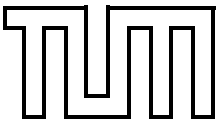
A Framework for Dynamically Adaptable Augmented Reality Systems

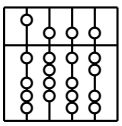
---

Thomas Reicher

Institut für Informatik
Technische Universität München



Institut für Informatik
der Technischen Universität München



A Framework for Dynamically Adaptable Augmented Reality Systems

Thomas Reicher

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Christoph Zenger
Prüfer der Dissertation:	
	1. Univ.-Prof. Bernd Brügge, Ph.D.
	2. Univ.-Prof. Gudrun J. Klinker, Ph.D.
	3. Univ.-Prof. Dr. Dr. h.c. Manfred Broy

Die Dissertation wurde am 27.11.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.04.2004 angenommen.

Abstract

In this dissertation I present a software framework for adaptable Augmented Reality systems in intelligent environments: the Distributed Wearable Augmented Reality Framework (DWARF). The framework is a reusable basis for the development of Augmented Reality applications and gives developers a system structure to customize existing or develop new components.

Augmented Reality is a human-computer interaction paradigm offering users new possibilities to interact with their environment. By tracking the position and viewing direction of the user and of real world objects in the environment, real world objects can be augmented with virtual objects. The most prominent example is the overlay of real world objects with computer graphics in a head-mounted display.

DWARF models Augmented Reality applications in four abstraction layers: application layer, inter-application layer, solution domain layer, and architectural style layer.

For the architectural style layer, I specify a contract-based peer-to-peer style. A system is a configuration of mutually interdependent distributed services. The connections between services are established by an active middleware based on an abstract service specification. Existing solutions for Augmented Reality are usually designed as monolithic or client/ server systems. DWARF uses an extension of component-oriented software engineering. Components on the user's computer and in the environment are both used as building blocks for Augmented Reality systems. Further on, I extend the component-based system approach to a service-based approach. An application is not built from instances of components, but from a configuration of services available on the user's computer and in the environment. An active middleware enables the dynamic adaptation of the system to changes in the environment by a reconfiguration of the system.

For the solution domain layer, I present an abstract reference architecture and a system of patterns for Augmented Reality systems. The foundation for them is a broad analysis of existing Augmented Reality systems.

Finally, as a case study for the applicability of the framework, I present the sub framework M^{3}ARF for mobile maintenance systems and a navigation application as part of such a mobile Augmented Reality maintenance system, DWARF Pathfinder. This case study covers the application layer, the inter-application layer, and the solution domain layer.

Kurzfassung

Diese Dissertation stellt ein Softwaregerüst für anpassbare Systeme der Erweiterten Realität in intelligenten Umgebungen vor: das Distributed Wearable Augmented Reality Framework (DWARF). Dieses Framework ist eine wieder verwendbare Basis zur Entwicklung von Anwendungen der Erweiterten Realität und gibt Entwicklern ein Gerüst vor, um existierende Komponenten anzupassen oder eigene hinzuzufügen.

Erweiterte Realität (Augmented Reality) ist ein Interaktionsparadigma, das Benutzern neue Möglichkeiten zur Interaktion mit ihrer Umgebung ermöglicht. Durch das Verfolgen von Position und Blickrichtung der Benutzer und realer Objekte in der Umgebung, kann die Welt der realen Objekte durch virtuelle Objekte erweitert werden. Das bekannteste Beispiel ist die Überlagerung von realen Objekten und Computergrafik in einer Datenbrille (Head-mounted Display).

DWARF beschreibt Anwendungen der Erweiterten Realität auf vier Abstraktionsschichten: Applikationsschicht, Interapplikationsschicht, Lösungsraumschicht und die Schicht des Architekturstils.

Für die Schicht des Architekturstils schlage ich einen vertragsbasierten Peer-to-Peer-Architekturstil vor. Ein System besteht aus einer Konfiguration von gegenseitig abhängigen verteilten Diensten. Der Aufbau von Verbindungen zwischen Diensten erfolgt durch eine aktive Middleware, die auf der Basis von Dienstbeschreibungen die Verknüpfungen herstellt. Bisherige Lösungen für Erweiterte Realitätssysteme sind meist entweder monolithisch oder als Client/Server-System aufgebaut. DWARF verwendet eine Erweiterung der komponentenorientierten Softwareentwicklung. Komponenten auf dem Anwenderrechner und in der Umgebung werden gleichberechtigt als Systembausteine verwendet. Zudem erweitere ich den komponentenorientierten Ansatz zu einem dienstorientierten Ansatz. Eine Anwendung besteht nicht aus Instanzen von Komponenten, sondern aus einer Konfiguration von Diensten auf dem Anwenderrechner und Diensten in der Umgebung der Anwenders. Eine aktive Middleware ermöglicht die dynamische Anpassung an Änderungen in der Umgebung durch eine Umkonfiguration des Systems.

Für die Lösungsraumschicht gebe ich eine abstrakte Referenzarchitektur und ein System von Entwurfsmustern für Systeme der Erweiterten Realität an. Grundlage ist eine umfassende Analyse von bestehenden Systemen.

Schließlich gebe ich als Fallstudie für die Anwendbarkeit des Frameworks das Teilframework M^{3}ARF für mobile Wartungssysteme und eine darauf aufbauende Navi-

gationsanwendung als Teil eines solchen mobilen Wartungssystems an, DWARF Pathfinder. Diese Fallstudie umfasst die Applikationsschicht, die Interapplikationsschicht und die Lösungsraumschicht.

Preface

The history of the DWARF project The starting point of developing an Augmented Reality system at the Technische Universität München was the course "Erweiterte Realität: Bildbasierte Modellierung und Tragbare Computer". It was part of the 1999 summer school organized by the Technische Universität München and the Friedrichs-Alexander Universität Erlangen-Nürnberg. Prof. Bernd Brügge and Prof. Gudrun Klinker had the vision that software engineering and Augmented Reality would have a stimulating influence on each other.

There, the idea was born to develop an own Augmented Reality system based on modern software engineering concepts. The project name was DWARF for Distributed Wearable Augmented Reality Framework.

The DWARF project started in earnest in spring 2000 and resulted in the first DWARF application, DWARF Pathfinder. It was demonstrated successfully in fall 2000. After that, the DWARF framework became the basis for several other Augmented Reality student projects at the Technische Universität.

But now DWARF has become a complex framework. It is the result of applying software engineering to develop a reusable framework for adaptable mobile Augmented Reality systems in ubiquitous computing environments. Since the first version of DWARF, a lot of development work and improvement has beeb done by several people on each of the different layers of the framework. There are several remaining research issues which are theses of their own. However, the basic concept of DWARF as distributed service-based system is still the same, and has proven valid for Augmented Reality system design.

Acknowledgements I would like to thank all people that supported me, and apologize to all which I had not enough time for.

The people I want to thank first are the original members of the DWARF team (the "seven dwarfs"): Martin Bauer, Asa MacWilliams, Florian Michahelles, Stefan Riß, Christian Sandor, Martin Wagner, and Bernhard Zaun. Without their efforts and talent, DWARF never would have been as successful as it appears to be now. The collaboration with them in the DWARF team has always been very enjoyable. In particular I want to thank Asa, who is the one who shares my interests and visions in respect to software architectures for Augmented Reality.

Next, I want to thank Prof. Bernd Brügge and Prof. Gudrun Klinker who had the initial idea to implement a mobile Augmented Reality system as part of an intelligent building. Their concepts for intelligent buildings developed within the OWL project at the Carnegie Mellon University, Pittsburgh and the Technische Universität München, and the vision of Augmented-Reality-ready buildings, developed into the idea of a completely distributed peer-to-peer Augmented Reality system. This concept has shown to be very flexible and open for future enhancements.

For the reading and the many valuable comments on previous versions of this work, I want to thank Prof. Bernd Brügge, Prof. Gudrun Klinker, and the former dwarfs and now colleagues Asa, the two Martins, and Christian.

I want to thank all my friends and my family who encouraged and supported me during the last years. There was not much time for them for too long.

And finally, I want to thank my two little women Gabi and Luisa for their endless patience with me during the creation of my dissertation. No one has supported me more on my way than they did. Thank you!

Overview

Contents

List of Figures

1 Introduction

An introduction into the research area and the technical problem, the identification of four abstraction layers, goals, hypotheses, approach, and contributions of this dissertation.

Software engineering is *"a systematic approach to the analysis, design, implementation and maintenance of software..."* [35]. This term was used first in 1967 by Friedrich L. Bauer in a meeting of the NATO science committee to describe the state of the art on the field of software development at that time: *"There is so much tinkering with software ... what we need is software engineering."* [12]. Since then, the term 'software engineering' has been retained to imply the need to develop software on the base on the types of theoretical foundations and practical disciplines, that are well established in other branches of engineering [101].

Software engineering is a modelling, problem-solving, knowledge acquisition, and rational-driven activity. Modelling helps to deal with complexity. Models are used to search for an acceptable solution and to collect, organize, and formalize data into information and knowledge. For the assumptions and decisions during the development process the context and the rationale behind them must be captured [22].

Since Augmented Reality evolved as a full-fledged topic of research, many Augmented Reality systems have been developed. Most of them are research prototypes that demonstrate a particular aspect, such as tracking or human-computer interface design. In most cases software engineering in general and the software architecture in particular was a secondary issue. The reuse of artefacts was done, at best, on the source code level.

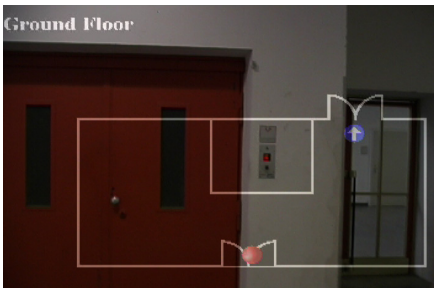
The subject of this dissertation is the application of software engineering to build a reusable and adaptable framework for Augmented Reality systems, the distributed wearable augmented reality framework DWARF. The focus is on mobile Augmented Reality systems in ubiquitous computing environments with the continuous example of maintenance of complex systems, such as machine-tools.

In the next section we start with an informal introduction of Augmented Reality (section 1.1). In section 1.2 we discuss several research fields that contribute to Augmented Reality. The following section 1.3 delves into several technologies that enable the realization of mobile Augmented Reality systems and ubiquitous computing. We

present the goals of this dissertation (section 1.4), the underlying hypotheses (section 1.5), and the chosen approach to prove these theses (section 1.6). In section 1.7 we introduce a schema with four abstraction layers for the modelling of complex Augmented Reality systems. We will use this schema throughout this dissertation. In particular we use it to specify the layers of the DWARF framework (section 1.7.3). After that list the contributions of this dissertation (section 1.8). Finally we give the outline of this work in (section 1.9).

1.1 What is Augmented Reality?

Traditionally, Augmented Reality has been seen as a human computer interaction paradigm, which provides users with a new way to interact with their environment. A core feature is the tracking of the user's position and viewing direction (called the pose) for the overlay of artificial artefacts over physical objects in the user's perception. As a result, the user perception of the world is augmented by virtual objects. The most prominent example is superimposing computer graphics on objects of the user's environment in a head-mounted display (HMD). The real world objects are represented by any kind of data that can be measured by sensors, and include primarily video, but also photographic images (visible or infrared), radar, X-ray, and ultrasound, as well as laser scanned range data.



(a) Indoors application

(b) Outdoors application

Figure 1.1: Screenshots through head-mounted display.

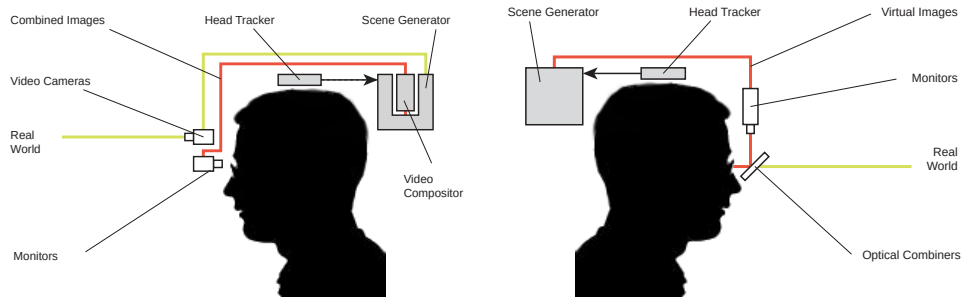
A widely used definition of Augmented Reality by Ronald Azuma in [7]

> *"... defines AR as systems that have the following three characteristics:*
>
> 1. *Combines real and virtual*
> 2. *Interactive in real time*
> 3. *Registered in 3-D*

This definition allows other technologies besides HMDs while retaining the essential components of AR."

This definition does not limit Augmented Reality to "visual" Augmented Reality but also includes audio. Nevertheless, visual Augmented Reality is most commonly used and was also used in this dissertation. As an example see the screenshots in figure 1.1 taken from the DWARF Pathfinder application. We will discuss Pathfinder in Chapter 5.

The main technical problem is how to present the virtual objects to the user. There are two different approaches: optical see-through Augmented Reality (figure 1.2(a)) or video see-through Augmented Reality (figure 1.2(b)). Both are usually implemented with a head-mounted display.



(a) Optical see-through HMD (b) Video see-through HMD

Figure 1.2: Diagrams of video and optical see-through head-mounted displays (from [13])

An *optical see-through head-mounted display* uses a half mirror to project virtual objects into the users field of view. In contrast, a *video see-through* head-mounted display captures images of the environment with a video camera, combines them with graphical objects from a 3D image generator and projects the resulting image to a head-mounted display directly in front of the eyes of the user.

The user tracking can be separated into two main approaches: *inside-out tracking* and *outside-in tracking*. Inside-out tracking is a technique where a tracking system on the user side tracks the position and orientation. An example is optical tracking by a camera mounted on the user's head that tracks markers in the environment. Typically, a optical tracking system such as the ARToolkit [73] with a head-mounted camera is used for this approach. Outside-in is the opposite approach. Tracking devices in the user's environment track the user from outside and send the information to the system. An example is tracking a marker on the user's head with an external camera system,

for example the A.R.T. system [3]. Inside-out and outside-in tracking can be combined to improve the tracking quality.

Applications of Augmented Reality can be found in several areas such as power plants service [77], the installation of wire bundles in planes [33], or in car assembly [122].

1.2 Augmented Reality and Contributing Research Fields

Tracking and visualization are only a part of the aspects of Augmented Reality. There are several contributing technologies that have to be combined.

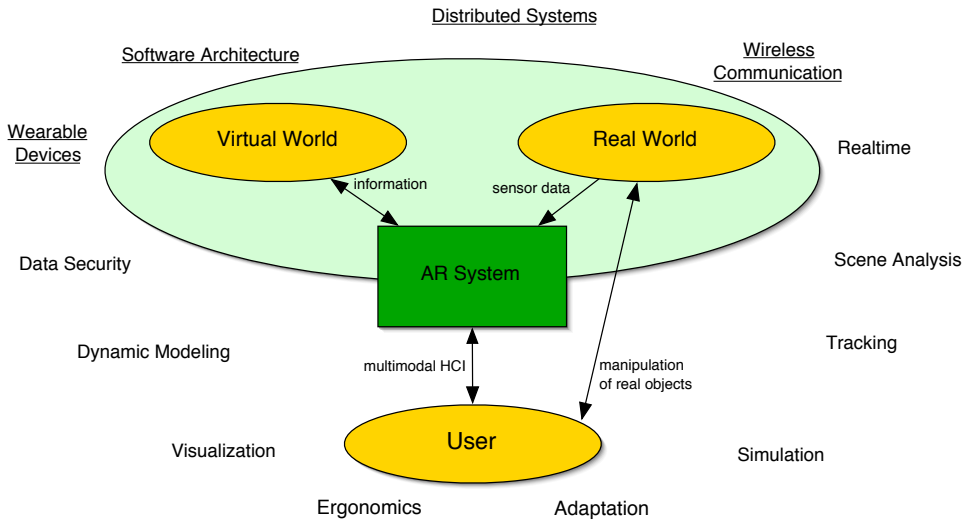


Figure 1.3: Informal model of Augmented Reality as multi-medial combination of real and virtual world data [76].

Figure 1.3 shows an informal model of an Augmented Reality system that combines real world and virtual world objects (or data). Real world data are accepted over *sensors* and data of the virtual world are queried from an information system. Real and virtual data are combined in human-computer interaction (HCI) devices.

There are several enabling technologies from various research fields used for an Augmented Reality system. These include software architectures as a subtopic of software engineering, distributed systems, wireless communication, wearable devices/wearable

computing, real time computing, scene analysis, tracking, simulation, adaptation, ergonomics, visualization, dynamic modelling, and data security.

In the scope of this dissertation the former four were involved: software architectures, distributed systems, wireless communication, and wearable devices.

An important addition in this model is the manipulation of real world data by the user. This manipulation must be sent back to the Augmented Reality system over the sensors. The feeding back of the manipulation of real world data into the system and the required reactions is a subject of research on its own and beyond the scope of this dissertation.

1.3 Enabling Technologies

Advances in four important fields are enabling new ways of human-computer interaction: computing power, network communication, user input/output devices, and awareness of the user's context. The most prominent context elements are the user's location and viewing direction.

Computing power. The technical advances in the microchip technology have changed the application areas of the computer. The size of the devices has shrunk while the computing power has increased enormously. Formulating *Moore's Law* in 1965 [99], Gordon Moore, founder of Intel Inc., expected a doubling of the number of transistors per square inch on a computer chip each fifteen months. While Gordon Moore retired some years ago, Intel's engineers believe that this law will still be valid for at least another ten years. This trend will enable a rich set of variants among computer chips ranging from comparable larger chips with more transistors to tinier chips with a smaller number of transistors. More transistors can not only be used to increase computing power but enable also new functionality. For example, Intel has plans to add radio transmission to microprocessors.

With the decreasing size of transistors the design space for microchips is increasing. There is a wide spectrum with maximum compute power for server processors such as the Intel Pentium IV or Itanium on one end and chips for embedded systems with comparable fewer possibilities on the other end. On the one side system size and energy consumption can be neglected while on the other side small physical size and energy consumption is very important. Available processors and micro chips are located all along this spectrum.

Network communication. In addition to the increasing computing power and the decreasing device size, advances in network technologies, especially in the field of wireless networking, have opened new possibilities. Some years ago 10 Mbit/s were

the maximum possible transfer rate in a LAN[1], now wired networks reach a transfer rate of one gigabit per second and wireless networks such as IEEE 802.11g outperform the rates of many existing wired networks.

Similar to computing devices along the power/size continuum, different types of networks for different needs have been developed. This is especially true for wireless networks where the transfer rate is correlated to the possible geographic network size and the needed energy for data transmission. The particular network types are usually separated according to the diameter of the network. Client devices, called *multi-network terminals*, with access to more than one network are becoming available. Depending on the application a particular network is dynamically chosen.

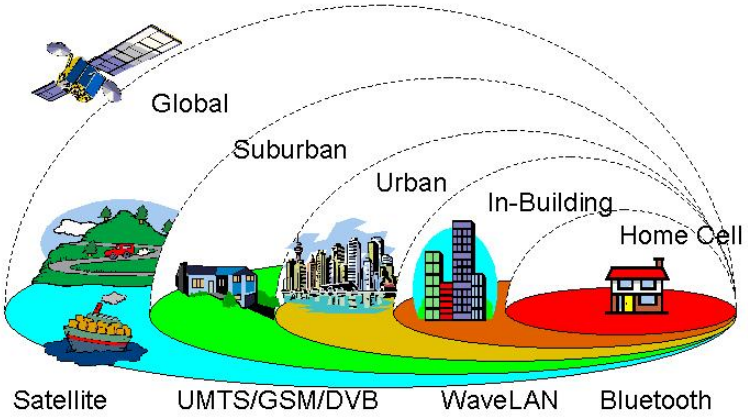


Figure 1.4: Network types

In figure 1.4 there are examples for several networks that could be available at the same time. Each network is suitable for a particular range that an application needs to cover. Starting with a Bluetooth [19] network for short range, low power networks for in-house use, each network type covers a larger geographical area. *WaveLAN* networks are usable for larger buildings with ranges of several hundred meters, UMTS[2], GSM[3] or Digital Video Broadcast (DVB) networks for the urban and suburban area, and satellite networks for global access. There are also gateways between sub-nets, for example between the UMTS networks of different network providers.

User input/output devices. The development of the 2-dimensional windows-based graphical user interface (GUI) in the 1970s was a big advance over the text-based

[1]Local Area Network
[2]Universal Mobile Telecommunications System
[3]Global System for Mobile Communications

user interface for mainframe access. A high-resolution monitor, keyboard and mouse allowed a much more intuitive interaction between human and computer. The enduring success and ongoing improvement of this concept has shown that windows-based GUIs provide suitable metaphors for desktop computing.

When computing devices become more mobile these well-known input/output devices are no longer appropriate. The need for lightweight devices and hands-free operations in a 3-dimensional world drives the development of new devices. Examples include head-mounted displays or speech recognition for pilots of fighter helicopters. Moreover, the move of the computer from the desktop in the office to the field requires the development of new devices that allows an interaction of users related to real world objects such as machines or cars near their current position. This is the subject of ongoing HCI (human computer interaction) research. Results of this research are *laser pointer*, *data glove* or *space mouse*. These devices are often combined for multi-modal input and multimedia output.

Context awareness. Dey defines the term context as follows: "*...Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.*" [34]. The computer may use this information for filtering, decision making or generally, trying to behave intelligently. The computer can gather the user profile, history, and actions easily with means such as logging or listening to user input and output devices. The perception of the user's environment is much more complicated as it requires techniques such as a GPS[4] to recognize the user's position or various tracking devices to perceive the user's viewing direction. Generally, the user's environment can be perceived by *sensors*.

1.4 Goals of this Dissertation

The goal of this dissertation was to develop a software framework for Augmented Reality to rapidly build systems that allow users with a mobile or wearable device to operate in a ubiquitous computing environment.

We identify three core requirements for such systems.

Real time performance for Augmented Reality. This is the primary requirement which decides upon the usability of an Augmented Reality system, although this requirement is not as strict for Augmented Reality systems as, for example, for airplane control systems.

Real-time systems can be classified into hard, firm, and soft real time systems [56, 141]. A real time system is called hard real time system if the consequences of not

[4] Global Positioning System

meeting a deadline within the system are catastrophic. Periodic tasks such as recalculating the airfoil position usually have deadlines of this kind. A deadline is called firm if the results produced are not useful after the deadline, but the consequences are not very severe. The deadlines of many aperiodic tasks belong to this category. Real time systems with a deadline, which is neither hard nor firm, are called soft real time systems. Failing to meet such a deadline does not have catastrophic consequences.

Real-time in the Augmented Reality context means that the update rate for the overlaid graphic must be acceptable by the user, which means a target value of 30 frames per second. The delivery and processing of position must happen in a specific time frame to be useful but the consequences are not severe. So Augmented Reality systems could be called firm real time systems. However, in the Augmented Reality community the update rate is a target rate. Currently there is no Augmented Reality system that can guarantee a time frame for position updates. Existing Augmented Reality systems do not use real time operating systems with real time scheduler for the underlying platform but relay on enough left over resources on a standard operating system. Indeed, on most systems recalculating the user position and updating the view are either the only user tasks of the platform or the resources are so high that it will never come in a critical range that leads to a drop or delay of frames.

The real time requirements require low latency communication between the participating subsystems of an Augmented Reality system. This means that the communication overhead of the middleware must be minimized. Traditional middleware techniques such as CORBA [109] or COM+ [94] do not support heterogeneous communication. There is only one defined communication protocol such as IIOP for CORBA and DCOM for COM+ for all communication purposes. This is insufficient for the transport of multimedia data such as video streams.

Seamless integration of components in the environment. Another requirement is the ability to combine services on mobile, wearable devices of individual users with services by an ambient, pervasive environment. An example are 'AR ready buildings', where buildings are equipped for user tracking for Augmented Reality [79]. This requirement is similar to ambient computing where services located in the user's current environment can be used dynamically, rather than being hard-wired to a specific application, device or user.

An additional requirement is the support of multiple interaction devices for input and output, local devices worn by the user as well as ambient, stationary devices. Wearable devices and environment components are treated the same.

To express that we want to integrate services from local components and components in the environment we use the term *service* instead of component. A Service could already be running when the user wants to use it or it can be started on demand

Adaptability to changes in the environment. A final requirement is that a mobile user must be able to freely enter and leave the access range of services in the environment. The user's infrastructure must be able to find and integrate new services. It must optimize the overall system functionality by connecting matching services and handling the loss of connection between services, such as when the user leaves a room. This requires the declaration of what a service needs and what it is able to provide.

1.5 Hypothesis

Most existing Augmented Reality systems are developed more or less as demonstrators for new tracking technologies or user interface approaches. In order to develop a new application everything must be written from scratch or be reused on the base of existing source code and the developer's knowledge how to use it.

We claim that it is possible to separate Augmented Reality systems into several abstraction layers and handle each layer independently.

Further on, we claim that we can construct Augmented Reality systems on the base of distributed components. Local components on the client system and components in the environment can be treated equally without giving up the real time performance requirement.

Traditionally, the design of mobile Augmented Reality systems has focused on standalone monolithic or on client/server systems. Communication with remote components was avoided or handcrafted, usually with network sockets. The reason has been that middleware was considered to cause too much communication overhead violating the real time requirement. Indeed, for tracking and visualization researchers were applying many tricks to get as many frames per second on the underlying hardware, particularly on mobile systems. Middleware overhead would make such efforts obsolete.

Furthermore, we claim that Augmented Reality systems share a common set of subsystems that can be specified as components. Different Augmented Reality systems with different goals will implement these components and their collaboration in different ways. In particular, we claim that for each subsystem several patterns can be identified and described.

1.6 Approach

We will develop an Augmented Reality framework on the base of a distributed component-based system. Therefore we separate the application specific issues from the Augmented Reality specific and the communication specific issues.

To address the communication specific issues we analyze existing work in distributed and mobile systems research, specifically component models, architecture description languages, and middleware frameworks. We then specify a new component model and

implement a middleware that supports this model. The middleware is based on the CORBA middleware and extends it with a new CORBA service.

For the Augmented Reality specific issues we analyze existing Augmented Reality systems and extract the core building blocks. The result is the specification of an abstract architecture for Augmented Reality systems and a set of architectural patterns.

From the abstract architecture we derive a sub framework for mobile maintenance applications. Finally, middleware and the building blocks are integrated in a concrete application, DWARF Pathfinder.

1.7 Software Abstraction Layers

A key concept in the design of a framework for Augmented Reality systems, is the notion of software architecture. A well-known concept to handle the complexity of systems is to divide the problem into several abstraction layers. Each layer can use abstractions of the underlying layers. The granularity of the abstractions increases from top to bottom.

1.7.1 Architecture Layers

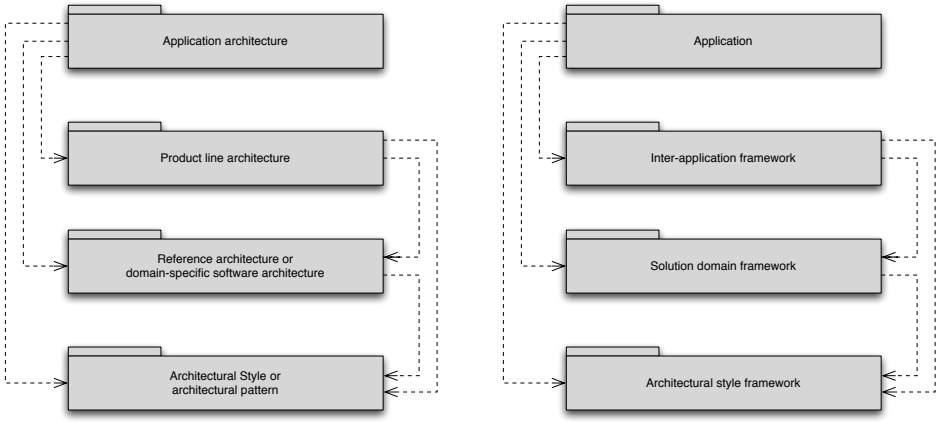
To apply this concept to software architectures we use an analysis of software architecture types by Hofmeister et al. [58]. They identify four different meanings of the term *software architecture*: (1) *(application) software architecture*, (2) *product line architecture*, (3) *reference architecture* [5], and (4) *architectural style* [6].

1. A software architecture describes the architecture for a particular system or product. There are application specific components, adaptations, configurations, and data.

2. A product-line architecture describes an architecture for a set of products that can be adapted for a series of similar applications, called a product line or product family of a particular company. Only minor adaptations for a customer or platform are needed.

3. A domain specific architecture/reference architecture describes an architecture for a system or subsystem in a particular domain. It is useful for certain domains but not for others. Nevertheless, such an architecture is part of many different applications. Such an architecture can be used as a starting point to develop an application in that domain. An example would be a reference architecture for web applications or Augmented Reality applications.

[5] or *domain-specific software architecture* synonymously
[6] or *architectural pattern* synonymously



(a) Relations between Hofmeister's architecture types.

(b) Transfer to relations between framework types.

Figure 1.5: Architectures and frameworks can be separated into a hierarchy of abstraction layers.

4. An architectural style/architectural pattern is *... a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done ...* [139]. It describes a general solution and is usually not domain specific. It is an abstract template and can be reused for several applications.

The different types of architectures are not mutually exclusive, but there are *builds-on* relationships between them. Reference architectures, domain-specific architectures, product-line architectures, and application software architectures all use architectural styles in their design. Product-line architectures and application software architectures can be based on a reference architecture or include parts of one. These in turn can be based on a reference or domain-specific architecture. And the architecture for a particular software can be based on a product-line architecture. Figure 1.5(a) illustrates these relationships.

The factor of the potential reuse is increasing across the layers from top to bottom. It is difficult to reuse the software architecture of a particular application for a new application but architectural styles can be reused for many applications.

1.7.2 Framework Layers

A framework is *"a set of classes providing a general solution that can be refined to provide an application or a subsystem"* [22]. Another definition includes the structure, and the flow of control and/or data and specifies a framework as *"a hybrid of architecture-level information and implementation"* [58, p. 9]. We think it is important to combine the architectural-level and the implementation-level information and define a *software framework* as *the combination of software architecture that can be refined to provide an application or a subsystem with a set of classes as a refinable partial implementation of it.* With this definition we can transfer the layer model for architecture types to a corresponding layer model for framework types. The exception is the (application-specific) software architecture. Usually the architecture for a specific application is not intended for reuse.

For an architectural style there is a framework that supports the usage of this style, the *architectural style framework.* For a reference or domain-specific architecture there is a framework that implements parts of the architecture and provides the base of a solution. We call it *solution domain framework.* A product-line architecture corresponds to a framework that provides components and structures that can be reused in several applications in the same application domain. So we call it an *inter-application framework.* Analogously to figure 1.5(a), figure 1.5(b) shows the relationships of the different types of frameworks.

Table 1.1 summarizes the attributes of the different framework types: the kind of building blocks of each layer, the granularity of them, and how specific they are for a particular application.

1.7.3 The DWARF Framework

We use the four abstraction layers defined above for the development of the DWARF framework for Augmented Reality systems. DWARF consists of four abstraction layers. On each layer sub-frameworks cover issues of the respective layer. Each of these frameworks builds on frameworks of the subjacent layers. Figure 1.6 shows each layer of DWARF and the frameworks developed for each of them. The upper layers use services from the subjacent layers, but not necessarily only from the one directly subjacent layer. The *architecture* of a particular layer could be independent from the other layers, in particular the subjacent layers. For example, components of the solution domain layer can be implemented with several architectural styles such as peer-to-peer or client/server. A particular *framework* on a particular layer however will not be independent since it combines design and implementation and will rely on subjacent layers.

(1) Application layer. The application layer consists of the application code, components, data, and bootstrapping-code for a particular software application. The appli-

Framework type	Kind of building blocks	Granularity of building blocks	Application specificity
Application	specific for a particular application	element specific, usually more complex	high
Inter-application Framework	for applications of the same type, e.g. navigation	coarse	middle
Solution Domain Framework	building blocks of applications of the same technical domain, e.g. Augmented Reality	middle	low
Architectural Style Framework	building blocks for a technical concept, e.g. distributed peer-to-peer systems	fine granular	none

Table 1.1: A comparison of different framework types. We compare them by the kind of building blocks, by the granularity of the building blocks, and by how application specific they are.

cation code is the part that provides the logic for the desired functionality. It initializes components, loads application specific data and executes bootstrapping routines for components provided by the underlying layers. Application components provide functionality for a particular application and are not part of the Augmented Reality framework. They must be developed especially for the application, but they provide a component interface. Bootstrapping-code is needed to initialize framework components and provide them with application specific data.

(2) Inter-application layer. The inter-application layer includes reusable building blocks and supporting components for several applications. The focus of this work is on the domain of Augmented Reality supported maintenance of complex systems. So this layer corresponds to the product-line architecture layer. For different application domains a different set of components and variants of general components are needed. For example, the needed quality of user tracking strongly depends on requirements of the particular application domain. Augmented Reality supported car design needs high precision tracking while augmented reality supported power plant maintenance is less demanding in user tracking but in taskflow dependent data.

(3) Solution domain layer. This layer defines element types and interactions specifically for a particular domain. They describe how domain functionality is mapped to the architectural elements. Within this dissertation the solution domain is *mobile*

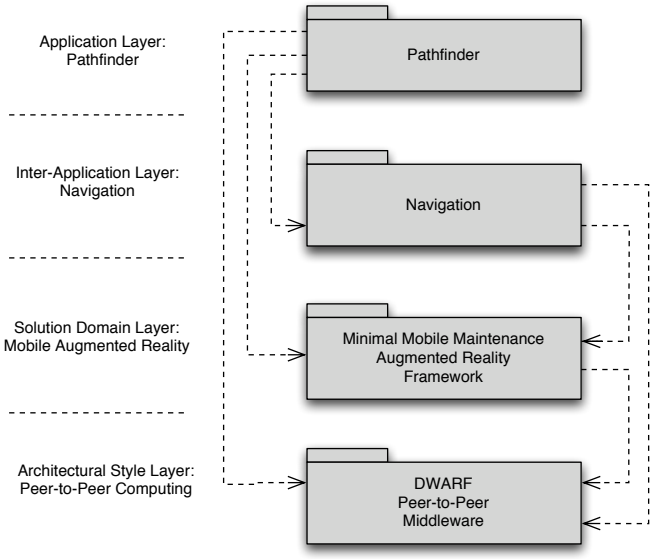


Figure 1.6: The DWARF framework and the relationships of the individual layers. The terms on the left hand side denote abstraction layers and the respective paradigms used in this dissertation. The right hand side shows the developed sub-frameworks of each of these paradigms.

Augmented Reality and the *Minimal Mobile Maintenance Augmented Reality Framework* is a framework for systems of this domain. For example, in Augmented Reality an architecture describes which elements are used for video-based user tracking and how they are interconnected: a video camera feeds images into a buffer. The buffer is read by an image processor that detects features. A tracking component calculates the user position and forwards it to the renderer, which in turn writes the data into the video buffer of the display component.

(4) Architectural style layer. Architectural styles define the element types that can be used to describe an architecture and how they interact. The term *style* is preferred over the term *pattern* to distinguish it from the general use of the term pattern in the sense of design patterns. An architectural style is not domain-specific and the particular style also puts some constraints on how functionality can be mapped to architectural elements. Examples of architectural styles are the Pipes-and-Filters style, the Blackboard style or the Software Bus style [24]. The architectural style layer of the DWARF framework consists of one particular style: *peer-to-peer computing*. The *DWARF Peer-to-Peer Middleware* implements this style.

1.8 Contributions of this Dissertation

The main contribution of this dissertation is a software framework for Augmented Reality systems. The focus is on mobile Augmented Reality systems that support workers in their daily tasks in the production and maintenance domains.

We developed a scheme of four abstraction layers to describe Augmented Reality systems. Each layer covers a specific aspect of an Augmented Reality system. The lowest layer, the architectural style layer, describes the component model. The second layer, the solution domain layer, describes the building blocks for systems in a particular solution domain. In our case this domain is Augmented Reality. The third layer, the inter-application layer, describes reusable building blocks that can be used for a family of similar systems. The fourth layer, the application layer, describes a particular application.

In addition, we have made contributions to each of the four layers.

Division of Problem into Layers of Concern. We developed a multi-layer framework that allows developers to describe Augmented Reality applications in four layers. These layers are from top to bottom: application layer, inter-application layer, solution domain layer, and architectural style layer. We then analyzed and worked on each layer independently. The focus, however, is on the solution domain layer and the architectural style layer.

Peer-to-Peer architectural style for dynamic, distributed service-based systems The disadvantage of traditional middleware solutions for distributed Augmented Reality systems was their communication overhead. Additionally, the middleware was not adaptable to communication requirements between the distributed components. So developers of Augmented Reality systems have not used middleware and have used self-developed low-level approaches based on network sockets instead. This approach led among other drawbacks to inflexible architectures simply because the developers chose to optimize for performance and sacrifice flexibility.

We developed a distributed peer-to-peer architectural style that combines architectural adaptability and flexibility with performance. We separate both issues into two steps. First, we establish the communication relationships between components. To this end, we treat components and connections between components both as first class objects. We adapt the system's transport facilities to the requirements of the components by letting the components select the desired type of connection and make the system establish it. Second, the connected components talk to one another over the communication means that the components chose by themselves.

This architectural style allows to model a Augmented Reality system as a configuration of collaborating peer services. Each collaboration of services is modelled as an Ability/Need pair. Such a pair expresses that one service has an ability that another

service needs. Connectors are first class objects and model the channel type over which services collaborate.

Middleware for the adaptable systems A middleware reads abstract service descriptions and connects services with matching Abilities and Needs. Connectors runtime objects connect the services. This decouples the connection implementation from the service implementations. So the active middleware can manage the connections independently from the services.

Analysis model for Augmented Reality systems For the solution domain layer we analyzed existing Augmented Reality systems. This analysis was done based on literature studies, a questionnaire that was sent to several Augmented Reality system developers, and interviews with Augmented Reality researchers. As a result of the studies, we specified an abstract architecture for Augmented Reality systems. This allows us to compare the architectures of existing systems with respect to this abstract architecture. We applied this approach to the architecture of the ARVIKA mobile Augmented Reality system [43], which is the foundation of a "Leitprojekt" funded by the German federal ministry for eduction and research (BMBF).

System of patterns for Augmented Reality systems The developers of Augmented Reality systems select different approaches to implement the subsystems of their systems. Some recurring approaches can be found in several systems. This allows to identify the abstract concept of an approach and describe it as a design pattern. The term software pattern covers architectural patterns as well as design patterns. In this dissertation we describe a system of patterns for Augmented Reality systems. Each pattern is described abstractly and there are several ways to implement them. An Augmented Reality system can then be described as the composition of several approaches.

M^{3}ARF framework for mobile Augmented Reality maintenance systems We developed a framework for a mobile Augmented Reality maintenance system and implemented a prototype based on this framework, DWARF Pathfinder. Pathfinder is a navigation application whose main purpose was to navigate a user from the subway station Königsplatz to the rooms of the Chair for Applied Software Engineering. The main non-functional goal of Pathfinder was the reuse as many existing components as possible and integrate them over the middleware into an Augmented Reality system. We analyzed the requirements of such a system and designed an architecture based on several Augmented Reality patterns. The prototype shows the applicability of the framework.

1.9 Outline

The outline of this dissertation is as follows:

Chapter 2: Problem Space Analysis We motivate our approach with a maintenance scenario. This scenario introduces an view on Augmented Reality that goes beyond the traditional definition of Augmented Reality. We analyze the scenario and specify the requirements for an adaptable Augmented Reality framework. An outcome of this chapter is the division of the problem into several layers of concern that can be treated separately. At the end we analyze related work in the research domains we want to combine: Augmented Reality, ubiquitous computing, and wearable computing.

Chapter 3: DWARF Mobile Augmented Reality Framework This chapter covers the solution domain layer Augmented Reality. We start with an analysis of existing Augmented Reality systems known from literature and a study that was done for the ARVIKA consortium [119]. This study revealed a core set of building blocks common to each of the analyzed systems. So basing on these building blocks we developed an abstract architecture for Augmented Reality systems. For example, components such as trackers or renderers can be found in every system. Then we analyzed two selected systems formally on the base of the abstract architecture, namely the ARVIKA mobile Augmented Reality system and DWARF Pathfinder. The analysis focuses on how the abstract building blocks were implemented in each system. The approaches to implement the building blocks and their relationships can be seen similar to patterns. We describe a set of approaches we identified and described in a way similar to design patterns.

Chapter 4: DWARF contract-based peer-to-peer middleware In this chapter we describe a contract-based peer-to-peer architectural style and the DWARF middleware for adaptable Augmented Reality systems. The architectural style specifies the building blocks for mobile Augmented Reality systems. It introduces the concept of Services, Needs, Abilities, and Connectors. These concepts are implemented by a middleware that manages Services and Connectors at runtime. We close with a section about a graphical notation of DWARF-based systems based on UML 2.0 component diagrams.

Chapter 5: DWARF Pathfinder The previous chapters give the outline for the design of particular Augmented Reality systems. We describe a specific sub framework for mobile Augmented Reality maintenance systems and demonstrate the usefulness with a demonstration application called DWARF Pathfinder. It is a tour guide system that guides a tourist from a subway station to a desired room on a university campus. We describe the overall Pathfinder system, give an overview of the participating DWARF

services and describe how they cooperate. Each service is an implementation of a building block and is based on an Augmented Reality design pattern.

Chapter 6: Conclusion. In this chapter we discuss the results of this dissertation and future work.

2 Exploration of the Design Space.

A motivating scenario, design space analysis, requirements analysis, design goals, and related work.

We see Augmented Reality as an interface to a virtual world around the user. For that the registration both of real and virtual objects in the physical world is central, while the traditional definition focuses on the user interface aspects [7].

In this chapter we illustrate our vision of Augmented Reality by a descriptive scenario that represents a class of applications where a mobile worker in a ubiquitous computing environment needs on-site information. For the implementation of that vision we argue that Augmented Reality, ubiquitous computing, and wearable computing techniques must be combined. Starting from a basic scenario without Augmented Reality, wearable computing or ubiquitous computing, we extend it step by step with new technologies into an example for our vision of Augmented Reality (section 2.1).

For the requirements analysis we analyze the design space for Augmented Reality systems on the base of the characterizing attributes of Augmented Reality, ubiquitous and wearable computing. We use six attributes: type of user device and user mobility from wearable computing, network connectivity and component localization from ubiquitous computing, and location awareness and media richness from Augmented Reality. This enables us to characterize our vision of mobile Augmented Reality systems compared to systems from related fields such as pure mobile and wearable computing (section 2.2). Based on the scenario and the design space analysis we formulate the non-functional requirements and the design goals of mobile Augmented Reality maintenance systems (sections 2.3 and 2.4).

We close with an overview of related work in Augmented Reality, ubiquitous computing, and wearable computing in section 2.5.

2.1 Maintenance of Complex Systems

The focus of this dissertation is on a framework for mobile Augmented Reality systems that support workers in their daily tasks in the production and maintenance domains. As an example, we present a scenario of Augmented Reality for machine maintenance. This example is similar to the maintenance scenario of the ARVIKA project [6]. We

start the description of the scenario without Augmented Reality and enhance it step by step with several technologies.

> *A service technician has the task to maintain a group of production machines that are distributed over a larger area. When a machine fails, the technician has to go to the malfunctioning machine, locate the error and its cause, and fix it. To find the error and its cause the technician typically needs a lot of information about that particular machine. The information is provided in form of specifications, drawings, or repair guides. It is either in the technicians head, written down in paper documents in a folder or stored electronically in a database. When he needs some information, he must retrieve the papers or go to a database terminal and look it up. Either way, the technician has to stop his work for a certain time, look up the information, and go back to work.*

The introduction of *mobile computing* allows users to store information on a mobile unit. The user does not have go to a database terminal for information access but could take the information with him to the location, where he needs it. This allows information access on-site. During physical work, the access to information used to be hands-free. This can be realized with a *wearable computer* [9, 11, 36]. A wearable computer is a special mobile computer that is worn by the user like clothing and provides facilities similar to a tool chest. To achieve this, a wearable computer needs a small form factor and input and output devices that allow hands-free interaction, for example speech input and voice output, or input devices such as data gloves (e. g. 5DT from Virtual Realities [1]) or a Twiddler input device [53]. The output is mostly via an HMD where information is blended into the user's view [149] [1].

> *With a wearable computer the technician can look up any information he needs without having to leave the task he is pursuing currently. He is interacting with the system hands-free via speech-input.*
>
> *With the help of an online maintenance guide book the technician could find out, that a broken fuse caused the reported error. This book can also guide the technician step by step through the maintenance process. The system tells him to change the fuse, but to do so he needs to reach the fuse behind a covering. This means that he cannot see the fuse but has to feel his way to it. And the same, which is even more difficult, to plug in the new fuse.*

Augmented Reality systems overlay real objects with virtual objects. Each virtual object is registered with respect to the real world and has a geometric extent. Ideally,

[1] Note that is not yet Augmented Reality as Augmented Reality requires the combination of real and virtual and a registration in 3D.

the user gets the impression that the virtual objects are part of the real world. Just as real world objects can be seen from different viewing directions the virtual objects can viewed from different directions.

> *The technician is equipped with a wearable Augmented Reality system. The online maintenance guide book does not only contain a description of where to find the fuse, but also an Augmented Reality model of the fuse which is projected at the location where the real fuse behind the shielding is located. The technician's position and viewpoint in front of the machine is tracked and the system renders the fuse in the head-mounted display as if he wore x-ray glasses.*

Some of the virtual objects are representations of devices in the user's environment. Already in 1981 Marc Weiser expected that in the future hundreds and thousands of computers would be embedded in everyday's devices and enhance their capabilities. They would surround the user and provide services without being noticed as computers. Weiser called this vision *ubiquitous computing* [169].

