identifier declarations for real and virtual objects.

```
#ifndef __DWARFCOMMON_IDL
#define __DWARFCOMMON_IDL

#pragma prefix "in.tum.de"

module DWARF {

  /*
   * This struct is the common Time format used between
   * all DWARF services.
   */
  struct Time {
    unsigned long seconds;
    unsigned long microseconds;
  };

   /*
   * This typedef gives an abstraction of system-wide Types
   * of observed objects.
   * Mainly used to instantiate PROTOs in VRML scenes
   */
  typedef string ThingType;

  /*
   * This typedef gives an abstraction of system-wide unique IDs
   * of observed objects
   */
  typedef string ThingID;

  /*
   * This typedef holdes the id of an virtual Object and will be used by
   * Model and View
```

```
   */
  typedef string VirtualObjectId;

};

#endif // __DWARFCOMMON_IDL
```

D Event Declarations

Service Developer Reference for Handling Events for the new Services

This appendix contains information which is only of interest for developers of services for the DWARF framework. It provides a reference on the event based CORBA interfaces.

D.1 ModelData

```
#ifndef __MODELDATA_IDL
#define __MODELDATA_IDL

#pragma prefix "in.tum.de"

#include <DWARF/DwarfCommon.idl>
#include <DWARF/PoseData.idl>

module DWARF {

  enum ModelAction { CreateModelObject, DeleteModelObject, PublishModelObject };

  struct ModelData {

    // the type of the modelAction
    ModelAction action;

    // The id of the virtual Object
    // Not always used: DELETE
    VirtualObjectId id;

    //  The type of the virtual Object
    // Not always used: CREATE
    string type;

    // The position of a object eventually involved in the action
    // Not always used: CREATE
    PoseData pose;
  };

};

#endif // __MODELDATA_IDL
```

D.2 SceneData

```
#ifndef __SCENEDATA_IDL
#define __SCENEDATA_IDL

#pragma prefix "in.tum.de"


#include <DWARF/DwarfCommon.idl>

module DWARF {

  typedef string Scene;

  enum SceneAction { CreateObject, DeleteObject, ReplaceScene, SuperImpose };

  struct SceneData {

    // The type of the modeAction
    SceneAction action;

    // The id of the virtual object
    VirtualObjectId id;

    // The id of the virtual objects parent
    // Not always used!
    VirtualObjectId parent;

    // The scene (as string) to display
    // Not always used!
    Scene newScene;

  };

};

#endif // __SCENEDATA_IDL
```

D.3 UserAction

```
#ifndef __USERACTION_IDL
#define __USERACTION_IDL

#pragma prefix "in.tum.de"


#include <DWARF/DwarfCommon.idl>

module DWARF {

  enum UserActionType { SelectVirtualObject, DeselectVirtualObject };
```

```
  struct UserAction {

    // the type of the modeAction
    UserActionType action;

    // The id of the virtual object involved in action producing
    VirtualObjectId id;

    // The id of the real object involved in action producing
    // Not always used!
    ThingID realObjectId;

  };

};

#endif // __USERACTION_IDL
```

E Glossary
Abbreviations and Term Definitions

AFS Andrew File System. Provides distributed file access in a local directory hierarchy.

API. Application Programmers Interface. Classes and libraries that are documented in a well structured format so that developers can easily reference functionality.

AR. *see* AUGMENTED REALITY

ARCHIE. Augmented Reality Collaborative Home Improvement Environment: This is the name of the application for which the calibration method was developed.

ART. Advanced Realtime Tracking: A tracking subsystems consisting of several infrared cameras and a Windows computer providing the tracking data.

Augmented Reality. A technique that uses virtual objects to enhance the user's perception of the real world.

AW. Augmented World. The real world enriched by virtual objects.

BLOB. Binary Large OBject. A generic data structure databases provide to store large unknown data.

CAD. Computer Aided Design. The use of well developed software systems for the graphical development of three dimensional Objects.