> *Virtual arrows guide the technician to the machine the technician looks for. In order to exchange the fuse, the electronic maintenance book[2] suggests to remove a shielding in front of the fuse and automatically contacts the computer in the machine to check for hazards. When the technician tries to remove the shielding, a warning is displayed in the HMD that shows that parts behind the shielding are still under electrical power. Thanks to that warning the technician first separates the machine from the electrical power supply. Then he safely replaces the broken fuse.*

The scenario contains two points where Augmented Reality and ubiquitous computing are combined. First, the Augmented Reality system of the technician contacts trackers in the environment for current position information. And second, it contacts the control unit of the machine to check for electrical hazards. In the first place the Augmented Reality system uses the external trackers itself, in the second place the Augmented Reality system provides an interface between the control unit of the machine and the user.

When a mechanic is maintaining a machine, he usually takes a set of tools with him that are right for the intended task. Each worker tries to avoid carrying unneeded tools. This is different to wearable computers. Traditional wearable computers are monolithic and do not allow the end user to adapt their setup to the expected tasks. The transfer of the tool metaphor to wearable computing would allow to do this.

[2] An example are Interactive Electronic Technical Manuals (IETM), a standard specified by the US military for electronic maintenance and repair instruction manuals [71, 159, 157]

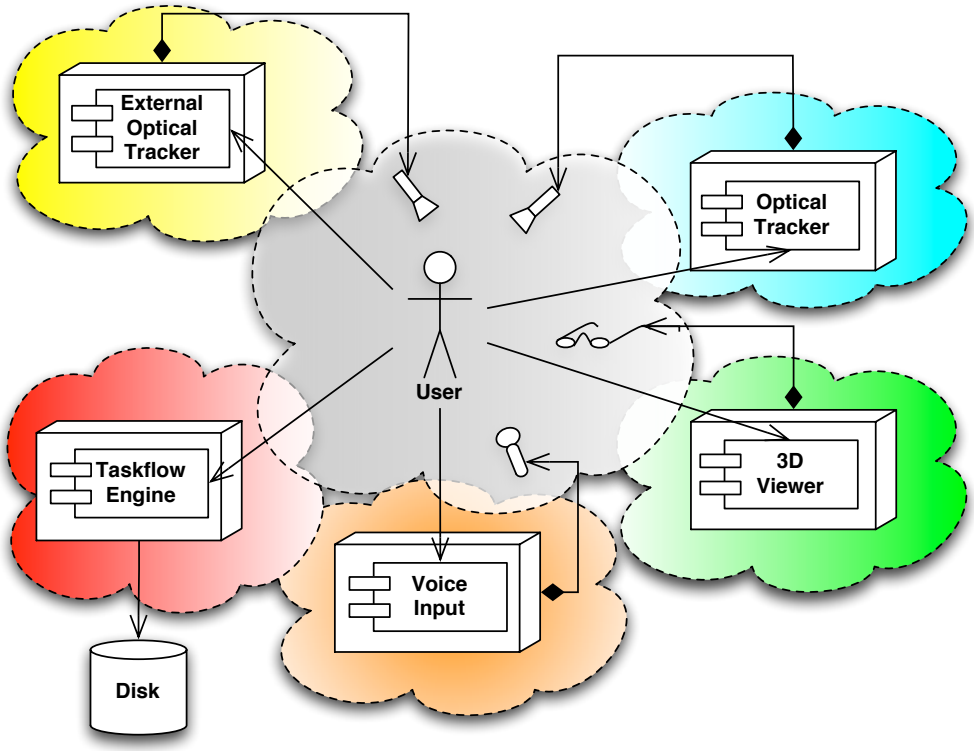


Figure 2.1: User in a ubiquitous computing environment.
Local and remote services are transparent. The services taskflow engine, optical tracker, 3D display and voice input are local services deployed on a mobile or wearable computer, a remote optical tracker services is integrated into the overall system transparently. The illustration also shows that hardware components such as head-mounted displays or video cameras are connected to the respective service which manages the access to it.

Each special purpose function of a wearable computer such as tracking for Augmented Reality could be encapsulated into an appliance, a boxed device. The appliances for different tasks could collaborate over a body-worn network. They might be called *copliance*s.

In a step further, the combination of wearable computing and ubiquitous computing allows to extend the capabilities of the wearable computer by the capabilities of appliances in the environment. Dynamically new capabilities are integrated as required to accomplish a user's task. Figure 2.1 illustrates the transparent integration of local and remote components for user tasks. The user and each component have an aura around them (indicated by the clouds around them). Where the aura of the user intersects auras of components, he can use the component.

> *Before the technician starts to go to the machine, he has a look at the work directives for the machine he should repair. He sees that he can use the Augmented Reality support and takes the tracking appliance out of the tool box. This appliance consists of a HMD with head-mounted camera and a box with an autonomous unit for the actual tracking and visualization software. He connects it to the tool belt and powers it on. After booting it connects to the body-network woven into the belt and is ready to work.*

Figure 2.1 illustrates the approach to have an appliance that serves a dedicated purpose. In the diagram, there are appliances for optical tracking, a 3D viewer, voice input, and an appliance that processes given taskflows from a database.

2.2 The Design Space for Augmented Reality Systems

In order to derive the design goals for mobile Augmented Reality systems we analyze the scenario from section 2.1 under six aspects: the type of the user device, the device mobility, the type of network access, the coupling of components, the grade of location awareness, and the richness of the user interface.

These six aspects set up a six-dimensional design space which can be used to compare the design of Augmented Reality systems with the designs for mobile systems, purely wearable systems or ubiquitous computing systems. Each dimension describes one design aspect with a discrete value.

2.2.1 User Device

The most visible item for the user is the device he is working with. The list starts with the Personal Computer (PC) and ends with connected appliances.

Stationary Personal Computer. The PC is the standard working device for desktop computing. It is used for stationary work and usually connected to a fixed network.

Notebook. Basically notebooks are PCs, the main difference between PC and notebook is that notebooks are designed for mobile use.

Personal Digital Assistant. Being smaller than notebooks, Personal Digital Assistant (PDA) are restricted in size and computing power but have a smaller form factor than notebooks. The application domains of PDAs are auxiliary services for a single user, such as date planner, email client or calculator. Although newly developed devices are getting more and more powerful, the applications on PDAs are restricted because of the small displays and compute power they have.

Wearable computer. A wearable computer is a computer that is worn by the user, either for business or for private use. There are already prototypes where the wearable computer is woven into the user's clothes [62]. The capabilities of wearable computers can be quite varying. They can be especially adapted for a particular task or powerful multi-purpose devices.

Appliance. Appliances are application specific devices, for example, calculators or translators. Hardware and software of the device are one unit that cannot be changed. This is the price for a device that is perfectly fitting for a dedicated purpose, cheaper, and simpler to use.

Connected appliances. Several appliances can be connected to cooperate in an application. Each appliance has a dedicated functionality which it contributes to the application. In short they could be named *Copliances*. This is the application of component-based software engineering to hardware-software co-design. The application of that design for wearable computing follows recommendations of the committee on electric power for the dismounted soldier in a study on electrical power requirements of the land warrior computer system of the US Army [100, p. 156]: "*... Army sponsored research should focus on developing embedded, dedicated computer systems, rather than adapting general purpose personal computers. Ideally, each sensor or subsystem should have its own processor and wireless transceiver, and user level programming should be minimized.*"

2.2.2 Device Mobility

The user mobility is related to the device type, but generally we can identify three values for mobility: *fixed*, *discrete*, and *continuously*. The particular value has significant consequences for the selection of other design attributes.

Fixed. The device is tied to a fixed location and the user cannot move with it.

Discrete. The user can move with the device but he can use it only while he stays at a discrete position (although he can be in a vehicle that moves). The reason is that either the device is portable but too obtrusive to be used while moving or that a network connection is only available at discrete locations.

Continuously. The user moves freely in a certain range and can use the device at any point in this range.

2.2.3 Network Access

The connectivity is important to describe the flexibility and quality of the network connection.

Disconnected. The device has no connection to the network and must work completely autonomously. An example is a PC without network access or an off-line mobile computer work in disconnected mode.

Fixed device. A fixed device is statically connected to one network. This is the standard configuration for workstations in a Local Area Network (LAN).

Spotty access. Mobile clients have network access at dedicated access points, so called *hotspots*. It is important to note that while the clients are mobile, the network can be accessed only at the hotspots. There is no connection handover when the user moves from one network to another. This is the mode that is supported by *Mobile IP* [114].

Roaming. A dynamic connection handover from one access point to another is provided. The consequence is that the client can be continuously online. The most prominent example are cell phone networks.

Multiple networks. There are various network types that were designed for different purposes, for example, cell phone networks (UMTS, GPRS[3]), digital broadcast networks (DAB (Digital Audio Broadcast), DVB (Digital Video Broadcast)), wireless networks (IEEE 802.11x), or body area networks (*Bluetooth*). Future user terminals will have access to several of these networks. The concrete network for a connection will be chosen on demand.

[3]General Packet Radio Service

2.2.4 Component Coupling

The combination of Augmented Reality and ubiquitous computing requires the dynamic coupling and decoupling of components on the client from components in the environment. In a first step the components have to find each other. There are several strategies beginning with a hard-wired connection up to dynamic connections based on the help of dedicated lookup services. The choice for a particular strategy depends on client mobility and the capabilities of the communication partners.

Hard-wired. This means that the connection between client and server is determined at design time and not at runtime. An example for this type are thin-client systems.

Fixed address. The communication partners use fixed addresses that uniquely identify them to each other. If the address of a communication partner changes, the configuration of the other communication partner must be updated to the new address.

Address alias. The calling partner knows the name of the communication partner and asks a *Naming Service* to look up its address. But the Naming Service is found over a well-known address that needs to be given in advance. This indirection is more flexible than fixed addresses because it allows to change the address associated with the name at the *Naming Server* instead of at every client. Examples for address lookup services include the Internet Domain Name Service (DNS) or the *CORBA Naming Service*.

Yellow pages. A *Yellow Page Server* manages information of different types. In this case a client requests information about a desired communication partner that fulfils a given number of constraints. The Yellow Page Server is found over a given name or address that the client needs to know. This type of lookup is more flexible than a simple name-based address lookup. Examples include Network Information Service (NIS), the Lightweight Directory Access Protocol (LDAP) [163] system or the *CORBA Trader Service* [106].

Broadcast. In an ad hoc network environment a system has no information about the infrastructure of the network. When it needs to communicate it first has to find out about its environment. It sends out broadcast messages that can be received by all systems that are in the same network. The same network means the same physical net, for example, the same Ethernet sub-net or the same WaveLAN cell. Any receiver of this broadcast can response to the call and accept it.

Lookup. This is a combination of Broadcast and Yellow Pages. In a first step a system sends out a broadcast for a Yellow Pages Server. Once such a server is found it is asked for the actual desired communication partner. Examples include the *Jini Lookup Service* [70] or the Service Location Protocol (SLP) [52]. This approach is typical for ad hoc environments where client systems do not have detailed knowledge about the surrounding environment [29].

2.2.5 Location Awareness

Another design criterion of mobile systems is location awareness. Four levels of location awareness can be identified:

Location unaware. The application does not know the location of the user. Location unawareness is typical for many mobile applications on notebook computers.

Two-dimensional awareness. The position of a user in a map can be described by the geographic latitude and longitude. This is sufficient for current location-aware applications, for example location dependent services that provide nearby filling stations for users in cell phone networks or a car navigation system.

Three-dimensional awareness. In addition to longitude and latitude the altitude can be important. With three geometric parameters the position can be determined. For example, Augmented Reality applications that want to visualize simple text widgets in the 3D space need the position of an object in space. This is enough when it assumes that the user looks at the widgets frontally from any direction.

Pose awareness. Augmented Reality applications need the user's position and orientation, called the *pose*. The pose is described in three values for the position and three values for the orientation. The pose is important to calculate the relative position of the user and objects in his environment. The relative position allows to adapt the rendering of objects in the HMD according to the user's current eye position and viewing direction.

2.2.6 User Interface

The richness of the user interface is another attribute to distinguish designs of Augmented Reality systems.

Text-based. Text-based interaction between user and system is sufficient for many applications. For example, the Remembrance Agent [125] is a simple application on a wearable computer that collects information from many sources such as email or

personal notes and offers information retrieval through a simple full-text index. It is implemented on the base of GNU Emacs [38].

Windows-based. In a windows-based user interface the user's screen consists of several windows which can be distributed freely across the screen. In an Augmented Reality system a 2D window can be attached to a position in the 3D space and rendered as if it was fixed to that position. An example are virtual post-its or stickies [77].

Simple 3D. The goal of simple 3D, icon-based augmentation is to present information to the user. Examples are three-dimensional arrows that show a moving direction for a navigation aid [122] or highlighted buttons on a machine [41, 137]. It should be clear at any time what is real and what is virtual. Simple 3D icons are a powerful way to give hints to the user. The icons are usually not very complex but they are rendered at the correct position. Occlusion is not considered. The user interacts with the system on the base of discrete icon objects.

Complex 3D. 3D-based augmentation with complex 3D models tries to mix real and virtual world. The goal is the perfect immersion of the user in his environment where the user is not able to distinguish between real and virtual objects. This requires advanced approaches such as considering object occlusion and light sources. Example systems include the Virtual Show Case [18], ArcheoGuide [144] or the Virtualized Environment Display Project [21].

2.3 Non-functional Requirements

We apply the system of six discriminants to the maintenance scenario in Section 2.1 and formulate the non-functional requirements for a mobile Augmented Reality system for maintenance.

User device: Connected appliances. In the scenario the technician uses a wearable computer for hands-free working with speech input, a head-mounted display for graphical output and user tracking. For input, output and tracking there are several alternatives. It should be possible for workers to add and remove devices that provide some functionality such as tracking.

In the scenario analysis we saw that an Augmented Reality system consists of several functional units. There are Augmented Reality subsystems and special purpose devices worn or carried by the user as well as devices in the environment that are dynamically connected to the system over the network. Some of these units are used only by the user and some of them are used by any user that connects to them. In our framework we bundle the hardware and the software of each system into a *module*. Each module

has a dedicated purpose in the system and should be useable like an *appliance*. The modules are combined dynamically into a *system configuration* of *connected appliances*. While a module is the physical representation of hardware and software, the logical building blocks of an Augmented Reality system are *services*.

Device mobility: Roaming users. The technician is moving across the site and is continuously guided by arrows that point in the right direction. This requires that he can take his client device, e. g. a wearable computer, with him and use it while he is moving. This is a key concept of mobile Augmented Reality. Current mobile computing is based on notebooks or PDAs, which allow users to carry the computer and use it at anywhere. Notebooks or PDAs do not support to focus on the computer and on the environment simultaneously, in particular the traffic. Mobile Augmented Reality applications must support this. The user's position must be tracked permanently and can be used for rendering of virtual objects in the user's head-mounted display.

Network access: Multiple networks. The client system continuously tries to receive tracking information and therefore it tries to get a network connection to contact the trackers.

There are several networks involved in the scenario. First, a wireless network that is available in the entire machine hall. Via that network the client system is supplied with navigational data. Second, short range networks exist for ad hoc connections between the client and devices in the environment. For example, the service worker gets information from the machine when he stands in front of the machine. This could be implemented over a short-range network such as Bluetooth [19]. Third, the client system itself consists of modules that are connected over a body network such as a wireless personal area network.

Component coupling: Lookup. In the scenario the client system has access to external position trackers. Let us assume that the technician is working in a larger site that cannot be covered by one tracker but several trackers, and each tracker covers a specific range. Then the client system has to change the external tracker it uses dynamically when the user comes into the realm of a new tracker. Also, when the user comes to the task where he should remove the shielding the client system has to connect to the machine dynamically and ask for possible hazards. In dynamic environments the setup and closing of connections between components requires a dynamic lookup of the currently available communication partners.

Location awareness: Registration in three dimensions. The mobile user is navigating through a larger area. While navigating his position is tracked by position trackers. When he arrives at the machine he sees 3D augmented objects. To be able to render objects in 3D from the correct perspective the client system must be aware

of the user's location and orientation.. In comparison, mobile computing with current location-aware services requires only the registration of the position.

Richness of user interface: Simple 3D graphics. In the scenario Augmented Reality is used to give hints and information to the user. The augmentations are simple icons such as arrows that guide the way to the machine, warnings with icons and text, and simple 3D graphics that show the position of the fuse's socket. Photo-realistic augmentation of the environment available in tourist guides such as ArcheoGuide [144] might cause accidents because it distracts the user's attention [137].

Comparison. To illustrate the differences between systems with the desired non-functional requirements and traditional stationary and mobile Augmented Reality systems we use a Kiviat graph [81]. Kiviat graphs can be used to show deviations, in for example the runtime behaviour of distributed systems with different distribution strategies. In figure 2.2 we use it to show that our approach aims towards a very flexible mobile system with simple 3D user interfaces.

Our system should be based on a flexible ad hoc component coupling over broadcasts in a multi-network environment, whereas traditional mobile systems employ at most coupling over a yellow pages service with spotty network access and stationary systems are usually coupled fixedly over a fixed network access.

Our approach targets for user-configurable mobile devices that can be used mobile. Traditional mobile Augmented Reality systems can only operate at discrete locations and are usually PDAs or wearable computers, stationary Augmented Reality systems are location fixed and usually a PC or workstation.

As all Augmented Reality systems we need the user's pose but as we target at maintenance tasks we want to use simple 3D user interfaces. This is similar to traditional mobile Augmented Reality systems, whereas stationary Augmented Reality systems mostly target at high-quality rendering and a complex 3D user interface.

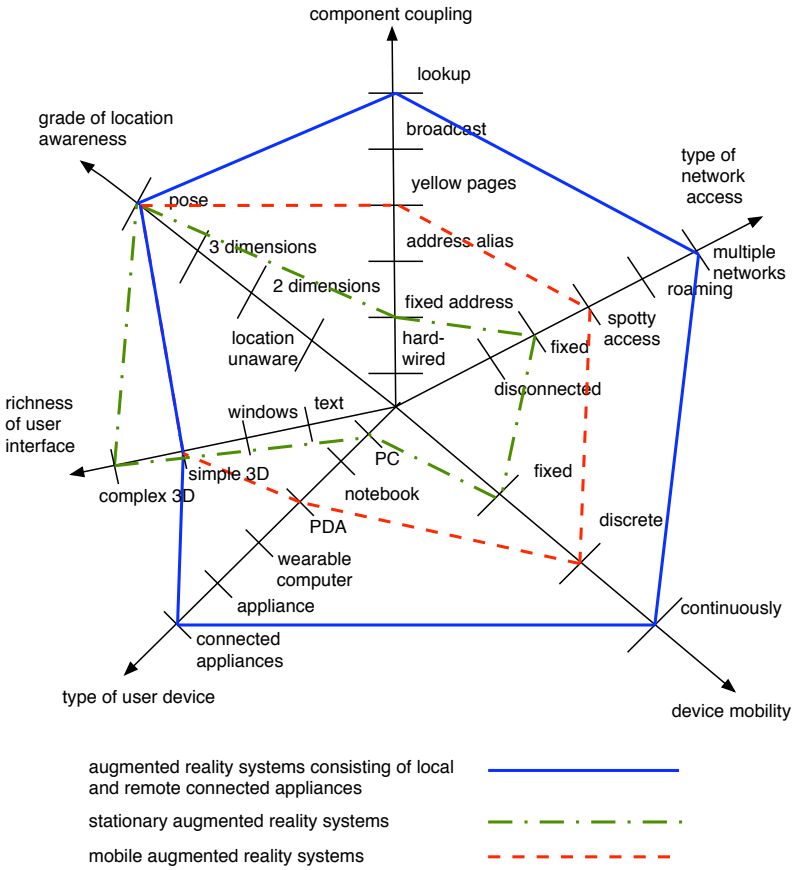


Figure 2.2: Design space for AR systems.
Different types of AR systems can be distinguished by the different instantiations of design attributes. Each dimension is directed from a common origin outwards. We placed simpler values closer to the origin and more complex values further to the outside. The graph shows examples for an AR system assembled with cooperating appliances, and traditional stationary and mobile, for example PDA-based, Augmented Reality systems. Examples for traditional stationary and mobile Augmented Reality systems are the stationary and mobile Augmented Reality systems of ARVIKA [43].

2.4 Design Goals

A recent study on software architectures for Augmented Reality systems [119] demonstrated that most existing Augmented Reality systems were developed to test and demonstrate specific Augmented Reality techniques. Only very few systems are based on a reusable framework for Augmented Reality systems such as Studierstube [145], ARVIKA [42], ImageTclAR [61], and the MR Platform [154].

Component-based software engineering [150] supports software reuse and the integration of local and external resources. *"... A component can always be considered an autonomous unit within a system or subsystem. It has one or more provided and required interfaces (potentially exposed via ports), and its internals are hidden and inaccessible other than as provided by its interfaces. Although it may be dependent on other elements in terms of interfaces that are required, a component is encapsulated and its dependencies are designed such that it can be treated as independently as possible. As a result, components and subsystems can be flexibly reused and replaced by connecting ("wiring") them together via their provided and required interfaces."* [108]. None of the frameworks for Augmented Reality mentioned above supports component-based software engineering, only reuse on the class level. Due to advantages of component-based software engineering over pure object-oriented software engineering, we chose to use component technology as the basis for the development of our Augmented Reality framework.

Ubiquitous computing services have two characteristics: First, they can be used by several users in parallel and second, the life cycle of a service is independent of its use. For example, a tracking service that covers a particular area is started at some point in time. Then it runs and waits for bypassing client systems that want to use it. Therefore we extend the component-based approach for our Augmented Reality framework to a service-based approach. Our Augmented Reality system consists of a set of distributed services instead of components.

Support of heterogeneous connections. We chose to design the envisioned Augmented Reality system as a system of communicating local and remote services. For a seamless integration the communication between both types must be treated equally. Moreover, different communication mechanisms must be supported, for example, asynchronous and synchronous communication as well as different implementations such as Common Object Request Broker Architecture (CORBA) or shared memory. The reason is that traditional communication concepts have performance penalties for multimedia systems such as Augmented Reality systems. The developer of a new Augmented Reality system should be able to select the most appropriate communication mechanism for each situation. Therefore we model a connection between services not as an association but as a first-class object, called Connector.

Connectors as first-class objects allow to separates the services of a system and the connections between them. We use this to separate system setup from system data

exchange. First, the system is set up by linking the services over a connector. After the linking process the services communicate directly over the established connection.

Connectors as first class objects are widely used for modelling distributed systems. For example, the reference model for open distributed processing RM-OPD uses *Binding Objects* for complex connections between two or more components [67], Jini from Sun Microsystems uses *Smart Proxies* [164] and several Architecture Description Languages (ADL) such as xAcme [135], Wright [4], and the Unified Modelling Language (UML) 2.0 [108] use connectors as modelling elements. Usually the connection between components is left to the developer of each component and therefore interwoven with the component.

Design by contract between services. Services usually depend on one another. A service needs support from other services (*Needs*) and in turn supports again other services through own *Abilities*. For the dynamic matching of service needs and abilities each service offers a *contract*. If the service gets the desired Needs from other services it provides own Abilities. The design of cooperating services is similar to Meyer's design-by-contract paradigm [91]. Meyer models contracts between components with 'requires' and 'assure' constructs. If the requirements are fulfiled by the caller then the component holds its assurances. In our approach we model a contract between a service and its environment. If the environment can accomplish the Needs then the service offers its Abilities. A difference to Meyer's paradigm is that his approach is used to ensure the correctness of method calls between objects, whereas Needs and Abilities are used to establish connections between components at runtime.

Loose coupling. In a dynamic environment, connections between services are only temporarily available. We assume that the service connections are transient and changing. This requires loose coupling between services and dynamic service lookup. The services specify what they need and what they offer in an abstract service description, and register themselves at the service registry. When a service needs another service it must ask the service registry for a connection to the required service.

Runtime reconfiguration by active middleware. The system developer should not be concerned with the dynamic configuration issues This should be supported by an active runtime environment or middleware that controls and manages the services and the connections among them.

2.5 Related Work

The framework described in this dissertation combines techniques from Augmented Reality, wearable computing and ubiquitous computing. In this section we give an overview of related work in these fields.

2.5.1 Augmented Reality Systems

The following discussion is based on a study on software architectures for Augmented Reality systems [119] conducted for the ARVIKA consortium [6]. In the scope of this study nearly twenty existing Augmented Reality systems, libraries, and applications were analyzed, namely ARVIKA [42], AIBAS [127], ArcheoGuide [65], AR-PDA [44], ARToolkit [17], Aura [46, 47], BARS [10], the Boeing wire bundle assembly prototype [33], EMMIE [25], ImageTclAR [61], MARS [90, 60], MR Platform [154], several prototypes from Siemens Corporate Research [138], STAR [142], Studierstube [145], Tinmith [152, 116], and UbiCom [113, 153].

We describe ARVIKA, STAR, MR Platform, and UbiCom in more detail. They are most relevant to us because they have a mobile setting or target mobile maintenance as application domain.

ARVIKA

The ARVIKA project [42] was supported by the BMBF (German Federal Ministry for Education and Research) and consisted of members from industry and research institutes. The goal was to research and develop Augmented Reality technologies that will support development, production and service of complex technical products. The project structure was application-driven with the focus on areas such as: automobile and aircraft development; production in automobile manufacture and aircraft construction; and service for large technical systems, particularly power stations and machine tools. The application-driven efforts were complemented by sub-projects for research on basic Augmented Reality technologies and a user-driven system design. The basic technologies support both high-end applications in product development and low-end mobile applications for skilled workers using belt-worn equipment in production and service environments [43].

The ARVIKA project develops two different types of systems: A stationary high-end system for lab environments and a mobile system for a web-based environment. The high-end system can be deployed on SGI IRIX, Linux and Windows platforms, whereas the mobile system runs on Windows 2000 and Windows CE for the client side and Windows and Linux for the server side. Synergy effects among the two solutions should be achieved by reusing implementations, interfaces and protocols for several subsystems, for example, the tracking subsystems [168].

Stationary high-end system The stationary high-end solution is intended for lab environments where high accuracy of tracking is mandatory.

All components of the high-end solution run on one system running SGI IRIX, Linux or Windows. On top of the operating system are the device integration interface IDEAL for local and remote devices, the AR Browser component for visualization, and the tracking components. Remote trackers can be used over the IDEAL interface.

The application-specific software runs on top of the Augmented Reality specific components. Figure 2.3 illustrates the layering for the high-end systems in a deployment diagram of the stationary system.

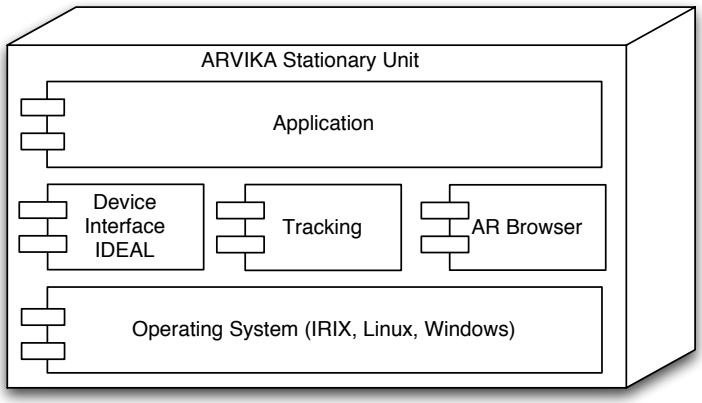


Figure 2.3: Architecture of the ARVIKA stationary solution.

IDEAL provides interfaces to connect various tracking and interaction devices. The connection is location transparent as IDEAL provides a socket-based interface, called the Low Level Device Interface (LLDI).

The high-end system runs on one physical machine, except for some tracking devices that are connected by socket interfaces. Thus, the underlying implementation uses local method calls and socket interfaces.

Mobile web-based solution Figure 2.4 illustrates the subsystem decomposition of the ARVIKA web-based system and the deployment on client and server nodes.

The focus of the ARVIKA mobile system is document access and presentation. Documents with Augmented Reality content are one type of documents among others (HTML documents, PDF documents, or CAD data). The ARVIKA system consists of a mobile client and a server, in particular a web client and a web server.

The client-side platform is targeted at mobile or wearable systems with Windows 2000 or Windows CE and provides the typical Augmented Reality functionality of position and orientation tracking and registered rendering. It has localization, tracking, graphics, and human-computer interaction techniques by several interaction devices such as a space mouse. The client environment is the Microsoft Internet Explorer 5, i.e. the client is a thin client with a web browser as the interface. The AR Browser needs access to operating system resources, e.g. access to the video camera. So the

Figure 2.4: Deployment of the ARVIKA web-based system. It is a typical three-tier architecture. The left side shows the components that run on the client side. The main components are the Microsoft Internet Explorer that controls several plug-ins and a client-side web server that acts as a proxy for the web server in the environment. The right hand side shows the web server that controls several Java Servlets. The Servlets in turn can connect to legacy systems such as external databases.

Augmented Reality subsystem (tracking and 3D visualization) is a local component with a wrapper around it to turn it into an ActiveX Control[4]. When a user loads a document that contains Augmented Reality scenes, the AR Browser is started and displays the scene registered in space.

The server handles the information management. It is based on the Apache Tomcat application server. Each component is realized as a Servlet with a HTTP address.

When a user loads a document with Augmented Reality support, the Augmented Reality subsystem is automatically loaded and the document is shown in the AR Browser.

Although it is a thin-client, some components need to be installed locally by the user. These components are Localization, Video Server, optical tracking and other tracking (Intersense, hybrid tracking, etc.), and the IDEAL device integration interface.

Following the web-based concept, different types of documents must be viewed with viewer plug-ins for the Microsoft Internet Explorer. Examples are an PDF viewer and a viewer for CAD documents.

The ARVIKA project developed components for Augmented Reality functionality such as tracking and rendering, but also several other supporting components. For Augmented Reality they developed an *AR Browser*, which is based on OpenSG and visualizes registered 3D virtual objects; a *Localization* component for the determination of the user's position (coarse grained tracking); an *Optical Tracking* component for the determination of the user's pose (fine tracking); the device integration interface *IDEAL*, which provides localization and instance independent interfaces for the integration and management of hardware devices such as trackers and input devices; and a *3D Interaction* component for user input in a 3D world, for example with a space mouse. As supporting components ARVIKA developed a *NetCollaboration* component, which builds on the AR Browser and displays annotations of the video image from a remote expert; a *Video Server* component, which delivers the video images gathered by a video camera over the network for the net collaboration; a *Context Manager*, which provides context information to client and server side components; and the *InfoService*, which acts as proxy server for server side components, for example the *InfoBroker*.

The ARVIKA server-side part was implemented with Java Servlets and Java Server Pages (JSPs) using the Java 2 Standard Edition and the Apache Tomcat servlet engine. These services are supporting services and do not belong to the core Augmented Reality system. ARVIKA put a lot of effort into such services for a better integration of Augmented Reality and information management. They developed the *InfoBroker* component, which provides a document model based on the structure of machines. The actual data are read from databases, files or third-party systems; the *Taskflow Engine* processes descriptions of an Augmented Reality supported taskflow, e.g. checking a machine; taskflows are modelled graphically with a *Taskflow Editor*; the server-side of the *NetCollaboration* component displays videos from the client-side to a remote expert,

[4] ActiveX is the Microsoft component model for browser-embedded components.

which can annotate the image; the *Annotation System* gathers user annotations and saves them; the *InfoService* component handles network issues and provides network transparency for other components; the server-side of the *Context Manager* provides context information to server-side components and collaborates with the client-side Context Manager; and the *UI Configuration*, which adapts information to be displayed to the user context, for example the device type.

The individual components are not connected directly but indirectly via the Context Manager component. It collects and distributes messages generated by the components to other interested components. This is basically a variation of the Observer pattern with a central event bus for message distribution.

The Kiviat graph in figure 2.2 shows the design attributes of the stationary and mobile ARVIKA Augmented Reality systems in comparison to the design attributes that we follow in this dissertation (section 2.4).

MR Platform

The MR Platform [154] is an environment for research and development of mixed reality and Augmented Reality and is based on the results of the Japanese Mixed Reality project. After the end of the project Canon Inc. has continued the project. The MR platform consists of a video see-through head-mounted display and a software development kit which includes libraries for registration, tracking, and video mixing. User interaction, inclusion of context knowledge, and the visualization of virtual objects are not included but left to third-parties. Additionally, the toolkit provides tools for sensor and camera calibration.

The user output subsystem of the Software Development Kit (SDK) contains rendering routines of the real world image for the video see-through HMD. For the visualization the SDK uses per default OpenGL drawing instructions but there are several computer graphics libraries such as OpenSG, OpenInventor or OpenGL that can be used instead.

The SDK is implemented as C++ class library on top of the Linux operating system and the Video4Linux image capturing library.

The MR Platform does not support higher-level concepts for the integration of remote sensors. The tracking devices, magnetic tracker and video camera, are connected through local method calls or socket connections. For the update of the sensors and the video image an update method must be called, usually within an update loop. Within each cycle, the user's position is calculated and the new image for the HMD is rendered. The processing of the current position is left to the application developer. For example, in the RV-Border Guards multi player game [111] the user position is transmitted to the state server that manages the distributed state of the game and the position of all players.

An application developed on top of the MR Platform must call the tracking-update-render loop. After each cycle the control flow returns to the application. That way

the application developer keeps the control over the control flow.

To embed the MR Platform into custom applications, a developer must manipulate the OpenGL output window, where the MR Platform renderer displays the images of the video cameras for video see-through.

STAR

STAR (Services and Training Through Augmented Reality) [142] is an EU funded project with Siemens, TU Delft, KU Leuven, University of Geneve, Realviz and EPFL. The project targets on the products for training, on-line documentation and planning purposes.

STAR develops several tools for the production of Augmented Reality content and an Augmented Reality runtime platform for the visualization of a sequence of maintenance tasks [51].

Figure 2.5 gives an overview on the system components of the STAR Augmented Reality system. STAR uses a *thin-client architectural style*. On the client, a video

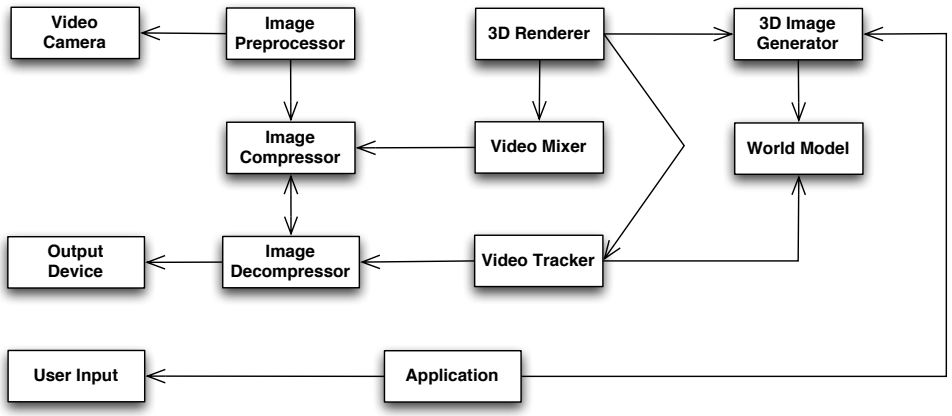


Figure 2.5: STAR system architecture

camera captures the image, the image is pre-processed (ImagePreprocessor), compressed (ImageCompressor) and transferred to the server. The server unpacks the image (ImageDecompressor), performs image processing, calculates the camera position (VideoTracker), and augments the image with virtual objects from a World Model (3DImageGenerator, 3DRenderer, and VideoMixer). The augmented image is again encoded, packed (ImageProcessor), and transferred to the client. The client unpacks (ImageDecompressor), decodes, and draws the image on an output device. User Input

is transferred to the Application, which collaborates with the 3DImageGenerator to update the shown virtual objects.

STAR executes the complete tracking process on the server. This enables thin clients with very low computing demands The disadvantage is a higher update latency because of the video transfer.

UbiCom

The Ubiquitous Communications (UbiCom) project [113, 153] is a multidisciplinary research project at Delft University of Technology. The program aims at carrying out research needed for specifying and developing mobile systems for mobile multimedia communications.

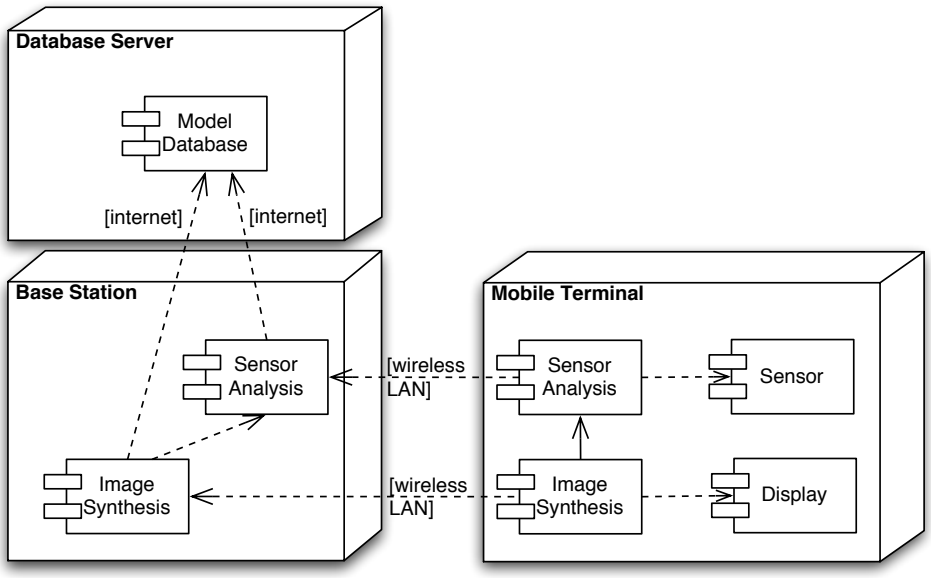


Figure 2.6: UbiCom system.
Sensors on the mobile terminal on the right hand node collect data such as video images for user tracking. The sensor data are pre-processed and then transferred to the base station. The base station conducts a further data analysis, contacts an external server for more data, for example, virtual models, and adapts them to the user's pose and generates a new image. This image is transferred to the client which mixes the generated image and the video image. The synthesized image is displayed on a HMD.

The project identified three important constraints for mobile Augmented Reality on a *ubiquitous communication system*, i.e. a mobile system which has always network access: low power on the mobile system, a system approach, and negotiated quality of service in throughput, delay, and power consumption.

Ubiquitous communication systems have several architectural properties in common. They are data driven with occasional feedback control, and they allow for local quality of service negotiations. The architecture recognizes three types of resources: communication, storage, and processing.

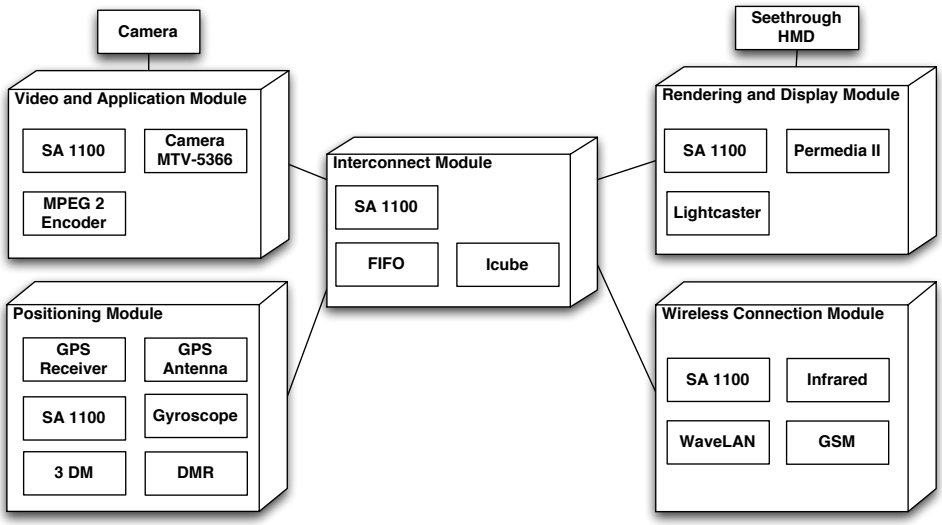


Figure 2.7: Hardware architecture of the UbiCom mobile terminal.
The mobile terminal consists of four functional modules and an interconnection module (middle node). A module is deployed on a LART board and provides a particular function for the overall system. The SA 1100 chip (StrongARM) is the core and can be found on every module. Additionally, there are function-specific building blocks such as an MTV-5366 camera decoder chip and an MPEG 2 encoder chip on each board.

UbiCom uses a control flow where the general *AR processing loop* of video gathering - image analysis - video synthesis - display is distributed on the mobile client and the background server. Figure 2.6 shows the distributed architecture. Sensors on the mobile station on the right hand side collect data such as video images for user tracking. The sensor data are pre-processed and then transferred to the base station. The base station conducts a further data analysis, contacts an external server for more

data, for example, virtual models, and adapts them to the user's pose and generates a new image. This image is transferred to the client which mixes the generated image and the video image. The synthesized image is displayed on a HMD.

The design of the mobile station uses a peer-to-peer approach where the software components are distributed over a set of several hardware units. Figure 2.7 illustrates that in a UML deployment diagram. The purpose to use a set of smaller hardware units instead of one large one are the lower resource requirements. The overall resource requirements for all UbiCom units are smaller than they were for one larger unit.

The mobile system consists of five hardware modules. Each of them has a SA 1100 computing unit to execute the Linux operating system and function specific code. The particular functions are *Positioning*, *Rendering and Display*, *Video and Application*, *Wireless Connection*, and *Interconnect*. Additionally to the SA 1100 unit each module has function specific hardware components. The Positioning module has a GPS receiver, a GPS antenna and Gyroscope for obtaining the user's position and orientation, the Video and Application module has a Camera MTV-5366 module to connect a video camera and a MPEG 2 encoder to stream video to the backbone unit (for video-based tracking). The construction of the Rendering and Display module and the Wireless Connection module is analogous. The Interconnect module serves as the data exchange backbone for the connected modules.

UbiCom distributes the user tracking over the base station and the mobile station. For video-based tracking the images are transferred to the base station. Additionally, the client uses an inertial tracker for calculating the pose and GPS for the position. Video-based rendering is only needed to correct the drift of the inertial tracker after some milliseconds. This reduces the data transfer between mobile station and base station.

2.5.2 Wearable Computing Systems

Wearable computers have been an issue of research at academia and industry for several years. Commercially available wearable computers [171, 69, 160, 32] and several research prototypes [16, 98, 48] are still constructed following the traditional design for desktop computers. Housing and electronic hardware are designed so that they can be worn at the body, for example in a belt or a vest. There is a central component that contains the CPU and controls the computer's functionality. All peripherals are connected to this central component over hardware interfaces such as USB, Firewire or proprietary connections. Special purpose hardware is connected to the central unit and the appropriate driver software is installed. Figure 2.8 illustrates this approach. For example, optical tracking requires a video camera and tracking software. The camera is connected to the wearable computer over particular interfaces and the tracking software and camera drivers are installed on the computer. A system administrator is usually needed to install software and drivers. This leads to unneeded complexity for the end users as configuration changes cannot be done by themselves.

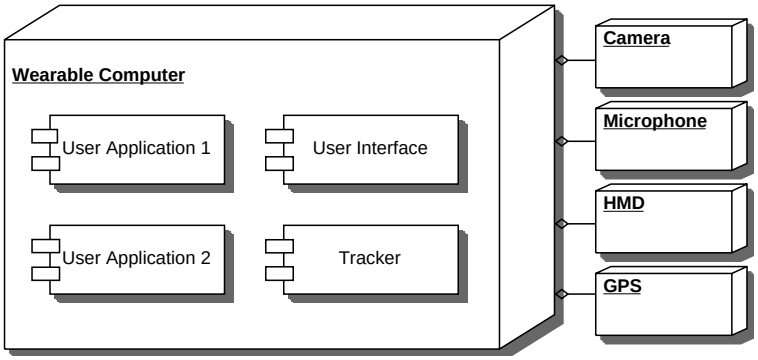


Figure 2.8: In the traditional deployment hardware and software are separated.

Other designs of wearable computers employ several modules with own micro processors connected over flexible data connections [68, 23, 26, 27]. These designs allow a more flexible deployment of the individual modules around the human body, but the architecture of the resulting computer is still similar to a traditional personal computer architecture. The single modules are assisting processors such as controllers for peripherals or network connection. They are not autonomous units with own operating system and higher-level functionality.

A wearable computer constructed from self-contained hardware components has been the goal of several research groups, examples involve the *Spot Computer* developed at Carnegie Mellon University[36], *MIThril* at MIT Media Labs[98], and the *LART (Linux Advanced Radio Terminal)* of the TU Delft[83]. All three of them have a modular concept based on the Intel StrongARM micro processor and use the Linux operating system. But they do not support the dynamic collaboration of appliances with dedicated functionality.

And these systems, except the LART do not have a comprehensive modular concept. The network topology is star shaped with a powerful central processor and several related smaller peripheral devices. These smaller devices process subtasks scheduled by the central unit.

The design of known wearable computers is basically similar to the design of workstations. The difference is that workstations are not wearable. The main disadvantages of the traditional design is the static system configuration and the nonuniform composition of components on the client system and components in the environment. There is no concept for the seamless integration of modules worn by the user and modules in the environment.

The only projects we are aware of that seems to go in a similar direction is the EU funded project 2Wear [133]. Goal of this project is to integrate input and output

devices in the user's environment for on-the-fly human machine interaction.

2.5.3 Ubiquitous Computing Systems

There are several approaches and technologies for ubiquitous computing systems today that share the idea of a space providing services for the user (GaiaOS [128, 55], Ninja [50], Aura [47], Oxygen [97], PIMA [8], or Cooltown [74]) [5]. In contrast to the decentralized approach we outlined in section2.4, most existing systems have a star-shaped architecture with central components.

Gaia OS The Gaia middleware infrastructure [128, 55] aims at supporting the development and execution of portable applications for *Active Spaces*. An Active Space is a physical space with well defined physical boundaries containing physical objects, heterogeneous networked devices, and users performing a range of activities. Gaia provides an infrastructure that supports the development of applications in such an environment and offers an abstract view as a single reactive and programmable environment.

Gaia offers support for the development of user applications in a ubiquitous computing environment by extending the MVC [82] pattern with respect to dynamic multi-input and output applications.

Oxygen The idea behind the MIT Oxygen project is that computational resources are freely available around a user as freely as oxygen. The devices that provide the computations are able to adapt themselves to the needs of the various users that want to use them. The software in such a system must be adaptable to changing users, environments, changes, and failures. The approach is to separate the description of the needed functionality from the specification of components that implement the features. Code, data objects, and specifications reside in a persistent data store.

The Oxygen software architecture is based on distributed objects. The components can be customized and they are replaceable. Oxygen uses an own component model called Pebbles [167]. Pebbles are described by interface specifications, informal descriptions, and potentially useful information that allows a location service, called Goals [166], to select a particular pebble needed for particular task.

Ninja The Ninja project aims to develop a software infrastructure to support the next generation of Internet-based applications. Central to the Ninja approach is the concept of a service, an Internet-accessible application. Key concepts of Ninja include: structured partitioning of state, *operators* and *paths*, automatic service composition, and mobile code for service deployment [50]. The Ninja architecture consists

[5] for a more complete list see `http://devius.cs.uiuc.edu/gaia/html/links.htm`

of *bases* (powerful workstation cluster environments), *units* (client devices), *active proxies* (adaptation units), and *paths* (connect units, services, and active proxies).

Aura Project Aura from Carnegie Mellon University is intended for pervasive computing environments involving wireless communication, wearable computing, and smart spaces. To accomplish the goals, Aura addresses every system level: hardware, operating system, application, and end users preferences. It applies to general concepts: *proactivity* and *self-tuning* [47].

To adapt to changing runtime conditions Aura has a task layer called *Prism* that represents user intent as a coalition of abstract services, configures tasks, and monitors and adapts resources. It consists of a Task Manager, a Context Observer, and a Environment Manager that implement these capabilities.

The existing approaches are quite different in detail, but each addresses the problems of component control, lookup and selection.

Component control. To set up a collaboration of services at runtime it must be possible to get information on them and control them. These requirements are met by component- and service-based systems such as the CORBA Component Model (CCM)[104], Microsoft's Component Object Model (COM+) [94] or Sun's Enterprise JavaBeans (EJB) [148] (based on the Java Beans component model [146]). The control is provided by the runtime environment and is needed to start/stop and modify components on demand, for example instantiate a new filter service for tracking. Web services extend this to web-based access methods [95, 147]. Oxygen[97] uses its own component model called *Pebbles* [167]. Aura [47] provides a software layer called *Prism* that monitors and controls components in order to adapt a system for user tasks to changing conditions. In Cooltown [74] persons, things, and places are connected via web services, the configuration is left to the service administrator. Most systems have an own component model that describes what attributes of components are well-defined functional interfaces, explicit context dependency, introspection, interfaces for the controlling runtime environment, usually the *component container*. A component container is an environment for components that provides services and controls them.

Component lookup and selection. In dynamic systems it is not enough to have a naming service that translates identifiers to references such as the CORBA Naming Service. To bind a component at runtime to a formerly unknown component, there must be an abstract description, and a lookup service must search for an implementation. Examples are CORBA with the CORBA Trader Service [106], Jini Lookup Service [164], UPnP [93], JXTA [49], or IETF's SLP [52]. These systems use simple attribute-value or a constraint-based language to formulate lookup. There are recent efforts to reuse knowledge management techniques to transport more semantic information, for example the Semantic Web research [54] or the W3C Web Services

Choreography Working Group [5, 151, 84]. In Oxygen Pebbles find each other by the *Goals* lookup service [166], Gaia provides a Space Repository component that stores information about available resources in the user's space.

Component connection. Current middleware systems such as CORBA IIOP, Microsoft DCOM, Java RMI or SOAP do not support heterogeneous connection types between components. More recent approaches try to make the middleware adaptable [165, 129, 37, 105, 30] to the connection requirements. Ninja [50] separates nodes from connections and enables the automatic construction of paths from node to node via several other nodes.

2.6 Conclusion

In this chapter we gave a motivating scenario of Augmented Reality supported maintenance of complex systems. We analyzed the design space, the non-functional requirements of such an Augmented Reality system and derived the design goals for a reusable software framework for mobile Augmented Reality systems in ubiquitous computing environments. Finally we gave an overview of related work in Augmented Reality, wearable computing, ubiquitous computing.

3 Reference Architecture and Design Patterns for Augmented Reality.

A reference architecture and design patterns for Augmented Reality systems.

This chapter covers the solution domain layer, the second layer of the four layer architectures abstraction framework laid out in section 1.7. The solution domain is mobile Augmented Reality.

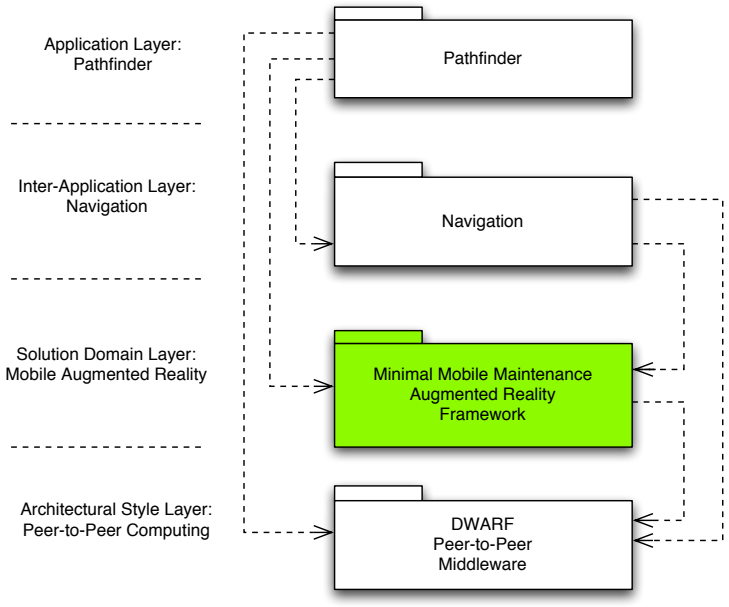
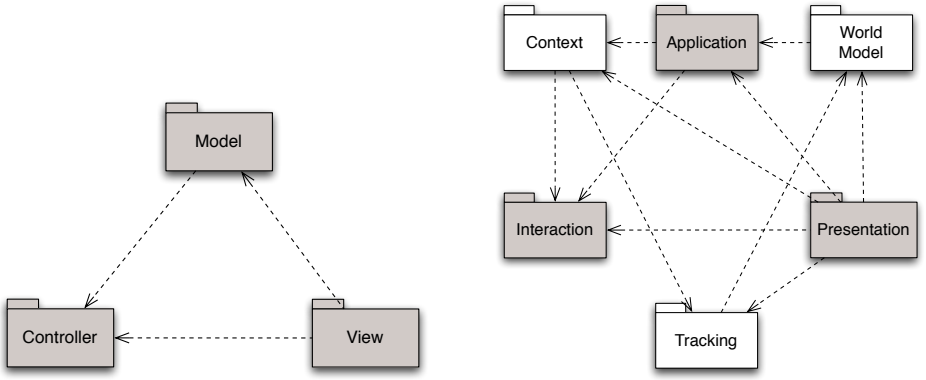


Figure 3.1: Chapter solution domain layer

Augmented Reality systems share a common basic architectural structure. In addi-

tion, many basic components and subsystems can be found in many different systems, e.g. various trackers or a scene graph. This is not surprising as all Augmented Reality systems are interactive systems and the core functionality of Augmented Reality is the same for all systems: tracking the user's position, mixing real and virtual objects, and processing and reacting on context changes and user interactions.

This allows us to specify a descriptive reference model for Augmented Reality systems (section 3.1). With this reference model we can analyze and describe existing Augmented Reality systems by mapping them to the reference architecture, (section 3.1.8). As an example we present a mapping of the ARVIKA mobile Augmented Reality system onto the reference architecture.



(a) Model-View-Controller-Pattern [82] as dependency graph

(b) Model-View-Controller as Application-Presentation-Control plus Augmented Reality specific subsystems for Tracking, World Model and Context

Figure 3.2: Extension of the MVC pattern with Augmented Reality specific subsystems

The structure that we use to model Augmented Reality systems is similar to the model-view-controller (MVC) pattern [82]. Figure 3.2(a) shows this pattern and the dependencies among the subsystems. The MVC pattern separates interactive systems into subsystems for the Model, Views of the Model, and a Controller for the data flow. The Model encapsulates application data, provides access and manipulation methods, and processes input from the controller; the View represents user information, updates on changes in the model, and creates the Controller; the Controller is related to the View, provides user input methods, forwards issues to the Model, and initiates changes in the View. To conform to terms used in HCI research we call the View *Presentation*, and the Controller *Interaction*. We extend this separation with specific extensions for

Augmented Reality and ubiquitous computing, in particular abstractions for *Tracking*, a *World Model*, and *Context*. To distinguish between the MVC Model and the World Model we use the term *Application* instead of MVC Model (section 3.1). We illustrate this in figure 3.2(b).

The similarity of our structure and the MVC pattern is not surprising, as AR systems are interactive systems by definition [7].

In section 3.2 we show that different Augmented Reality systems use different approaches to implement the subsystems of the reference architecture. We identify a set of different approaches and describe them as patterns (based on the description of design patterns in [45, 136]). The result is a catalogue of patterns that can be used to implement Augmented Reality subsystems. Every Augmented Reality system has additional components but the core of each Augmented Reality system can be implemented by a combination of these patterns.

3.1 An Augmented Reality Reference Model

We decompose an Augmented Reality system into six core subsystems. Each subsystem provides a particular functionality for the whole system. These subsystems refer to the three subsystems of the MVC pattern plus additional three that are specific for Augmented Reality systems.

Application subsystem. An important issue of frameworks is the integration of application specific functionality. The abstract Application subsystem is a placeholder for all application specific code. This subsystem complies with the model subsystem of the MVC pattern (section 3.1.2)

Interaction subsystem. The Interaction subsystem gathers and processes any input that the user makes deliberately. We distinguish it from other input such as by moving and changing the position. This subsystem complies to the controller subsystem of the MVC pattern (section 3.1.3).

Presentation subsystem. The Presentation system displays system output for the user. Besides 3D augmentation this also supports other media such as 2D text or speech. This subsystem complies to the MVC view subsystem (section 3.1.4).

Tracking subsystem. Tracking the user's pose is a key functionality of Augmented Reality systems. The tracking subsystem is responsible for tracking the user's pose and update depending subsystems such as the presentation subsystem. Sensor input in general is part of the Controller subsystem in the MVC pattern, but because tracking is of prominent interest in Augmented Reality systems we decided to partition the Controller into a Tracking subsystem for position and pose information and subsystems for the other information (section 3.1.5).

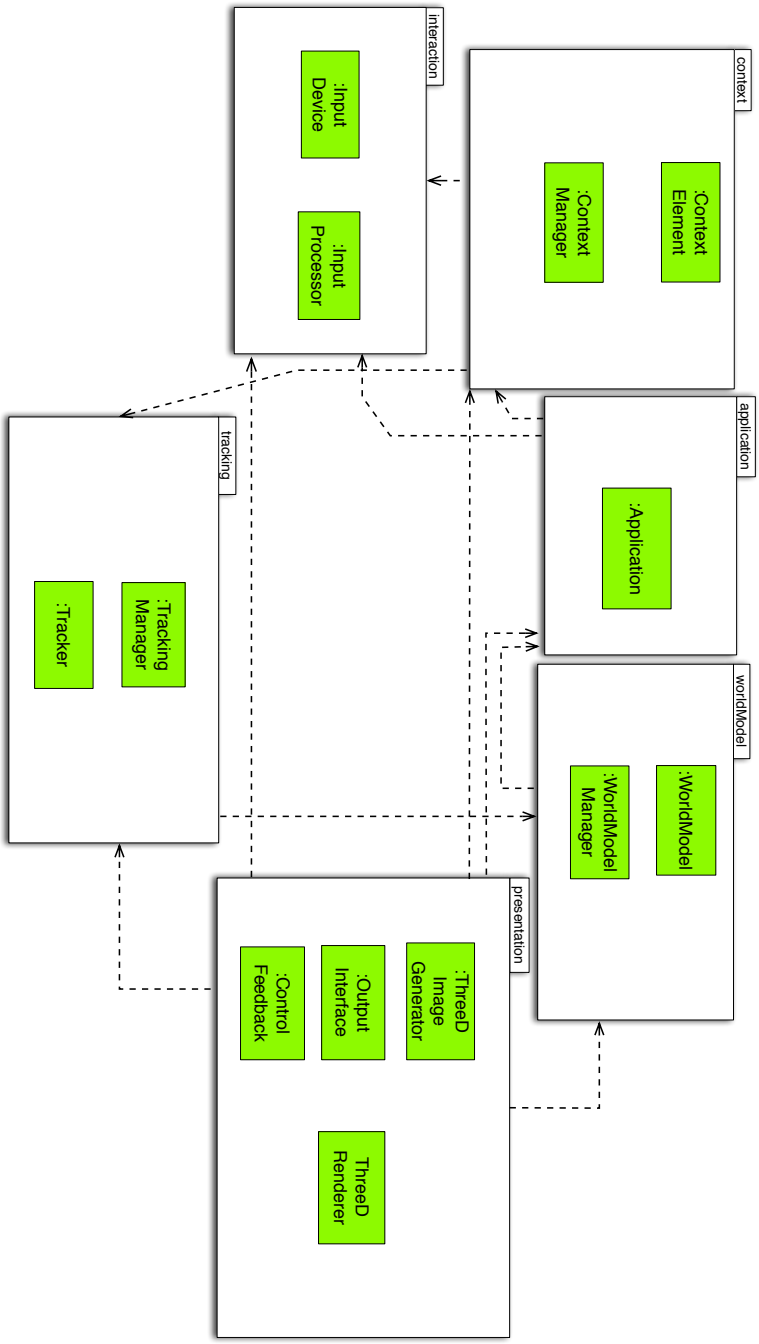


Figure 3.3: Subsystem decomposition of the reference model.

Context subsystem. The Context subsystem collects different types of context data and makes it available to other subsystems. Examples include user preferences and the current user task. The context is a similar case as tracking. In MVC it would belong to the Controller subsystem. Similar to tracking in Augmented Reality systems it has a special relevance for ubiquitous computing systems and we decompose it into an own subsystem. The user's pose is of course part of the user's context. But again, tracking is very important for Augmented Reality, so we partition the Context subsystem into a Context subsystem and a Tracking subsystem. The pose belongs to the Tracking subsystem, and the remaining context data belong to the Context subsystem. We model context not as part of the Application subsystem because context is relevant for several subsystems besides the application, for example the Presentation subsystem. The handling of context changes is specific for the components of each subsystem, in particular the application (section 3.1.6).

World model subsystem. In Augmented Reality the user moves in the real world and obtains information linked to real world objects or user positions. Information about the world are stored in a world model. In the MVC schema this subsystem would be part of the model. Similar to tracking, the world model is of a particular interest for Augmented Reality systems design, so we partition the MVC Model into a two parts with an own World Model subsystem for geometric models (section 3.1.7).

Each subsystem consists of several classes. Figure 3.3 gives an overview of the identified subsystems and the main classes for each of them in a UML class diagram [108]. Subsystems are shown as UML packages which contain classes. We chose to model them as packages to express a collaboration between the classes within a package. Each class in turn may be realized by other classes. There are dependency relationships between the subsystems, illustrated with dashed lines. A dependency shows that a subsystem relies on interfaces of another subsystem.

3.1.1 Overview

Figure 3.3 gives an overview of the subsystems and the dependencies between them. Now, figure 3.4 shows refinements of the classes by derived classes, aggregations and associations between classes. The connections between classes are annotated with labels that show what type of data is exchanged. To keep the complexity of the diagram reasonable, we left out some details and swapped them out to detailed diagrams of the individual subsystems.

We analyze each subsystem and model it with a class diagram. The models describe which classes appear in each subsystem, what their task is and how they relate to each other. The classes themselves are treated as functional black boxes.

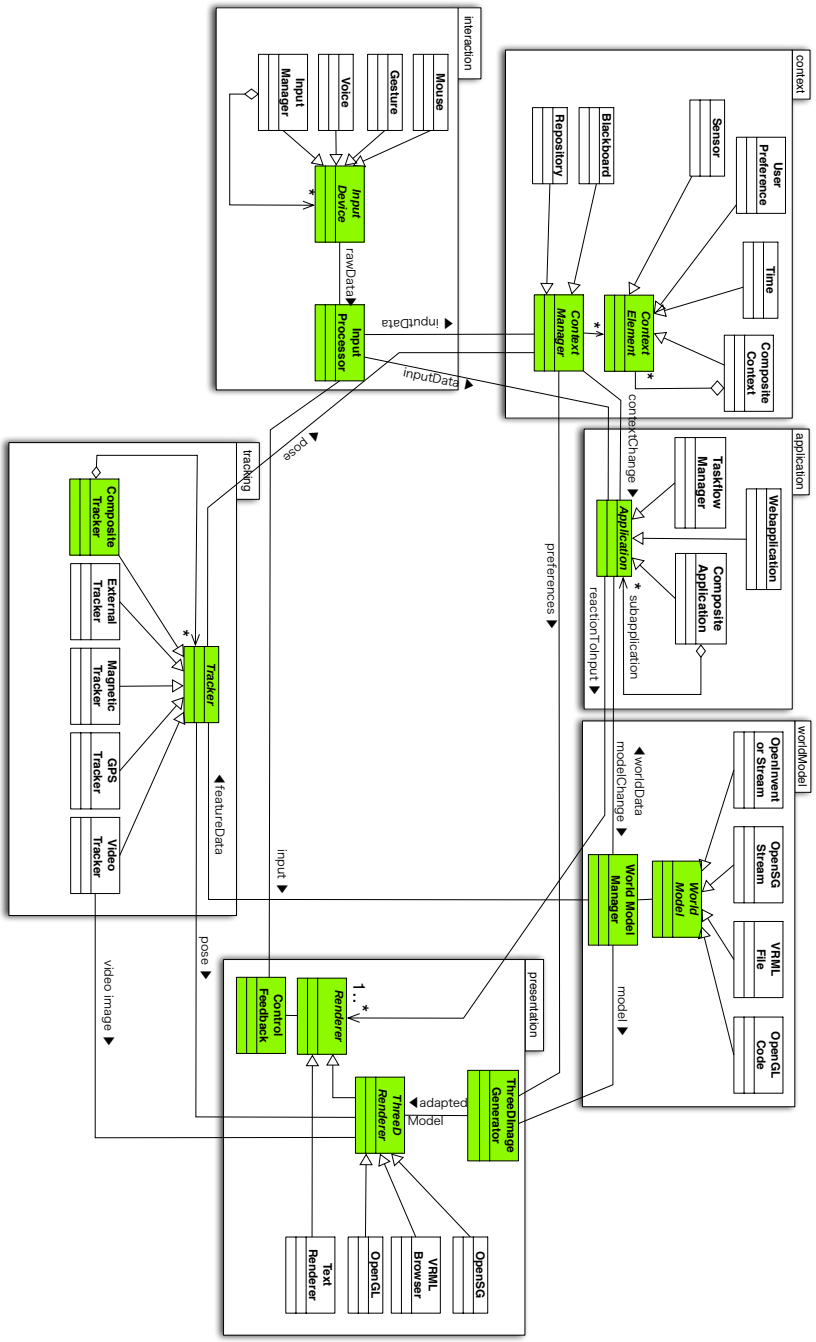


Figure 3.4: Abstract Augmented Reality architecture.

Each of the six subsystems is explained in the sections 3.1.2–3.1.7

Note that we kept the layout of the subsystem diagram in figure 3.3. We will keep the same layout when we describe the existing Augmented Reality systems, to show what subsystems and classes of the reference model they implement.

3.1.2 Application Subsystem

The Application subsystem encapsulates application functionality and data. It consists of several sub-applications and classes that also provide user functionality. Figure 3.5 illustrates the principal structure of that subsystem. This subsystem is not specific to Augmented Reality, but communicates with other subsystems, particularly the Tracking and Rendering subsystems. The structure of the Application subsystem follows the Composite pattern.

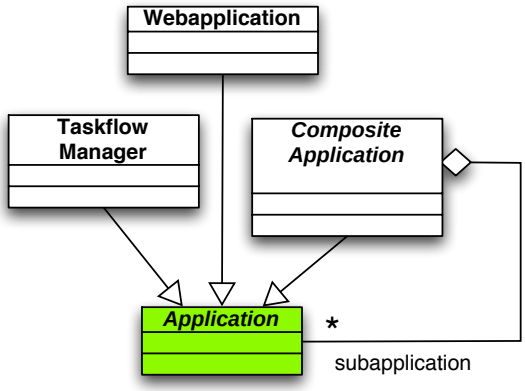


Figure 3.5: Application subsystem.
The Application subsystem is represented by the abstract Application class. A particular application can be implemented in several ways. Examples include Webapplication and Taskflow Manager. Also, a Composite Application can consist of sub-applications.

Figure 3.5 shows the refinement of the Application subsystem. The abstract Application class can be refined in several ways. Examples include Webapplication and Taskflow Manager as in the ARVIKA system [42].

The Application subsystem corresponds to the Model subsystem of the MVC pattern, so the Application controls the functionality of the Augmented Reality system. To get information about the world outside the user data are read from the World Model. After start-up it subscribes to context changes (from the Context subsystem), user interactions (from the Interaction subsystem), and changes in the world model

(from the World Model subsystem) and reacts on them. This corresponds to the Observe Pattern described by Gamma et al [45]. It processes the input, and sends commands to the Presentation subsystem as reaction to the input and to the World Model subsystem for changes in the model.

3.1.3 Interaction Subsystem

User input is any input that the user issues to control the system. Therefore input through the Tracking subsystem is not taken into account. Figure 3.6 illustrates the structure of this subsystem. The architecture of this subsystem is based on the Perception subsystem from a conceptual architecture for situation-aware interactive assistance systems developed by Kirste and Rapp [75]. It follows the pipes-and-filter architectural pattern [24, pp 54]. Raw input from input devices is processed through a pipeline into a recognized input token. The token is then forwarded to higher level classes such as the Application.

User input can be achieved through several input devices, e.g. Mouse, Gestures, Voice or combinations thereof through an Input Manager class. The raw input data is forwarded to an Input Processor that uses Data Analysis and Modality Fusion classes to interpret the raw input data. Context Information can be used for the data interpretation. The interpreted input is then forwarded to the Application and the Context Manager class. As a shortcut for instant user feedback, the recognized input can be forwarded to the Presentation subsystem, particularly to a Control Feedback class that visualizes the input, e.g. mouse input as movement of the mouse pointer.

3.1.4 Presentation Subsystem

The Presentation subsystem corresponds to the MVC View and models the human-computer interface. User output can be achieved through several modalities, such as 2D and 3D graphics, text, voice or sound. For Augmented Reality the graphical presentation is the most important, but there are others such as audio. Other output interfaces can be used simultaneously.

In general, the Presentation subsystem is setup as a pipes-and-filter system. Raw graphical data from the World Model are processed in a pipeline of several classes into a registered image that is displayed on a HMD or computer screen. The filters use additional data such as the pose. Figure 3.7 shows this pipeline.

Figure 3.8 illustrates the Presentation subsystem. Core of this subsystem is the ThreeDRenderer class, a refinement of the abstract Output Interface class. It renders the virtual objects that augment the user's perception in the correct distance, position and orientation based on the user's pose. The virtual objects are created by the ThreeDImageGenerator. It reads models from the World Model and adapts them from the World Model according to context information from the Context Manager, e.g. the required level of detail. The user's pose comes from the Tracker. For rendering in 3D

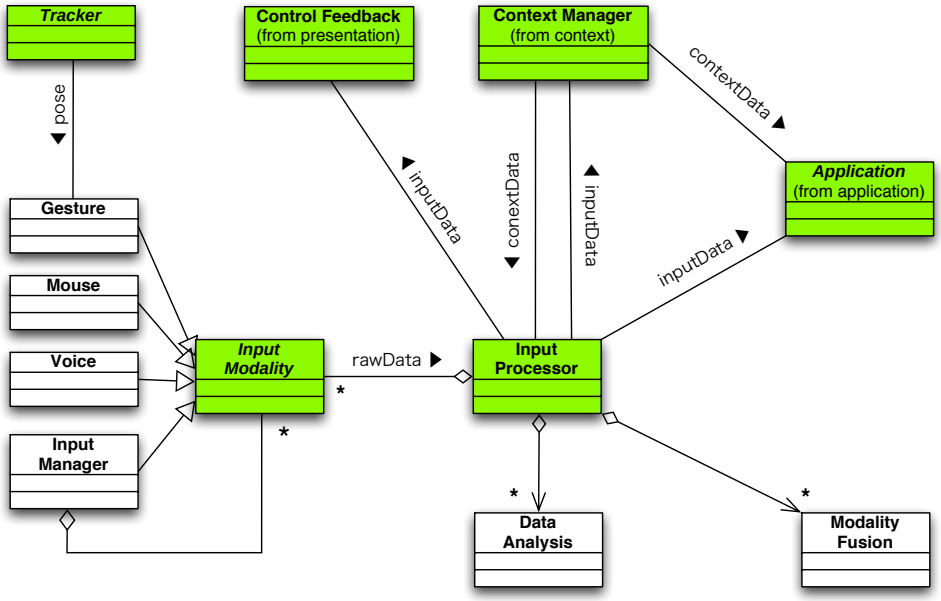


Figure 3.6: Interaction subsystem.
Input data are created by an Input Device (for example Mouse, Gesture, Voice or combinations thereof from an Input Manager that combines several input devices to an abstract input device). An Input Processor further processes the input data with Data Analysis and Modality Fusion and forwards it to the Application, the Context Manager and the Control Feedback class. The Control Feedback class is part of the Presentation subsystem and visualizes the recognized input.

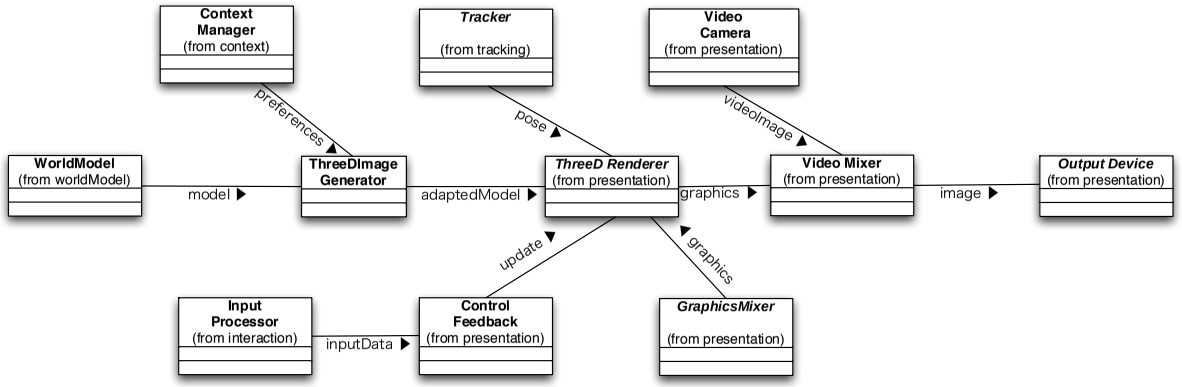


Figure 3.7: Presentation pipeline.
The presentation subsystem is set up as a pipes-and-filter system. The ThreeDImageGenerator reads the model from the WorldModel and adapts it according to preferences from the ContextManager. Such preferences might be level of detail or possible resolution of the output device. The adapted model is handed over to the ThreeDRenderer, which combines it with data from the ControlFeedback and the GraphicsMixer. The combined model is adapted to the current pose from the Tracker and forwarded to the VideoMixer. The VideoMixer combines the image of a live video stream from a video camera and generated image and sends it as the final output to an OutputDevice, which might be a head-mounted display or a monitor.

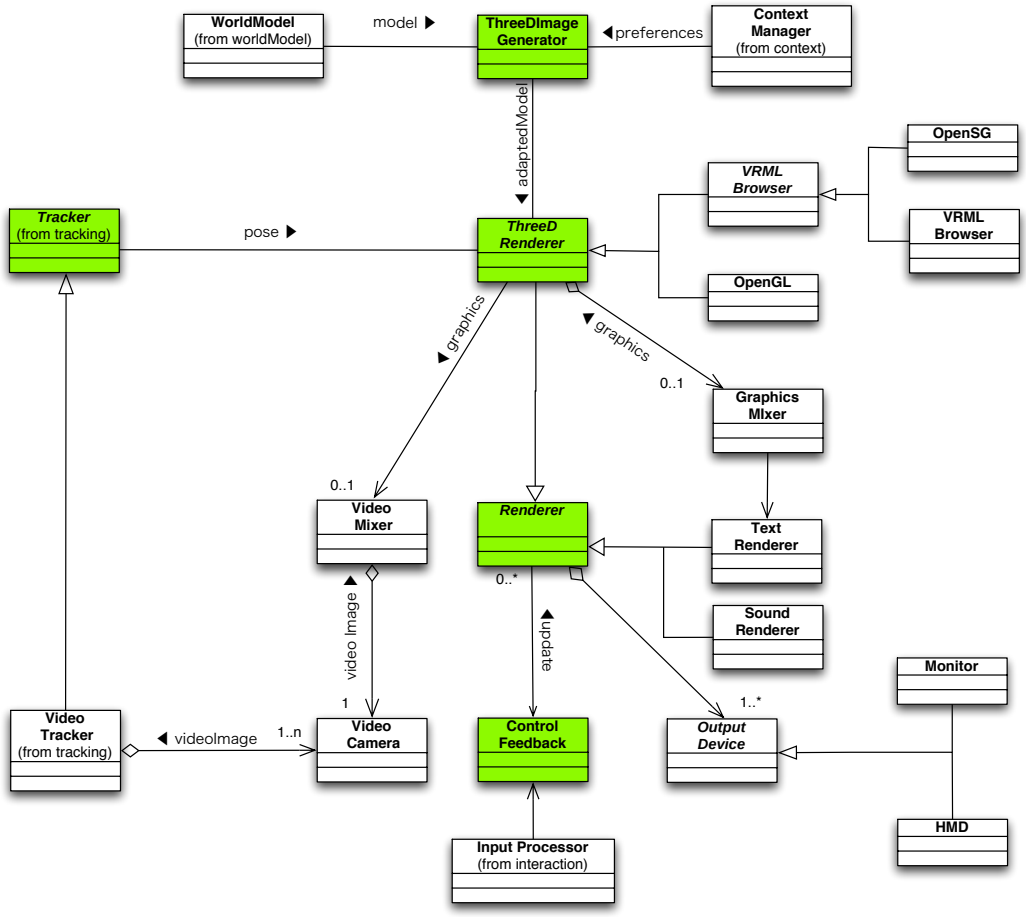


Figure 3.8: The core of the Presentation subsystem is the ThreeDRenderer, an Output Interface for 3D graphics. Images are generated by a ThreeDImageGenerator, which adapts virtual objects from the World Model to the context. The ThreeDRenderer can use a Graphics Mixer to embed non-3D data into the scene. Trackers send the pose to the Three-D Renderer to enable an update of the view. For video see-through a Video Mixer overlays images from the Video Camera and the images of the ThreeDRenderer. For optical see-through, this class is not needed. In both cases, the rendered images are emitted by an Output Device. This can be a head-mounted display (HMD) or a computer screen.

usually a Scene Graph is used. There are several possible concrete 3D renderer libraries available such as OpenSG Graph [110], OpenInventor [143], a VRML browser [66] or OpenGL [170]. If non-graphical data should be embedded into the 3D scene such as textual data from a Text Renderer, a Graphics Mixer combines graphics and non-graphics into one representation.

The Control Feedback class is part of the Output Interface. It displays user interactions without interpretation. This is analogous to mouse or keyboard input in a window system. The operating system tells the Output Interface, for example a window system, to update the mouse position or to display a new character. This is done independently from any further processing by the application.

For optical see-through Augmented Reality the graphic is emitted by an Output Device, for video see-through Augmented Reality the output data is then forwarded to a Video Mixer, which mixes the videos images from a camera with the virtual images. The mixed images are then emitted by the Output Device. An Output Device can be a head-mounted display or a computer screen.

3.1.5 Tracking Subsystem

Figure 3.9 shows a class diagram of the Tracking subsystem. Tracking is achieved by processing data from a sensor, for example a video camera. So tracking might be seen as part of the Interaction subsystem (which complies to the MVC controller) or the Context subsystem. We want to treat tracking as a separate subsystem for the following reasons: First, as said before, tracking is important in Augmented Reality systems. Indeed, many of existing Augmented Reality system have only tracking sensors and no other ones. Second, we want to separate deliberate user input via a control device and undeliberate input via a sensor. Changing the pose through the natural human movement needs not to be treated by the MVC model as it does not change it. Third, the pose is part of the user's context, but a very important one. Many sensors can only be used to calculate the pose. So we decided to create an own subsystem for tracking which collaborates with the Context subsystem, the Interaction subsystem, and the Presentation subsystem.

Tracking can be achieved by several techniques. For example, there are video-based trackers, magnetic and inertial trackers, GPS, external trackers from the environment, or combinations thereof, i.e. hybrid trackers. The diagram shows only some examples of them. A Tracking Manager can be used to combine the input from several trackers to higher level tracking data.

The trackers can use the World Model Manager from the World Model subsystem to get information about the environment they are working in. For example, for feature-based optical tracking the Video Tracker needs information about the features it should look for in the video images. The result of the tracking, the pose, is forwarded to the ThreeDRenderer that updates the visualization, the Context Manager as one part of the user's context, and the tracking-based user interaction. For example, user input

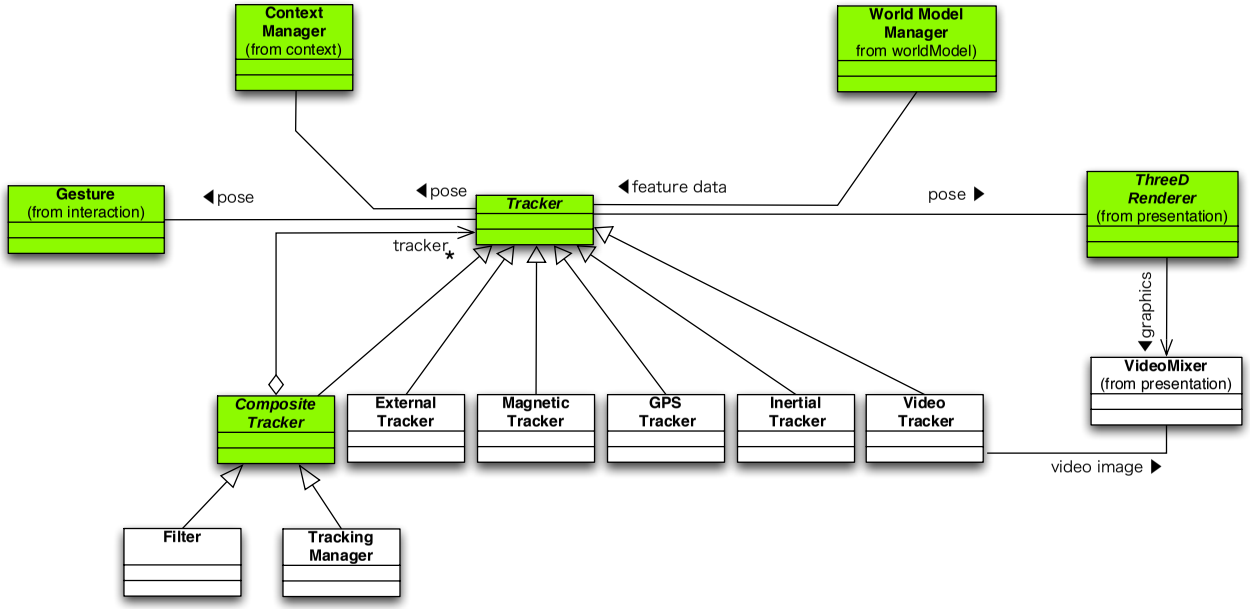


Figure 3.9: The Tracking subsystem processes sensor data from tracking devices and calculates the pose. There are several types of trackers such as Video Tracker, Magnetic Tracker, Inertial Tracker, External Tracker, GPS Tracker, or a Tracking Manager. The latter combines tracking data from several simple trackers to improve tracking accuracy. For the initialization the trackers need information from the World Model. The result of the tracking process, the pose, is forwarded to the ThreeDRenderer that updates the visualization, the Context Manager for use in other subsystems, and the Gesture class.

through hand gestures requires hand-tracking.

As a special tracker the Video Tracker can be shortcut with the Video Mixer from the Presentation subsystem for video see-through Augmented Reality where both must share the video image.

3.1.6 Context Subsystem

Modelling the user's context is still a field of research. We use the following definitions for the terms *context* and *context-aware* defined by Dey for the Context Toolkit:

Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves. [34, pp 22]

A system is context-aware if it uses context to provide relevant information services to the user, where relevancy depends on the user's task. [34, pp 23]

There are several facets of user context. For different applications, different parts of the user's context are needed. Therefore we model the Context subsystem as a system with a Context Manager that collects different types of context information which we call Context Elements. Examples are User Preferences, Sensor Data, the current Time, the Pose, Resource Information such as the type of user device, Domain Knowledge or the user's current Task. Context Elements can be combined to Composite Context Elements.

The Context Manager stores this information and makes it accessible to other classes. Context Processors may read context information, process it and generate new context information. Examples for Context Managers are Blackboards [31] and Repositories [140]. The Context Manager collaborates with the Application, the ThreeDImageGenerator, the Tracker, and the Input Processor.

Figure 3.10 illustrates the Context subsystem.

3.1.7 World Model Subsystem

The World Model subsystem (figure 3.11) stores and provides information about the world around the user. It is user and application independent. The typical structure for a world model is a scene which is often represented by a tree. The tree scheme is usually relaxed to a graph in order to express relationship between objects on different hierarchy levels, for examples in scene graphs. A scene graph is a hierarchical object-oriented data structure for graphical objects, used for example in OpenInventor [143]. This is an application of the Composite design pattern.

A scene consist of Virtual Objects, Real Objects (representations of real world objects), Features (feature information for trackers), and sub-scenes (see right-hand side of figure 3.11). Virtual Objects, Real Objects and Features have several attributes

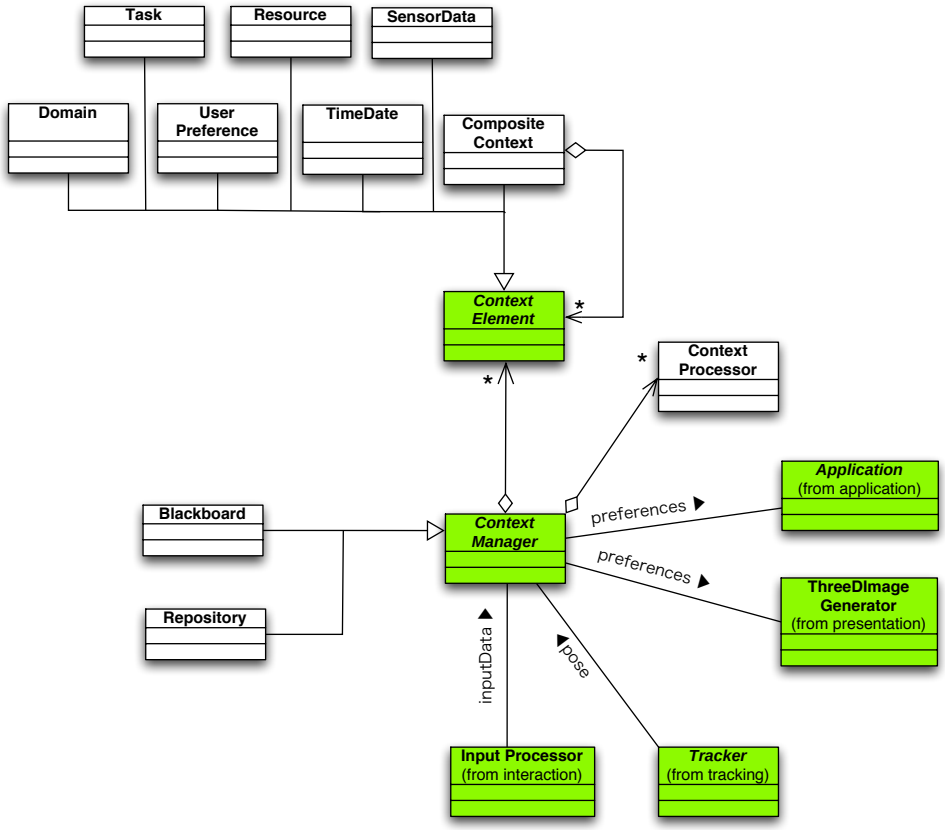


Figure 3.10: Core of the Context subsystem is the Context Manager. It collects different Context Elements, e.g. User Preferences, Sensor Data, the current Time, Tracking Data, Resource Information, Domain Knowledge or the user's current Task. The Context Manager stores this information and makes it accessible for other classes. Context Processors may read context information, process it and generate new context information.

that describe their graphical representation and other relevant information. A common important attribute of these classes is the Position relative to a common World Coordinate System specified by the owning World Model.

Existing Augmented Reality systems use rather simple implementations of world models. Usually they are file-based using various formats such as VRML [66] files, OpenGL [170] classes, and OpenInventor [143] or OpenSG streams. VRML is mostly used as there are many tools that create 3D models in VRML format and nearly every other library or tool has import functions for VRML. Some more recent approaches use a database for saving world model data [124, 59].

At runtime a World Model Manager controls the access to the World Model. The ThreeDRenderer gets information about the graphical representation of real and virtual objects for rendering and the Trackers get information about the Features they have to look for from the World Model Manager. World Model objects are not only graphical objects but representations of any object that have a spatial position. The Application can access the World Model to read and change the non-graphical data of these objects (see figure 3.11).

3.1.8 Mapping of the ARVIKA System onto the Reference Architecture

As an example we map the software architecture of the ARVIKA mobile Augmented Reality system that we described in section 2.5.1 onto the reference architecture. First we analyze to which architectural layer each ARVIKA component belongs to. This reveals which components are used for Augmented Reality functionality and which are application specific.

The ARVIKA components can be classified into the four abstraction layers as follows:

Application layer. The ARVIKA project consists of several application-driven sub-projects for production, development, and maintenance. Each sub-project developed its own specific application components.

Inter-application layer. The InfoBroker, Workflow Engine, NetCollaboration, and Annotation System are specific to the application domains maintenance and service.

Solution domain layer. Reusable components for Augmented Reality systems are Localization for the determination of the current user position, the Context Manager for the access of context information for different components, UI Configuration for the adaptation of the user interface to different devices and users, the Device Integration Interface IDEAL for the integration of different devices, mainly sensors, the Video Server to distribute the video image to remote components, Tracking for pose tracking, and 3DInteraction to connect various interaction devices. These components are not specific for a particular application

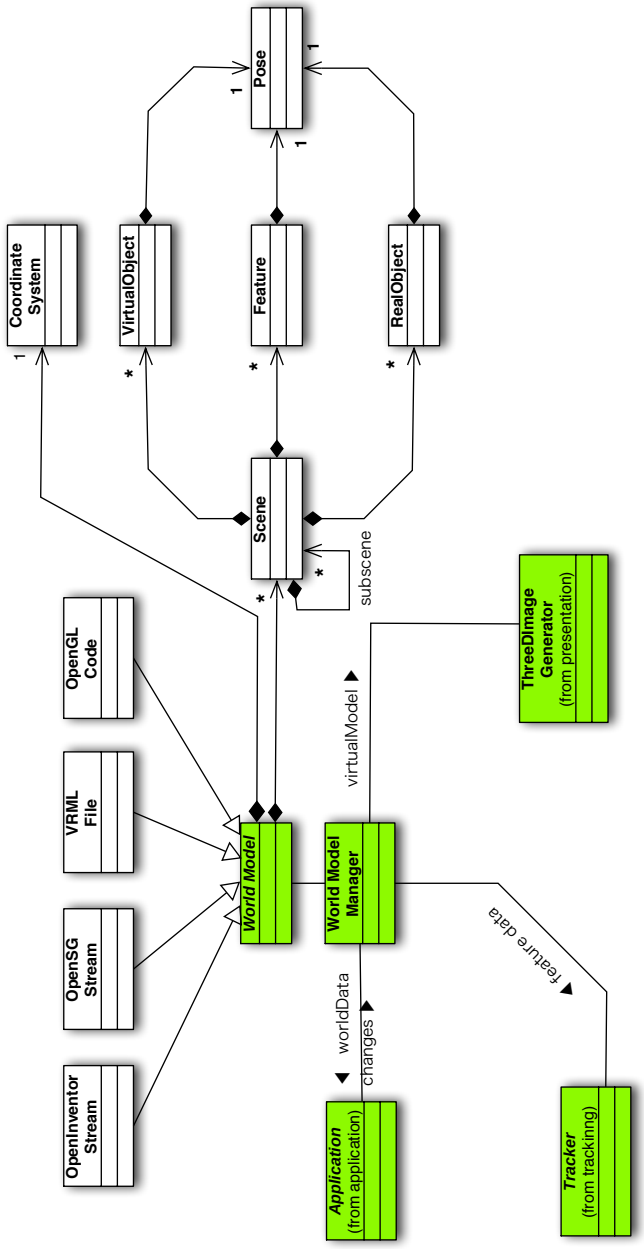


Figure 3.11: The World Model can be a VRML File, OpenGL Code, OpenInventor Stream or OpenSG, to name some. A World Model consists of Scene objects, in each scene are Virtual Objects, Features, and Real Objects. Each of them as a Position in the coordinate system of the World Model. A World Model Manager controls the access to the World Model.

domain, but can be reused in new application domains for the ARVIKA mobile Augmented Reality system.