CORBA. Common Object Request Broker Architecture. CORBA is a specification for a system whose objects are distributed across different platforms. The implementation and location of each object are hidden from the client requesting the service.

DAG. Directed Acyclic Graph. A data structure extending a tree by additional directed edges which are arranged, so that no cycles exist.

DBMS Database Management System. A software system that facilitates the creation and maintenance of a database or databases, and the execution of computer programs using the database or databases.

DDL. Data Definition Language. A formal description for manipulating data structures in a database.

DWARF. Distributed Wearable Augmented Reality Framework.

ER. Entity Relationship. A simple graphical notation to describe attributes in entities and their relationship between each other.

GIS. Geographic Information Systems. Environments that provide support for acquiring, modeling, managing, analysis and presentation of spatial data and information.

GPL. GNU Public License. A open source license enforcing developers using this license to offer their products as well under this license.

HMD. *see* HEAD MOUNTED DISPLAY

Head Mounted Display. A display device similar to glasses. Its user either sees only the display or the display information projected optically onto the real world (See-Through Head Mounted Display)

IDL. Interface Definition Language. A quasi programming language to define the overall structure of compilable interfaces between different applications that communicate via CORBA.

NFS. Network File System. A file system extension for distributed file access with insecure data transferring.

OpenGL. An API for simple programming of three dimensional computer graphics available on most operating systems.

ORB. Object Request Broker. A library that enables CORBA objects to locate and communicate with one another.

PDF. Portable Document Format. A wide distributed standard for electronic versions of book and documents.

RAD. Requirements Analysis Document. A document describing the requirements of a software project and the way they were derived.

SMB. Server Message Blocks. A specification for remote access to file systems.

SDK. Software Development Kit. A abbreviation Sun uses among other vendors to name and version their product Java.

SLP. Service Location Protocol. A protocol, the DWARF servicemanagers use to find themselves.

SMS. Spatial Model Server. Decentral autonomous components of the Nexus platform that store spatial information for a certain area.

STL. Standard Template Library. A API for C++ that provides efficient implemented templates for any kind of data containers.

SQL. Structured Query Language. A almost intuitive formal language providing access to relational databases.

Tracker. A device determining the position and orientation of a tracked object.

UDP. User Datagram Protocol. A relative lightweight connectionless network communication protocol.

UIC. User Interface Controller. A service containing configurable Petri Net for discrete event handling.

URL. Uniform Resource Locator. A String giving the exact location of a file in a network, including protocols to access and handle them.

VLIT. Virtual Litfasssäule. A virtual advertising column used in Nexus.

VR. *see* VIRTUAL REALITY

VRML. Virtual Reality Markup Language. Allows the convenient description of virtual objects and scenes for AR and VR applications.

Virtual Reality. A computer based technology that allows its user to act in purely virtual environments.

WWW. World Wide Web. Internet based information access service.

XML. Extensible Markup Language. XML is a simple, standard way to delimit text data with so-called tags. It can be used to specify other languages, their alphabets and grammars.

Bibliography

[1] *ARVIKA Homepage*. http://www.arvika.de/.

[2] *DB4O Object-oriented database*. http://www.db4o.com.

[3] *DWARF Project Homepage*. http://www.augmentedreality.de.

[4] *ESRI Homepage - ArcView*. http://www.esri.com.

[5] *GNU GPL Homepage*. http://www.gnu.org/licenses/licenses.html#GPL.

[6] *Goods Homepage*. http://www.ispras.ru/~knizhnik/goods.html.

[7] *IBM DB2 Homepage*. http://www-3.ibm.com/software/data/db2/.

[8] *IBM's TSpaces Homepage*. http://www.almaden.ibm.com/cs/TSpaces/.

[9] *Informix Homepage*. http://www-3.ibm.com/software/data/informix/.

[10] *Linda Tuple Spaces Homepage*. http://www.cs.york.ac.uk/linda/.

[11] *Loria Homepage*. http://www.loria.fr/~gsimon/gilles_anglais.html.

[12] *MySQL Homepage*. http://www.mysql.org/.

[13] *OMG CORBA Homepage*. http://www.corba.org/.

[14] *Oracle Homepage*. http://www.oracle.com.

[15] *PostgreSQL Homepage*. http://www.postgresql.org/.

[16] *Request For Comments*. http://www.rfc.net.

[17] *SapDB Homepage*. http://www.sapdb.org/.

[18] *SGI Inventor Homepage*. http://www.sgi.com/software/inventor/.

[19] *S.O.D.A Simple Object Database Access Homepage*. http://sodaquery.sourceforge.net/.

[20] *SQL W3School Homepage*. http://www.w3schools.com/sql/default.asp.

[21] *UML Unified Modeling Language Homepage*. http://www.uml.org/.

[22] *W3C XML Homepage*. http://www.w3.org/XML/.

[23] *W3C XMLQuery Homepage*. http://www.w3.org/XML/Query.

[24] *Sun Microsystems: JavaSpaces Technology*, 2000. http://java.sun.com/products/javaspaces.

[25] K. AHLERS, A. KRAMER, D. BREEN, P. CHEVALIER, C. CHRAMPTON, E. ROSE, M. TUCERYAN, R. WHITAKER, and D. GREER, *Distributed Augmented Reality for Collaborative Design Applications*, Eurographics '95 Proceedings, Maastricht, (1995).

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

[27] M. BAUER, *Distributed Wearable Augmented Reality Framework (DWARF) Design and Implementation of a Module for the Dynamic Combination of Different Position Tracker*, Master's thesis, Technische Universität München, 2001.

[28] M. BAUER, B. BRÜGGE, G. KLINKER, A. MACWILLIAMS, T. REICHER, S. RISS, C. SANDOR, and M. WAGNER, *Design of a Component-Based Augmented Reality Framework*, In IEEE and ACM International Symposium on Augmented Reality, (2001).

[29] K. BECK, *Extreme Programming - Das Manifest*, Addison-Wesley, 2000.

[30] K. BECK and M. FOWLER, *Extreme Programming - Planen*, Addison-Wesley, 2001.

[31] J. BLOCH, *Effective Java*, Addison-Wesley, 2002.

[32] D. A. BOWMAN and C. A. WINGRAVE, *Design and Evaluation of Menu Systems for Immersive Virtual Environments*, in VR, 2001, pp. 149–156.

[33] I. BRONSTEIN, K. SEMENDJAJEW, G. MUSIOL, and H. MÜHLIG, *Taschenbuch der Mathematik*, Verlag Harri Deutsch, 1995.

[34] B. BRÜGGE and A. H. DUTOIT, *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[35] A. BUTZ, C. BESHERS, and S. FEINER, *Of Vampire Mirrors and Privacy Lamps: Privacy Management in Multi-User Augmented Environments*, Technical Report, (1998).

[36] R. CAREY and G. BELL, *The Annotated Vrml 2.0 Reference Manual*, Addison-Wesley Pub Co, 1997.

[37] M. K. DALHEIMER, *Programming with Qt*, O'Reilley Verlag GmbH & Co. KG, 2002.

[38] N. DAVIES, S. WADE, A. FRIDAY, and G. BLAIR, *Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications*, in Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP 1997), 1997.

[39] B. DOUGLASS, *Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison-Wesley, 1995.

[40] B. ECKEL, *Thinking in Java*, Addison-Wesley, 2000.

[41] E. B. (EDITOR), *Information Appliances and Beyond*, Morgan Kaufmann Publishers, 2000.

[42] J. M. C. (EDITOR), *Human-Computer Interaction in the New Millennium*, Addison-Wesley Pub Co, 2001.

[43] S. FEINER, B. MACINTYRE, and T. HÖLLERER, *Wearing It Out: First Steps Toward Mobile Augmented Reality Systems*, First International Symposium on Mixed Reality (ISMR 1999), (1999).