Architectural style layer. The overall structure of the ARVIKA system is a web-based client/ server system. The mobile Augmented Reality client is built upon a web-browser with plug-ins that has access to various web services that run on a webserver. The Augmented Reality subsystems (tracking and 3D rendering) are realized as local applications. Via the IDEAL device interface various devices can be connected locally or remotely via sockets.

For the mapping onto the reference architecture we consider the components of the application layer for the Application subsystem and the components of the solution domain layer for the remaining five subsystems. The analysis shows that ARVIKA covers each subsystem of the reference model.

Figure 3.12 shows the mapping of ARVIKA onto the reference architecture. The diagram shows that ARVIKA covers each of the six subsystems of the reference architecture. The main ARVIKA Applications are WorkflowManager and InfoBroker. The WorldModel uses VRML files and proprietary marker configurations. The Microsoft Internet Explorer is the main component for presentation and interaction. The Internet Explorer is drawn in the Interaction subsystem and the Presentation subsystem to show the double role of it but actually it exists only once. The Tracking subsystem consists of IDEAL connected sensors and a local VideoTracker component. The ContextManager stores user preferences, device type information, and workflow state.

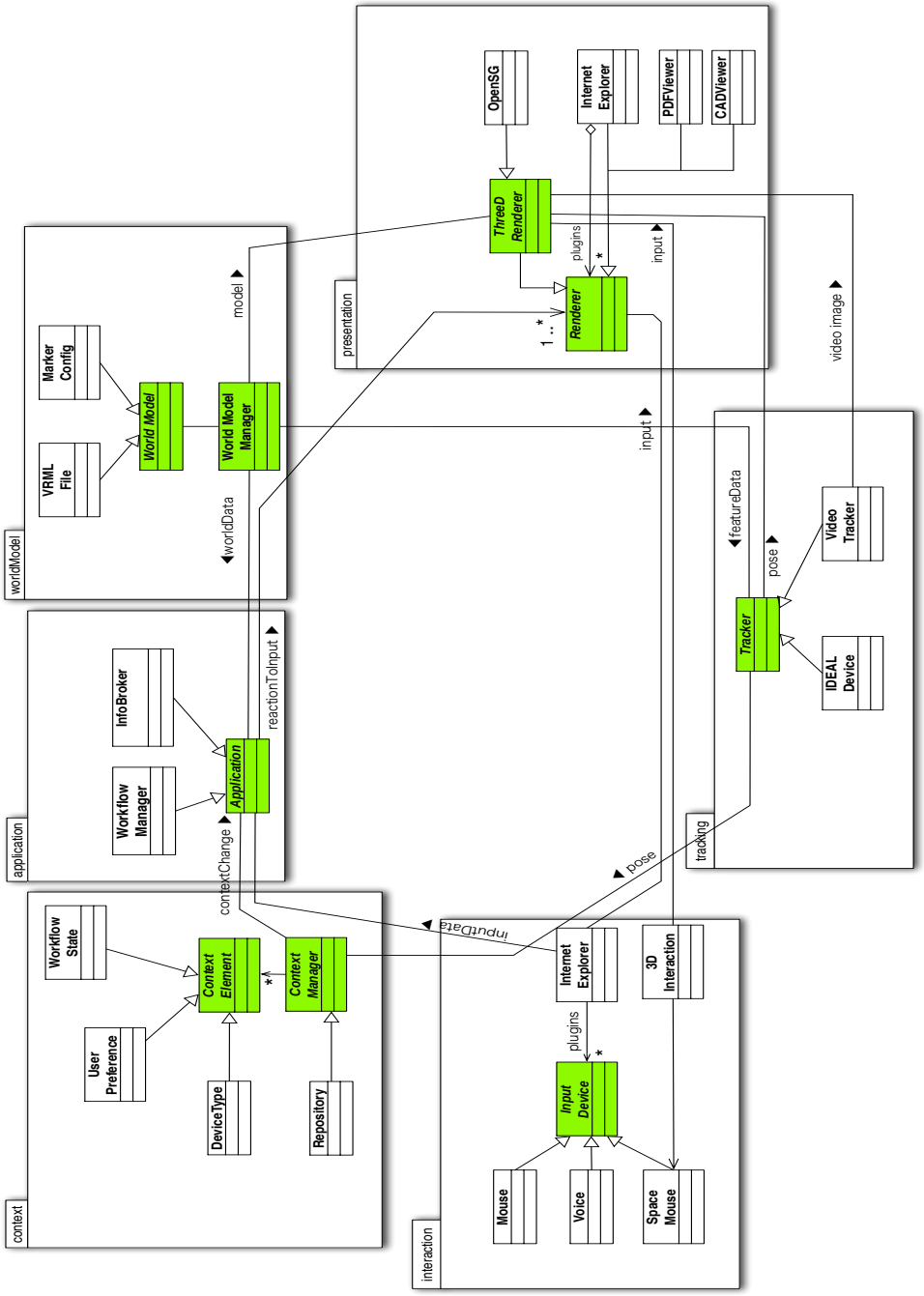


Figure 3.12: ARVIKA architecture mapped onto reference architecture.

3.2 Architectural Patterns for Augmented Reality Systems

The abstract reference model of the previous section describes the general components and structure of Augmented Reality systems. However, depending on the functional requirements of a particular system, some of the components may be left out. For example, the Video Mixer component is not required for optical see-through Augmented Reality.

On a subsystem level, however, the developers of existing Augmented Reality systems use different techniques and building blocks to implement the subsystems, e.g. tracking or presentation. An analysis of existing systems reveals that several techniques and building blocks recur in various existing systems—sometimes explicitly, such as when two systems use a common library, and sometimes implicitly, when different developers apply the same basic techniques. The selection depends on the non-functional requirements and the design goals.

These techniques and building blocks can be extracted from existing systems and described as abstract reusable patterns for Augmented Reality systems design. This is heavily based on the idea of *software design patterns*. Patterns are structured descriptions of successfully applied problem-solving knowledge: *A software architectural pattern describes a specific design problem, which appears in a particular design context, and presents a generic solution scheme. The solution scheme specifies the involved components, their responsibilities, relationships and the way they cooperate* [24, pp 8].

3.2.1 A Catalogue of Patterns for Augmented Reality Systems

We classify the found patterns into six problem categories which comply to the six subsystems. Here we follow Buschmann's [24, pp 362] approach to specify categories that support the search process of developers and use the subsystem decomposition as the base for the problem categories.

Additionally, the list of patterns can be separated into two types: First, patterns that are specific for Augmented Reality and describe good practices for the design of Augmented Reality systems. Second, patterns that can be found in several existing Augmented Reality systems with modifications for Augmented Reality. These patterns complement the pattern system for a high-level description of Augmented Reality systems. So although a pattern might already be well known as a general pattern for system design, for example the Blackboard pattern, we present it in the Augmented Reality context.

Table 3.1 gives an overview of the patterns we found ordered by the defined categories.

Subsystems	Augmented Reality Patterns
Application	Central Control Scripting Scene Graph Node Tracking-Rendering-Loop Web Service Multimedia Flow Description
Interaction	Handle in Application Use Browser Input Functions Networked Input Devices Operating System Resources
Presentation	3D Markup Low-level Graphics Primitives Scene Graph Video Transfer Multiple Viewers Proprietary Scene Graph
Tracking	Tracking Server Networked Trackers Direct Access
World model	Example Class Scene Graph Stream Object Stream Marker File Dynamic Model Loading
Context	Blackboard Repository Publisher/Subscriber Context Pull

Table 3.1: A collection of Augmented Reality patterns.

3.2.2 A Scheme for the Description of Patterns

We describe each pattern by name, goal, motivation, a description, usability, consequences, collaborations, and known use. This follows the scheme of describing architectural or design patterns, e.g. as used by Gamma et al. [45].

Name The name of a pattern should be descriptive and in the best case in use in several systems.

Goal The goal is a short description for the target use of the pattern.

Motivation This section describes why the pattern was developed.

Description We describe each pattern informally by its tasks and structure. We do not yet have formalisms such as UML static and dynamic diagrams for each pattern.

Usability Describes when and how each pattern can or cannot be used.

Consequences The advantages and disadvantages of the pattern.

Collaborations Other patterns that can or must be used in combination to this pattern.

Known use Projects and systems that use the pattern.

A short description of each pattern can be found in the appendix A. Here we give some examples of interesting Augmented Reality patterns. We describe the patterns *Scene Graph Node* and *Scripting* for the Application subsystem, and the *Scene Graph* pattern for the Presentation subsystem.

Scene Graph Node Pattern

Goal: Embed application in world model.

Motivation: In Augmented Reality, user interaction is connected with the physical environment. Consequently applications are often linked to places in the real world. With this pattern, the application is seamlessly embedded in the environment.

Description: A scene graph models the world around a user as a tree of nodes. Each node can be of any type, usually graphical objects such as spheres. But there are also non-graphical objects that include control code.

Collaboration: Uses scene descriptions in scene graph format, may be implemented with in a scripting language, may be implemented as event-call-back.

Usability: In combination with the Scene Graph pattern for rendering.

Consequences: The scene graph-based approach for an application handles the control flow to the underlying scene graph platform, e.g. Open Inventor. On the other side this approach offers a relatively easy possibility for the implementation of shared applications for locally nearby users. One 3D interface can be shared among several users but displayed for each user from a different view.

Known use: Studierstube [134], Tinmith [115]

Scripting Pattern

Goal: Quickly develop new applications.

Motivation: The real-time constraints of a user application are often not very strong, so that it is possible to quickly develop new applications in a scripting language supported by a powerful environment.

Description: For the development of an application, there is a scripting wrapper around all components that have performance constraints. These components are written in compiled languages such as C++ and offer scripting interfaces.

Collaboration: Can implement the Scene Graph Node pattern.

Usability: The development of scripted applications allows rapid prototyping but demands powerful components that implement important functionality. The disadvantage is that the scripting approach is not suited for very complex applications.

Consequences: A script interpreter is needed, as well as (possibly) a special scripting language for AR.

Known use: ImageTclAR [112], Karma [40], Coterie [85], MARS [39], EMMIE [25]

Scene Graph Pattern

Goal: Use a rendering component that allows more complex and dynamic scenes.

Motivation: For the representation of 3D environments, scene graphs have shown to be a reasonable choice. The level of abstraction is higher than for OpenGL, but they are much more powerful and flexible than VRML browsers with a limited application programming interface. Most scene graph components can read VRML based descriptions of scenes.

Description: Examples are (Open) Inventor, OpenSG, Open Scene Graph.

Collaboration: Use Scene Graph Node pattern for the application.

Usability: Use a scene graph if you don't need the flexibility and low-level graphics access that OpenGL provides but want to render more complex scenes and need more dynamic access that a VRML browser offers.

Consequences: Can restrict the possibilities for modelling the application.

Known use: ARVIKA [42], Studierstube [134]

3.2.3 A System of Patterns

One of the goals of software patterns is to provide a common vocabulary for system developers to discuss and compare the different approaches they use. Similar to words in a vocabulary, patterns do not exist in isolation; there are interdependencies among them. Patterns can be integrated into a system of patterns: *"A system of patterns for software architecture is a collection of patterns for software architecture, combined with rules for their implementation, combination, and practical application for software development."* [24, pp 360] Buschmann et al. formulate several requirements on a system of patterns: it must contain a sufficient number of patterns, each pattern should be described consistently, the pattern system should show the relationship between patterns, the patterns should be ordered adequately, the pattern system should support the construction of new systems, and support its own evolution. We support these requirements by the catalogue of patters and the schema for the descriptions of individual patterns.

To give an overview of the relationships between the individual patterns we use a directed graph. Each pattern is part of this graph along with labelled arrows indicating direction and type of the relationships. Figure 3.13 shows the identified patterns and their relationships. This illustration is similar to the one used in Gamma et al. [45]. To support the locating of patterns we show the associated subsystems.

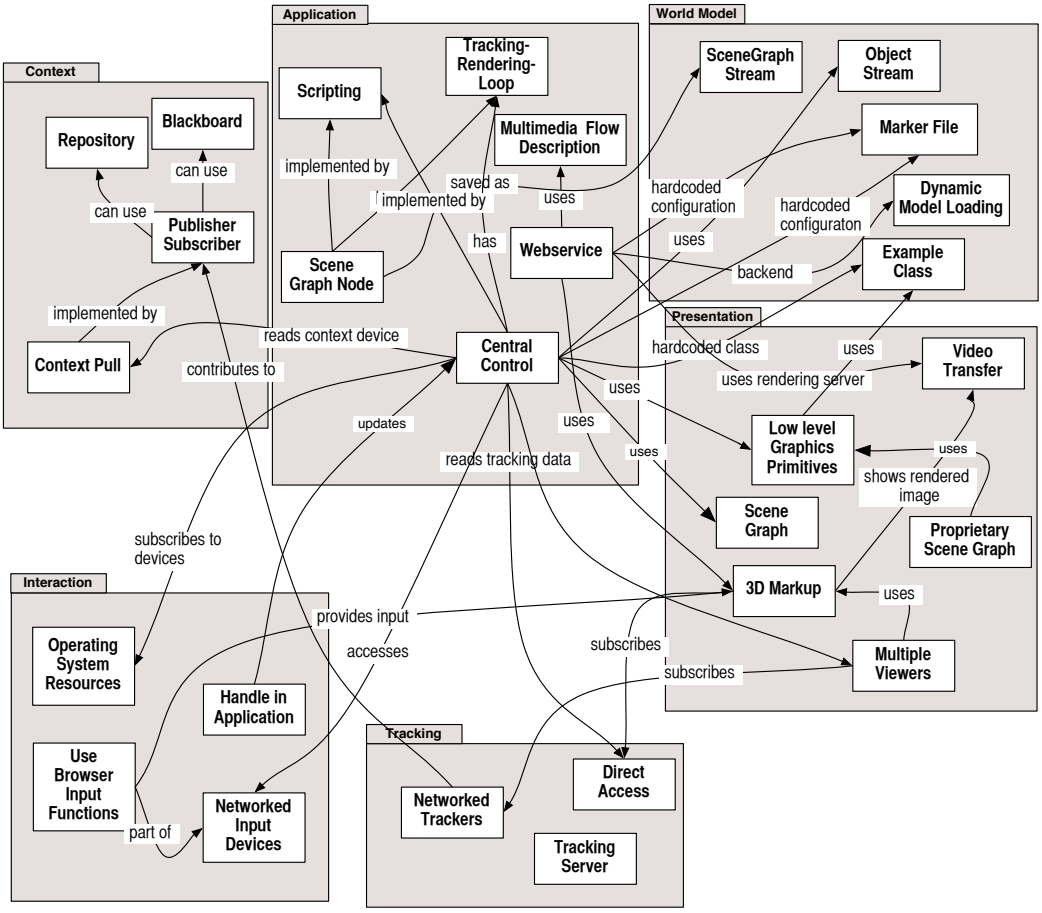


Figure 3.13: Relationships between the individual patterns for Augmented Reality systems. Several patterns are used in combination within an Augmented Reality system. One pattern might require the use of other patterns or prevent the usage.

3.3 Conclusion

This chapter covered the second layer of section 1.7, the solution domain layer. We presented an abstract architecture and a system of software patterns for Augmented Reality systems. The abstract architecture is based on the notion, that the core functional requirements for Augmented Reality systems are basically the same. This allowed us to identify recurring components and collaborations. Architectures for concrete Augmented Reality systems are an adaptation of this abstract architecture to comply to application specific functional and non-functional requirements. The system of patterns for Augmented Reality system is based on the non-functional requirements which differ for each system. As well as different systems have some of the non-functional in common, they also share some recurring approaches to fulfil them. These approaches can be described as architectural patterns for the design of Augmented Reality systems.

4 The DWARF Contract-based Peer-to-Peer Architectural Style

A contract-based peer-to-peer architectural style, a supporting middleware, and a graphical notation for distributed AR systems.

This chapter covers the architectural style layer, bottom layer of the four layer architectures abstraction framework laid out in section 1.7.

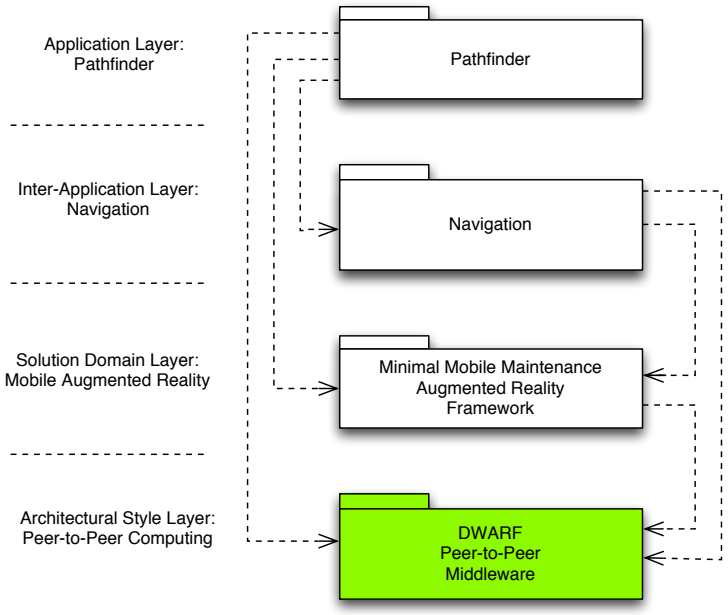


Figure 4.1: Architectural style layer chapter

Software architects use a number of styles in the design of software architectures.

Each style is suitable for some classes of problems, but none is a general solution for all [139]. A goal of this dissertation was to find the appropriate styles for mobile Augmented Reality systems. In section 2.4 we discussed the requirements and design goals for these systems. They were: development of a service-based framework, connectors as first-class objects, design by contract, dynamic service lookup, and runtime reconfiguration. Therefore we propose to use a *contract-based peer-to-peer architectural style*. The main building block of this style are Services which we call DWARF Services. In the following sections we present this style (section 4.1) and the supporting DWARF middleware (section 4.2). For the description of the style we follow the outline of Buschmann et al. in [24].

We introduce a graphical notation for systems based on the proposed architectural style in section 4.3. This notation is based on UML 2.0 component diagrams.

Finally we describe an example for the implementation of a DWARF Service in section 4.4

4.1 A Contract-based Peer-to-Peer Architectural Style

A peer-to-peer architectural style allows to build a system configuration from a set of collaborating peer components or services. To establish a collaboration, each participating peer specifies to the environment what needs it has, what abilities it can offer, and how it wants to communicate. We call a collaboration on the base of specified Needs and Abilities respectively a *contract*. Actually, there are two types of contracts. There is one contract between a service that provides an Ability and another service that has a Need for that Ability. This contract could be called a *Collaboration Contract*. And there is a second contract between a service and the environment: If the environment can fulfil all Needs of a service, then the service will offer its Abilities. This contract could be called a *Requires/Provides Contract*. We illustrate this in figure 4.2.

Example. In 2.1 we describe a scenario for mobile Augmented Reality maintenance systems. A possible system configuration could include an optical local tracker, a remote position tracker, and a local presentation subsystem for user output. The optical tracker and the presentation subsystem communicate over shared memory, because they must share the incoming video images for image analysis and video overlay. The remote position tracker and a local tracking manager communicate over asynchronous CORBA Events.

Context. A (distributed) component-based system with different communication requirements between different component connections, for example a multi-media system.

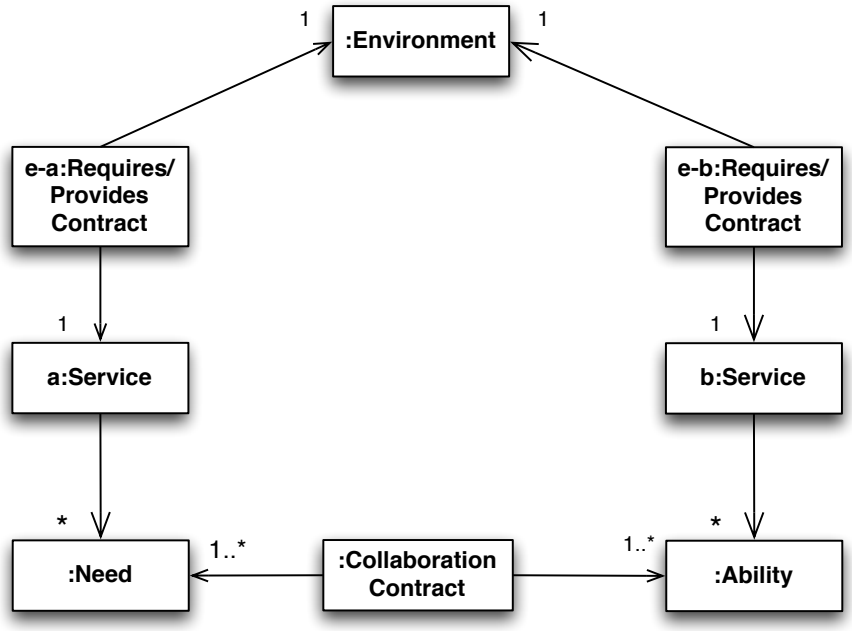


Figure 4.2: The concept of the contract-based peer-to-peer architectural style. A Service holds two contracts: a Requires/Provides Contract with its Environment and a Collaboration Contract with another Services over a Need/Ability pair.

Problem. In the last chapter we learned about the building blocks of Augmented Reality systems. In a ubiquitous computing environment some of these blocks are local and some are remote, for example in so called 'Augmented Reality-ready buildings' where the building is equipped with sensors for Augmented Reality [79]. A mobile Augmented Reality system must be adaptable and able to dynamically find services in the environment. It must optimize the overall system functionality by connecting matching services on the mobile system and in the environment. It must also deal with the loss of connection between services, such as when the user leaves a room or turns a component off in the mobile system. Ideally, the middleware handles services running on hardware worn by the user and services running on devices in the environment without difference. The requirement of system adaptability requires a loose coupling between the services.

As we have shown in section 2.4, Augmented Reality building blocks must communicate efficiently to meet the real-time requirement for Augmented Reality systems.

The most efficiently communicating system is where every single connection between communicating components is adapted specifically. The disadvantage is that this often leads to a mix of heterogeneous communication mechanisms in one system.

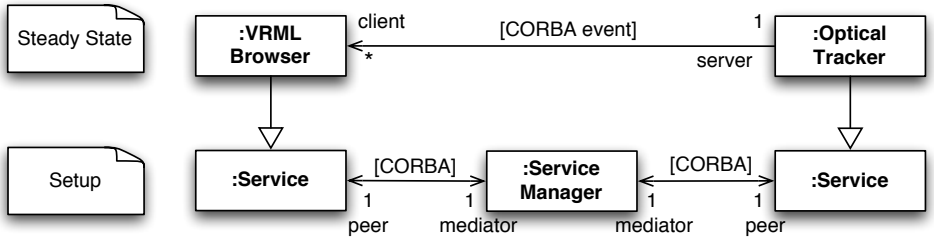


Figure 4.3: Connection layers for efficient communication.
In the setup phase the Service Manager establishes a connection between two peer Services. In the following steady state phase the Services communicate directly without involvement of the Service Manager. In this example, the Services use a client/server connection over CORBA events. The server sends events to the client.

Solution. The contract-based peer-to-peer architectural style consists of a combination of several techniques. Systems are built by distributed services that assemble themselves into a configuration for a complete system. The functionality for the services to find each other and to establish connections is provided by a *mediator*. A mediator arranges the Collaboration Contract between the Services.

Specification of contracts between DWARF Services. In a component or service-based system, services cooperate over the mutual provision of abilities. Traditionally the possibilities to declare the offered abilities and needed abilities (needs) from another service are restricted to names of interfaces and the signatures of the supported methods. The declaration of dependencies on the base of exported and imported interfaces does not allow to transport semantic data such as tracking accuracy. A simple method to transport semantics are the setting of attributes for each ability and predicates for each need. Each service offers a contract to the mediator: when the mediator fulfils all needs with the required quality, i.e. it establishes connections to all needed services that meet the predicates and the desired connection type, then the service provides its abilities, i.e. the mediator can establish connections to the service for other services.

Support of component-specific communication methods. Traditional middleware technologies such as CORBA [103] or COM+ [94] do not support heterogeneous means of communication as we discussed in 2.5.3. But to adapt the system's transport facilities to the requirements of the components it is best to let the components select

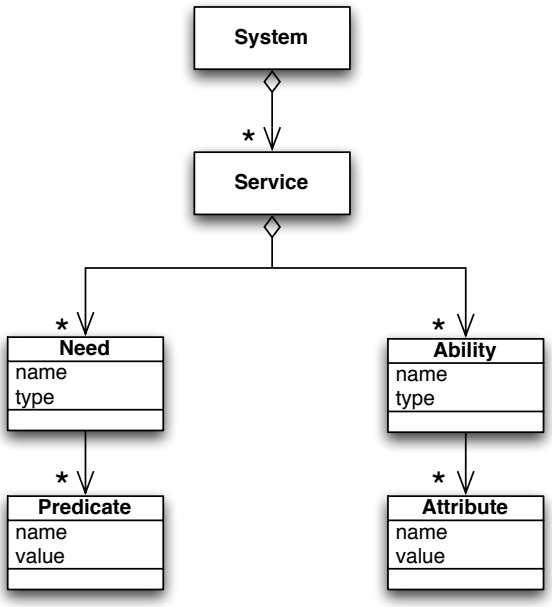


Figure 4.4: A System consists of Services. Each Service has Needs and Abilities. A Need describes a requirement by Predicates. An Ability describes what the Service can provide for other Services. The quality of service is expressed by Attributes. Needs of one Service can be satisfied by Abilities of another Service.

the desired type of connection and let the system establish it. This includes access to low-level and high-level communication methods such as remote method invocation and sending events.

Separate system setup and steady state. In the setup phase the mediator looks up all required services and sets up the connections between them. Then the connected services communicate with each other over the established connection specific connectors (figure 4.3). After the services were started and configured to a system there is only little need for reconfiguration. Most connections between services stay fixed after they were established. For example, a local optical tracker sends position updates to a local display service thirty times per second as long as they are connected. This two-phase approach combines adaptability with dynamic system reconfiguration by a mediator, and performance with heterogeneous direct peer-to-peer connectors that by-pass the mediator.

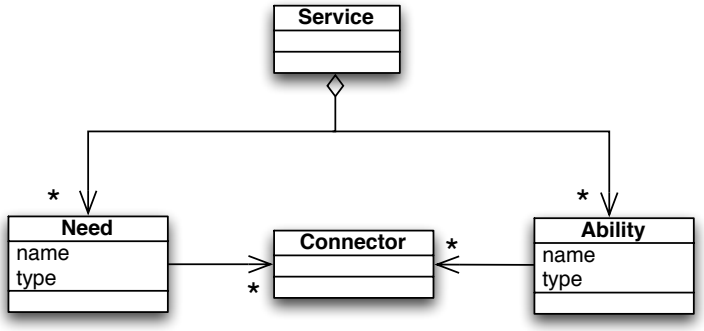


Figure 4.5: Model of the contract-based peer-to-peer architectural style.
A Service has Needs and Abilities. Matching Needs and Abilities are tied over a Connector which provides the required communication means. The Collaboration Contract of figure 4.2 is modelled as a Connection between Services.

Structure. A DWARF *System* consists of a set of *Services*. A mediator connects independent DWARF Services at runtime to build a system configuration. Figure 4.4 presents the elements of the architectural style. The mediator as a runtime object is left out here.

System. In this style a System is a configuration of distributed cooperating Services. Instead of components we use the term Services because of the integration of Services provided by devices in the environment.

Services. Services are the building blocks of systems. They are similar to components as they are used like components to compose a system but they are different to components in one aspect. A component is a software construct that is instantiated to be used in the context of one system. A Service is instantiated at some time to be used by many clients.

Needs and Abilities. Needs and Abilities model the offers of a Service to other Services and the requirements from other Services. When the Needs of a Service can be fulfilled, then the Service can offer its Abilities. A match of a Need by an Ability leads to a Connection between the Services. An Ability might be used by several needing Services.

Attributes. To refine the specification of an Ability attribute/value pairs can be used. For example, a Tracking Service could provide an Attribute `update_rate` with the value `30`.

Predicates. Predicates are used for the selection of a Service. Needs have Predicates that are applied over the Attributes of the Abilities to select a matching one.

In the example above, the Need of a Rendering Service could provide the predicate `update_rate>=30`.

Connectors. Connectors are first class elements. They allow to specify the desired communication mechanisms and instantiate the corresponding system communication resource. Usually connectors are solely handled by the runtime system without possibility to intervene by the involved services. In figure 4.5 the model is enhanced by a Connector. Ability and Need have a reference to Connectors. To decide when to model something as a Need and when to model it as an Ability, consider on which side of the communication an actor is on. Thus, it is natural to model a head-mounted display as *needing* position data, since this data is used to generate output for the user. On the other hand, a printing Service has the *Ability* to accept print jobs, since the user has documents that he wants to print out. This example shows that the direction of data flow does not necessarily coincide with the direction of a dependency between two Services.

Following the 'where is the actor' guideline, a Service's Need in one application may become an Ability in another. For example, when a user watches a television show, his television *needs* images to display in order to make the user happy. During commercials, this changes. The viewer actor would be happier without commercials, but the television has the *Ability* to show images, which another actor, the advertiser, is paying for.

If the middleware is aware of this kind of distinction, it makes it easier to design applications around fulfilling the user's desires.

Related work. The contract-based peer-to-peer architectural style is a new combination of several well-known techniques.

The *Peer-to-Peer architectural style* has been used in many systems. Prominent examples are change platforms, examples involve the file sharing platform Gnutella [72] and SUNs peer-to-peer framework JXTA [49].

There are several component models available for component-based software engineering. The most prominent are the CORBA Component Model, Microsoft .NET, Sun Java ONE. We presented them already among several approaches from research in section 2.5.3. These models enable ubiquitous computing with distributed software components. The DWARF contract-based peer-to-peer style is not directly supported by these models. Our style complements them and could be added to each of them.

Using *components and connectors as first class objects* is an established technique in component-oriented software engineering research. Several authors use this concept in their respective version of an architecture specification language (we gave several examples in section 2.4). The described style uses this approach not only for specification purposes but also for connector objects at runtime. Connections are not created by the components themselves but by the runtime environment.

The *Mediator Pattern* is a well-known pattern in pattern literature [45]. We use it

for decoupling individual DWARF Services.

Design by contract was firstly introduced by Bertrand Meyer [91]. He uses it to specify assertions on objects in Eiffel. We use a modification of this paradigm. A service offers a contract to the mediator and not directly to other services. This contract offer is used at runtime to establish a connection to other services.

4.2 The DWARF Peer-to-Peer Middleware

So far we have discussed the contract-based peer-to-peer architectural style for the design of mobile Augmented Reality systems. It gives the basic outline for the design of a system and specifies which component types a developer can use for the system design. Now we will address the DWARF middleware that implements this style.

In the specification of the architectural style in section 4.1 we use the abstract class Environment which acts as a mediator between the individual services. The responsibilities of this abstract class are taken over by the DWARF *middleware*. The particular responsibilities are recorded by several use cases in section 4.2.1, and an analysis of the functional and non-functional requirements (sections 4.2.2 and 4.2.3).

This analysis is followed by an object model of the DWARF middleware 4.2.4 and the system design. The DWARF middleware consists of several subsystems for the individual tasks of the middleware (section 4.2.5). It coordinates between matching services and sets up the communication. It uses descriptions of the individual services and their communication needs to do that automatically. For example, a Display Service and a Tracking Service can be connected together without user interaction.

Finally we describe several possible deployment strategies for the middleware (section 4.2.6) and persistent Service descriptions on the base of XML (section 4.2.7). The DWARF middleware is implemented on the top of CORBA which allows the variable deployment of the middleware.

4.2.1 Use Cases

The following use cases describe the functional model of an Augmented Reality system based on Needs and Abilities. Figure 4.6 shows the relationships between use cases.

The first two use cases are from the point of view of a Service, which is started manually and needs data from other Services to operate.

Use case name:	**StartManually**
Participating actor:	Initiated by `User` Communicates with `Service`
Flow of Events:	1. *Entry condition:* The `User` starts the `Service` manually by starting the appropriate executable.

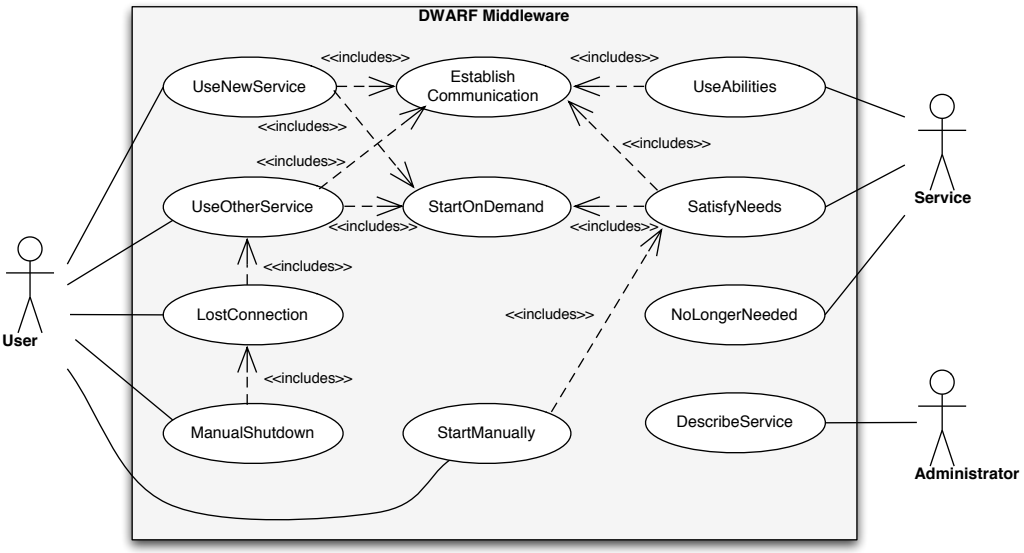


Figure 4.6: Use cases for the DWARF peer-to-peer middleware.
The dashed arrows indicate "includes" relationships between use cases; the solid lines indicate use cases being initiated by actors.

2. The `Service` checks whether the middleware has a Service description with its name, i.e. whether it has been described by an `Administrator` using the *DescribeService* use case.
3. If there is no such description yet, the `Service` describes itself to the middleware using an appropriate interface.
4. The `Service` registers itself by name.
5. The *SatisfyNeeds* use case is invoked to satisfy the `Service`'s Needs.
6. *Exit condition:* If the `Service`'s Needs can be satisfied, the middleware tells it to start running.

Once the Service has registered itself with the middleware, the middleware will try to fulfil the Service's needs.

Use case name:	**SatisfyNeeds**
Participating actor:	Initiated by `consumer:Service`
	Communicates with `suppliers:Services`

Flow of Events:

1. *Entry condition:* The `consumer` is a Service that needs data from another Service in order to operate. It has been loaded and has registered itself with the middleware, using the *StartManually* or *StartOnDemand* use case.
2. The middleware attempts to find other Services that can deliver the data this Service needs. If necessary, it starts them on demand with the *StartOnDemand* use case.
3. For each Need of the `consumer` Service that the middleware satisfies, it establishes communication between the appropriate `supplier` and the `consumer`, using the *EstablishCommunication* use case.
4. *Exit condition:* The Service is connected to enough other Services to satisfy all of its Needs, provided that enough other Services are available.

The next three use cases reflect what happens on the other end: a Service is started on demand, and one of its Abilities is used. For the Service to be started on demand, it must first be described by an administrator.

Use case name: **DescribeService**

Participating actor: Initiated by `Administrator`

Communicates with —

Flow of Events:

1. *Entry condition:* The `Administrator` places a valid XML Service description in a specified directory for the middleware.
2. *Exit condition:* The middleware loads the Service description and can now start the described Service on demand.

Now that the Service has been described, it can be started when its Abilities are needed by other Services.

Use case name: **StartOnDemand**

Participating actor: Initiated by `firstConsumer:Service`

Communicates with `supplier:Service`

Flow of Events:

1. *Entry condition:* The `supplier` is a Service that can deliver data to other Services. The Service has previously been described by an administrator, using the *DescribeService* use case. Enough other Services are available to satisfy all needs of the `supplier` Service. A `firstConsumer` Service needs data that the `supplier` can supply, and has also been started and registered with the middleware.
2. The middleware loads the `supplier`, using the executable file

specified in the Service description.

3. *Exit condition:* The Service registers itself with the middleware by name.

Once the desired Service has been started, its Needs are satisfied, and then it can offer its Abilities to other Services.

Use case name: **UseAbilities**

Participating actor: Initiated by `firstConsumer:Service`

Communicates with `supplier:Service`

Flow of Events:

1. *Entry condition:* The `supplier` is a Service that can deliver data to other Services. It has been loaded and has registered itself with the middleware, using the *StartManually* or *StartOnDemand* use case. Enough other Services are available to satisfy all needs of the `supplier` Service. A `firstConsumer` Service needs data that the `supplier` can supply, and has also been started and registered itself with the middleware.
2. The middleware satisfies all of the `supplier`'s needs, using the *SatisfyNeeds* use case.
3. The Service is notified that it should start operating.
4. The middleware establishes communication between the `supplier` and the `firstConsumer`, using the *EstablishCommunication* use case.
5. *Exit condition:* The Service is running and can provide its requested Ability to the `consumer`.

When a Service's Abilities are no longer needed, the middleware can shut it down.

Use case name: **NoLongerNeeded**

Participating actor: Initiated by `lastConsumer:Service`

Communicates with `supplier:Service`

Flow of Events:

1. *Entry condition:* At least one Ability of the `supplier` is being used, as per the *UseAbilities* use case. The last consumer using this Ability, `lastConsumer`, disconnects.
2. The middleware waits for a timeout specified in the Service description so that other consumers can reconnect to the Ability. If another consumer connects again during this time, this use case ends.
3. The middleware disconnects the Needs of the `supplier` from all other Services.

4. The middleware frees the communication resources used by the Service.
5. The middleware notifies the Service that it should stop running.
6. *Exit condition:* The Service unregisters itself from the middleware and terminates.

Of course, a Service can be shut down by the user, as well.

Use case name: **ManualShutdown**

Participating actor: Initiated by `User`

Communicates with `Service`

Flow of Events:
1. *Entry condition:* The `User` instructs the `Service` to shut down.
2. The `Service` unregisters itself from the middleware and terminates.
3. The middleware frees the communication resources used by the `Service`.
4. *Exit condition:* For any Services that were using this `Service`'s Abilities, the *LostConnection* use case is used.

The next two use cases deal with the middleware's dynamic behaviour when the user roams around in an intelligent environment. First, a new Service can be used *in addition to* an old one.

Use case name: **UseNewService**

Participating actor: Initiated by `User`

Communicates with `newSupplier, consumer:Service`

Flow of Events:
1. *Entry condition:* All needs of the `consumer` Service have been satisfied, using the *SatisfyNeeds* use case. The `consumer`'s Service description allows the Need to be satisfied more than once (e.g. a Tracking Manager that can combine multiple position data inputs). The `User` enters a new wireless network, where the middleware finds a `newSupplier` that can satisfy one of the `consumer`'s needs.
2. The middleware establishes communication between the `newSupplier` and the `consumer`, using the *EstablishCommunication* use case.
3. *Exit condition:* The `consumer` can use the Abilities of the `newSupplier`.

Second, a new Service can be used *instead of* an old one.

Use case name: **UseOtherService**

Participating actor: Initiated by `User`

Communicates with `newSupplier, oldSupplier, consumer: Service`

Flow of Events:

1. *Entry condition:* The `consumer` is currently using the `oldSupplier` to satisfy one of its Needs. The `user` either enters a new wireless network, where the middleware finds a `newSupplier` that can satisfy one of the `consumer`'s needs better than the `oldSupplier`, or the `oldSupplier` is disconnected.
2. The middleware disconnects the `oldSupplier` from the `consumer`. This can lead to the *NoLongerNeeded* use case being used for the `oldSupplier`.
3. If necessary, the middleware starts the `newSupplier` with the *StartOnDemand* use case.
4. The middleware establishes communication between the `newSupplier` and the `consumer`, using the *EstablishCommunication* use case.
5. *Exit condition:* The `consumer` can use the Abilities of the `newSupplier`.

If a connection is lost, the middleware notifies the Services depending on it.

Use case name: **LostConnection**

Participating actor: Initiated by `User`

Communicates with `oldSupplier, consumer:Service`

Flow of Events:

1. *Entry condition:* The `consumer` is currently using the `oldSupplier` to satisfy one of its Needs. The connection is lost when the `User` leaves the wireless network.
2. The middleware notifies the `consumer` of the lost connection.
3. If a new supplier is available that can satisfy the `consumer`'s need, the *UseOtherService* use case is used.
4. *Exit condition:* If a new supplier is available, the `consumer` can use its Abilities.

The last use case is never initiated directly by actors; it is included by other use cases.

Use case name: **EstablishCommunication**

Participating actor: Initiated by other use cases

Communicates with `supplier:Service, consumer:Service`

Flow of Events:

1. *Entry condition:* The `supplier` and `consumer` Services have both been loaded and have registered themselves with the mid-

dleware. The consumer has a Need that the supplier has a matching Ability for.

2. The middleware sets up the necessary communication resources for communication. This can be, for example, an event channel, a shared memory block or a remote interface reference. The choice of communication resources depends on the Service's preferences and their location in the network.
3. If this is the first consumer to be connected to the `supplier`, the middleware notifies the `supplier` that its Ability is desired, and supplies it with the properly configured communication resources. Otherwise, the `supplier` is not notified.
4. The middleware notifies the `consumer` that its Need can be satisfied, and supplies it with the properly configured communication resources.
5. *Exit condition:* The `supplier` and `consumer` Services are both connected to the communication resources and can communicate with one another.

4.2.2 Functional Requirements

The DWARF middleware has to realize the coupling of the distributed services to build a configuration of services. In particular the services have to be *located, connected*, and *managed*.

Locate Services

One main area of functionality for the middleware is locating services. Note that the term *location* refers to logical locations such as network addresses, not to physical *positions*.

Advertise Services DWARF Services have Abilities that they offer to other Services. For example, trackers provide position data. The middleware must advertise these Services on the network so that other Services can find and use them.

Discover Services Conversely, the middleware must discover Services that have been advertised, so that the head-mounted display, for example, can receive position data from a tracker.

Abstract Description of Services In order to reduce coupling between the individual DWARF Services, the Services should access each other only through well-defined interfaces. The middleware can maintain an abstract description of each DWARF

Service, including the interfaces it supports and what kind of other Services it depends upon. Aside from formalizing the dependencies between Services, this also allows the middleware to start Services on demand, since it knows details of the DWARF Services from the Service descriptions before a Service actually runs.

Expose Needed and Provided Quality of Service For the real-time requirement on Augmented Reality, it is important that the position data are accurate and have a low temporal lag. This can be described in *quality-of-service attributes* of Abilities and *quality-of-service predicates* of Needs. The middleware must be able to use the attributes, to find the Services that meet the predicates best. The attributes can be static (such as the resolution of a video camera) or dynamic (such as the current accuracy of a GPS device),

Roaming and Handover In mobile systems, new Services can become available and the connection to old ones can be lost. These situations are not exceptions but the normal case for mobile systems. To avoid user interventions, the middleware must be able to handle these situations autonomously by changing the current Service configuration dynamically. For example, when the user walks out of the range of one video camera, another camera should take over.

Connect Services

Once Services have been located, they can communicate with one another with support from the middleware. Again, this functionality can be divided up:

Manage Communication Resources The middleware should provide a general mechanism for managing communication resources. This way, neither one of two communicating Service have to deal with allocating event channels, network sockets, etc., making implementation of the DWARF Services easier. Additionally, this allows the middleware to configure the communication resources in order to optimize the communication in a group of Services—for example, by migrating an event service from one network node to another.

Extendable Communication Support One of the design goals in section 2.4 was the support of various communication means to be able to select the best one between the respective Services.

Event-based communication is flexible, since it frees the sender from having to know who will receive the event, and the receivers from having to know who sent it. It is a useful mechanism where small chunks of information do not have to be sent reliably to one or several receivers, for example Position events. The DWARF middleware must provide a mechanism for event-based communication. It also must manage the publish-and-subscribe mechanism for events, so that the

individual Services do not have to implement this themselves. The reason is that the middleware also manages the autonomous connect and disconnect between Services.

Remote method calls. To support access to complex services, in particular non-stateless services, the calling of methods on remote objects must be supported. For example, a World Model Service could be realized as a CORBA server.

Extensibility to other communication methods. In order to keep the middleware open to future requirements it should be extendable for additional communication methods, such as shared memory blocks (when two Services are on the same host) or streaming video.

Manage Services

After locating Services and connecting them, the Services have to be managed at runtime.

Starting and Stopping Services The middleware should be able to offer Services' abilities on the network even when the Services are not running, and start the Services on demand when other Services need them. Analogously, the middleware should be able to shut Services down when they are no longer needed.