[44] M. FIORENTINO, R. DE AMICIS, G. MONNO, and A. STORK, *Spacedesign: A Mixed Reality Workspace for Aesthetic Industria l Desgin*, In Proceedings of the IEEE and ACM: ISMAR 2002, (2002).

[45] G. W. FITZMAURICE, H. ISHII, and W. BUXTON, *Bricks: Laying the Foundations for Graspable User Interfaces*, in CHI, 1995, pp. 442–449.

[46] M. FJELD, M. BICHSEL, M. RAUTERBERG, and I PRESS, *BUILD-IT: A Brick-based Tool for Direct Interaction*, 1986.

[47] M. FJELD, N. IRONMONGER, S. G. SCHÄR, and H. KRUEGER, *Design and Evaluation of Four AR Navigation Tools Using Scene and Viewpoint Handling*.

[48] M. FJELD, S. G. SCHÄR, D. SIGNORELLO, and H. KRUEGER, *Alternative Tools for Tangible Interaction: A Usability Evaluation*.

[49] M. FOWLER, *Refactoring*, Addison Wesley, 2000.

[50] D. FRITSCH, D. KLINEC, and S. VOLZ, *Positioning and Data Management Concepts for Location Aware Applications*, Technical Report, (2000).

[51] E. GAMMA, R. HELM, R. JOHNSON, and J. VLISSIDES, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[52] D. GELERNTER, *Generative communication in Linda*, in ACM Transactions on Programming Languages and Systems, 7(1): 80-112, 1985.

[53] GOSLING, BOLLELLA, BROSGOL, DIBBLE, FURR, HARDIN, and TURNBULL, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[54] J. GOSLING, *JavaSpaces - Innovative Java Technology that Simplifies Distributed Application Development*, WhitePaper.

[55] S. GRIBBLE, M. WELSH, J. VON BEHREN, E. BREWER, D. CULLER, N. BORISOV, S. CZERWINSKI, R. GUMMANDI, J. HILL, A. JOSEPH, R. KATZ, Z. MAO, S. ROSS, and B. ZHAO, *The Ninja Architecture for Robust Internet-Scale Systems and Services*, Computer Networks 35, (2001).

[56] K. B. HAYES, *Communication Behaviors of Co-located Users in Collaborative AR Interfaces*.

[57] C. HESS, M. ROMAN, and R. CAMPBELL, *Building Applications for Ubiquitous Computing Environments*, Pervasive 2002, (2002).

[58] O. HILLIGES, *Development of a 3D-View Component for DWARF based Applications*. Systemenwicklungsprojekt, Technische Universität München, 2003.

[59] C. HOFMEISTER, E. NORD, and D. SONI, *Applied Software Architecture*, Addison-Wesley, 2000.

[60] R. JEFFRIES, A. ANDERSON, and C. HENDRICKSON, *Extreme Programming - Installed*, Addison-Wesley, 2001.

[61] H. KATO, M. BILLINGHURST, and I. POUPYREV, *ARToolKit version 2.33 Manual*, 2000. Available for download at http://www.hitl.washington.edu/research/shared_space/download/.

[62] A. KEMPER and A. EICKLER, *Datenbanksysteme*, Oldenburg, 2001.

[63] C. KULAS, *Usability Engineering for Ubiquitous Computing*, Master's thesis, Technische Universität München, 2003.

[64] M. KURZAK, *Tangible Design Environment*, Master's thesis, Universität Stuttgart, 2003.

[65] R. LANGENDIJK, *The TU-Delft research program "Ubiquitous Communications"*, 21st Symposium on Information Theory, (2000).

[66] F. LOEW, *Maintainance Task Engine with ARToolKit*. Systemenwicklungsprojekt, Technische Universität München, 2003.

[67] K. LYONS and T. STARNER, *Mobile capture for wearable computer usability testing*.

[68] A. MACWILLIAMS, *Using Ad-Hoc Services for Mobile Augmented Reality Systems*, Master's thesis, Technische Universität München, 2001.