Service Status Information The middleware must provide a mechanism to gather status information on the Services. This information is needed for monitoring and control purposes. For example, for decisions about Service exchange due to decreasing quality of service.

4.2.3 Non-functional Requirements

The DWARF middleware has to make a trade-off between two contradicting requirements: efficiency and flexibility. On the one side it must support the Augmented Reality real-time requirement and on the other side it must provide the flexibility for combining wearable computers and intelligent environments (see the design goals in section 2.4). Additionally, the middleware must be very stable.

The middleware has a central role in setting up and maintaining system configurations from a set of Services. The middleware must be able to arrange for communication between a wide variety of Services and under a wide variety of circumstances. For that the middleware must additionally provide high stability,

Low Latency The event-based communication that the middleware supports must have a very low latency, so that position data can be transmitted from the trackers to the head-mounted display quickly enough that the user does not begin to feel sick. Not all events have to have this low latency, but a position event

should not require more than 15 milliseconds (equivalent to half a video frame at 30 frames per second) from the time the tracker sends it until the time the display receives it.
Of course, the non-event-based communication should be fast as well.

Throughput The middleware must be able to handle fairly large volumes of data in communication. For example, if an external server does the image processing, the video image must be transferred from the clients camera to the server efficiently. Example systems are UbiCom [113] and STAR [142].

Fault Tolerance The Augmented Reality system must be able to tolerate failures of network connections that are frequent in mobile environments. Indeed, it should shield the Services from these failures as much as possible and perform seamless handover [1].

Robustness The middleware will connect Services together that it may never have known about before, such as when a mobile user walks into a new intelligent building. In the case that one such Service fails, the middleware must be able to deal with it, so that the other Services can continue running. Where possible, the middleware must shield the other Services from the failure by either starting replacement Services or caring for a graceful shutdown of dependent Services.

Reliability The middleware must perform its tasks reliably, which means that it must not connect Services that do not match. If it tries to supply a printer with position data from a voice recognition system, neither side will work properly.

Scalability The middleware should be able to support a network of mobile computers. The design should be scalable so that the middleware can run on large networks with many mobile (wearable) terminals.

Ad hoc Connectivity When a user walks through a room with an ad hoc wireless network, the middleware should be able to (briefly) establish a connection between the wearable system and stationary Services. This allows the user to collect information from the environment while walking by.

4.2.4 Object Models

The object model for the DWARF middleware refines the general model given by the contract-based peer-to-peer architectural style (section 4.1).

[1] This non-functional requirement leads to the important design decision of *distributing* the middleware

Services

A Service can be started and stopped by a user or by the middleware, as in the use cases *StartManually* and *StartOnDemand*. It is registered with the middleware, so it can advertise its functionality to other Services, and find the external functionality it needs. Services can be stopped manually or when they are no longer needed, as in *ManualShutdown* and *NoLongerNeeded*.

Services are not sub classed into "supplier" and "consumer" Services, even though these roles show up in the use cases. This simple division would limit the modelling power of the middleware: we have to be able to model Services that can play both roles simultaneously. This is accomplished by the next two objects identified.

Needs and Abilities

Following the structure of the contract-based peer-to-peer style, a Service can have zero or more Needs, and zero or more Abilities (shown in figure 4.5).

For example, an Optical Tracker Service attached to a digital video camera might have the following Needs and Abilities (figure 4.7).

- the Ability to provide position data,
- the Need for updates on context changes,
- and a Need for a description of the world that it is supposed to recognize.

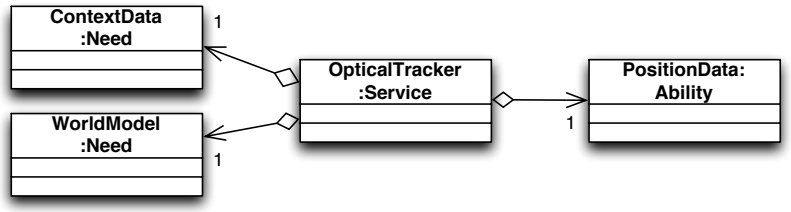


Figure 4.7: Object model of an Optical Tracker Service with two Needs for context data and a world model and one Ability for the provision of position data.

The middleware must satisfy the Needs of all Services with the Abilities of others. Thus, at any given time, the middleware must maintain a dependency graph between Services, as shown in Figure 4.8 on the facing page. This is where the middleware performs most of its work, as described in the use cases *SatisfyNeeds*, *UseAbilities*, *UseNewService*, *UseOtherService*, and *LostConnection*.

In addition, the middleware can start and stop Services on demand, since it knows when a Service is needed by others, and when a Service's Needs are satisfied.

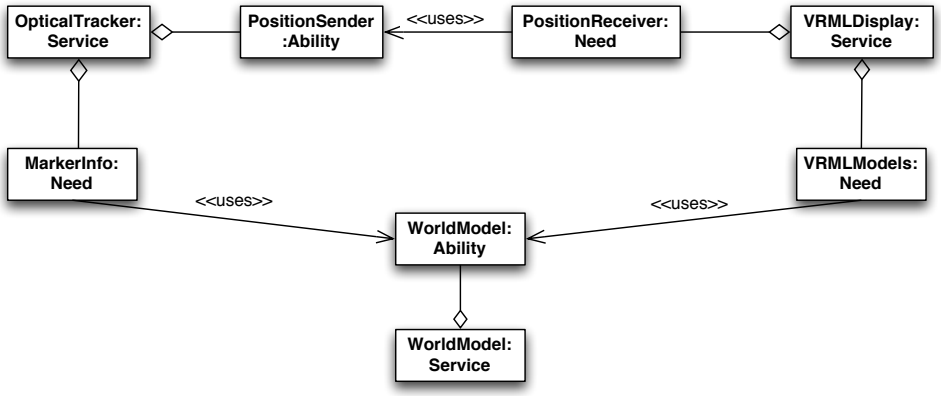


Figure 4.8: Example dependency graph between Services. A VRML-based three-dimensional display Service needs models of real and virtual objects to display, which come from the World Model Service. It also needs position data from the Optical Tracker, which in turn depends on the World Model as well for descriptions of markers to recognize.

Service Descriptions

Even when a Service is not running, it can be described. Parts of this description are hard-coded into the Service itself, and parts can be adjusted according to the application the Service belongs to.

In other middleware systems, this description is often implicit, i.e. coded into the Service. If, however, this description is formalized as a *Service Description* class, it can be given to the middleware by an administrator (use case *DescribeService*). This conserves resources by letting Services be located before they are started, and also makes it possible to package Services as well-described stand-alone components. A Service Description can include the following information:

- a Service's Abilities, i.e. the functionalities it provides
- a Service's Needs, i.e. functionalities of other Services that this Service depends on
- quality-of-service information (as attributes)
- attributes of the Service, such as a printer's paper size
- supported communication protocols
- how to start the Service, e.g. command line.

Figure 4.9 shows the currently supported model for Service Descriptions. A Service Description links to Descriptions of all Needs and Abilities of the Service. Each Need and Ability Description links to Descriptions of all supported Connectors.

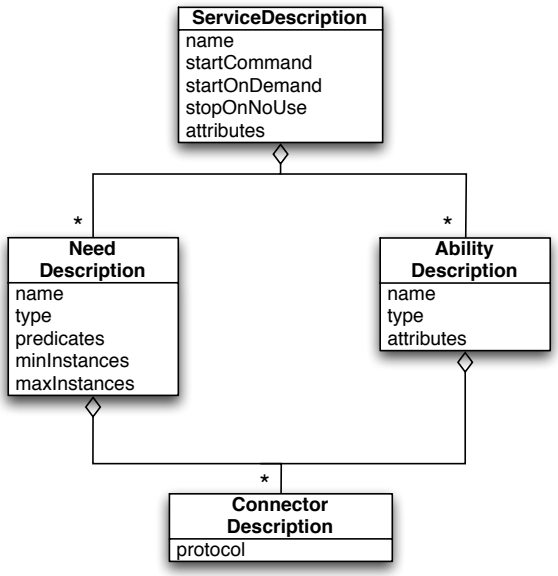


Figure 4.9: Model of Service, Need, Ability, and Connector Descriptions. A Service Description links to Descriptions of all Needs and Abilities of the Service. Each Need and Ability Description links to Descriptions of all supported Connectors.

Some elements of this description are static (e.g. pages per minute) and can be queried even when the Service is not running, others can change dynamically while the Service is running (e.g. amount of toner remaining or number of pending print jobs). The middleware must provide a mechanism for these attributes to be accessed in a uniform fashion.

Communication Resources

There are many different ways that Services can communicate with one another. This includes different basic communication methods such as notifications, sockets and shared memory. Entity Objects involved here are communication resources such as notification channels, shared memory blocks, and sockets(4.10).

For each supported communication resource Connectors for each side of a connection

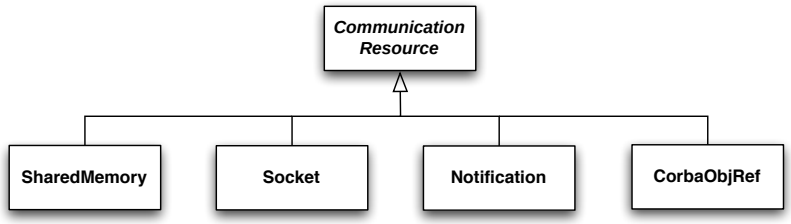


Figure 4.10: DWARF is open to support different communication methods such as notifications, sockets and shared memory.

must be provided. Currently DWARF provides connectors for CORBA object references and structured notifications over the CORBA Notification Service (figure 4.11).

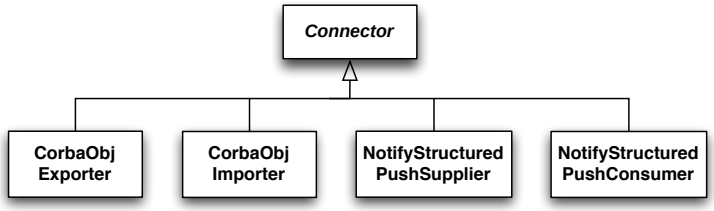


Figure 4.11: For each supported communication resource a pair of corresponding connectors must be provided, for example, export and import connectors for CORBA objects references and push suppliers and consumers for CORBA notifications.

Service Manager

Each Service must be able to access the middleware. In order to give the DWARF middleware a coherent interface, we defined an object called the *Service Manager*.

Upon start-up, a Service registers itself with the Service Manager, from where it can gain access to all of the DWARF middleware's functions. The Service Manager must know the Service descriptions for the Services that register themselves with it. If the Service Manager has not received the Service description in the form of an XML[2] file from an administrator, the Service must describe itself upon start-up.

[2]eXtensible Markup Language

Connectors

In the use case *EstablishCommunication*, the middleware needs to be able to deal with the various communication protocols the Services use.

Since the DWARF middleware is supposed to be extensible to support many different protocols, we have encapsulated the functionality for establishing and managing communication into *Connectors*.

The middleware creates Connectors when Services want to communicate with one another. These allocate and configure the necessary communication resources on both ends of the connection. The Services can access the communication resources (event channels, shared memory blocks) through the Connectors, and use them to communicate with one another. This is shown in figure 4.12.

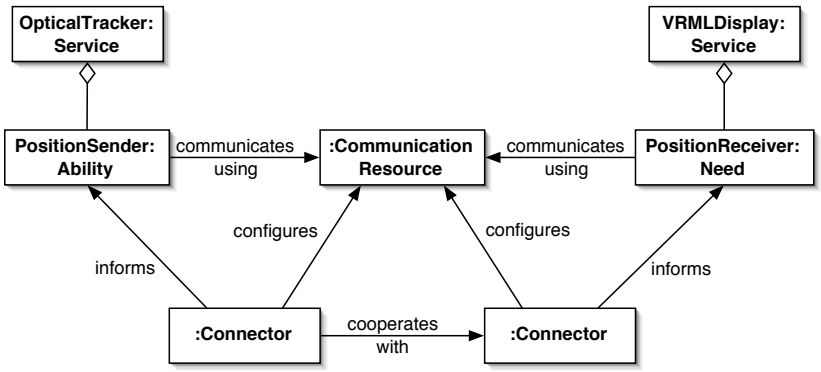


Figure 4.12: Communication between Services using Connectors.
A VRML display receives Position Data from an Optical Tracker. The data is sent using some communication resource (such as an event channel). This is configured by two cooperating connectors. The two Services receive information about the communication resources from their Connectors, so they do not have to know each other directly.

The abstract Connector interface encapsulate different protocols over sub classing and makes the middleware much more extensible, since protocol-specific connectors can be added later. Connectors also reflect the control flow in the use case *EstablishCommunication* as Services do not actively request connections to other Services. Instead, the middleware establishes a connection and delivers information about this connection to the Service. This information is encapsulated in a Connector.

Active Service Descriptions

When a Service becomes active, the middleware needs to keep track of it. A simple Service Description is not enough for this—we need an *Active Service Description*. This object knows the description of a Service and acts on its behalf. It coordinates the rest of the middleware functionality in satisfying the Needs of a Service and offering its Abilities to other Services. It encapsulates the functionality of the Service life cycle, described on page 104.

4.2.5 System Design

The last section analyzed the requirements for the contract-based peer-to-peer architectural style. Based on these findings this section covers the system design of the DWARF middleware components. From an AR system's point of view, the basic functionality is that of a *mediator* from the *mediator design pattern*, but additional steps were necessary to make this functionality available in a distributed system.

Distributed Mediating Agents The use of the mediator pattern for the middleware in the DWARF system has one obvious disadvantage for distributed systems: the mediator has to know all other subsystems that it connects. The consequence is that it could become a central component that cannot easily be distributed onto different network nodes. This would limit the user's ability to turn off arbitrary hardware components—if the mediator is turned off, the whole system falls apart. It would also break the *robustness* design goals.

The solution for this problem is to extend the mediator to a *Distributed Mediating Agent*.

In software technology the term *agent* is used in different ways. Here, we use it to mean software components running on the various computers in a network that proactively communicate with one another on behalf of other components.

It turns out that the functionality of mediating between the DWARF Services can be achieved just as well by distributed mediating agents as by a central mediator. These local mediating agents obviously have to communicate with one another, on a peer-to-peer basis, in order to set up communication between Services running on different nodes of the network. Services that use the mediating agents see these agents collectively as a single mediator.

The middleware that we have designed is active, in that the mediating agents proactively search for other Services, create connections, start and stop Services.

Subsystem Decomposition

The Mediating Agents of the DWARF middleware can be divided into three distinct subsystems for service management, location, and communication as shown in fig-

ure 4.13.

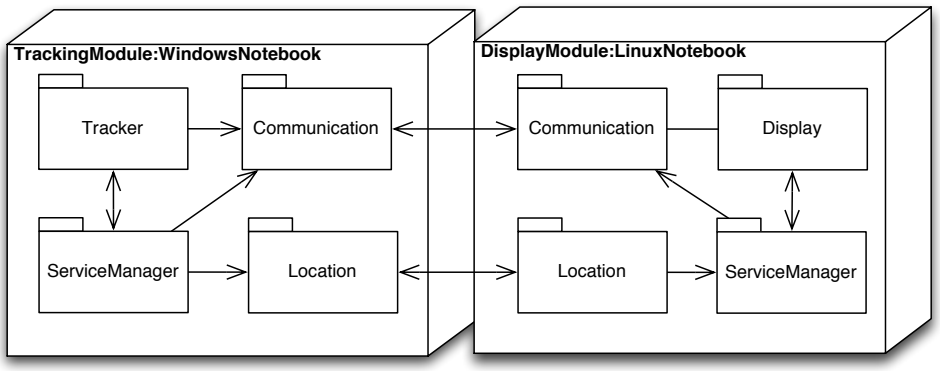


Figure 4.13: Subsystem decomposition of the DWARF middleware.
On each DWARF system there are a Service Manager, a Communication, and a Location subsystem. The DWARF Services communicate only with the Service Manager which acts as a facade. The display needs position data, which the tracker can deliver. The Communication and Location subsystems communicate across network boundaries, the Service Manager and the DWARF Services do not.

Communication Subsystem Encapsulates the functionality of communication between known Services. This includes network communication, but also interprocess communication on single machines.

Location Subsystem Advertises and finds Services that are distributed over the network, so that they can communicate with one another. Note that the term *location* refers to logical locations such as network addresses, not to physical *positions*.

Service Manager Maintains descriptions of active and inactive Services. Starts and stops Services on demand. Coordinates between communication and location subsystems and the DWARF Services.

There are several reasons for the decomposition into three subsystems.

The middleware has two main functions, finding Services and letting them communicate. These two functions have widely different requirements: communication has to be *fast* (especially for AR), whereas location has to be *flexible*. By using two different subsystems for communication and location, these competing functionalities are separated into distinct subsystems. This makes it possible to optimize separately for speed

and for flexibility, and to use different existing components for Service location and communication between Services, e.g. the Service Location Protocol and the CORBA Notification Service.

Separating these two subsystems from the rest of the middleware also separates the subsystems that use network communication from those that do not. Thus, only the communication and location subsystems have to deal with low-level network communication and problems such as loss of connectivity. The Service Manager and the other DWARF Services only communicate locally, making them both simpler and more reliable.

The rest of the middleware is part of the Service Manager. It needs to coordinate between the DWARF Services, the communication subsystem and the location subsystem. This includes managing Service descriptions, starting Services, and so on. In the future, it might be desirable to further decompose the Service Manager into smaller subsystems (such as storage of Service Descriptions).

The objects of the models in section 4.2.4 are distributed into the three subsystems. The location subsystem does not contain any problem-domain objects, since the decision to create a separate location subsystem was made after requirements analysis.

Communication Subsystem

The communication subsystem (figure 4.14) encapsulates the functionality of communication between Services. It manages communication resources such as event channels, shared memory blocks and TCP[3] sockets.

The design of the communication subsystem follows the *abstract factory* design pattern. Connectors for various communication protocols are created by *Connector Factories*. There is one Connector Factory for each communication method, for example, one for the CORBA Notification Service and one for the exchange of CORBA interfaces. The Connectors have a common interface, but specialized Connector Factories may have additional protocol-specific interfaces. A Service must know the protocol-specific interface of the particular Connector type to be able to use it.

Connectors The communication subsystem, i.e. a Connector Factory creates Connectors when instructed to do so by the Service Manager. A Connector manages communication resources at one end of a connection between one Service's Need and another Service's Ability. The Service Manager can actively tell the Connector to connect to another Service, or it can tell it to passively wait for incoming connections. Once a connection has been established, the Service Manager passes the Service's Need or Ability a reference to the Connector.

The Service's Need or Ability then determines the protocol that the Connector supports, and extracts a reference to the protocol-specific communication resources from

[3]Transmission Control Protocol

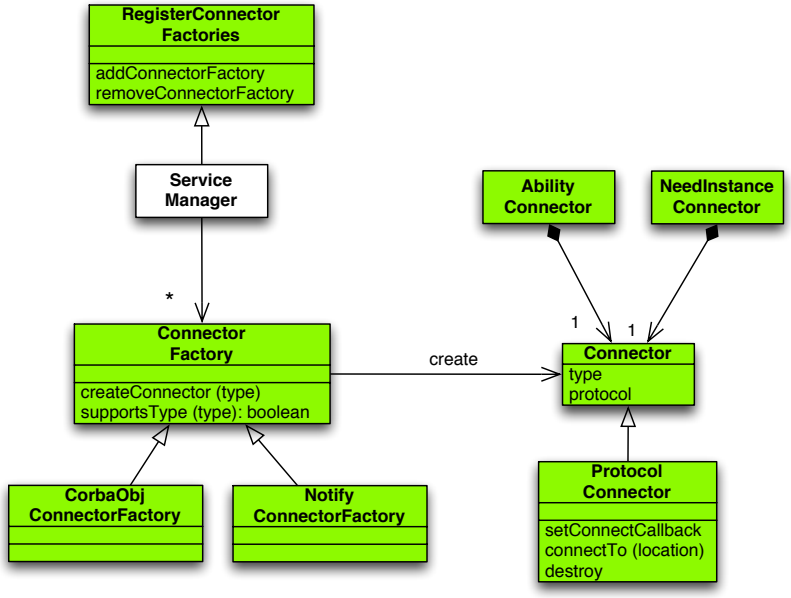


Figure 4.14: Communication subsystem.
The classes of this subsystem are highlighted. A protocol-specific Connector Factory creates Connectors of the supported type. For each connection side, an Ability Connector and a Need Instance Connector respectively hold the Connection instance in behalf of an Ability or a Need respectively. Connector Factories that can establish connection must register with the Service Manager over the Register Connector Factories interface.

the Connector. It then accesses these communication resources directly, bypassing the Connector, to communicate with the other Service. The Service thus must be able to "speak" the protocol itself, but it does not have to deal with managing the infrastructure, e.g. allocating event channels or shared memory blocks. DWARF Services can access non-DWARF systems and vice versa, as long as a standard communication protocol, e.g. HTTP[4], is used.

A Service only receives one Connector per protocol for an Ability, even if more than one other Service connects to it to use that Ability. In contrast, a Service receives multiple Connectors for a Need if the Need is satisfied by connecting to multiple other Services.

The data flow between two Services goes directly through the communication re-

[4]Hypertext Transport Protocol

sources (event channels), once it has been set up by the Connector. The DWARF middleware thus causes no extra overhead once the connection has been established. Figure 4.3 illustrates this concept. This makes communication flexible and fast at the same time.

Since Connectors can be created before the Service is running, the middleware can accept connections on behalf of a Service and only then start the Service, saving resources.

Location Subsystem

The location subsystem (figure 4.15) provides the basic functionality of locating Services. This level of abstraction allows to use existing Service location technology (e.g. SLP) for the implementation.

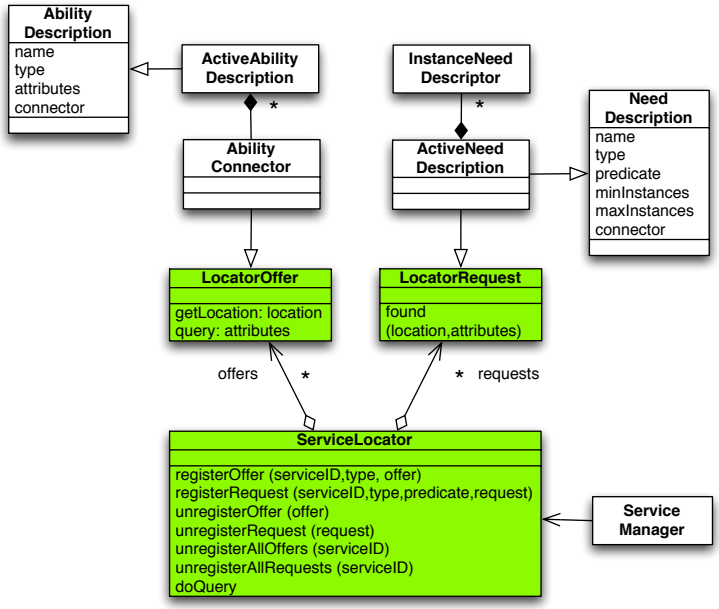


Figure 4.15: Location subsystem.
A Service Locator interface must be implemented by every component that provides Service lookup. It manages Service offers and Service requests in form of Locator Offer and Locator Request interfaces. These are call-backs to query for provided attributes and to inform about matching offers.

Thus, the location subsystem does not know about communication protocols, con-

nectors, Services, Needs, or Abilities. It simply deals with *Offers* and *Requests*.

Offers consist of a location and a set of attributes, and

Requests are predicates over these attributes.

The location subsystem periodically tries to find Offers to match its Requests, either from among its own Offers or by using network service discovery mechanisms to find Offers that other location subsystems have advertised.

The Service Manager creates Offers and Requests from each Service's description and registers them with the Service Locator. It maps a Service's Abilities onto Offers, telling the location subsystem to advertise these on the network. Analogously, it maps a Service's Needs onto Requests, which the location subsystem tries to answer. An Offer is registered as Locator Offer, a Request as Locator Request which are call-backs to query for provided attributes and to inform about matching offers.

The communication protocol between the location subsystems is one main method of communication between the mediating agents on different network nodes. The two Service Managers do not communicate directly with one another. An example for a Service Locator is a wrapper service around an SLP Directory Agent.

The simple model of Offers and Requests that the location subsystem uses makes it easier to implement, because existing service location technologies use this model. Also, the location subsystem's functionality is broad enough so that the Service Manager itself does not have to match up Needs and Abilities, simplifying the design of the Service Manager. Direct communication between the Service Managers of different mediating agents would not provide large enough benefits to make it worth the additional complexity.

Service Manager

For the interface of the Service Manager we used the *Facade design pattern*. The interface of the Service Manager provides the central access point of the DWARF middleware for DWARF services. When a DWARF Service starts up, it must find the Service Manager, but once it finds that, it can access all other middleware functionality over the Service Manager, which provides operations to describe DWARF Services, register and connect them.

Figure 4.16 shows the layout of the Service Manager subsystem. The Service Manager holds a set of Active Service Descriptions, as identified on page 95. An Active Service Description exists throughout the life cycle of a Service and represents the Service within the Service Manager. It holds a state which reflects the state of the represented Service (as shown in 104).

Coordination The Service Manager coordinates the other subsystems. It creates Requests and Offers in the location subsystem to satisfy a Service's Needs and to make

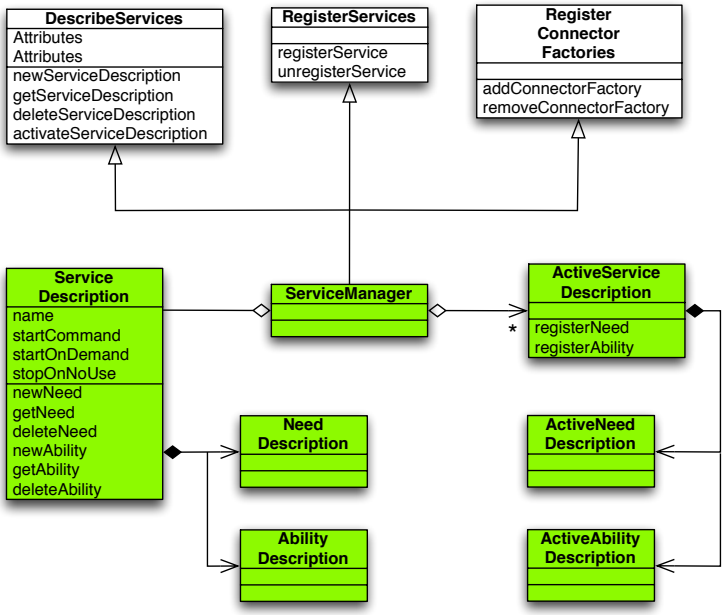


Figure 4.16: Service Manager subsystem.
The Service Manager is the facade of the subsystem and implements the interfaces for the middleware functionality: Describe Service to create new Service Descriptions, Register Service to activate already described Services and Register Connector Factory to register a new Connector Factory. Inactive Services, i.e. described but not yet registered Services, are managed in a list of Service Descriptions, registered Services in a list of Active Service Descriptions.

its Abilities available to other Services. It instructs the communication subsystem to connect the Services together that the location subsystem locates.

Description and Registration An administrator can describe a Service before the Service is started. Services are described by creating Service Descriptions, which describe a Service's attributes, its Needs and Abilities, and the communication protocols it supports.

When a Service starts, it registers itself with the Service Manager, which then associates the running Service with its stored description.

Starting and Stopping The Service Manager can create connectors for a Service's Abilities before the Service is started. When such a Connector is connected to, the Service Manager can start the Service, potentially loading it across the network. When all other Services have disconnected themselves from the Service's Abilities, the Service Manager can shut down the Service. This start-on-demand and stop-on-no-use mechanism conserves resources.

Dynamic Models

We explain the collaboration between a DWARF Service and the middleware with UML collaboration diagrams. The first diagram shows the course of action for a Display Service that needs Position Data. The second diagram shows the opposite side, a Tracker Service that provides Position Data.

In the first example, the display has not been described to the middleware yet. This description can be done by the administrator in advance or it must describe itself after the registration. In the second example, the tracker is described to the middleware by an administrator so it can be started on demand.

Note that these two scenarios are complementary: when Alice starts the display, the middleware finds and starts the tracker automatically. This could be modelled in one large collaboration diagram, but to keep things simple, we have used two smaller ones.

Note also that the interactions shown here do not include the details of locating a Tracking Service, which are internal to the middleware. Internally, the Service Manager calls an external location service that implements the ServiceLocator interface to look up for a registered Tracking Service. In our case this is an SLP Directory Agent.

Display Needs Position Data Figure 4.17 shows how a viewer that feeds a head-mounted display starts up and receives position data from a tracker.

1. Alice starts the display Service.
2. The Service describes itself to the Service Manager, which creates an Active Service Description object for the Service.
3. The Service registers itself with the Service Manager.
4. After having found an appropriate tracking Service to provide position data for the middleware, the Active Service Description creates a Connector to handle the communication.
5. The Connector allocates an event channel and configures it to receive position data from the tracker's event channel.
6. The Active Service Description gives the display's Need a reference to the Connector.
7. The display retrieves a reference to the event channel from the Connector.
8. The display's `PosReceiver` connects itself to the event channel.

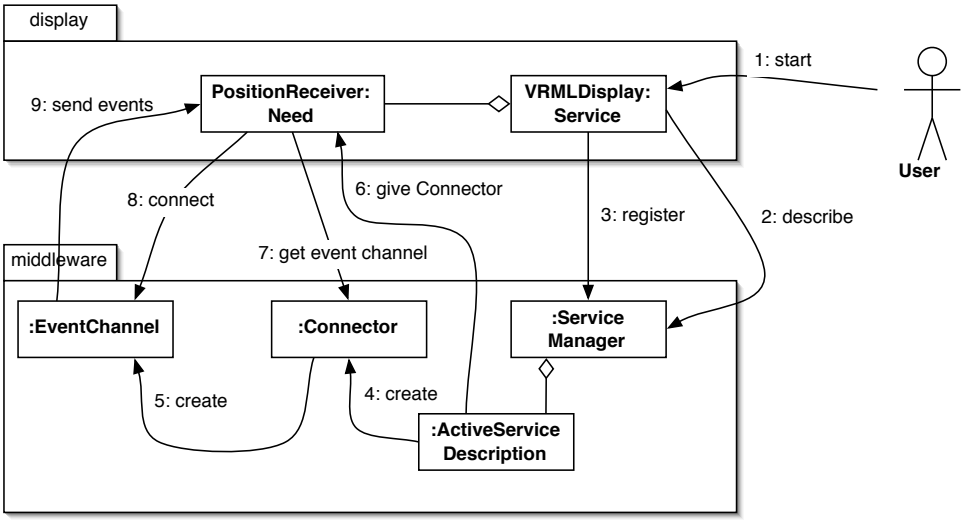


Figure 4.17: Example interaction between a display service and the middleware (UML collaboration diagram).

9. The event channel forwards incoming position events to the display.

Tracker Provides Position data Figure 4.18 on the following page shows how an optical head tracker is started on demand and supplies position data [5].

1. Joe, the administrator, describes the tracking Service to the middleware.
2. The optical tracker's Active Service Description creates a Connector to handle incoming communication requests.
3. The Connector creates an event channel and configures it to receive position data from the tracker.
4. The Active Service Description, having been notified that another Service needs position data, starts the tracker Service.
5. The Service registers itself with the Service Manager, which associates it with its Active Service Description.
6. The Active Service Description gives the tracker's Ability reference to the Connector.
7. The tracker retrieves a reference to the event channel from the Connector.

[5] For simplicity, this tracker is modelled as having no Needs itself—in reality, however, most DWARF trackers need at least a World Model to know what they are tracking.

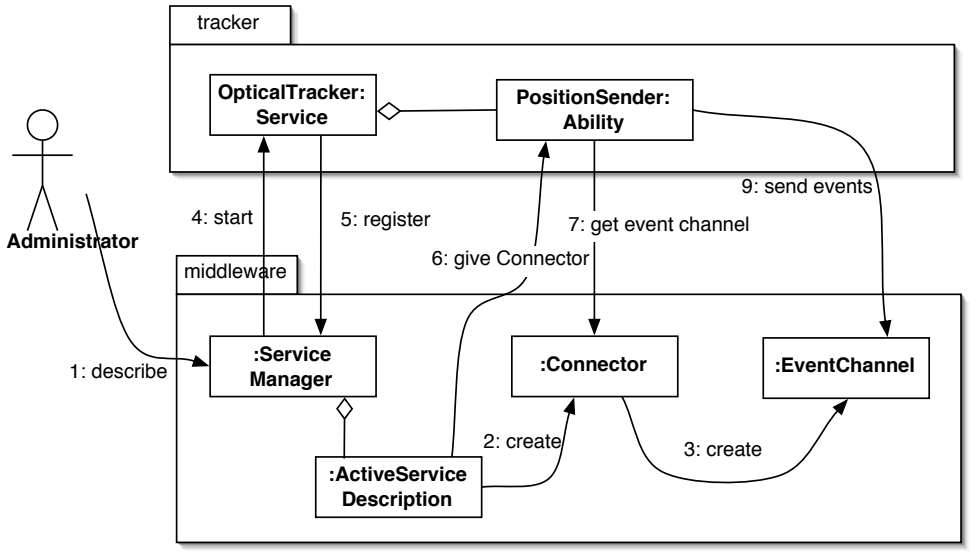


Figure 4.18: Example interaction between a tracker Service and the middleware (UML collaboration diagram).

8. The tracker's `PositionSender` sends its position events to the event channel.

State diagram of a DWARF Service DWARF Services can be in several states, the stages of the *service life cycle*. In each state of the life cycle they have different requirements of the middleware.

The states of the life cycle are shown in figure 4.19.

Undescribed The middleware does not know anything about the Service because it has not been described yet.

UndescribedLoaded The user has loaded the Service, but this Service has not been described yet. The Service describes itself to the middleware.

Inactive An administrator has described the Service to the middleware, but some of the Service's Needs are not yet satisfiable. The middleware tries to find Services to satisfy the Service's Needs.

LoadedManually The user has loaded the Service manually, and it has been described, but some of the Service's Needs are not yet satisfiable. As soon as the middleware has found other Service to satisfy the Service's Needs, it establishes the appropriate communication and starts the Service.

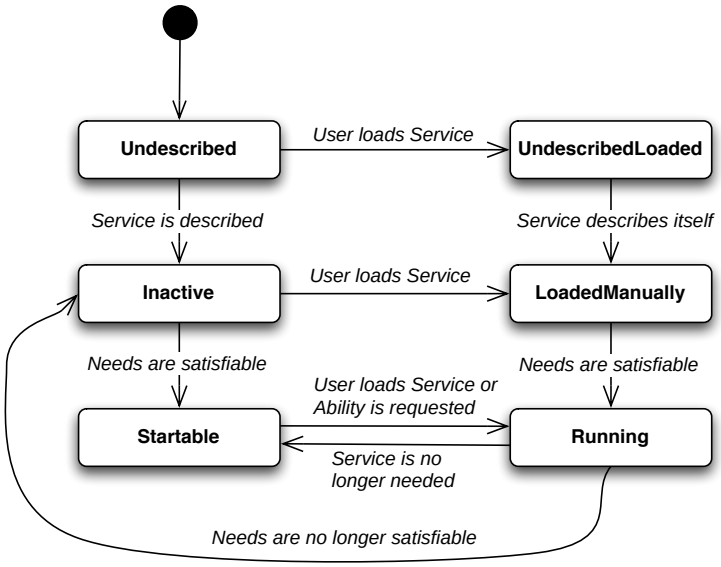


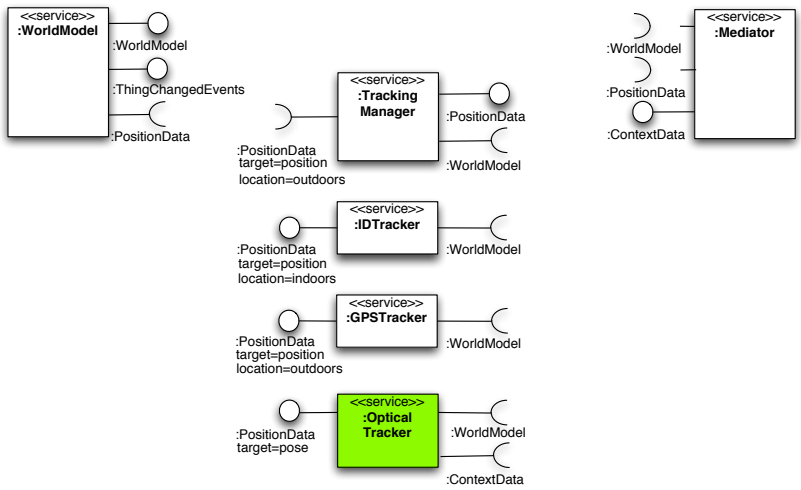
Figure 4.19: Dynamic model of a DWARF Service (UML state diagram).

Startable The middleware has found other Services that can satisfy this (not yet loaded) Service's Needs. As soon as an Ability of this Service is requested, the middleware establishes the communication to satisfy the Service's Needs, loads the Service and starts it.

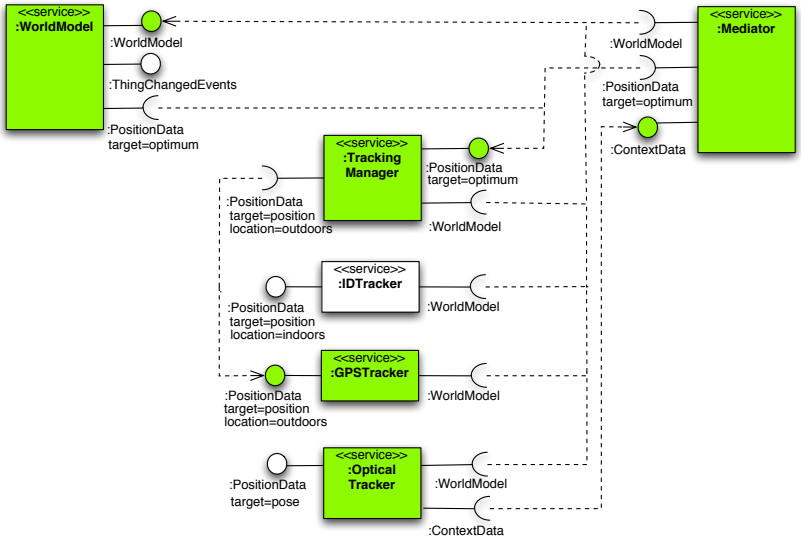
Running The Service is loaded, the middleware has satisfied its Needs, and told it to start running. When the Service is no longer needed, or when its Needs are no longer satisfiable, it stops.

Traversing Service Dependencies In finding, starting and arranging communication between Services, the middleware creates and traverses a *dependency graph* between Services. Since locating other Services happens before actually using them, the middleware must traverse this dependency graph twice, once for locating and once for starting. In the first traversal, a possible configuration of Services using other Services is found, but Services that are not running are not actually started yet. In the second traversal, the Services are started one by one as they actually start using one another and communicating.

An example of the setup of the Service dependency graph is shown in the two parts of figure 4.20. The Optical Tracker is needed by another Service (not shown) and should be activated (figure 4.20(a)). This requires to set up the Service dependency graph.



(a) Interdependent Services before the setup of the Service dependency graph



(b) Interdependent Services after the setup of the Service dependency graph

Figure 4.20: Setting up the Service dependency graph

.

The Optical Tracker needs a World Model and Context Data which can be delivered by the Meditator Service; the Mediator Service also needs a World Model and additionally optimal Position Data (from the Tracking Manager); the Tracking Manager requires raw Position Data from an outdoor tracker; the GPS Tracker is such an outdoor tracker and needs a World Model. Before the first traversal of the Service dependency graph, all Services are located. After the first traversal, the Service dependency graph is set up internally. This means that the middleware holds all possible connections between Services. To activate the Optical Tracker, the middleware does a second traversal in which it activates the required Services and establishes connections between them (figure 4.20(b)).

The two-phase traversal, setting up an internal graph and actually starting Services, allows browsing for potential Services while they are not started yet. In the example, the ID Tracker is not connected, since the Tracking Manager prefers to use the outdoors GPS Tracker.

4.2.6 Hardware/ Software Mapping

The DWARF middleware was developed on top of the CORBA middleware. This allows to deploy the DWARF middleware on various hardware platforms with different processor architectures (such as Intel x86, Motorola PowerPC, or StrongARM) with widely different memory sizes and processing power. Since one of the requirements of DWARF was to support collaboration between mobile systems and devices in the environment, in particular AR-ready intelligent environments (section 1.4), the middleware must run on systems ranging from PDA-sized wearables to large servers within a building.

Additionally, different methods of deploying the middleware subsystems onto the hardware are possible—not every middleware subsystem has to be running on the local machine.

Here, we will show different possible configurations of DWARF middleware components with different hardware demands, ranging from high to low demands. The main target platform is a wearable computer but we also want to support smaller devices with less resources.

Different possible hardware software mappings are shown in figure 4.21, clockwise from the top left. The top left platform belongs to the class of server computers, the top right to the class of notebooks, the bottom right to the class of wearable computers, and the bottom left to the class of PDAs.

Fully Local Middleware Deployment Ideally, one Service Manager, one location subsystem, and one full communication subsystem run on each node of a networked system. This way, all communication between Services and the middleware is local, and the Services do not explicitly have to deal with losing network connectivity. This, however, has the highest memory requirements.

Remote Communication Subsystem To be able to use systems with less memory and processing resources, it is possible to move parts of the communication subsystem (such as an event service) onto other, larger, hosts. Local DWARF Services can still access the event service using CORBA—indeed, they will not even notice that the event service is not local, unless the network connection fails.

Note that if the DWARF Services running on a particular wearable system do not use an event service at all, the event service can be removed from the communication subsystem entirely. The communication subsystem is modular by design, as shown in Section 4.2.5.

Remote Mediating Agent This deployment configuration has hardly any local middleware at all. Each DWARF Service only has an ORB[6] for using CORBA. The Services then must be configured to use the mediating agent of another host but they must know that host's network name. This means that the machine must have some form of reliable network connection to the machine whose DWARF middleware it is using, otherwise the local Services will not even be able to communicate with one another.

Use of Low-Level Protocols A system that cannot even accommodate an ORB implementation cannot run DWARF Services in the usual way. It can, however, run Services that cooperate with the rest of the system, similar to the way the DWARF system can use external Services such as printers. So it can use DWARF Services but cannot use the support of the DWARF middleware. To do this, a Service must natively support the low-level protocols that the DWARF middleware uses. This is the Service Location Protocol (SLP) and a communication protocol such as HTTP. Since both SLP and HTTP require hardly any more resources than a TCP/IP stack, this allows very limited systems to cooperate with other DWARF Services. Of course, finding appropriate other Services, establishing connections, and so on all have to be implemented by the Service itself.

4.2.7 Persistent Data Management

In this section, we describe the persistent data stored by the middleware and the infrastructure required to store it.

The middleware hardly needs to store anything persistently, as the distributed mediating agents are designed to be running constantly on each network node. Status information for Services, communication ports etc. are all transient, as the middleware connects Services together dynamically.

The only persistent objects of the middleware are the Service Descriptions. As said in section 4.2.5, the Service Manager describes the Offers and the Requests to the

[6] Object Request Broker

Location subsystem. There are two ways: either a Services describes itself to the Service Manager by calling the methods of the `DescribeService` interface, or the Service Manager reads in persistent Service descriptions. The last method allows the Service Manager to advertise the Abilities of Services that have not been started yet. We have designed the data contained in Service Description so that these can easily be stored in XML files.

Each Service Description is stored in its own XML file. These XML files can reside in a file system directory that the Service Manager has access to. The Service Manager reads the files in this directory upon start-up and regularly checks it for changes, loading any new Service descriptions. This way, installing a new Service is as easy as installing the executable file and copying the XML Service description into the Service Manager's directory.

An XML Service Description contains a description of a Service, its attributes, Needs and Abilities, with the communication protocols they support. The format is specified as an XML DTD[7].

```
<!ELEMENT service (attribute|need|ability)*>
    <!ATTLIST service
        name CDATA #REQUIRED
        startOnDemand (true|false) "false"
        stopOnNoUse (true|false) "false"
        startCommand CDATA ""
        isTemplate (true|false) "false"
    >
    <!ELEMENT attribute EMPTY>
        <!ATTLIST attribute
            name CDATA #REQUIRED
            value CDATA #REQUIRED
        >
    <!ELEMENT need (connector)*>
        <!ATTLIST need
            name CDATA #REQUIRED
            type CDATA #REQUIRED
            predicate CDATA ""
            minInstances CDATA "1"
            maxInstances CDATA "1"
        >
    <!ELEMENT connector EMPTY>
        <!ATTLIST connector
            protocol CDATA #REQUIRED
        >
    <!ELEMENT ability (attribute|connector)*>
        <!ATTLIST ability
            name CDATA #REQUIRED type CDATA #REQUIRED
        >
```

Listing 4.1: XML DTD for service descriptions

[7] Data Type Definition

An example Service description for an Optical Tracking Service is the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="OpticalTracker"
    startOnDemand="false" stopOnNoUse="false"
    startCommand="/opt/dwarf/optical-tracker/bin/optical-tracker">
    <need name="RoomChangedEvent" type="ContextData"
        predicate="hopcount>=0"
        minInstances="0" maxInstances="100">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="World" type="WorldModel" predicate="World=beautiful"
        minInstances="1" maxInstances="100">
        <connector protocol="CorbaObjImporter"/>
    </need>
    <ability name="PositionEventSender" type="PositionData">
        <attribute name="how" value="fine"/>
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
</service>
```

Listing 4.2: Service description for an Optical Tracking Service

In the example above, the Service name is `OpticalTracker`. The Service can be started on demand by the Service Manager (`startOnDemand="true"`), it can also be stopped when it is no longer used (`stopOnNoUse="true"`). It can be started by issuing the given start command, and stopped by sending a kill signal. The Service has one Need (`WorldModel`) and two Abilities (`PositionData` and `VideoData`). Needs and Abilities have a type and a name, where the types are used for Service matching and the names are identifiers of Service instances. The `WorldModel` Need has a predicate and the Service needs exact one `WorldModel` (`minInstances="1" maxInstances="1"`) over a CORBA RPC connection. The `OpticalTracker` wants to play the client part and imports the reference of the `WorldModel` (`CorbaObjImporter`). The declaration of the Abilities is analogous. In the example, both Abilities support two communication means each. Any of two respective two can be used to access this Service.

Service Descriptions are small, as shown in listing 4.2, and thus fit comfortably into text-based XML files. Using XML files showed the best trade-soff between expressiveness and complexity. Binary files would save only a marginal amount of space, and would necessitate extra editing tools. Storing the data in some larger database system would introduce more unneeded complexity. Databases are useful for large amounts of data and a controlled data access. In our system both cases do not apply.

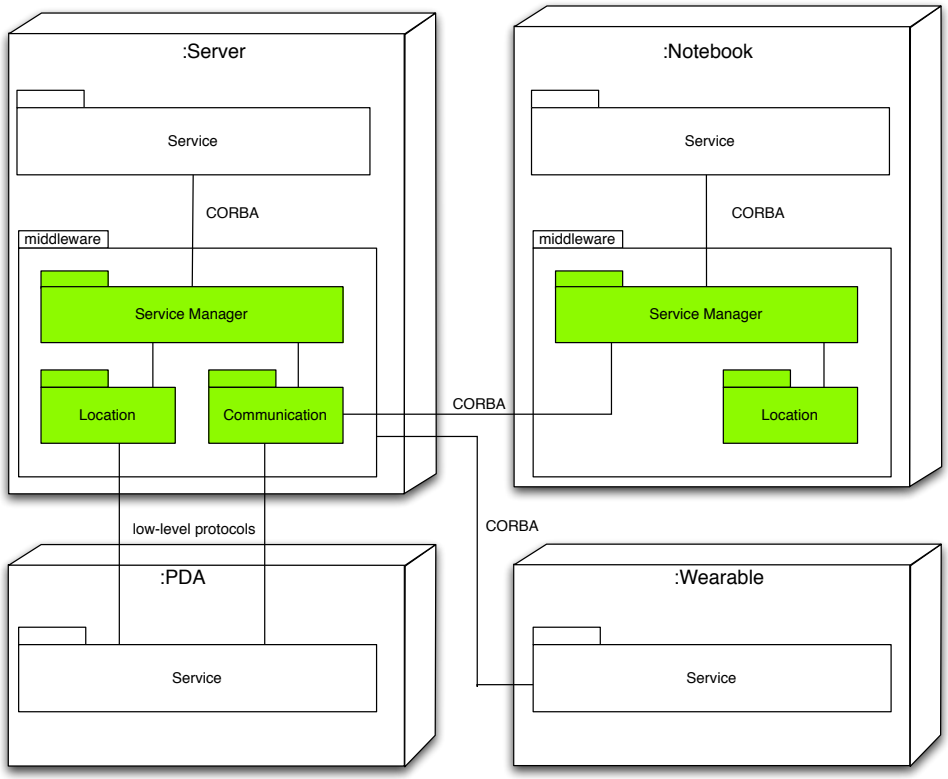


Figure 4.21: Deploying the subsystems of the DWARF middleware in a four machine configuration.
Each machine shown belongs to a different hardware class with a different hardware configuration and different middleware components. Note that the DWARF Services on the wearable computer communicates with the middleware as a whole using CORBA, whereas the Services on the PDA only use low-level protocols.

4.3 A Graphical Notation for DWARF Systems

The development of systems based on DWARF Services must be supported by an appropriate modelling technique. In particular, it must be able to model an individual DWARF Service with Needs, Abilities, Attributes, and Connectors, and it must be possible to model a DWARF-based system consisting of several DWARF Services. Up to now we have used standard UML diagram types such as class diagrams. The goal is to map the model of the contract-based architectural style to a compact graphical notation.

As a starting point we use UML Version 2.0. This is a major revision of earlier UML version and supports *component-based software engineering*. The UML 2.0 specification consists of two parts, the UML 2.0 Infrastructure specification [107] and the UML 2.0 Superstructure specification [108]. The UML 2.0 Infrastructure specification defines the foundational language constructs required for UML 2.0, whereas the Superstructure specification complements it with the user level constructs

4.3.1 DWARF Service Modelling

UML 2.0 provides a new diagram type, component diagrams, for the specification of components. Note, that component diagrams and components are introduced for system modelling and differ considerably to components in UML 1.x deployment diagrams. In UML 1.x components are implementation artefacts such as executables or libraries deployed on hardware nodes. In UML 2.0 a component is "*A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behaviour in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics)*" [108, p. 6].

A UML 2.0 component diagram consists of components. Each component can be realized by other components or classes. A component specifies its external access points by ports. And a port is a set of provided and required interfaces. Ports can be used to specify the mapping of external interfaces to internal realizing components. If it is not the goal to model the internals of a component, then the ports can be left out of the diagram and the interfaces are connected directly to the component.

The example DWARF Service in section 4.2.4 could be modelled in UML 2.0 as shown in figure 4.22. The Optical Tracker Service is modelled as UML component, the Needs as required interfaces with dependency associations, and the Ability as provided interface with implements association.

UML 2 allows a more compact notation with circles for provided interfaces and semi-circles for required interfaces. This allows more compact diagrams when details such as attributes can be left out (figure 4.23).

As we have seen, the mapping is straight forward. Both DWARF and UML specify

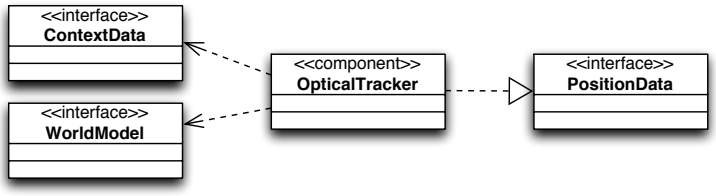


Figure 4.22: Model of an Optical Tracker Service in UML 2.0. The Optical Tracker Service is modelled as UML component, the Needs as required interface with dependency associations, and the Ability as provided interface with implements association.

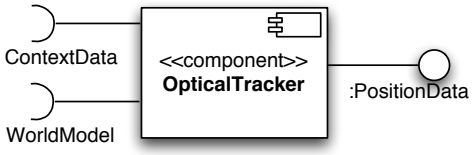


Figure 4.23: Compact model of an Optical Tracker Service in UML 2.0.
This diagram models the same DWARF Service as figure 4.22 but uses the more compact circle and semi-circle notation for interfaces.

the provided and required interaction points of Service and components respectively. The easiest way for the extension of UML diagram types are the use of stereotypes. A stereotype can be used to adapt an existing UML element or to introduce a new one. We use this approach to distinguish the DWARF elements from the elements of UML component diagrams. DWARF Services are denoted by boxes with the stereotype `<<service>>`, Needs by `<<need>>`, and Abilities by `<<ability>>`.

A key feature of DWARF is the use of connectors as first class objects. UML connectors in component diagrams allow to express a relationship between a required and a provided interface. In particular, UML provides *delegation connectors* and *assembly connectors*. A delegation connector models the delegation of a request from the component to a realizing internal component or class. An assembly connector models the connection between a provided interface of one component with a required interface of a second component. Both types of connectors do not allow to express detailed semantics of a connection.

Further on, UML connectors are binary objects, i.e. they can be used to associate two components. They cannot be used to express the possible connector types that one component or Service supports. Here we need a unary object. UML supports the

use of *association classes* to express the semantics of an association between classes or components. An association class is tied to an association between two or more classes and as such it is also a binary object. There is no support in UML to connect interfaces with association classes, besides the fact that interfaces are also classifiers and as such they can have associations of their own. So such an interface/connector combination can be expressed but this is not very elegant.

Instead we use the following concept: DWARF Connectors are modelled as classes like association classes. To express the relationship between a Service and a Connector we use the compact notation UML already uses for *templates*. A template is a parameterized element, i.e. one or more parameters are unbound. DWARF Connectors are bound to Needs and Abilities, which we model as templates with the connectors as parameter. Graphically we express it as dashed box for the Connector located at the upper right corner of a Need or Ability box. Figure 4.24 shows the same example with the Optical Tracking Service with the new notation.

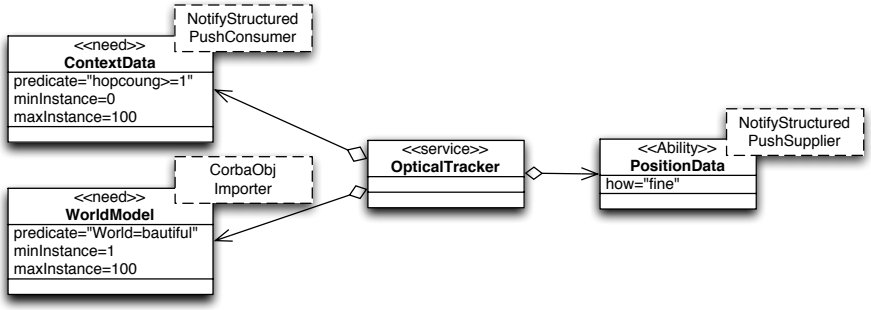


Figure 4.24: Model of an Optical Tracker Service with DWARF UML extension for Connectors. Each connector is modelled as dashed box in the upper right corner of the associated Need or Ability.

Similar to the compact circle and semi-circle notation for compact diagrams we can use circles and semi-circles with dashed boxes for the connectors (figure 4.25). For overview diagrams with several DWARF Services the dashed boxes may be left out.

4.3.2 System Modelling

A DWARF system consists of a set of Services that are connected via Needs and Abilities. Again, we use UML 2.0 component diagrams as basis. Component diagrams provide connectors to connect individual components. Graphically, a connector is drawn as a circle for a provided interface encapsulated by a semi-circle for a required interface. A circle and a semi-circle can also be connected over a dashed arrow for

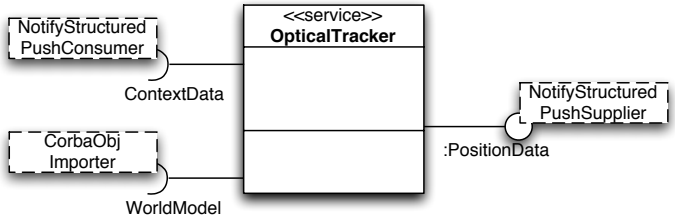


Figure 4.25: Compact model of an Optical Tracker Service with DWARF UML extension for Connectors. Each connector is modelled as dashed box in one of the upper corners of the associated Need or Ability.

dependency directed from the semi-circle to the circle. This expresses that the required interface depends on the provided interface.

We use the same concept to model the connection between a Need and an Ability. As we explained, Needs can be drawn as semi-circles, Abilities as circles. In a system model we draw DWARF Services as boxes with the compact circle and semi-circles notation for Abilities and Needs and draw a dependency arrow from a Need to a collaborating Ability. As an example see figure 4.26. This is a section of the system model for the DWARF Pathfinder system we will present in chapter 5. Such a model describes the communication relationships within a DWARF system. Therefore we call this view onto a DWARF system as the *communication view*. A more complex example is the communication view of DWARF Pathfinder which can be found on page 130.

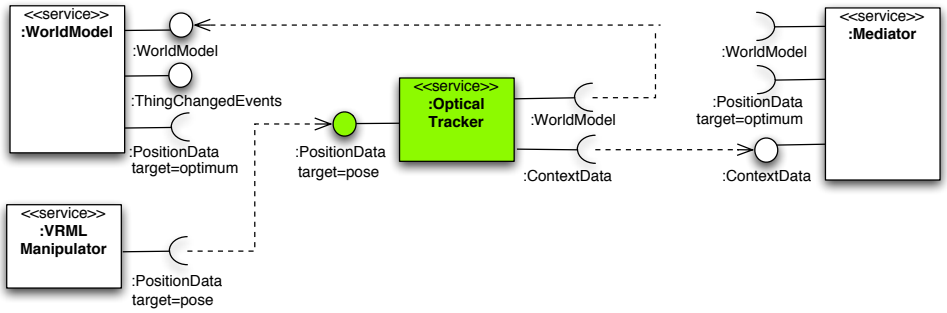


Figure 4.26: The integration of the Optical Tracker Service via Needs and Abilities. To retain a compact model we use circles and semi-circles to model Abilities and Needs. A Need/Ability pair is connected by a dependency association expressed with a dashed arrow.

4.4 An Example for a Customized DWARF Service

Developers of new DWARF Services must implement the `Service` interface. For each Need and Ability there must be an associated class that implements the interface `Need` and `Ability` respectively with the appropriate Connectors. Figure 4.27 shows an example for an Optical Feature Tracker Service in a UML class diagram. This Service has two Needs and one Ability with one supported Connector each. The Need for a World Model has a CorbaObjImporter Connector, The Need for Context Data a NotifyStructuredPushConsumer Connector, and finally the Position Data Ability has a NotifyStructuredPushSupplier Connector.

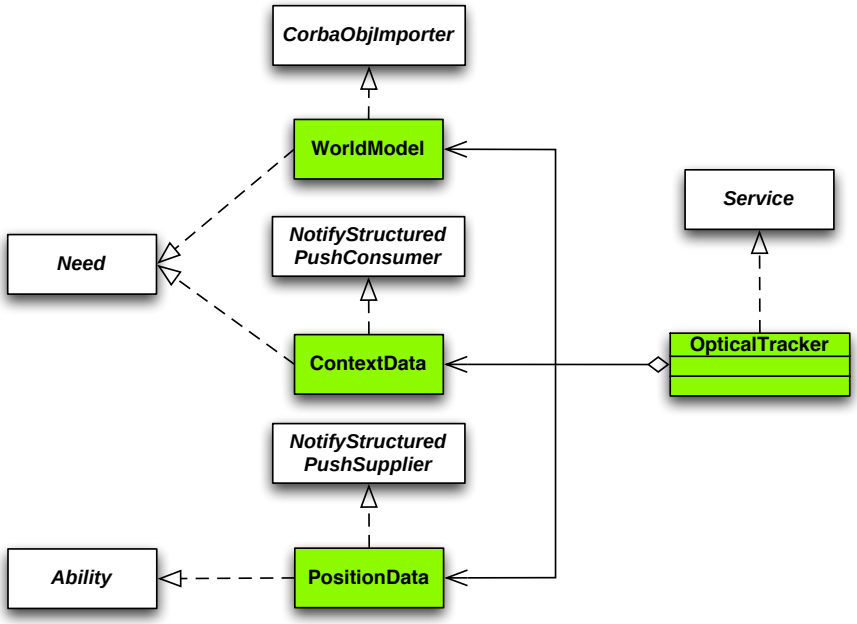


Figure 4.27: Customized DWARF Services: Example Optical Tracker Service. This Service has two Needs and one Ability with one possible Connector each. The Need for a World Model has a CorbaObjImporter Connector, The Need for Context Data a NotifyStructuredPushConsumer Connector, and finally the Position Data Ability has a NotifyStructuredPushSupplier Connector.

This is the ideal solution. To simplify the development of prototypical Services, a new Service can implement the interfaces for Needs, Abilities and all supported Connectors on its own. Instead of delegates for each Need and Ability it implements

the interfaces for Needs, Abilities, and Connectors and registers itself as the respective implementing object with the Service Manager. For technical reasons resulting from the implementation with CORBA, a new Service cannot implement each interface directly but the interface `ServiceAndNeedAndAbilityIsNotifySupplierConsumer`. Figure 4.28 shows the model of the simplified version of the Optical Feature Tracker Service.

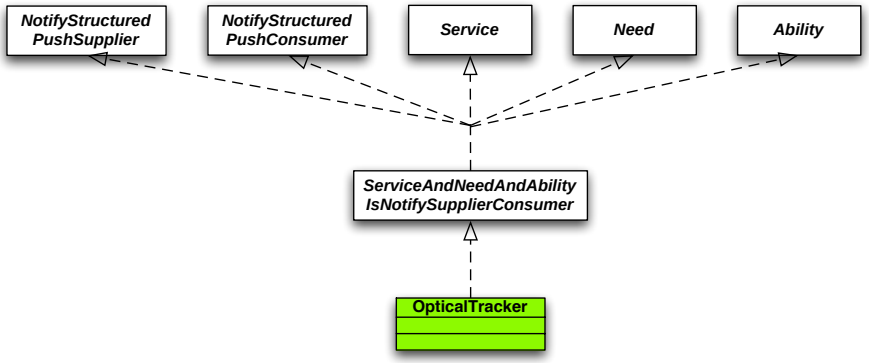


Figure 4.28: With the simplified way to build a customized Service, the new Service must implement the interfaces for Service, Needs, Abilities, and all Connectors on its own. For technical reasons resulting from the implementation with CORBA, a new Service cannot implement each interface directly but the interface `ServiceAndNeedAndAbilityIsNotifySupplierConsumer`.

4.5 Conclusion

In this chapter we presented a contract-based peer-to-peer architectural style for distributed and adaptable systems. This pattern describes a model for system configurations of peer-to-peer distributed services. It is a specialization of the Peer-to-Peer pattern [24]. In the contract-based variant pattern peers collaborate on the base of contracts between peers and the environment. An active middleware is described that mediates between the DWARF Services and connects them for collaborations. Connections between services are first-class objects at design and runtime. This allows to change connections and adapt a system at runtime. Finally we presented a graphical notation for this style based on UML 2.0 component diagrams.

5 A Case Study for the DWARF Framework

The M^{3}ARF sub framework of DWARF for mobile AR maintenance systems and the DWARF Pathfinder application.

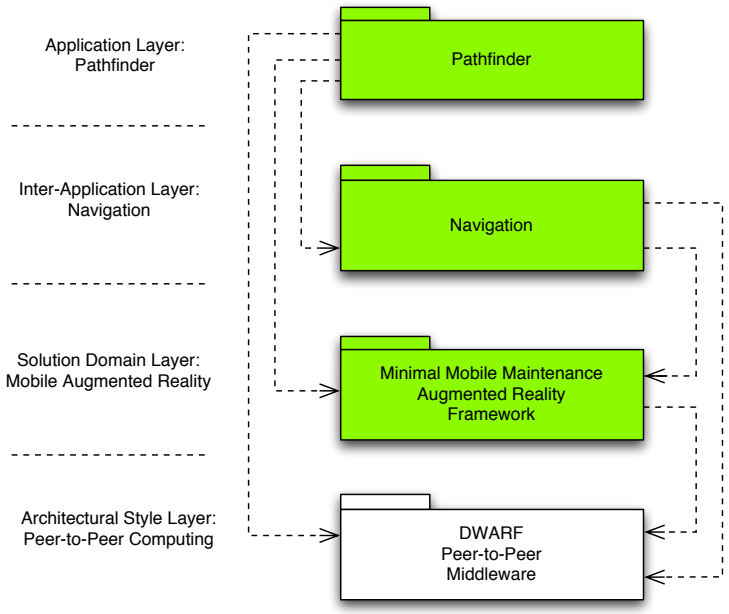


Figure 5.1: This chapter covers the application layer, the inter-application-layer, and the solution domain layer.

The abstract Augmented Reality architecture and the system of Augmented Reality patterns introduced in the previous chapters can be used to design application domain

specific sub-frameworks for particular Augmented Reality systems. In this chapter we describe such a sub-framework for mobile Augmented Reality maintenance systems, the *Minimal Mobile Maintenance Augmented Reality Framework (M³ARF)*, and a demonstration application called DWARF Pathfinder. M³ARF supports the rapid prototyping of Augmented Reality-supported maintenance applications as described in section 2.1 with a core of required Augmented Reality Services. It includes several Augmented Reality patterns that are particularly applicable for rapid prototyping.

The purpose for building Pathfinder was to demonstrate M³ARF in a coherent application [14]. It covers parts of the motivating scenario in a modified form. Instead of guiding a worker to a machine, Pathfinder guides a tourist over a campus, and instead of overlaying a machine with maintenance information, Pathfinder overlays printing instructions on a laser printer. The Pathfinder scenario is described in section 5.1.

A second purpose of Pathfinder was the implementation of a first set of reusable DWARF Services upon which other sub-frameworks and applications could be built. We developed each DWARF Service from scratch and often they are a technology demonstrator on their own. We give an overview of the M³ARF framework in section 5.2 and an overview of each subsystem in section 5.3. Details for each Service can be found in appendix B.

The implemented Services include a core of reusable Services for Augmented Reality systems as well as application and domain specific Services. Therefore, this chapter covers the upper three abstraction layers: application layer, inter-application-layer, and solution domain layer. The basis of M³ARF at the architectural style layer is the contract-based peer-to-peer architectural style and the DWARF middleware.

The Pathfinder prototype and the included DWARF Services were developed in several student theses. In this chapter we describe the core principles behind each Service and the integration into the M³ARF framework to provide a coherent view for the reader of this dissertation. Details for the individual Services beyond the description here can be found in the respective student theses. We reference them at the particular Service description.

5.1 The Pathfinder Scenario

A visitor is headed for a meeting in a room on the campus of the Technische Universität München. His mobile Augmented Reality system navigates him to the meeting room and lets him print out his handouts while he is on the way.

We made the choice for the scenario because of three goals:

- The scenario should involve not only all Augmented Reality core services (tracking, interaction, and presentation), but also demonstrate the dynamic use of services,

- it should take place in different types of environments to show the flexibility of the framework,
- and it should be a test driver for all the DWARF Services developed within the Pathfinder project.

The scenario demonstrates a subset of the visionary scenario on page 2.1. Nevertheless, it covers the core features of a mobile Augmented Reality system in a ubiquitous computing environment.

Scenario: **Demonstration Scenario**

Actor instances: `Fred: User`

Flow of Events:

1. Fred is invited to a meeting with some software engineering students at the TU München. He is equipped with a backpack with two laptops, a head-mounted display with an attached digital video camera, a headset and microphone for voice input, a GPS/-compass combination and a Radio Frequence ID (RFID) tracking device. Fred has a handout on one of his laptops, and has already registered the handout to be printed as soon as he reaches the main building of the university. The students Fred is supposed to meet with have told him to take the subway to the Königsplatz station.
2. Fred emerges from the Subway station and walks towards the exit. As he comes within reach of an information terminal on his way, an option appears on his display letting him download personalized navigation instructions to the meeting room. He says "yes" to accept this data transfer. He sees a message that the download is in progress. After a while a message appears saying that the data transmission is complete.
3. On Fred's head-mounted display, a three-dimensional map of the area appears. It shows his own position with a red dot and rotates as he turns, showing his current orientation. A blue arrow indicates his destination. Fred uses this map to guide him to the entrance of the university campus.
4. As Fred reaches the university campus, an option appears to let him send off the print job for his handouts by wireless LAN. Fred confirms this by saying "yes" again.
5. Inside the building, Fred is guided by a schematic two-dimensional map, indicating which room he is currently in (the position is read from RFID tags), to the hallway outside of the meeting room.
6. Here, he sees a red arrow appear in his head-mounted display, pointing to one of two printers, which has printed his handouts.

7. Fred picks up his handouts from this printer, says “ready”, and a three-dimensional blue arrow appears, pointing him to the meeting room.
8. Fred enters the meeting room, takes off his head-mounted display and backpack and greets the students.

In the Pathfinder scenario we can identify one actor, the Pathfinder user, and derive the following use cases: (figure 5.2):

1. receive navigation instructions,
2. outdoor, indoor and in-room navigation,
3. use dynamically found services such as printers,
4. and receive instructions over a HMD.

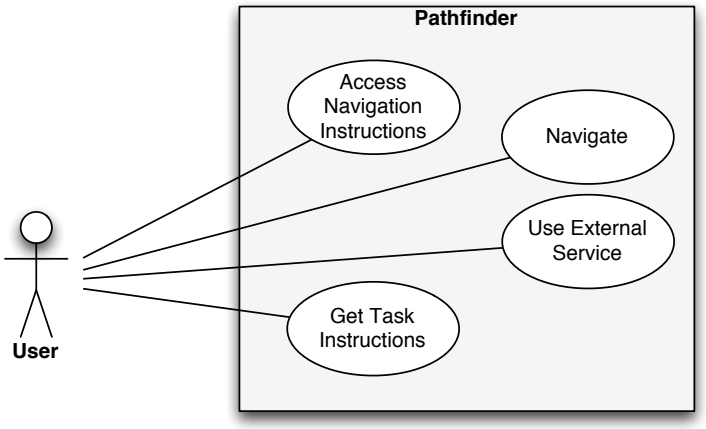


Figure 5.2: The use cases of the Pathfinder application.

5.2 The Minimal Mobile Maintenance Augmented Reality Framework

The purpose of the Minimal Mobile Maintenance Augmented Reality Framework is the support of rapid prototyping of applications such as Pathfinder. As such it must

provide DWARF Services for the Augmented Reality core functionalities tracking and registered video overlay. Therefore we need *Tracking Services* and Services for the *User Interface*. Further on, we need an *Application Service* that implements the application functionality and is responsible for system initialization and bootstrapping. For the navigation functionality we need a Service for the interpretation of navigation instructions and a DWARF Service that provides the access to an external server providing such instructions. And finally, for the dynamic use of external non-DWARF services we need another DWARF Service that manages the service access.

5.2.1 DWARF Services for M^{3}ARF

We can map the use cases of the Pathfinder scenario to the following set of DWARF Services including Services for Augmented Reality for the M^{3}ARF, inter-application Services for navigation, data access, the access to external, non-DWARF services, and application specific Services (ordered by subsystems):

Application. For Pathfinder the Application Service is provided by the *Pathfinder Application Service*. It is responsible to bootstrap the system and coordinate the involved Services. It uses the *Application Component* pattern. This is a typical approach for distributed systems. The application functionality is encapsulated into a distributed component that collaborates with other distributed components.

The navigation instructions are provided by an external information terminal (a PC located near the subway station) in the form of a taskflow model. The instructions are downloaded via the *Bluetooth Communication Service*. The Bluetooth Communication Service is not an Augmented Reality specific DWARF Service but it can be reused for other kinds of systems with local information points for data access. As such it belongs to the inter-application layer of the DWARF framework. We assumed that the information terminal is a black box which can generate navigation instructions on demand, and transfer them via Bluetooth.

The navigation instructions are executed by the *Taskflow Engine*. A taskflow model contains a description of the tasks the user has to do or wants to do. It receives events from the application when a step is performed and then triggers the generation of the user interface for the next step. This allows complex taskflows for maintenance or navigation to be stored in a structured and uniform format. This is an application of the (*Multimedia Flow Description* pattern). The advantage of this pattern is that the content for the Augmented Reality system is described in a high-level declarative language. This allows a faster development of new content. Internally the Taskflow Engine uses a *Web Service* approach that provides the application content from an Internet webserver.

The framework provides mechanisms for the application to select and access external, non-DWARF services based on the current context of the user and the system, such

as geographical location. The selection of a printer for the handouts are managed by the *Context-Aware Packet Routing Service*.

The Services of the application subsystem are covered in more detail in the sections 5.3.2 and 5.3.3).

Presentation and Interaction. The goal of M^{3}ARF's interaction and presentation subsystems was to support different types of viewers depending on the content and to reuse existing viewer components [132]. User interface scenes for initialization, navigation, graphical augmentations, and service selection are rendered in the *User Interface Engine*. The user interfaces for applications built with M^{3}ARF are described at a high level in terms of functions, operations and messages. This description is then converted into actual interface elements for the currently running user interface devices. These devices allow multimodal user interaction, i.e. flexibly combining different user input and output devices depending on the situation and the preferences of the user.

To address the requirements we use a web-based approach and write adaptors to integrate third-party web components with DWARF. The central component was a web browser with plug-ins for different views. In particular, Pathfinder uses a VRML plug-in for 3D graphics controlled over the External Authoring Interface (EAI), a plug-in for speech recognition, and an HTML frame for text and untracked graphics. Thus, Pathfinder uses the *VRML Browser* and *Multiple Viewers* design pattern for output and *Browser Input* and *Input Manager* for input. Technically, a User Interface Engine combines input and output control.

We present the combined Interaction and Presentation subsystem in section 5.3.4.

Tracking. The framework contains support for selected tracking hardware devices as well as the possibility of accessing future devices using the same interfaces. Several *Tracking Services* determine the user's position, both indoors and outdoors and the output of the trackers is accessible by the application, including quality of service parameters such as accuracy and lag. The framework also includes methods for post processing and combining the output of the trackers with filtering and prediction algorithms

M^{3}ARF uses a combination of different tracking modalities. There is no hybrid tracking, each modality covered a specific part: GPS tracking for outdoors, room tracking to track the location within a building, and optical tracking for the near-range. Each tracker worked independently and as part of a distributed system. As a distributed system Pathfinder could use the approach of *Networked Trackers* combined with the approach to use a *Tracking Manager* that coordinates the trackers.

Position trackers are described in section 5.3.5 and the Optical Feature Tracker in section 5.3.6.

World Model. The *World Model* receives a model of the TUM campus and the relevant rooms, which were also downloaded from the *Bluetooth Information Terminal*. A World Model stores all information the system has about its environment. All relevant objects of the real world the system is operating in are represented as objects in the World Model. In addition, virtual objects such as stickies (location-fixed textual tags) that the system uses to augment the user's reality are stored in the World Model.

The World Model for Pathfinder is adapted for navigation scenarios, so it is rather small. Therefore we could use a simple file-based world model. The tracking information was based on a proprietary format for saving information about the environment such as the building and the area, and VRML files for the representation of 3D scenes. The approach is therefore called *Configuration File*

The World Model subsystem is described in section 5.3.7.

The various functionalities provided by the framework are designed to interact automatically. Using a model of the real and virtual world, the tracking information and the taskflow descriptions, the user interface can display relevant information to the user, registered in three dimensions in real time.

5.2.2 Classifying the Services into the DWARF Framework

The identified Services for each subsystem can be classified into one of the four abstraction layers of the DWARF framework. Figure 5.3 illustrates this classification.

The core Augmented Reality Services are part of the M^{3}ARF framework. They are the User Interface Engine, the *World Model*, the *Tracking Manager*, the *GPS Tracker*, the *ID Tracker*, and the *Optical Tracker*.

Additionally there are Services which do not belong to the Augmented Reality core system but can be used for several applications. As such they belong to the inter-application layer. They are the Bluetooth Communication Service for data access, the *Taskflow Engine* for executing navigation tasks, and the CAP Router for external service access.

The Pathfinder application itself on the application layer consists of the Pathfinder Application and the helper Service Mediator.

5.2.3 Mapping M^{3}ARF to the Reference Architecture

The architecture of Pathfinder as M^{3}ARF-based system can be mapped to the Augmented Reality reference architecture. Figure 5.4 shows the mapping of the Pathfinder DWARF Services with several support classes mapped onto the subsystems of the abstract Augmented Reality architecture. This view is called *architectural view*. The diagram shows that M^{3}ARF covers the subsystems of the reference model except the

Context subsystem. Additionally, several classes of the other subsystems in the reference architecture are also missing. For example, there is no Video Mixer class for combining captured video and computer graphics for video see-through Augmented Reality. For details on each subsystem and Service we defer to the following sections on the individual DWARF Services of M^{3}ARF.

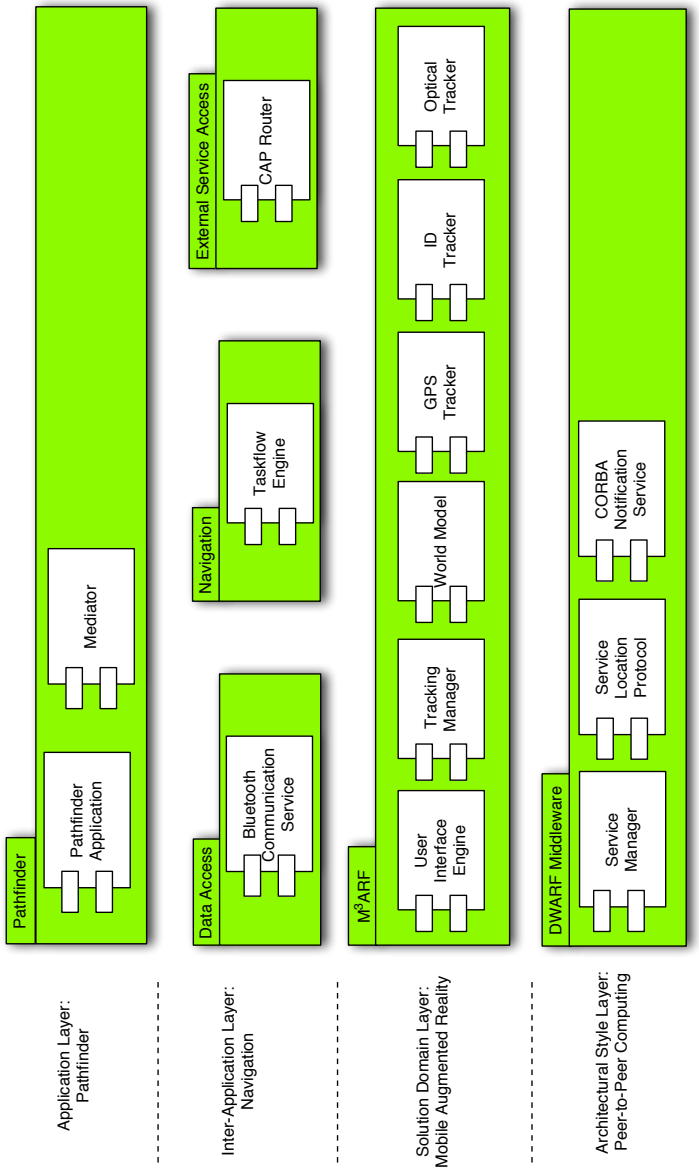


Figure 5.3: The Pathfinder DWARF Services.
The services are allocated to the respective abstraction layer according to figure 1.5. The green coloured packages contain services that are part of M³ARF.

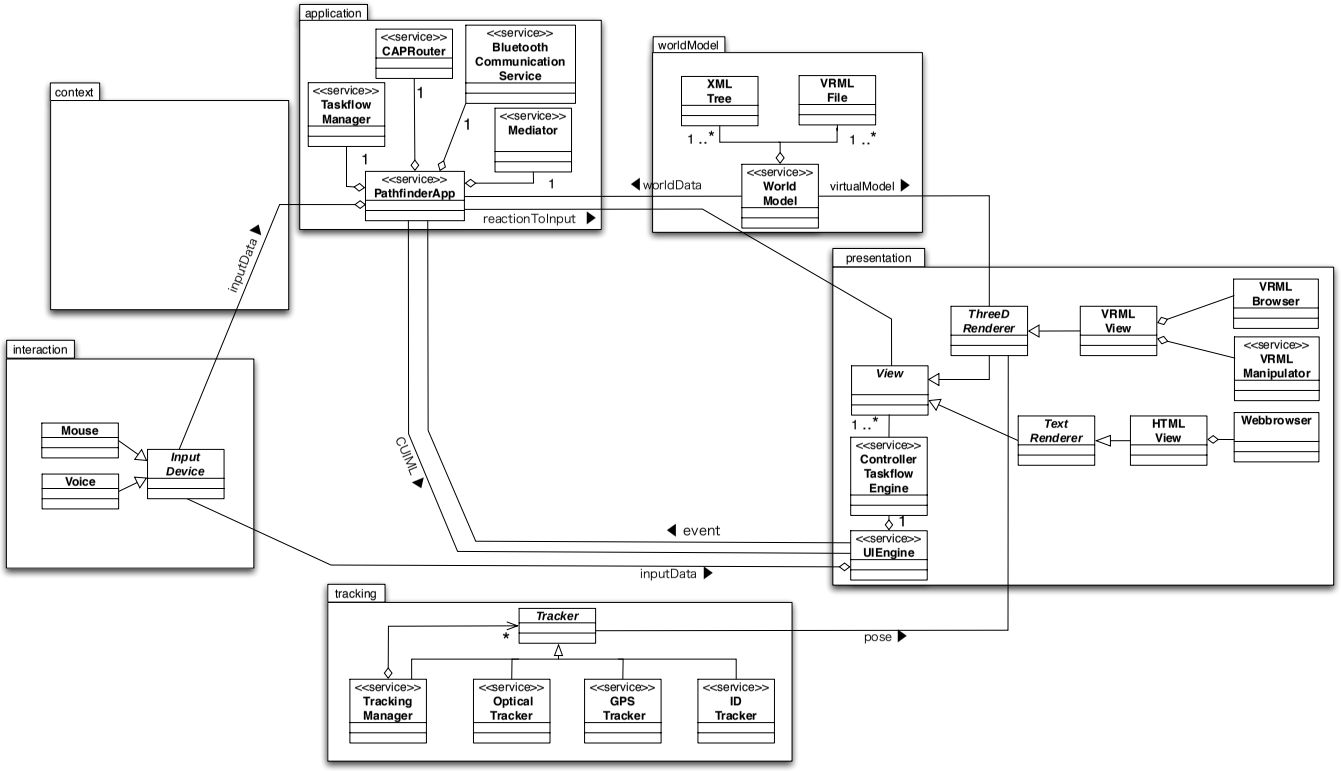


Figure 5.4: The Pathfinder architecture mapped onto the reference architecture.
This diagram shows the distribution of the services among the subsystems. Note that there is no Service in the Context subsystem.

5.3 The Pathfinder DWARF Services

Pathfinder is based on the M^{3}ARF sub framework plus additional application-specific Services. In this section we first give an overview on the whole DWARF system and the connected DWARF Services. We do this with a connection view introduced in section 4.3.1. After that we present each individual DWARF Service.

5.3.1 The Connection View of Pathfinder

The *connection view* in figure 5.5 shows the connections between all the services involved in Pathfinder. In some sense, the connection view shows M^{3}ARF systems from the perspective of the architectural style layer, and the architectural view M^{3}ARF from the solution-domain layer above it.

We go into detail about the individual Services with the specification of the Needs, Abilities and Connectors in appendix B. Here we give an overview on which Service collaborates with which other Services.

For example, the Tracking Manager Service provides the Ability for enhanced Position Data, but it itself has the Need for simple Position Data and a World Model. The Need for simple Position Data can be fulfiled by the Services IDTracker, GPSTracker, and OpticalTracker, the Need for a World Model by the World Model Service. The provided Position Data Ability is used by the World Model Service and the Mediator Service.

5.3.2 Pathfinder Application

Pathfinder Application Service

Some of the application's functionality had to be implemented specifically for the demonstrator:

- bootstrapping functionality that lets the user select (via a scene in the User Interface Engine) navigation instructions to download, initiates the Bluetooth data transfer, and passes the downloaded data to the DWARF Services,
- mediating functionality between the CAP Service and the User Interface, for the confirmation and status reports for the print job,
- generation of *ContextData* events from the tracking data, indicating which room the user is in, which causes the Taskflow Engine to move to the next navigation step,
- and generation of *ContextData* events indicating the availability of the wireless network.

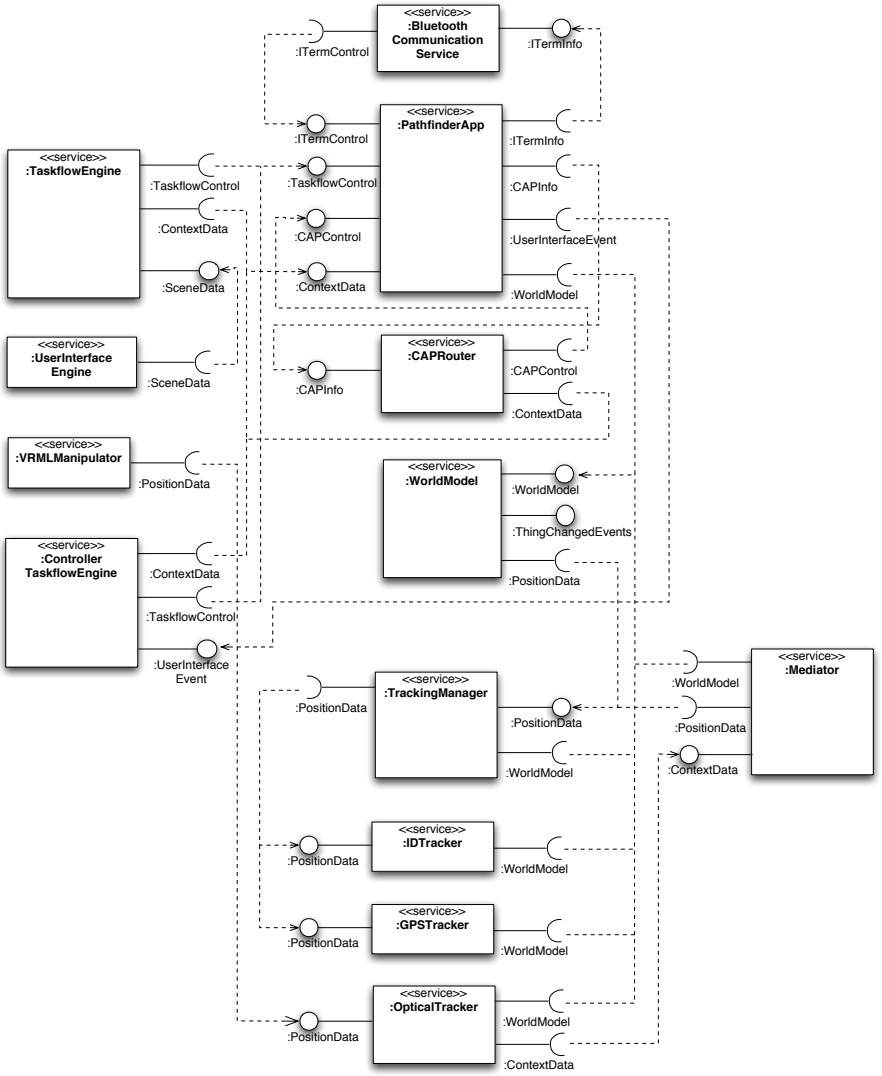


Figure 5.5: Pathfinder connection view.
This view shows the Need/Ability contracts between the individual DWARF Services.

Details on the DWARF Pathfinder Application Service and the helper Mediator Service can be found in [87].

Context-Aware Packet Routing Service

Besides predefined task flows, the user of a DWARF system should be able to spontaneously use external services, such as printing in an unknown environment. For example, in the fist step of the scenario, the user registers a print job to be printed when he enters the campus of the university. This is handled by the Context-Aware Packet (CAP)*context* Routing Service. Here, the user defines a service he would like to use, such as 'print out this document at a printer on the way to the meeting room', and the CAP Routing Service takes care of it.

The major problem for this task is the user's current printer configuration. Even technically simple tasks such as printing can lead to huge problems in unknown computing environments, as a lot of contextual information such as the preferred paper size has to be regarded for successful execution.

The basic idea of the CAP service is to encapsulate such information in packets that are further processed by software devices that route them in a suitable way. For the printing example, all of the user's configuration data, e.g. paper size, preferred colour model etc., are stored in such as packet. The CAP router gathers all information necessary for an optimal fulfilment of the given task of printing from the other DWARF subsystems and executes the print job. The packet also contains the contextual conditions under which the print job should be started. In this case, that means "when the user is within the TUM's wireless network."

Further details on the CAP Service can be found in [92].

Bluetooth Communication Service

The scenario includes an information terminal that allows the user to download location dependent data. To make the interaction with the information terminal as painless as possible for the user, we chose to communicate wirelessly using Bluetooth.

For this, we used the file transfer mechanism of the DWARF *Bluetooth Communication Service*. In the scenario, the mobile system tells the *information terminal* that its user wants to go to the meeting room (and print something along the way), and the information terminal sends a compressed file which contains:

- a geographic and geometric description of the area of the TUM campus, including locations of printers along the way, which is loaded into the World Model,
- a Taskflow which guides the user towards the meeting room step by step and room by room, which is loaded into the Taskflow Engine,

- and abstract user interface descriptions of the navigation scenes to be displayed during this navigation process, which are stored locally so the Taskflow Engine can send them to the User Interface Engine at the appropriate time.

The Bluetooth Communication Service is described in [172].

5.3.3 Taskflow Engine

The basic idea behind the development of the taskflow engine was to provide an easy-to-use possibility for the description of structured flows of tasks. The advantages of this concept arise immediately if we think of maintenance applications using Augmented Reality technologies. Most maintenance tasks are characterized by a fixed flow of steps that have to be performed one after another. This feature is very useful for other application domains as well, such as navigation.

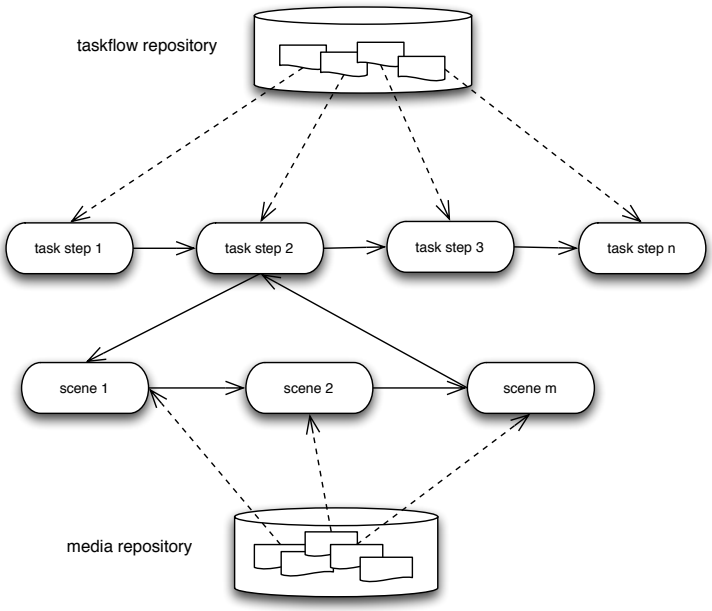


Figure 5.6: General idea of the Taskflow Engine Service.
A taskflow repository contains information about the single steps of a taskflow. For each step several Augmented Reality scenes or other documents provide information for the user.

Initially, the TaskflowEngine Service was specified and developed as part of the ARVIKA Augmented Reality system [118]. Goal of this component is the easier de-

velopment of Augmented Reality content as before. Basic idea is that in industrial domains Augmented Reality-supported work is integrated into a predefined taskflow or sub-taskflow. Typical scenarios from the ARVIKA project such as laboratory experiments, assembly or maintenance schedules follow a pre-planed flow of steps and require a description of this process flow. Examples can be found in Production Planning Systems (PPS) and IETMs [155, 156, 158].

A particular taskflow is specified in an XML-dialect that we developed for that purpose, the Taskflow Definition Language (TDL). At runtime a taskflow description is loaded from the repository and interpreted by the *Taskflow Engine*. The Taskflow Engine tells the client system, in the case of ARVIKA a web browser and the ARBrowser, which documents and Augmented Reality scenes should be loaded and displayed.

A taskflow is composed of several tasks in a given sequence. The single steps of such a taskflow are linked with information from legacy systems, Augmented Reality scenes or other types of documents. Figure 5.6 illustrates the idea. A *Taskflow Repository* contains descriptions of taskflows and the steps of the taskflows, *Media Repositories* provide documents that may be displayed for each step.

For DWARF Pathfinder the ARVIKA Workflow Engine was reused. The navigation of a user to a predefined goal is similar to the guidance of a service man through a maintenance order. The single tasks of a workflow became the location points the user had to follow in the navigation scenario, the information documents became augmented information about the environment at each location

Internally, the taskflow engine may be seen as a state machine that switches to new states when it is triggered by certain incoming events. Every state has information associated with it, which can be sent to the user interface to be displayed. This could be a textual or graphical description of a navigation task, e.g. 'go up the stairs', or an animation of how to repair a certain machine part. By evaluating incoming events such as changes in the user's location or a spoken 'done' command, the taskflow engine switches to new task descriptions.

In the Pathfinder application, the Taskflow Engine receives and stores navigation instructions in the form of a taskflow from the information terminal via the Bluetooth Communication Service. This taskflow represents the process of navigating from the subway station to the meeting room. States in this taskflow are, for example, "outside", "in the hallway on the first floor" and "in the meeting room". Every state has a textual or graphical description of a navigation task, e.g. "go up the stairs" or an image of the stairs that have to be taken. This is sent to the User Interface Engine.

The Taskflow Engine reacts to incoming *ContextData* Events with updates on the user's position, which are generated by the application from the tracking data. By evaluating them, it can switch to new states of the navigation task.

Further details on the taskflow engine can be found in [126].

5.3.4 User Interface Engine

Augmented Reality is a technology which combines a real-world scene with virtual objects created from a computer. A desired feature of the presentation layer of all of those systems is multi-modal human-computer interaction. Figure 5.7 shows the idea of a multi-modal system.

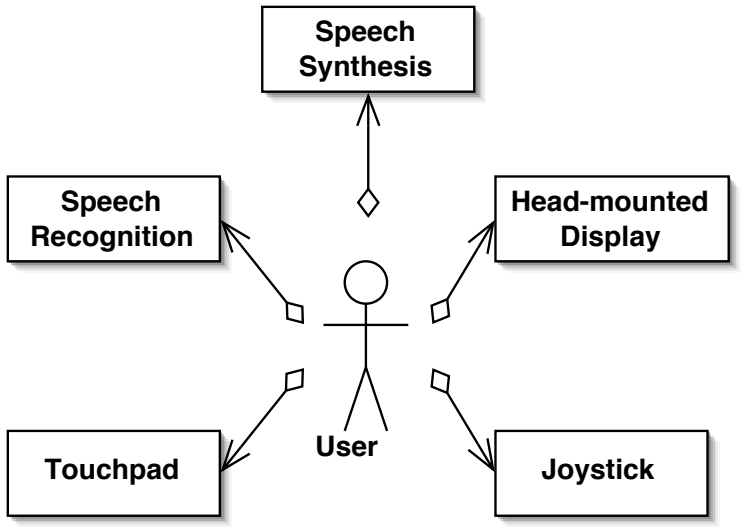


Figure 5.7: Several I/O channels for multi-modal interfaces

A definition of multi-modal system is given by Nigay and Coutaz [102]:

> *"In the general sense, a multi-modal system supports communication with the user through different modalities such as voice, gesture and typing. Literally "multi" refers to "more than one" and the term "modal" may cover the notion of "modality" as well as that of "mode"*
>
> 1. *Modality refers to the type of communication channel used to convey or acquire information. It also covers the way an idea is expressed or perceived, or the manner an action is performed.*
> 2. *Mode refers to the state that determines the way information is interpreted to extract or convey meaning.*
>
> *(. . .) multi-modality is the capacity of the system to communicate with the user along different types of communication channels and to extract and convey meaning automatically. (. . .) a multi-modal system is able to model the content of the information at a high level of abstraction."*

This means that the system should support user input and system output by various ways. User input could use components such as speech or gesture recognition, system output could use components such as voice output, 3D graphics, and text. These components have different features and properties as well as common ones. This can be used to combine different components to complement each other. For example, in a tour guide the directions are shown in 3D graphics in a head-mounted display in form of an arrow, but information about a tourist attraction is given in text and 2D graphics. The common features of devices allow the user to interact via more than one respective input or output component. Another example is user input by pressing the button "OK" or by saying "OK" to a speech recognition component.

The recurring task of developing a multi-modal HCI should be simplified by a presentation-free language for describing human-computer interaction: the Cooperative User Interfaces Markup Language (CUIML) This language should allow to use various input and output components in combination and in addition to each other. The User Interface Markup Language (UIML)[2] is an abstract language for user interface definition that was used as a starting point for CUIML. In DWARF we use CUIML for the definition of the presentation and control subsystems of Augmented Reality systems.

The core idea of CUIML is to factor out the structure of a HCI that is common to all different wearable setups. When the structure changes, only the passage in CUIML that contains the structural information has to be changed. As the concrete views that form the HCI for different hardware setups are generated automatically, the changes are minimized by this approach.

The DWARF User Interface Engine is responsible for interaction with the user. For the Pathfinder application, it displays three different kinds of scenes:

- outdoor *navigation*, showing a three-dimensional map of the TUM campus, rotated to match the user's orientation,
- indoor navigation, showing two-dimensional maps of rooms, halls, stairs and doorways,
- and three-dimensional highlighting of a specified object, for example the printer used for the handouts.

Only the last scene, overlaying an arrow over the printer, fits the classical definition of AR [7]. However, we found that for the task of navigation, schematic maps can provide a better overview than virtual arrows floating in mid-air would.

The information on which scene to display when comes from the taskflow engine.

Additionally, the User Interface Engine can display status information such as "print job started" and processes user input by voice recognition, in order to start the Bluetooth download or to confirm the print job.

The main task of the User Interface Engine is to display and process the user interface scenes provided by the application or the taskflow engine using *multi-modal* human-computer interfaces. The DWARF framework has been designed to support a large variety of application domains. In consequence, we can not rely on a fixed class of user interface devices such as head-mounted displays or usual computer terminals. It may even be possible that the output device changes during the runtime of an application.

To handle these constraints, the User Interface Engine separates the description of the user interface from its actual instantiation. The input of the engine consists of a XML-based description of the user interface's functionality that does not contain much information about its final look and feel. This input is then transformed or *rendered* to a concrete user interface displayed on a single or multiple available devices. This approach allows high flexibility and reduced development time for highly platform independent Augmented Reality applications.

Concrete User interface Devices supported by the user interface engine include a Virtual Reality Modelling Language (VRML)-based three-dimensional interface, a Hypertext Markup Language (HTML)-based interface for displaying textual and graphical information in two dimensions, and a voice recognition system that allows the user to give commands to the system.

For a more detailed discussion of the User Interface Engine, see [130].

5.3.5 Tracking Manager and Position Trackers

The tracking subsystem provides methods for determining the position of the user or other tracked objects. It is divided into two layers; there are several more or less simple trackers that provide location information. This information is collected by the next layer called the tracking manager, where the possibly contradictory data is combined and according to the reliability of the data the most probable position is computed as well as the accuracy of this measurement.

We use two classes of tracking devices. The first class are simple devices that give less than the six-dimensional data (three translational and three rotational components) necessary for real three-dimensional registration, but have a large range of operation or require only limited computing power. The second class are trackers that deliver six-dimensional data, but which are often constrained in rage or need a large amount of computing power.

Tracking Manager For an application using the tracking subsystem, a tracking manager is basically all it needs to get the most accurate position.

A tracking manager collects the information of all relevant trackers and calculates a more accurate real position out of this data. This takes the general accuracy of the trackers into account as well as the time when the measurement was done and the update frequency of the information.

One important feature of the tracking manager is the ability to dynamically add and remove trackers, which is invisible to the application.

In addition, facilities such as sensor fusion or movement prediction may be added to the tracking manager. In short, its task is to make the whole of the trackers more than the sum of its parts.

GPS Tracker The GPS tracker uses the output of a standard GPS receiver to determine the position and orientation. The position data consists of two-dimensional coordinates (the altitudinal measurement is very imprecise) and an orientation angle from a magnetic compass. This GPS tracker only works outdoors, and only when it has an unobstructed view of the GPS satellites.

Radio Frequency ID Tracker For indoor navigation, we originally intended to use RFID tags, which can identify each doorway as the user walks through it.

ID tags are unpowered devices that are attached to known locations in the real world, like doors or significant points in hallways. A special RFID-Tag reader mounted on the user's wearable computer reads the tag's identification every time the user passes by. This way, it is possible to obtain precise location information, although only for a short moment.

Unfortunately, it was not possible to find and obtain such tags in the narrow time frame of the project. As an alternative, we implemented a software simulation of RFID tags—a "manual tracking Service" in which an assistant entered the ID of the room the user was entering. Porting this software to use RFID tags should be trivial.

For every Augmented Reality application its very important to know at every time the accurate pose of the user. Unfortunately, no system exist at the moment that gives the required accuracy both in position and in viewing direction at the same time and is scalable to large areas as well [7].

The tracking subsystem for DWARF is a tracking framework that provides an application with position data for any object of interest, including the user, other persons, and moving or static objects. It is used as an abstraction layer between the application and the actual hardware tracker used to retrieve the position. On the other hand, the tracking subsystem can be used as a testing platform for testing new tracking hardware as well as new algorithms for data fusion, dead reckoning and movement prediction by simply implementing the right interfaces.

The output of each single tracker can not be used directly by an application without knowing detail about the available trackers and their properties. To detach this dependency on actual hardware and to simplify the tracking process for the application, the tracking subsystem is responsible for collecting the data from the various connected tracker, to check if its valid, then to filter it and generate the model parameters describing the tracked object. It also offers the service to predict the position at a given time based on this model.