[69] A. MACWILLIAMS, C. SANDOR, M. BAUER, M. WAGNER, B. BRÜGGE, and G. KLINKER, *Herding Sheep: Live System Development for Distributed Augmented Reality*, Technical Report, (2003).

[70] A. MACWILLIAMS and T.REICHER, *Decentralized Coordination of Distributed Interdependent Services*, Technical Report, (2003).

[71] Z. MARX, *Interaction Elements for Studierstube*, Master's thesis, Vienna University of Technology, 2002.

[72] D. J. MAYHEW, *The Usability Engineering Lifecycle*, Morgan Kaufmann Publishers, 1991.

[73] A. MEIER and T. WÜST, *Objectorientierte Datenbanken*, dpunkt.verlag, 1997.

[74] F. MICHAHELLES, *Designing an Architecture for Context-Aware Service Selection and Execution*, Master's thesis, Ludwig–Maximilians–Universität München, January 2001.

[75] P. NEUMANN, *Computer Related Risks*, Addison-Wesley, 1995.

[76] D. NICKLAS, M. GROßMANN, T. SCHWARZ, and S. VOLZ, *Architecture and Data Model of NEXUS*, Technical Report, (2002).

[77] M. OBREITER, *Analyse und Konzeption von Tuple Spaces im Hinblick auf Skalierbarkeit*, Telecooperation Office, Universität Karlsruhe, 2002.

[78] A. OLWAL, *Unit - A Modular Framework for Interaction Technique Design, Development and Implementation*, Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2002.

[79] G. REITHMAYR and D. SCHMALSTIEG, *Open Tracker - An Open Software Architecture for Reconfigurable Tracking based on XML*, Technical Report, (2000).

[80] G. REITHMAYR and D. SCHMALSTIEG, *Mobile Collaborative Augmented Reality*, In Proceedings of the IEEE and ACM: ISAR 2001, (2001).

[81] L. RICHTER, *Design and Practical Guideline to a CORBA-based Communication Framework*, 2002.

[82] S. RISS, *A XML based Task Flow Description Language for Augmented Reality Applications*, Master's thesis, Technische Universität München, 2000.

[83] M. ROMAN, C. HESS, R. CERQUEIRE, A. RANGANTHAN, R. CAMPBELL, and K. NAHRSTEDT, *A Middleware Infrastructure to Enable Active Spaces*, IEEE Pervasive Computing, (2002).

[84] J. RUBIN, *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, John Wiley & Sons, 1994.

[85] J. RUMBAUGH, I. JACOBSON, and G. BOOCH, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[86] J. RUMBAUGH, I. JACOBSON, and G. BOOCH, *The Unified Modeling Lnguage User Guide*, Addison-Wesley, 1999.

[87] C. SANDOR, *CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces*, Master's thesis, Technische Universität München, 2000.

[88] M. SANTIFALLER, *TCP/IP und ONC/NFS*, Addison Wesley, 1998.

[89] B. SCHIENMANN, *Kontinuierliches Anforderungsmanagement: Prozesse - Techniken - Werkzeuge*, Addison-Wesley, 2002.

[90] D. SCHMALSTIEG, A. FUHRMANN, G.HESINA, Z. SZALAVARI, L. M. ENCARNACAO, M. GERVAUTZ, and W. PURGATHOFER, *The Studierstube Augmented Reality Project*, Technical Report, (2000).

[91] D. SCHMALSTIEG, A. FUHRMANN, G. HESINA, and W. PURGATHOFER, *Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics*, Technical Report, (2000).

[92] D. SCHMALSTIEG, G. REITMAYR, and G. HESINA, *Distributed Applications for Collaborative Three-Dimensional Workspaces*, Technical Report, (2002).

[93] H. SCHREIBER, *Performant Java*, Addison-Wesley, 2002.

[94] B. SHNEIDERMAN, *Designing the User Interface*, Addison-Wesley Publishing, 1997.

[95] K. SHOEMAKE, *Quaternions*, Department of Computer and Information Science University of Pennsylvania Philadelphia, PA 19104 (1991).

[96] F. STRASSER, *Personalized Ubiquitous Computing with Handhelds in an Ad-Hoc Service Environment*. Systemenwicklungsprojekt, Technische Universität München, 2003.

[97] B. STROUSTRUP, *The C++ Programming Language*, Addison-Wesley Pub Co, 2000.

[98] D. SVANÆS and W. VERPLANK, *In search of metaphors for tangible user intefaces*, in Proceedings of DARE 2000 on Designing augmented reality environments, ACM Press, 2000, pp. 121–129.

[99] A. TANENBAUM and J. GOODMAN, *Computerarchitektur*, Addison-Wesley, 1999.

[100] B. F. TANG A., OWEN C. and M. W., *Comparative Effectiveness of Augmented Reality in Object Assembly*, in Proceedings of ACM CHI 2003, April 5 April 10, 2003, Ft. Lauderdale, Florida, US, 2003.

[101] M. TÖNNIS, *Data Management for AR Applications*, Master's thesis, Technische Universität München, 2003.

[102] A. TRIPATHI, *Augmented Reality Application for Architecture*, Master's thesis, University of Southern California, 2000.

[103] VARIOUS, *IWAR 1999*, Proceedings of the IEEE International Workshop on Augmented Reality, (1999).

[104] VARIOUS, *ISAR 2000*, Proceedings of the IEEE and ACM International Symposium on Augmented Reality, (2000).

[105] VARIOUS, *ISAR 2001*, Proceedings of the IEEE and ACM International Symposium on Augmented Reality, (2001).

[106] VARIOUS, *ISMAR 2002*, Proceedings of the IEEE and ACM International Symposium on Mixed and Augmented Reality, (2002).

[107] S. VOLZ, D. FRITSCH, and D. KLINEC, *NEXUS: Spatial Model Servers for Location Aware Applications on the basis of ArcView*, Technical Report, (1999).

[108] S. VOLZ and M. SESTER, *NEXUS - Distributed Data Management Concepts for Location Aware Applications*, Technical Report, (2000).

[109] S. VOLZ, M. SESTER, D. FRITSCH, and D. KLINEC, *NEXUS - Eine Plattform für ortsabhaengige, verteilte Geodatennutzung*, Technical Report, (2002).

[110] M. WAGNER, *Design, Prototypical Implementation and Testing of a Real-Time Optical Feature Tracker*, Master's thesis, Technische Universität München, 2000.

[111] A. WEBSTER, S. FEINER, B. MACINTYRE, W. MASSIE, and T. KRUEGER, *Augmented Reality in Architectural Construction*, 1996.

[112] M. WEISER, *The Computer for the 21st Century. Scientific American*, Scientific American, (1991), pp. 94–104.

[113] J. WERNECKE, *The Inventor Mentor: Programming Object Oriented 3D Graphics with OpenInventor, Release 2*, Addison Wesley, 1994.

[114] J. WERNECKE, *The Inventor Toolmaker: Extending OpenInventor, Release 2*, Addison Wesley, 1994.

[115] J. WÖHLER, *Driver Development for TouchGlove Input Device for DWARF based Applications*. Systemenwicklungsprojekt, Technische Universität München, 2003.

[116] M. WOO, J. NEIDER, T. DAVIS, D. SHREINER, and O. A. R. BOARD, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, Addison-Wesley Pub Co, 1999.

[117] P. WYCKOFF, S. MCLAUGHRY, T. LEHMANN, and D. FORD, *T-Spaces*, in IBM Systems Journal, 1998.

[118] B. ZAUN, *A Bluetooth Communications Service for DWARF*, 2000.

[119] B. ZAUN, *Calibration of Virtual Cameras for AR*, Master's thesis, Technische Universität München, 2003.

Index