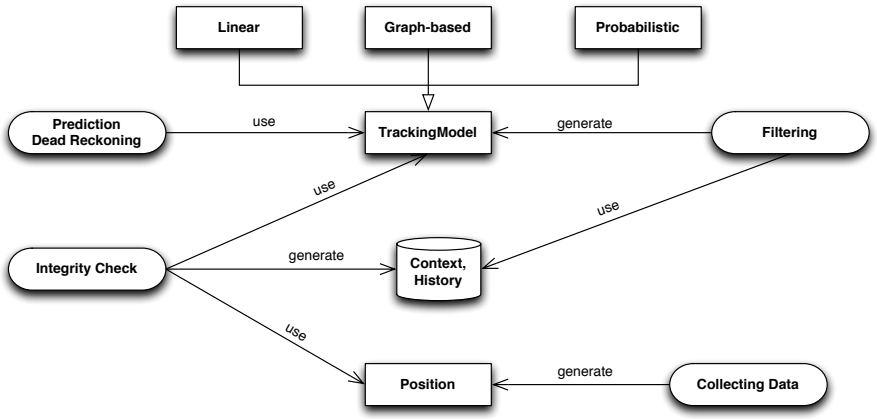


Figure 5.8: Activity diagram of data cooking of the Tracking Manager

Current tracking systems for Augmented or virtual reality are in most cases either commercial products that are if at all only to a small extend configurable or they are research projects that are usually limited to their particular research domain [123]. The proposed tracking framework provides an object-oriented design for the overall tracking process that tries to one the one hand be flexible to use any kind of tracking device, even not yet existing ones, and on the other hand to move away from the usual filter theory approach.

The difference between prediction and dead reckoning is the fact that dead reckoning is necessary where temporarily no data is available and prediction is needed when the data is not yet available. Therefore, prediction can be used later to validate the model and the parameters.

More information about the Tracking subsystem can be found in the diploma thesis of Martin Bauer [13].

5.3.6 Optical Feature Tracker

For the AR highlighting of the printer that printed the user's handouts, we needed accurate and fast tracking. For this, we used the Optical Tracking Service.

The *Optical Tracker* processes live video input from a camera mounted on the user's head and determines its orientation and position in real time. This six-dimensional data is crucial for performing 'real' augmented reality applications.

The basic principle of the optical tracker is simple. The video stream is analyzed for markers that are attached to known locations in the three-dimensional world. As a result, correspondences between two-dimensional image points and three-dimensional

real world points are established. Sophisticated algorithms are now used to compute the camera's six-dimensional pose out of these correspondences.

Optical tracking is a computationally expensive task. To ensure stable high performance of this crucial service for exact registration, the optical tracker should be executed on a dedicated Central Processing Unit (CPU) and deliver its result over a reliable network connection to the user interface.

The area around the printer was measured accurately, and the optical tracker could detect carefully-placed fiducial markers on and around the printer. It used this to calculate the user's position, and sent position data to the VRML display component of the user interface.

Figure 5.9 shows a general overview of the subsystems involved in the optical tracker and the shared memory areas they use to communicate with each other.

Further details on the Optical Tracker are described in [162].

5.3.7 World Model

The geographical and geometric information from the information terminal is stored in the mobile system's DWARF World Model Service.

The DWARF system needs to store data about real and virtual objects in a well organized fashion. The first place to store all data describing the user's natural and virtual environment is the World Model Service. It can be seen as a large database that holds entries for every real or virtual object. Examples for real objects are buildings, floors, furniture in rooms etc. Virtual objects may consist of virtual stickies attached to real objects or highlighting information such as virtual arrows to indicate the directions the user has to take [77].

The crucial point for all these objects is their three-dimensional position and orientation towards each other. The World Model service provides facilities that allow easy description and computation of these relations.

The World Model represents the world as a tree of Things, such as a campus, buildings, rooms, furniture items and so on. Each Thing has a geometrical relation to its parent Thing, allowing the relative position of all Things to be computed. Each Thing can also have arbitrary attributes associated with it. Useful attributes include services associated with Things, or VRML descriptions of the Thing's appearance. This tree-shaped structure is stored in an XML dialect.

As almost all DWARF components rely on such data, the World Model is a heavily used component. In consequence, efficiency was one of the major design goals.

The DWARF WorldModel component may be seen as the central database holding all necessary information about the real and virtual objects in the user's environment. As such, the specific requirements for AR systems have to be kept in mind and the World Model's interfaces have to be defined in a way that facilitates the development of AR applications.

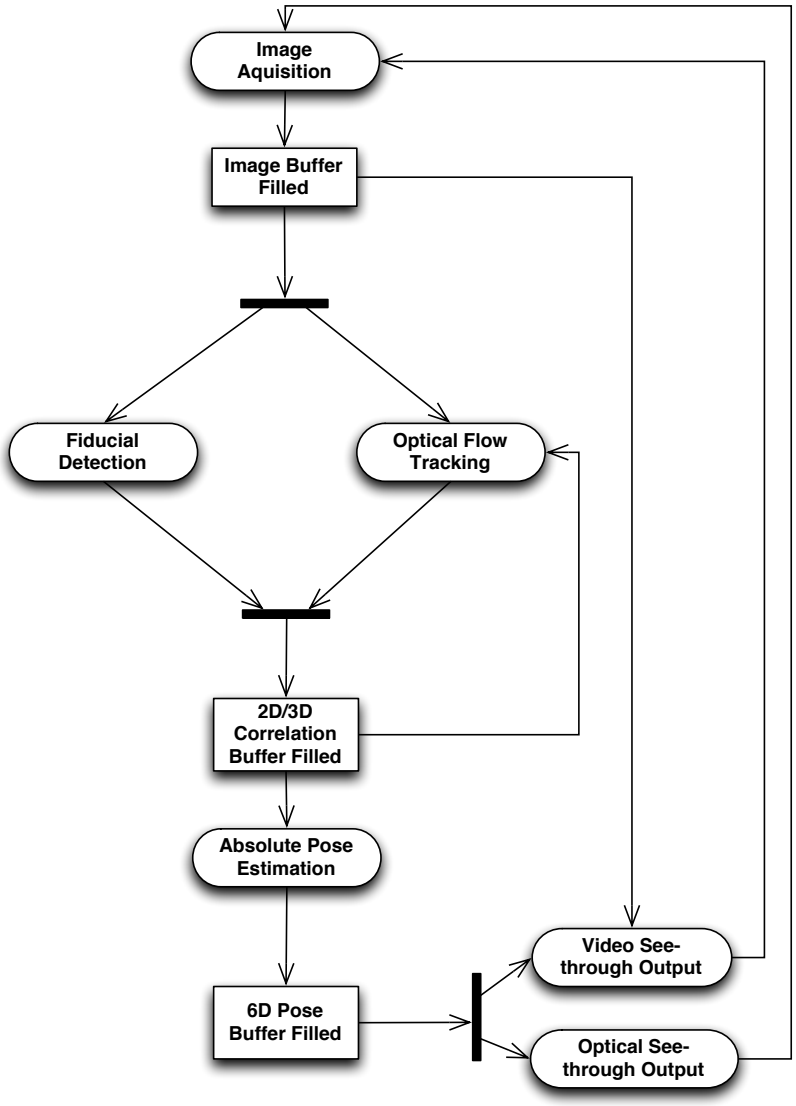


Figure 5.9: Basic concept of the Optical Tracker Service. Note that this diagram presents the dynamic behaviour of the Optical Tracking subsystem, the actual underlying architecture is not object-oriented and separates data from functionality.

From the requirements for the World Model Service, the system design follows straightforward. The core component of the information about a real or virtual object is its pose. Perhaps the single most important functional requirement for the World Model is the ability to compute one object's pose relative to an arbitrary other.

The system design of the World Model is centred around this capability. In the remainder of this section, we will explain the basic approach we chose to handle this problem and give some details about the subsystem decomposition and the persistent data management.

Real and virtual objects can be grouped hierarchically in a tree data structure (this could be modelled using the Composite design pattern). To give an example, consider a table in a room at the TUM campus. The top-level object may be, at the DWARF system designer's choice, something like a map of Munich or a Universal Transverse Mercator Projection (UTM) coordinate system [57]. One child of this top-level object should be the TUM campus. The coordinates of this campus may well differ from the UTM coordinates so we have to store rules how to convert the UTM system to the TUM campus system.

Again, the campus object has children. We may want to take every floor of every building at the campus as a child. These floor objects will then have single rooms as children. Finally, the table we are looking for is represented as a child of the room it is standing in.

Using this general structure, it is easy to add objects without knowing their position in the top-level coordinate system. If we add a virtual TV set to the table in our room, we only have to give the coordinate transformation from the virtual object to the table in order to allow the World Model service to compute the TV set's pose relative to every other object in the tree structure.

Details on the World Model can be found in the diploma thesis of Martin Wagner [162].

5.4 Service Deployment

The DWARF Services can be distributed onto several computing devices; the middleware will let them find each other, as long as they have a network connection. This allows computation-intensive services such as an optical tracker to run on dedicated hardware which can be added to or removed from the system at run time.

This way, the user can tailor the mobile Augmented Reality system to the requirements and leave out the hardware modules that are not needed for a particular application.

DWARF takes advantage of existing software components. For example, the user interface devices use existing VRML rendering and voice recognition software, and the middleware makes use of CORBA and third-party event services.

The DWARF services can run on many different platforms; the choice of hardware

and operating system for a particular Service depends heavily on the availability of drivers for the specialized hardware (such as digital cameras or GPS receivers). The performance-intensive and resource-constrained Services are written in C or C++, the others in *Java*.

Figure 5.10: A side view of our prototype wearable computer built with DWARF. Note that two laptops are used, and that there are no trailing cables. The display on the lower laptop was kept open so that the user's view could be shown to an audience as well

The Pathfinder prototype was deployed on two laptops with attached peripheral devices. They were mounted on a mobile frame one on top of the other as shown in figure 5.10.

- A Sony Vaio PCG-C1XD PictureBook (upper laptop in the figure) with a Pentium II-400 processor and 128 MB of RAM, running Microsoft Windows 98, with
  - a Sony DFW-VL 500 FireWire Camera for optical tracking,
  - an Ericsson Bluetooth device for access to the information terminal,
  - and a PC Card Ethernet adapter for access to the other laptop.
- A Dell Inspiron 5000 (lower laptop in the figure) with a Pentium III-450 processor and 192 MB of RAM, running SuSE Linux, with
  - a Sony PLM-S 700 Glasstron see-through head-mounted display,
  - a Garmin eTrex Summit GPS receiver,

– a Lucent WaveLAN wireless network PC Card adapter (using Apple AirPort base stations),

– and a PC Card Ethernet adapter for access to the other laptop.

All devices were battery-powered, and the time of operation was about two hours.

The deployment of the DWARF Services on the laptops is shown in figure 5.11. For the middleware, we used the "no local mediating agent" deployment on page 108 for the Sony Vaio, where all the middleware was on the Dell Laptop and there was a reliable Ethernet connection between the two.

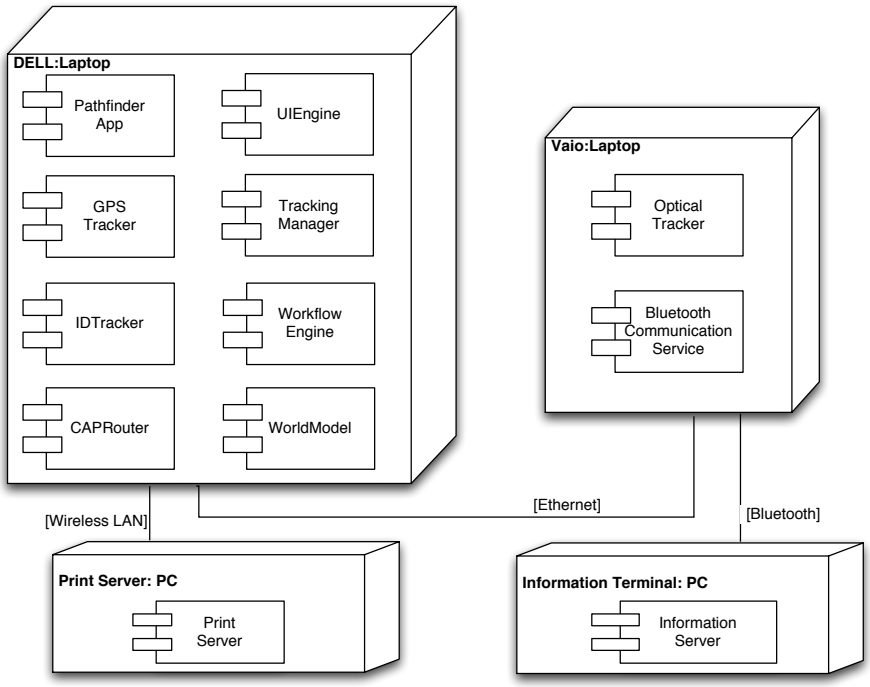


Figure 5.11: Deployment of the DWARF Services in the demonstration system. It consists a Sony Vaio running Microsoft Windows 98, a Dell laptop running SuSE Linux, a PC as Information Terminal Server connected over Bluetooth to the Vaio laptop, and a PC as Print Server connected over wireless LAN to the Dell laptop. The two laptops communicate over an Ethernet cross cable.

5.5 Conclusion

In this chapter we described a case study for the DWARF framework. We present the M^{3}ARF sub framework for mobile Augmented Reality maintenance systems and a demonstrator that implements a subset of the maintenance scenario, DWARF Pathfinder. This framework is a subclass of the abstract reference architecture and uses several Augmented Reality software patterns for the implementation of the Augmented Reality components. As system model the framework uses the contract-based peer-to-peer architectural style.

The case study shows the applicability of the DWARF framework and the underlying peer-to-peer architectural style.

6 Conclusion

Results and future work.

The goal of this dissertation was the development of a framework for adaptable mobile Augmented Reality systems in ubiquitous computing environments. A subclass of ubiquitous computing environments are so called *Augmented Reality-ready buildings* where external trackers can be integrated into the client's tracking process [79]. A special requirement was the ability to change the system configuration on demand to adapt to changes in the environment. For example, external resources such as tracking systems should be integrated dynamically as soon as they come within reach and be released when they get out of reach.

Another requirement was the ability to develop Augmented Reality applications with the framework that are fast enough for interactive use. This means that the overhead caused by the framework had to be as low as possible to enable real-time Augmented Reality systems. So we had to find a good balance between design for adaptability and flexibility on the one side, and performance on the other, similarly to finding the balance between high-quality and real-time constraints as described in [80].

Results

The result of this dissertation is the Distributed Wearable Augmented Reality Framework (DWARF), a reusable basis for the development of Augmented Reality applications. The DWARF framework divides Augmented Reality systems into four abstraction layers to handle technical, Augmented Reality, application domain, and application specific aspects separately.

Contract-based peer-to-peer architectural style. The first layer, called the architectural style layer, is based on the contract-based peer-to-peer architectural style for dynamically adaptable distributed systems. It allows us to model a system as a configuration of distributed services. Each service provides a contract which specifies possible collaborations with other services. The architectural style is implemented by the DWARF peer-to-peer middleware that connects services for collaboration on the base of these contracts. The chosen approach allows the dynamic configuration and

adaptation of Augmented Reality systems and is also fast enough to develop interactive Augmented Reality systems. This is achieved by decoupling the system configuration from the system communication. In the first step, the services find each other and establish the communication. In the second step, the actual communication takes place and data are exchanged. After that, the middleware is no longer involved and therefore does not induce communication overhead.

Augmented Reality reference architecture and design patterns While the first layer of the DWARF framework is domain independent, the second layer, the solution domain layer, is domain specific, which is Augmented Reality in the context of this dissertation. An abstract reference model for Augmented Reality systems describes the subsystems and their relationships of Augmented Reality systems [121]. This abstract model allows us to identify design patterns to implement these subsystems and components. An Augmented Reality system is the composition of Augmented Reality subsystems and an individual Augmented Reality subsystem is the composition of several Augmented Reality patterns [120]. The M^{3}ARF framework has a minimal set of the required DWARF Services for mobile Augmented Reality maintenance systems and includes several design patterns which are particularly suited for rapid prototyping of Augmented Reality systems.

M^{3}ARF framework and DWARF Pathfinder To demonstrate the feasibility of the DWARF framework we presented a case study with the M^{3}ARF sub framework for mobile maintenance systems and the prototype of a navigation application, DWARF Pathfinder. This application covers the application layer, the inter-application layer, and the solution domain layer. The third layer, the inter-application layer, contains reusable, more complex services that are suited for a specific application domain. Services of this layer use Services of the solution domain layer. An example from DWARF Pathfinder is the Taskflow Service for user navigation which collaborates with the User Interface Service.

Future Research Directions

The DWARF framework provides a useful foundation for the development of Augmented Reality systems. This was demonstrated successfully by building several Augmented Reality applications based on DWARF, for example FixIT [78], TRAMP [89] or SHEEP [131].

Nevertheless, the framework can be extended in several research directions.

Contract-based architectural style. The developed architectural style and the middleware for the dynamic configuration of Augmented Reality systems from distributed

services have been demonstrated as a powerful approach to develop flexible and adaptable Augmented Reality systems. It was developed for Augmented Reality systems on wearable computers with the possible integration of external resources. The assumption was that the available resources do not change too quickly, and as a consequence a system configuration could be assumed to be relatively stable. At system start-up the original set of DWARF Services is connected, afterwards only some, such as Services for tracking, are replaced dynamically. It is unclear, if the chosen concept is still valid in a highly dynamic environment. The next challenge is to see how the approach scales when it is applied to a new class of systems with potentially hundreds or thousands of available services.

The evolution of the contract-based peer-to-peer architectural style is still ongoing to enhance the Service selection strategies towards context-based selections [88].

DWARF Middleware. In any distributed system, deadlocks are an issue. If there is a cyclic dependency between Services, the DWARF middleware will be able to solve the deadlock by starting up the Services in an unspecified order, and connect them. The Services themselves may still deadlock, however—for example, if Services need to have data from other Services before they can send data themselves, and this dependency forms a cycle. Therefore, when designing DWARF Services, such dependencies should be considered carefully.

The DWARF middleware currently does not address all stages of the Service life cycle, such as installation and upgrade. It only deals with starting and stopping Services and arranging for communication between them. Managing these additional steps of the Service life cycle would be a useful future extension of the middleware.

Security. Another open issue is security. DWARF-based systems are still research systems and security has not been a major design goal. However, in future the middleware should address security issues, especially when multiple mobile users share Services in the same wireless network.

Evolution of the DWARF framework. M^3ARF was the first sub framework and Pathfinder the first application developed on the base of the presented DWARF framework for Augmented Reality systems. Other systems followed. For each system new DWARF Services have been developed and existing ones have been improved. So DWARF is evolving into a real framework with mature and reusable Services. Particulary, the User Interface and Tracking Services have improved a lot. For other domains such as games, new sub frameworks that customize already available Services and add their own DWARF Services must be developed.

Architectural patterns for Augmented Reality systems. New Augmented Reality systems for new application domains with yet unknown approaches will appear. There-

fore the catalogue of patterns must be updated regularly, quantitatively as well as qualitatively. New patterns must be added to the catalogue and existing ones must be updated or measured again. Such a step was done by the pattern community for approaches that have existed for a long time. To do that for a still rather young field as Augmented Reality is promising but would require the efforts of the whole Augmented Reality community. One possibility is to adapt the pattern shepherding concept used in the pattern community. Shepherding is the collaboration of one that is experienced in the description of patterns ('the shepherd') with one that identified a new pattern ('the sheep'). Collaboratively they improve the description of the new pattern. The concept to describe a reusable solution as patterns is not very well known in the Augmented Reality community, but there several developers that have developed solution that could be described as pattern. So the task is to identify a possible pattern in an existing Augmented Reality system and ask the author to join a shepherding process (as a sheep).

Development tools. With an underlying architectural style for distributed Augmented Reality systems and concrete middleware, the next challenge is to develop new software tools. For example, a graphical editor for the visual design would allow to specify the overall system and generate the contract files and code skeletons of the services. This could be combined with other tools that check the correctness of system designs. The key concept in this context is the OMG's Model Driven Architecture (MDA) specification [96]. The key idea of the MDA is the generation of executable code from machine-readable application and data models. The idea is already used in some domains such as telecommunications and real-time systems. A widely supported specification is the ITU's Specification and Description Language (SDL) [64]. The MDA tries to transfer the success of SLD to software engineering in general. The challenge is to develop a DWARF profile for the MDA and generate most part of a new DWARF system from a system model.

Content for Augmented Reality. Currently, the creation of new content for Augmented Reality and the dynamic access to it, for example over database interfaces is not yet solved satisfyingly. This is true for Augmented Reality systems in general and DWARF in particular. Here we see the need to research the connection of Augmented Reality systems with authoring environments such as DART [86], to implement the world model with database systems [124] and to connect enterprise-wide information systems [77].

Development of a wearable multi-computer. Another challenge is to build a wearable multi-computer based on the DWARF concept for dynamic module integration. Such a multi-computer would be composed of several loosely coupled self-contained computers worn by the user and several external computers in the environment. Each

unit could be restricted to serve a specific function, similar to an appliance [15]. This design promises a very flexible system with a straight-forward integration of external resources. New developments on the field of FPGA[1] allow us to develop such a prototype. Possible technical solutions are described in [117]. FPGA-based functionality-specific modules could also include the required middleware for dynamic collaboration.

[1] Field-Programmable Gate Arrays

A Design Patterns for Augmented Reality Systems

Architectural Patterns for the Application, Interaction, Presentation, Tracking, Context, and World Model subsystems.

This section is a catalogue of patterns that we have identified so far. The patterns are ordered by subsystems and described with name, goal, motivation, description, usability, consequences, and known use.

Application Subsystem

Central Control pattern

Goal: Keep the flow of control.

Motivation: The main parts of an application are independent of Augmented Reality. Indeed, the Augmented Reality specific components are only one part among others, and only used to visualize some content.

Description: Write the application in a high-level programming language, explicitly describing what happens when.

Usability: Use this pattern if it is necessary to keep the control flow, for example to guarantee real-time constraints for non-AR subsystems, e.g. reacting to external events. The disadvantage is that it is up to the application developer to implement the continuous update of registration and the rendering.

Consequences: The modifiability of the application is low.

Known use: MR Platform, ARToolKit

Tracking-Rendering-Loop pattern

Goal: Let an AR library do the tracking and rendering and call the application within the tracking-rendering loop.

Motivation: The tracking and rendering must be done in a regular loop that updates the user's view based on her motion. Embed the application into this loop.

Description: To alleviate the development of AR applications some libraries provide the needed low-level functionality to update the user's view regularly. The application's task is to provide hooks that can be called within the update loop and that might react on changes in the view.

Usability: With a library for tracking and rendering.

Consequences: The control flow is managed by the update loop of the tracking-rendering system.

Known use: VD2, ARToolKit

Web Service pattern

Goal: Treat AR as one type of media among others.

Motivation: For content-based applications the web-based approach has been proven to be a reasonable approach. AR scenes and world model information can be seen as an AR document. A scene such as an arrow that points to a particular button in front of the user is then described in document that is loaded from a web server.

Description: The control flow is situated on a web server and implemented within a web service. This web service is published under a particular web address and the answer of the service is rendered on a web client. If the answer contains Augmented Reality content then the AR component is activated to display the given AR content.

Usability: This approach can be used where the focus is on displaying various types of content and load them dynamically from a server.

Consequences: The client and the server must be connected. If a connection cannot be guaranteed then there must be a proxy available locally that emulates the server. This approach should be combined with a scene-based rendering component, e.g. a VRML or custom AR browser.

Known use: ARVIKA

Multimedia Flow Description pattern

Goal: Use high-level description language to describe AR scenes.

Motivation: For the development of multimedia content, there are several formats that simplify the creation of new content by providing high-level concepts such as timers. Examples for such languages are SMIL and Macromedia Flash. Additionally to multi-purpose languages for multimedia content, there are domain specific languages for particular fields—for example, description languages for Workflows or for technical manuals (IETMs).

Description: A high-level markup language provides domain specific components and concepts that help quickly create new content. For example, to support a training scenario for unskilled workers, the AR system should visualize a sequence of AR scenes and other documents. To describe such a scenario, the content creator has to combine workflow steps and add content to each step. An execution engine for workflows reads such a description and controls the presentation of the current working step.

Usability: This approach can be used for applications with a meta-model for the description of documents and their relationship and dependencies combined with a component that reads and executes such descriptions.

Consequences: The complexity of this approach is higher for simple applications. This approach should be combined with a scene-based rendering component, e.g. a VRML or custom AR browser.

Known use: STAR, ARVIKA, DWARF

Interaction Subsystem

Handle in Application pattern

Goal: Keep the system architecture simple; provide high-fidelity interfaces.

Motivation: The simplest way to handle user input for a specific application is to hard-code it into the application logic itself. Also, this allows custom-tailored input styles for each application.

Description: Include input handling code in the application code, with explicit references to the types of input devices.

Usability: Within a main executable application.

Consequences: Potential for high-fidelity interfaces; reduced modifiability.

Known use:

Use Browser Input Functions pattern

Goal: Take advantage of existing functionality.

Motivation: When using a browser for rendering, you can take advantage of its input features.

Description: VRML browsers can send events out through the EAI interface when the user clicks on on-screen objects with the mouse or when the gaze direction coincides with certain objects. Other browsers provide similar functionality.

Usability: Together with a browser-based output subsystem.

Consequences: Separating input from output modalities and integration into multi-modal systems can be difficult. Since wearable systems do not generally have a mouse, the mouse movement must be simulated with another input device.

Known use: ARVIKA

Networked Input Devices pattern

Goal: Combine the data of various input devices to form multi-modal user interfaces.

Motivation: Multi-modal interfaces require many simultaneous input devices. Modelling all possible combinations is exponentially complex.

Description: Provide an abstraction layer for input devices and a description of how the user input can be combined; interpret this description using a controller component. Use middleware to find new input devices dynamically.

Usability: Good when combined with multiple viewers for output.

Consequences: Allows integration of new input devices at run time or when building new systems.

Known use: DWARF

Presentation Subsystem

VRML Browser pattern

Goal: Use a rendering component that can display simple virtual scenes.

Motivation: The usage of a VRML browser is a simple way to display virtual scenes. The standardized VRML format, a markup language for the description of virtual worlds, allows to use a lot of tools for authoring virtual worlds and to reuse components that can render descriptions of virtual worlds.

Description: Use a third-party VRML browser to display 3D information. Use the External Authoring Interface (EAI) that is part of the VRML standard to modify the scene and set the viewpoint based on tracking data.

Usability: A VRML browser component can be used if the complexity of the scenes is relatively low and the browser is only used as a rendering engine.

Consequences: The advantages of using a VRML browser are the standardized format and the reuse of tools for authoring and the reuse of existing components. This allows rapid prototyping of Augmented Reality system based on VRML scenes. The disadvantages are that the EAI is restricted to relatively simple operations and that tying the VRML browser to the rest of the system may be tedious. Also, the rendering performance of VRML browsers is not as high as that of native OpenGL.

Known use: STAR, DWARF

OpenGL pattern

Goal: Use a standardized library to render 3D objects and keep maximum flexibility and control.

Motivation: OpenGL is the standard low-level library for 3D graphics. While higher-level approaches, particularly scene graphs, provide a more powerful interface for 3D worlds, OpenGL provides the most flexibility to the application programmer. Scene graphs use their own control path user applications have to comply with. By using OpenGL, the developer can implement his own control flow.

Description: OpenGL provides low-level 3D constructs. The application developer creates new objects and tells the render to display them [170]. With the information from the trackers the scene can be rendered with the correct viewing direction and distance.

Usability: Usable on nearly all systems.

Consequences: Modifiability of system is low.

Known use: ARToolKit

Proprietary Scene Graph pattern

Goal: Use a rendering component that can render scenes and provides a customized programming interface.

Motivation: There may be reasons to develop an own scene graph for rendering, e.g. the provided interfaces are not satisfying or the control over the data flow, that is often controlled by the scene graph, should be kept.

Description: The Tinmith system uses an own scene graph for graphics rendering on top of OpenGL combined with an own concept for object access through an own addressing schema. Each node of the scene graph has the same abilities to serialize and address them as the other objects in the system.

Usability: An own scene graph may be useful if the developer wants to control the implementation and the behaviour of the scene graph. For example, in the Tinmith system each object of the system can be made persistent and it can be reached over an address schema. The nodes of the scene graph are objects of the same type and there have the same attributes.

Consequences: The scene graph has to be developed from the ground up. For example, to make scenes persistent, either a parser for a standard format like VRML has to be developed or a proprietary format which cannot reuse scenes developed by standard tools such as VRML editors.

Known use: Tinmith

Video Transfer pattern

Goal: Offload the video rendering to a server, transfer it to the client and present it there.

Motivation: To reduce hardware requirements of the client system, use a rendering server in the environment and transfer the completely rendered images to the client. Particularly suited for video see-through AR.

Description: The client gathers videos through one or two head-mounted cameras, encodes them (e.g. MPEG 2), compresses them, and transfers them to the server. The server uncompresses the video images, processes them (calculates the camera position and orientation), augments, encodes and compresses the images. The images are sent to the client, decompressed and shown on the HMD.

Usability: This can be used with a good network connection and a strong rendering server.

Consequences: The advantage is that the hardware requirements for the clients are very low. Even PDA-class devices can be used. The disadvantage is that the client and the server must have network connection.

Known use: STAR, MR Platform, AR-PDA.

Multiple Viewer Classes pattern

Goal: Use different media types for different types of information.

Motivation: Although the central requirement of AR is displaying three-dimensional information, other types of output devices such as speech synthesis or wrist-worn displays are also useful.

Description: Provide an abstraction layer for different types of viewers (AR, speech, text etc.) that can handle certain document types. Then provide the viewers with the appropriate documents.

Usability: This approach can be used whenever multiple output media are desired, unless the rendering complexity is so high that the 3D viewer cannot be parameterized.

Consequences: Advantages are a modular system design and the possibility of integrating additional output devices at run time or in new systems. Disadvantages are the extra complexity of a viewer abstraction layer.

Known use: ARVIKA, DWARF

Tracking Subsystem

Networked Trackers pattern

Goal: Combine the data of various trackers in the environment without knowing the physical location.

Motivation: The best results for tracking can be achieved by combining various tracking methods. Usually, the possibilities to connect tracking devices physically to the client device or a tracking server are restricted. As a solution, wrap every tracker with a software interface that masks the location of the tracker by providing a network-accessible interface. To gather the data of all trackers, each tracker is accessed remotely. If supported by middleware, the lookup can be done by name, not by network address.

Description: For each tracking device, provide a wrapper that uses middleware concepts such as CORBA. The wrapper provides an interface to the tracker and registers itself in the network. Components that need a tracker (consumer) look for them through middleware services and connect to them. The components search for the trackers by name, not by address. Once connected, the tracker and the consumer communicate transparently.

Usability: This approach can be used when the client system can connect to tracking devices over the network.

Consequences: The advantage is that virtually any number of tracking devices of different types can be combined. The disadvantage is the overhead of network communication.

Known use: ARVIKA, Studierstube, DWARF.

Operating System Resources pattern

Goal: Directly connect tracking devices to client system.

Motivation: If no network connection is available then only devices that are physically connected to the client device can be used. Particularly inertial trackers and video cameras are small and can be used.

Description: The tracking devices are accessed through drivers for the operating system.

Usability: The client device must powerful enough to execute the tracking algorithms.

Consequences: The advantage is that no network connection is needed but only devices that can be connected physically to the client device can be used. And the computing must be done completely on the client device.

Known use: MR Platform, ARToolKit

Tracking Server pattern

Goal: Use a centralized tracking server to reduce hardware requirements on the client system.

Motivation: There are many different tracking technologies available: magnetic, video based, inertial, or combinations thereof. Calculate the user's pose from raw data may require significant computing power. One strategy to avoid this is to offload the computing to a server in the user's environment and only transfer the result to the client system.

Description: A tracking server in the user's environment performs resource intensive computations and returns the results to the client.

Usability: Use when integrating a commercial tracking system.

Consequences: Availability of a connection to the server.

Known use: STAR, UbiCom.

World Model Subsystem

OpenGL Code pattern

Goal: Describe virtual objects in OpenGL source code and load objects at runtime from library.

Motivation: To be able to render objects in an AR system, it is enough to write some OpenGL code, compile it and feed the code to the renderer. For testing purposes, this will suffice.

Description: The developer creates OpenGL code and calls the OpenGL rendering engine to display it. For correct registration with the user's pose, the position and angle of the virtual camera that looks at the scene can be changed. This is usually done in rendering-tracking-update loop.

Usability: When only small scenes need to be presented, simple OpenGL code can easily be written and tested, or existing code for scenes may be reused.

Consequences: The advantages are that it is easy to test an AR system. However, this approach is not very scalable as OpenGL is a fairly low-level library for 3D.

Known use: ARToolKit

Scene Graph Format pattern

Goal: Use a standard format with authoring tools to generate content and rendering engines to display them.

Motivation: Scene graphs are the standard component to display virtual environments. Each scene graph can read scene graph descriptions in several formats, where the most well-known is the VRML format.

Description: With an authoring tool a content developer creates the model of a virtual scene. In industrial context scenes created with CAD tools can be simplified and reused. The scene description is saved in the file system and given the AR system for processing. Scene graphs are usually stored on the file system.

Usability: Can be used where a scene graph is used for rendering, the scene does not change too often, and the scenes are discrete without interconnection with each other.

Consequences: Simplifies the creation of new content as authoring tools can be reused and fits perfectly to the scene graph component.

Known use: ARVIKA, DWARF, MR Platform.

Object Stream pattern

Goal: Serialize world model in main memory to disk.

Motivation: Serialization is a well-known technique to save runtime objects. This can be reused for objects in a scene graph at runtime.

Description: The runtime environment allows to serializing objects to disk. The next time the application is started the objects are recreated by deserializing them. Recursively, a whole scene graph can be loaded from disk.

Usability: Usable where the world model is created by the system user and the system only has to read own formats.

Consequences: An object stream can be read only by systems that knows the classes of the objects. Thus, an object stream is a proprietary format.

Known use: Tinmith

Configuration File for Marker Positions pattern

Goal: Read configuration data for the trackers from a flat file.

Motivation: The easiest way to tell a tracker what to look for and how to interpret it is to write it to a file and read it every time the system starts.

Description: At system start-up or any time the system comes into a new environment the trackers have to know what to look for and how to interpret it. So the tracking component is told the file that describes the markers or any natural features it has to look for.

Usability: Used if only a small number of markers are needed. If many markers are needed the same problems as with file-based scene graphs will happen.

Consequences: Does not scale well for data more complex than marker positions or for dynamically changing data.

Known use: ARVIKA, MR Platform, ARToolkit

Dynamic Access pattern

Goal: Use abstract description of world model from a database.

Motivation: For larger descriptions of the user's environment a collection of file-based scene descriptions may not be enough. For example, finding a particular scene by filename requires a name schema that could become quite complex depending on the number of scenes and the complexity of them. Especially in mobile environments, this can happen quickly.

Description: Instead of loading a particular scene from a file, the system has access to a database system. This system contains information about the environment, e.g. in a geographical schema. Part of the information are graphical information and marker information. The system queries for the graphical information that belongs to a discrete database object and passes it on to the rendering component. The same is true for marker information (to the tracking component) and real world objects (e.g. for occlusion).

Usability: World models in a database can be combined with graphical models that are also saved in the database. This decouples the application dependent model, e.g. the model for a machine to be repaired, from the graphical information needed to render objects that are part of the model.

Consequences: A database server is a heavy-weight component compared to file system but offers more possibilities to model larger environments. Especially useable for mobile applications a database model based on geographical information system (GIS) concepts combined with graphical models for concrete objects.

Known use: ARVIKA, ArcheoGuide, MARS.

Context Subsystem

Blackboard pattern

Goal: Gather and process context information.

Motivation: Blackboards are a well-known pattern to gather information from various sources, apply rules and create new information. This is particularly suited where a lot of raw data with low-level information must be refined in several steps to higher abstract information. Consumers and producers of information are decoupled.

Description: Information producers write information to the Blackboard, a central component. Information consumers read data from the Blackboard, process them and may write new, higher abstract information to the Blackboard.

Usability: Good if information from many sources must be analyzed and filtered.

Consequences: Blackboards are a central component which might become a bottleneck. As an advantage, the participating components do not need to know each other.

Known use: MIThril

Repository pattern

Goal: Provide a central means for information exchange.

Motivation: Many components need context information, other produce context information. A Repository component is a central component that is well-known by all components. It acts as a information storage.

Description: Components that produce context information write to the repository. Components that are interested into context information read from the repository. The repository uses an addressing schema to manage the information. Each kind of data is written and read by providing its address.

Usability: Good if data from many sources must be stored and read.

Consequences: Similar to Blackboards, Repositories can be a bottleneck. But again the participating components do not need to know each other.

Known use: ARVIKA

Publisher/Subscriber pattern

Goal: Provide a means to distribute context information.

Motivation: When context data does not need to be saved for a certain period of time, it can be distributed at once without need for a temporary storage.

Description: Context providers connect as publishers to a central messaging service, context consumers as subscribers. The context providers write the new context information to a particular channel which distributes it to the connected subscribers.

Usability: May be used where there is no need to store context information. Could be used in combination with a context repository. Some data may be saved, some delivered at once.

Consequences: Again might be a bottleneck and publishers and subscribers do not need to know each other.

Known use: DWARF, ARVIKA

Context Pull pattern

Goal: Connect components that need context information directly with source.

Motivation: For simple cases a centralized service for data exchange may not be needed. For example, if only location context is needed a direct link to the tracking subsystem may be enough.

Description: An interested component directly queries the context producer component or it registers itself as subscriber. The subscriber list is managed by each component privately.

Usability: Suited for reduced context sensitivity.

Consequences: If the amount of context sources and context consumers increases ad hoc hardwired direct connections will lead to unmanageable code. The coupling between consumer and producers is very tight, which makes them very interdependent.

Known use:

B Details on the Pathfinder Services

Reference documentation for DWARF Services used in Pathfinder: Pathfinder Application, CAP Router, Bluetooth Communication Service, Taskflow Engine, Tracking Manager, Optical Tracker, User Interface Engine

---

This chapter contains some technical details for the DWARF Services that are part of the M^{3}ARF sub framework and were used for the Pathfinder application as described in chapter 5. It provides a coherent view on the Services in one place and with a uniform structure. For each Service we present the use cases, the functional requirements, the system design, the integration into the system, and the XML service description. The DWARF Services for the Pathfinder application, the CAP Router, and the Bluetooth Communication Service are not part of the M^{3}ARF framework and we only present the XML service description and abstain from more details.

B.1 Pathfinder Application

Service Description

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="PathfinderApp" startOnDemand="false"
    stopOnNoUse="false" startCommand="">
    <need name="needITerm" type="ITermInfo"
        predicate="" minInstances="0" maxInstances="10000">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="needCAP" type="CAPInfo" predicate=""
        minInstances="0" maxInstances="10000">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="needUserInterfaceEvent" type="UserInterfaceEvent"
        predicate="" minInstances="0" maxInstances="10000">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="needWorldModel" type="WorldModel" predicate=""
        minInstances="0" maxInstances="10000">
```

```
        <connector protocol="CorbaObjImporter"/>
    </need>
    <ability name="abilityITerm" type="ITermControl">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
    <ability name="abilityTaskflow" type="TaskflowControl">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
    <ability name="abilityCAP" type="CAPControl">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
    <ability name="abilityContextData" type="ContextData">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
</service>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="Mediator" startOnDemand="false"
    stopOnNoUse="false" startCommand="">
    <need name="needPositionData" type="PositionData"
        predicate="" minInstances="1" maxInstances="10000">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="needWorldModel" type="WorldModel"
        predicate="" minInstances="1" maxInstances="10000">
        <connector protocol="CorbaObjImporter"/>
    </need>
    <ability name="abilityContextData" type="ContextData">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
</service>
```

B.2 Bluetooth Communication Service

Service Description

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="BluetoothCommunicationService" startOnDemand="false"
    stopOnNoUse="false" startCommand="">
    <need name="BluetoothEventReceiver" type="ITermControl"
        predicate="OS!=Win98" minInstances="0" maxInstances="100">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <ability name="BluetoothEventSender" type="ITermInfo">
        <attribute name="batz" value="bargl"/>
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
```

```
</service>
```

B.3 CAP Router

Service Description

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="CAPRouter" startOnDemand="false"
    stopOnNoUse="false" startCommand="">
    <need name="egal" type="CAPControl"
        minInstances="0" maxInstances="4711">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="auchegal" type="ContextData"
        minInstances="0" maxInstances="4711">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <ability name="wurscht" type="CAPInfo">
        <attribute name="batz" value="bargl"/>
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
</service>
```

B.4 Taskflow Engine

Use Cases

The following use cases provide a generalization of the taskflow system's external behaviour. The use case diagram in figure B.1 shows the interdependencies between the use cases. These use cases do not cover the possibilities of the taskflow description language but the possibilities and reactions a taskflow controlled system has to provide.

In fact it was not easy to find more detailed use cases at all because most of the time a user will not be working with the taskflow directly. He will either manipulate it through a not further specified GUI or the taskflow just manages the control flow in another system, in which case the user might not even be aware of handling a taskflow.

CreateTaskflow. Please note that in the case that the `TaskflowCreator` is a legacy system with an export filter it has to have access to a document repository. In most cases a combination of both would be most effective: A product management system exports a rough outline of a taskflow and a human taskflow creator fills in the gaps and links AR-documents to it.

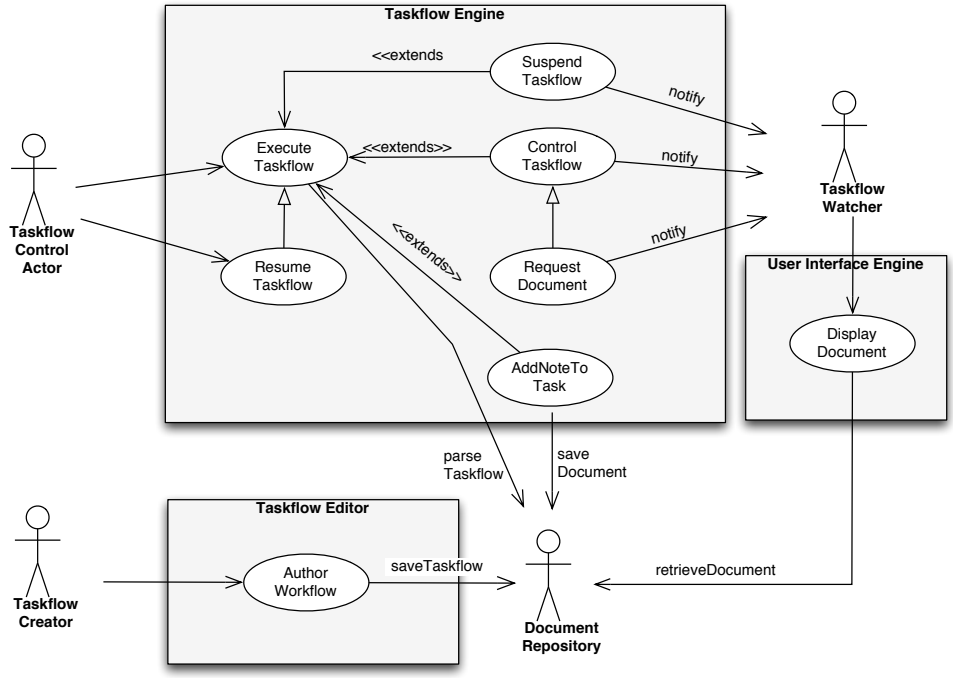


Figure B.1: TaskflowEngine and -Editor boundaries (UML use case diagram)

Use case name: **CreateTaskflow**

Participating actor: Initiated by `:TaskflowCreator`
Communicates with `:DocumentRepository`

Flow of Events:

1. *Entry condition:* The `TaskflowCreator` starts the graphical Taskflow Editor and creates the first task.
2. Create additional tasks and connect them with each other, so that the connections describe the taskflow.
3. To every task in the taskflow the `TaskflowCreator` may add links to documents either from a document repository or from an AR-scene repository. Each document may have properties that may influence the taskflows behaviour.
4. Connections between tasks can have a condition.
5. It is also possible to add links to information that should always be available.

6. *Exit condition:* When finished the taskflow has to be transferred to the taskflow server.

ExecuteTaskflow. A user directly manipulates the taskflow through a web browser-based GUI. This is an example of how a taskflow might be presented to the user and how it should react to the users input. Additionally, there could be sensors that monitor and control a taskflow.

Use case name:	**ExecuteTaskflow**
Participating actor:	Initiated by `TaskflowControlActor` Communicates with `TaskflowWatcher`
Flow of Events:	1. *Entry condition:* The `TaskflowControlActor` chooses a taskflow. 2. The taskflow description is loaded and parsed. 3. The taskflow is started and the user is shown the first task. 4. In each task the user is presented a list of available documents upon entering a task. 5. The user may control the taskflow with the available options (`ControlTaskflow` use case). 6. He may also suspend the taskflow (`SuspendTaskflow` use case). 7. Or he may add a note (`AddNoteToTask`-use case) to the current task. 8. *Exit condition:* The Worker has seen all mandatory documents of the last task and has received acknowledgment that the data entered in the process has been saved.

ControlTaskflow. The control over a taskflow is part of the execution of it. For example, a user can go to the next task or the previous one.

Use case name:	**ControlTaskflow**
Participating actor:	Initiated by `TaskflowControlActor` Communicates with `TaskflowWatcher`
Flow of Events:	1. The user is shown a control bar that offers task specific functions (i.e. next task, previous task, show documents, call help, ...). 2. The user may also choose from a list of documents available in the current task. 3. Whenever the user activates any of these options a request is sent to the taskflow.

4. `TaskflowWatchers` have to be notified about changes in the Taskflow.

RequestDocument. Load a document that is linked with the current task.

Use case name: **RequestDocument**

Participating actor: Initiated by `TaskflowControlActor`
Communicates with `TaskflowWatcher`

Flow of Events:
1. *Entry condition:* The user requests a document to be shown.
2. The request for the document is sent to the taskflow, so that it may react appropriately. For example it may note that the user has read some security instructions.
3. The `TaskflowWatcher` is sent a request to display a document (`DisplayDoxcument` use case).
4. *Exit condition:* The `TaskflowWatcher` is displaying the requested information.

DisplayDocument. Show it.

Use case name: **DisplayDocument**

Participating actor: Initiated by `Web browser as a TaskflowWatcher`
Communicates with `DocumentRepository`

Flow of Events:
1. *Entry condition:* The `TaskflowWatcher` received a request to display a document.
2. The `TaskflowWatcher` connects to the `DocumentRepository` and loads the document.
3. The `TaskflowWatcher` displays the document.

SuspendTaskflow. Taskflows can be suspended and resumed later.

Use case name: **SuspendTaskflow**

Participating actor: Initiated by `TaskflowControlActor`
Communicates with

Flow of Events:
1. *Entry condition:* The `TaskflowControlActor` activate the SuspendTaskflow function.
2. The user has to acknowledge his intention to suspend the taskflow.

3. To ensure that an other worker is able to continue the taskflow later the current user has to enter a note that includes a description of the completed tasks and a reason for the suspension.
4. Other `TaskflowWatcher`s have to be notified of the taskflow's suspension.
5. The system saves the taskflow's current state and again presents a list of available taskflows. This should now also include the suspended taskflow which is marked accordingly.

ResumeTaskflow. Resume a suspended taskflow.

Use case name: **ResumeTaskflow**

Participating actor: Initiated by `TaskflowControlActor`

Communicates with

Flow of Events:
1. *Entry condition:* The `TaskflowControlActor` activate a suspended taskflow.
2. The taskflow is presented to the user as in the `ExecuteTaskflow` use case. The initial task is the same as the one when the taskflow has been suspended.

AddNoteToTaskflow. This use case requires that a common document repository is used and referenced by a taskflow. This repository is not part of the taskflow system. This use case was not implemented neither for ARVIKA nor for DWARF Pathfinder.

Use case name: **AddNoteToTask**

Participating actor: Initiated by `TaskflowControlActor`

Communicates with `DocumentRepository`

Flow of Events:
1. *Entry condition:* The `TaskflowControlActor` activates the AddNote function.
2. The `TaskflowControlActor` enters a note (text, video, graphic or audio) and acknowledges his input.
3. The note is saved to a document repository and linked to the taskflow as a new document with additional information specifying author and date.

Functional Requirements

The following list of functional requirements has been extracted (and translated) from the ARVIKA Taskflow Engine specification [118]. They "... describe the interactions between the system and its environment independent of its implementation" [22, 4.3.1 Functional Requirements].

1. Creation of taskflow descriptions with a graphical user interface.
2. It should be possible to define taskflows that are more complex than just a linear sequence of documents, although the first release will only offer simple taskflows.
3. Definition of sub-taskflows to reduce the complexity of a taskflow and to provide a mechanism for simple AR scenes.
4. Interface to production-planning-systems PPS: In many cases a PPS will plan the taskflow for the worker so that the Taskflow Engine should have an interface that allows other systems to add new taskflows on demand. Additionally the Taskflow Engine also should have an interface that allows the user to add information from a PPS or other production relevant tools.
5. Linking of documents to single tasks in the taskflow: There is a number of supplementary documents for each step in the taskflow that support the user to fulfil the task.
6. Interface to document-systems: Various ARVIKA partners already had company wide access to product documentation over an intranet. These systems may be coupled with a production-planning-system. The taskflow system should provide an interface to enable the access to the document-system.
7. Definition of events and conditions to change from one task to another: Events and definitions may be defined in the taskflow's description that should be interpreted by a Taskflow Engine. Such events may lead the user to the next task in the taskflow or invoke an other action, i.e. a sub-taskflow.
8. The Taskflow Engine will trace the users way through the taskflow and log every task so that the user is able to suspend the taskflow at any time. The Taskflow Engine keeps the current state and notifies the next user that the taskflow can be resumed at that task.

The following functional requirements are not part of the ARVIKA specification but have been included to be able to easier cope with future problems and scenarios.

1. The user should be able to activate a supporting display: Documents that are not suited for the display on HMDs should be displayed on a secondary display nearby. This display has to be identified by the user.
2. Multiple `TaskflowWatchers` may observe the same taskflow.

System Design

The Taskflow Engine Service is designed using the MVCpattern.

Figure B.2 provides an overview of the subsystem decomposition based on the MVC architecture and enhanced by a facade that hides the *Model*'s structure from the *View* and *Controller* subsystem. The following sections describe each subsystem from bottom to top.

The model subsystem. The model subsystem keeps an object representation of the taskflow. The model is optimized to allow easier processing of requests and to enable the TDL's reaction mechanism. To reduce the coupling with the facade subsystem the model is accessed through the central `FsaTaskflow` class. The `FsaTaskflow` keeps a `CommandQueue` to serialize the `TaskflowCommand`s which request a change in the taskflow.

The facade subsystem. One of our major design goals was to keep the system as adaptable as possible and modifiable as possible. The facade subsystem serves exactly that purpose by providing a simple API for `View`s and `Controller`s on one hand and on the other hand by reducing the coupling to the underlying `Model`. The intent of a facade is to "provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use" [45, p. 185 Facade].

The facade itself is provided by the `FsaTaskflowEngine`. The name comes from the fact that it is a descendant from the `TaskflowEngine` and thus responsible to forward `ControlTaskflow` requests to the `Taskflow` and that it provides a facade for a taskflow based on a Finite State Automaton.
What follows is a description of the Taskflow Engine's interface divided into two parts. The first describes the methods that a `Controller` may use to modify the state of a taskflow, the second part describes the methods that allow communication between the `TaskflowEngine` and the `View`s.

The controller subsystem. To control and manipulate a taskflow or to request the display of a document a controller has to access the `FsaTaskflowEngine` and call the appropriate methods.

The only controller we implemented is part of the prototype adapter for the ARVIKA system. The controller is a `Servlet` that allows to access the `TaskflowEngine` from a web-browser by translating special HTTP request into calls for the facade's methods. The `View` counterpart of this adapter are formed by the `Workspace` and the `FsaWebRenderer` which are described in the next section.

The view subsystem. Views that want to be notified have to implement one or more of the listener interfaces in the view subsystem. Each listener monitors different as-

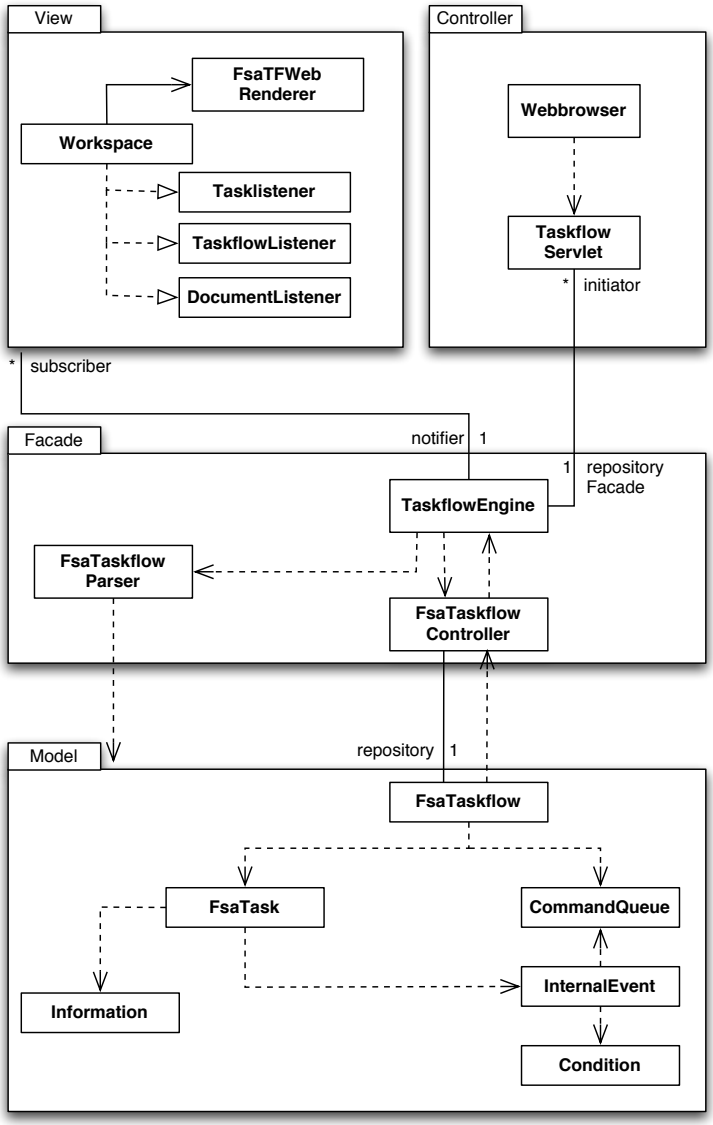


Figure B.2: MVC architecture of the Taskflow Engine enhanced by a facade

pects of the taskflow. To be notified a view has to register itself by calling the Taskflow Engine's `addListener` method. Every time the monitored aspect changes the view is notified which is the essence of the *MVC* pattern.

System Integration

Figure B.3 shows the specification of the Taskflow Engine Service. It requires a *TaskflowControl* and *ContextData*. If it finds these two services it will provide *SceneData*.

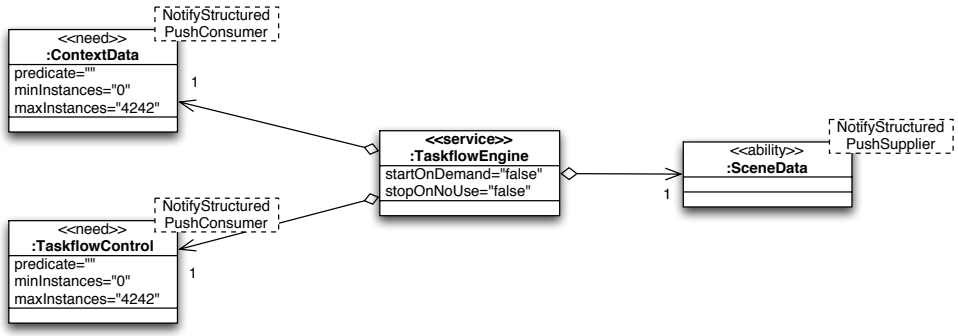


Figure B.3: Taskflow Engine Service specification.

Service Description

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="TaskflowEngine" startOnDemand="false"
       stopOnNoUse="false" startCommand="">
    <need name="contextData" type="ContextData"
          predicate="" minInstances="0" maxInstances="4242">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="taskflowControl" type="TaskflowControl"
          predicate="" minInstances="0" maxInstances="4242">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <ability name="sceneData" type="SceneData">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
</service>
```

B.5 User Interface Engine

Use Cases

CreateHCI. This use case describes the typical creation of a multi-modal HCIs.

Use case name:	**CreateHCI**
Participating actor:	Initiated by `HCICreator` Communicates with `HCIRepository`
Flow of Events:	1. *Entry condition:* The `HCICreator` has several specification documents that describe the semantics that should be supplied by the HCI 2. The `HCICreator` specifies for, all supported hardware configurations, the Views that should be displayed to the `HCIUser`. 3. He adds the synchronization logic by specifying the controller that describes the different states in which the HCI can be. 4. The `HCICreator` maps the Views to concrete markup languages for the output components. 5. The resulting documents are saved to the `HCIRepository`. 6. *Exit condition:* The `HCIRepository` contains all documents needed to build the current HCIs.

ModifyHCI

Use case name:	**ModifyHCI**
Participating actor:	Initiated by `HCICreator` Communicates with `HCIRepository`
Flow of Events:	1. *Entry condition:* The `HCIRepository` contains a HCI description that should be adjusted because of changed specifications. 2. The `HCICreator` retrieves the formerly saved documents. 3. The changes are integrated in the descriptions. 4. The changed documents are saved back to the `HCIRepository`. 5. *Exit condition:* The `HCIRepository` contains all updated documents needed to build the current HCIs according to the new specification.

RenderHCI

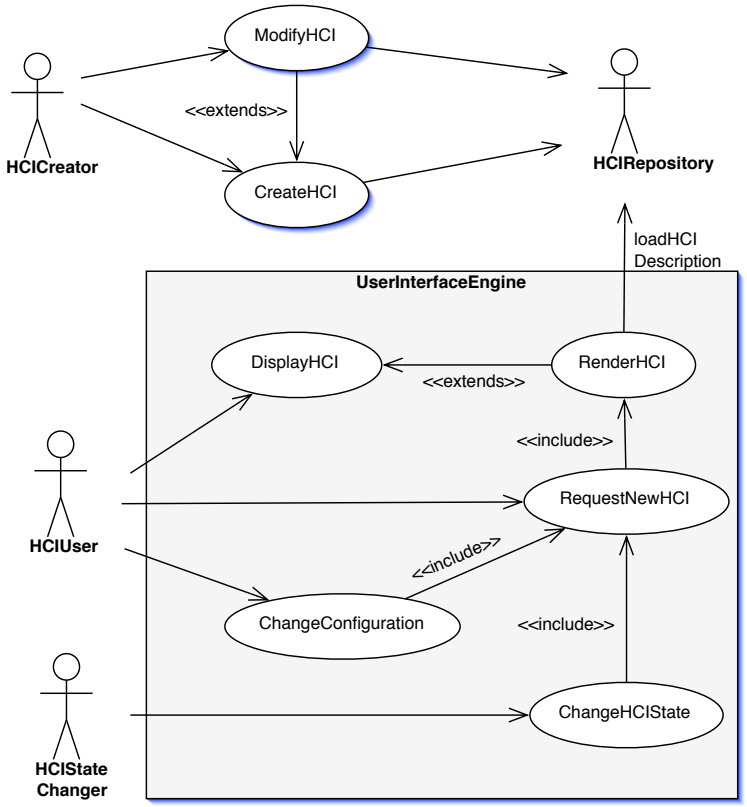


Figure B.4: Use cases of the User Interface Engine

Use case name: **RenderHCI**

Participating actor: Initiated by `HCIRepository`
Communicates with

Flow of Events:

1. *Entry condition:* The `HCIRepository` contains the documents that are needed to create a HCI according to the hardware configuration of the wearable.
2. The `HCIRepository` supplies the input needed for the transformation process (abstract descriptions, mapping information).
3. A HCI is built according to the above mentioned information.
4. *Exit condition:* The HCI components are generated and kept in the memory of the wearable.

DisplayHCI

Use case name:	**DisplayHCI**
Participating actor:	Initiated by `HCIRepository` Communicates with `HCIUser`
Flow of Events:	1. *Entry condition:* The "Render HCI" use case has been applied and thus the HCI components are kept in the memory of the wearable. 2. The HCI components are activated. 3. The components are visible to the `HCIUser`. 4. *Exit condition:* The HCI components are displayed to the user.

RequestNewHCI

Use case name:	**RequestNewHCI**
Participating actor:	Initiated by `HCIUser` Communicates with
Flow of Events:	1. *Entry condition:* The *DisplayHCI* use case has led to a display of the HCI views. 2. The `HCIUser` interacts with the HCI in a way that leads to the necessity of a new HCI. 3. The *RenderHCI* use case is triggered. 4. *Exit condition:* The newly generated HCI components are displayed to the user.

ChangeConfiguration

Use case name:	**ChangeConfiguration**
Participating actor:	Initiated by `HCIUser` Communicates with
Flow of Events:	1. *Entry condition:* The *DisplayHCI* use case has led to a display of the HCI views. 2. The user changes his current wearable configuration. 3. The *RequestNewHCI* use case is triggered with the updated hardware configuration. 4. *Exit condition:* The newly created HCI has to be displayed as described in the *DisplayHCI* case.

ChangeHCIState

Use case name:	**ChangeHCIstate**
Participating actor:	Initiated by `HCIChanger` Communicates with
Flow of Events:	1. *Entry condition:* An external event appeared that should change the semantics of the HCI. 2. The `HCIstateChanger` delegates an event to the controller mechanism of the HCI. 3. The controller requests a new HCI if it is not configured to deal with the event. 4. *Exit condition:* The *RequestHCI* use case is triggered.

Functional Requirements

Multi-modality of the system A desired feature of the presentation layer multi-modal human-computer interaction. This means that the system should support user input and system output by various ways. User input could use components such as speech or gesture recognition, system output could use components such as voice output, 3D graphics, and text. These components have different features and properties as well as common ones. This can be used to combine different components to complement each other. For example, in a tour guide the directions are shown in 3D graphics in a head-mounted display in form of an arrow, but information about a tourist attraction is given in text and 2D graphics. The common features of devices allows the user to interact via more than one respective input or output components. Another example is user input by pressing the button "OK" or by saying "OK" to a speech recognition component.

Short response times of the presentation layer An important rule for HCIs is that the response times should be short. The user should get feedback of his actions as quickly as possible. Long response times are annoying to the user and can lead to rejection of the whole system.

Easy creation of HCIs The creation of a HCI should be as simple as possible. Often human factor specialists and cognitive psychologists are involved in the creation of user interfaces. To give them the chance to help in the creation process, they should not have to learn complex programming syntax.

Reuse of parts of HCIs The creation is simplified even more, when common parts of the HCI can be reused, for example from libraries that contain semantic descriptions

of several common tasks.

I/O hardware is subject to change There are two points that have to be considered: First, I/O hardware of one category differs dramatically. For example HMDs differ in resolution, field of view, opacity etc. The user interface description should support as many different hardware versions as possible. Second, during the usage of a wearable system by a user, the user might switch off parts for various reasons. Or he might add components for I/O at runtime. This requirement deserves serious consideration, as a static system would not find the acceptance of users

Simple Adjustment of the design of presentations Because of the differences in I/O hardware, the designer of the views has to make many adjustments to the presentation of the semantics of the HCI. Usability tests and cognitive experiments as well as field studies have to be done. The results of those investigations have to be easily integrable into the presentation, as the designer has to create several versions that are used in the tests.

High-level modelling of HCIs and their underlying logic In the introduction of the requirements analysis, we already pointed out that the information that describes the interaction with the user should be modelled at a high level of abstraction.

Transforming the high-level descriptions to device-dependent representations This requirement follows directly from the previous one. To interact with the user, the high-level models, as described above, have to be transformed according to the context.

Synchronization of multiple views A wearable computer consists of numerous I/O devices. To keep the information presented on the different devices in a consistent state an instance that synchronizes these different views has to be provided. This recurring problem has been formalized with the MVC design pattern.

System Design

Figure B.5 gives an overview of the system. The system is divided into two parts: the front-end and the back-end. The back-end is responsible for generating the front-end, which are all the components that run at the wearable.

Taskflow Engine The Taskflow Engine is a component that is from another part of the DWARF system and was developed by Stefan Riss [126]. The responsibility of the Taskflow Engine is to keep track of the user's current task and accordingly send CUIML documents to the User Interface Engine (UIE). It is important to notice that in DWARF the user is always in a state that is linked to a CUIML document.

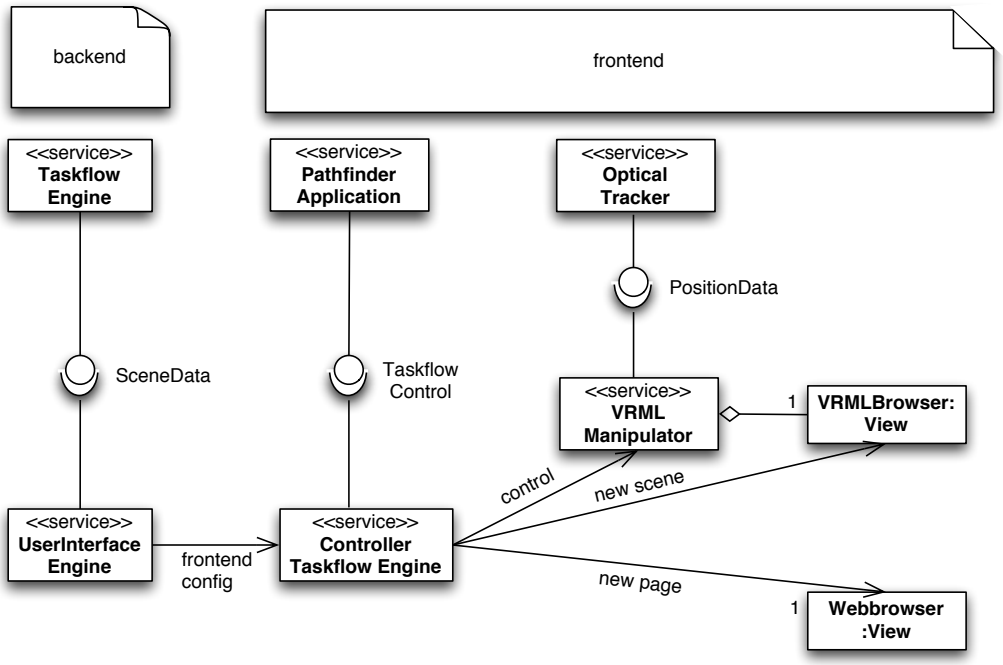


Figure B.5: Conceptual view of the User Interface Engine

The internal state of the Taskflow Engine can be changed by events that are triggered by other DWARF Services that observed a change of the user's context.

User Interface Engine The User Interface Engine is responsible for transforming CUIML documents to the components needed for the front-end. This is done by triggering the appropriate renderers.

View In our concept, a view is the display of markup language. The view can send events to the Controller when the user interacts with it, for example by clicking on a link. To enable this kind of communication an event adapter has to be generated which is not shown in our diagrams for simplicity reasons. The concept of describing and rendering of the views has been adopted from UIML.

Manipulator Another enhancement over UIML is the concept of manipulators. To achieve faster response times, minor changes to the view should be done by the

manipulators instead of rendering new CUIML descriptions to the according view. Views described by markup languages can be accessed using the Document Object Model (DOM) [28]). For other types of views, proprietary access mechanisms have to be found. For example, the Reflection API can be used to access Java objects at runtime.

Controller Taskflow Engine The central component on the wearable is the ControllerTaskflowEngine. It embodies the central instance to synchronize the views and keep track of the current state of the HCI. This is the same component as the Taskflow Engine–it was reused because the requirements were very similar.

System Integration

Figure B.6 shows the specification of the User Interface Engine It consists of the three sub-Services User Interface Engine, Controller Taskflow Engine and VRML Manipulator. The User Interface Engine Service requires Scene Data, the Controller Taskflow Engine requires Context Data and a Taskflow Control and provides User Interface Events, and the VRML Manipulator requires Position Data.

Service Descriptions

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="UIEngine" startOnDemand="false"
       stopOnNoUse="false" startCommand="">
    <need name="currentScene" type="SceneData"
           predicate="" minInstances="0" maxInstances="4242">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
</service>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="ControllerTaskflowEngine" startOnDemand="false"
       stopOnNoUse="false" startCommand="">
    <need name="contextData" type="ContextData"
           predicate="" minInstances="0" maxInstances="4242">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="taskflowControl" type="TaskflowControl"
           predicate="" minInstances="0" maxInstances="4242">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <ability name="userInterfaceEvent" type="UserInterfaceEvent">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
```

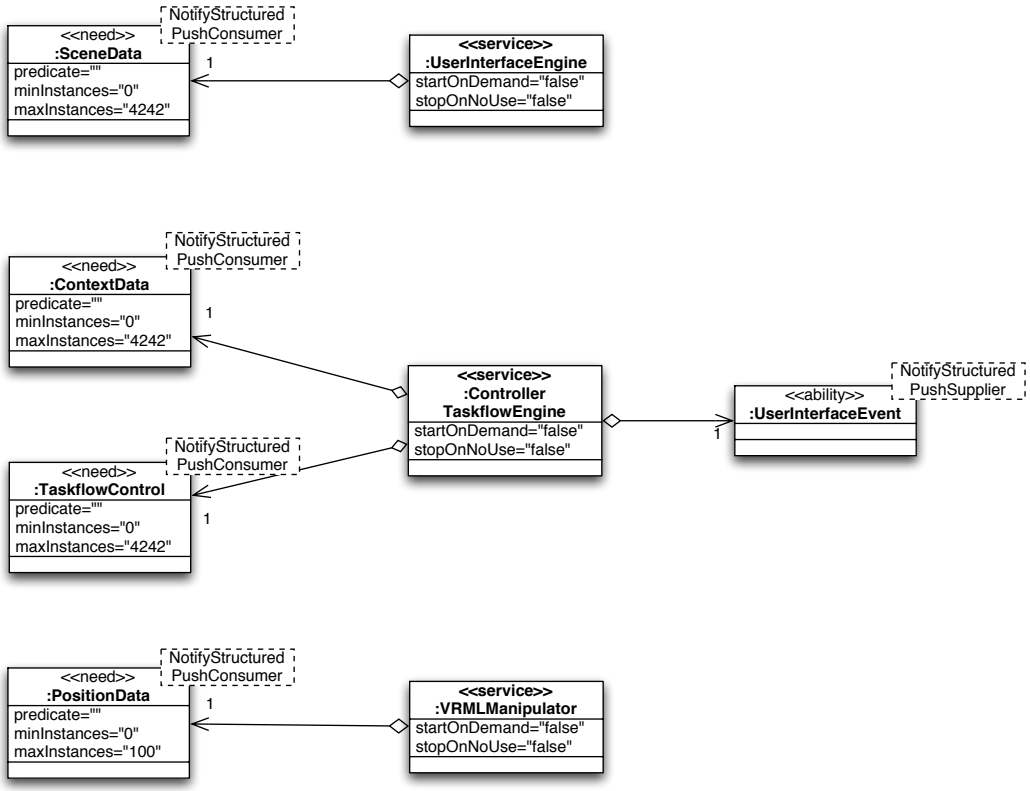


Figure B.6: User Interface Engine Service specification.

```
</service>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="VRMLmanipulator" startOnDemand="false"
       stopOnNoUse="false" startCommand="">
    <need name="PositionDATA" type="PositionData"
          minInstances="0" maxInstances="100">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
</service>
```

B.6 Tracking Manager and Position Trackers

Use Cases

Sensor Fusion. Sensor Fusion or *Data Cooking* is the overall process of receiving data from the various external tracker, filtering the data to get more accurate results and predicting positions where no measurements are available. This use case combines together all other use cases, though the other us cases not necessarily have to be really initiated by this use case but can be self-running independent processes that use the same data as well.

Use case name:	**Sensor Fusion**
Participating actor:	Initiated by `Tracker` Communicates with `Position User`
Flow of Events:	1. A position event that has been emitted by a tracker is received. 2. The position data contained in the event is converted into a common coordinate system and time-base. 3. Using older information, the data is checked for validity. 4. The data is filtered together with the stored data to generate a tracking model of the user 5. While no new input from the tracker is received, the generated model is used to compute new position events. 6. If desired, predictions into the future are made using the computed model. 7. The new position events are sent to the Position User.

Dynamic Handover Sometimes the user leaves the range of a tracker or enters the range of a new one. At this point, the output of both trackers has to be combined.

Use case name:	**Dynamic Handover**
Participating actor:	Initiated by `Tracker` Communicates with `---`
Flow of Events:	1. As the user walks around, his position is tracked by some trackers. 2. The output of those trackers is merged together. 3. When a new tracker suddenly shows up, depending on the tracking model the position data from this tracker is used separately or together with the other outputs and the tracker is added to the set of currently active trackers.

4. Eventually the user leaves the range of one of the active tracker. Then this tracker produces no output anymore. Depending on the tracking model this may be detected and the tracker removed from the set of active trackers.

Collecting Data. In a scenario where several users equipped with mobile tracking devices or using stationary tracking services, usually the position data sent over the network. The event service ensures that once the tracking subsystem of a user only receives position events concerning this user. Nevertheless, this positions may be in different coordinate systems or even of a different time-base.

Use case name:	**Collecting Data**
Participating actor:	Initiated by `Tracker`
	Communicates with `History`
Flow of Events:	1. A position event is received by the tracking subsystem.
	2. The coordinates are converted to a common coordinate system for all positions.
	3. The measurement time is adjusted using the time stamp of the event and the specified latency
	4. The converted data is stored in the History.

Integrity Check. Sometimes a tracker constantly produces wrong, but repeatable results. These errors can be removed by adding a constant offset or by building a lookup table, for example. This can be considered as an additional calibration step for each tracker. Additionally, a tracker producing complete nonsense can be revealed in this step. The algorithms here may vary and are usually based on empirical observations or probabilistic methods.

Use case name:	**Integrity Check**
Participating actor:	Initiated by `Tracker`
	Communicates with `Tracking Model, History`
Flow of Events:	1. New position is stored in the history.
	2. If some calibration exists for this tracker, the offset is applied to the data.
	3. Using the tracking model, the data is checked if it appears to be valid.
	4. The calibration is updated using this information.
	5. If the data appears to be nonsense, it is discarded.

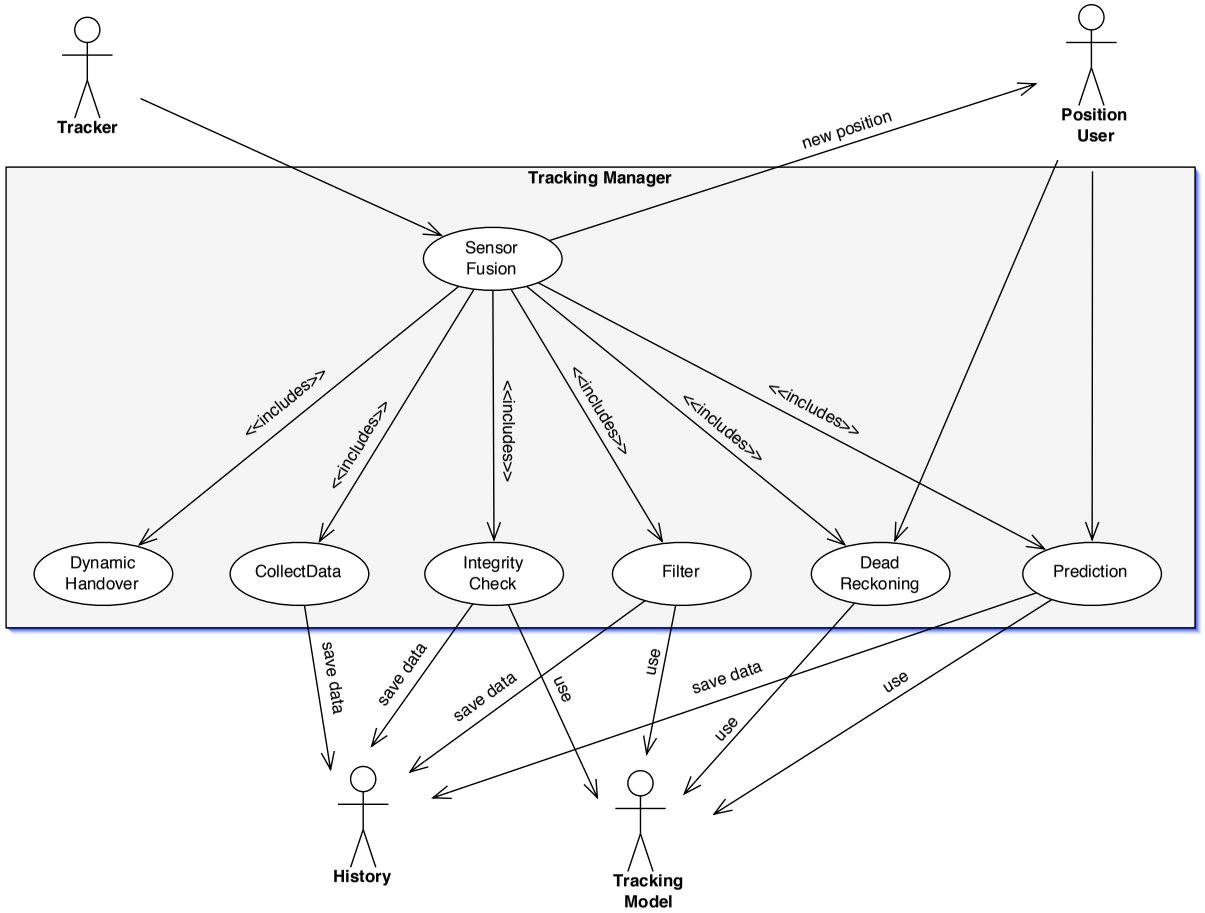


Figure B.7: Use cases of the Tracking Manager.

Filtering. The filtering step is the most important step during data cooking. In this step a model of the users motion, the 'tracking model' is applied to the measurements to adjust the parameters. Models may be simply mathematical functions, like linear movements using in least-square methods or Kalman filtering. More sophisticated models include some kind of world knowledge like assuming that a car only moves on roads and not somewhere else. Adaptive learning algorithms known from robotics can be used as well.

Use case name:	**Filtering**
Participating actor:	Initiated by `Tracker` Communicates with `Tracking Model, History`
Flow of Events:	1. New valid data is stored in the history. 2. According on the algorithm and the tracking model, the new parameters are calculated 3. The calculated parameters are stored with the model.

Dead Reckoning. Dead reckoning occurs when for some period of time the sensors are unable to provide position data. This may happen for example when a car equipped with a GPS receiver drives through a tunnel and looses temporarily the connection to the satellites. Similarly, when the output frequency of the actual tracker is lower than the desired tracking frequency, dead reckoning is necessary between two measurements to ensure a higher output rate.

Use case name:	**Dead reckoning**
Participating actor:	Initiated by `Application` Communicates with `Tracking Model, Position User`
Flow of Events:	1. The application needs current position data. 2. According to the algorithms and the tracking model, an estimate of the current position is calculated based on the stored parameters. 3. The data is sent to the Position User

Prediction. Prediction is necessary, when for some task an estimate of a position in the future is necessary. On space missions to Mars for example, the transmission of a control command to a vehicle takes several minutes to reach the vehicle. Therefore, a prediction into the future is necessary to get the position at the time the control command will reach the vehicle.

Use case name:	**Prediction**

Participating actor: Initiated by `Position User`
Communicates with `Tracking Model, Position User`

Flow of Events:
1. The application needs future position data.
2. According to the algorithms and the tracking model, an estimate of the current position is calculated based on the stored parameters.
3. The data is sent to the Position User
4. The data may be stored to validate the Tracking Model and the parameters in the future.

Functional Requirements

Although the application needs always the accurate pose of the user, its not that important to know how that information is retrieved form the external world. Nevertheless, information about the accuracy of the pose data as well as the delay or update frequency can help the application to change its behaviour according to the actual situation. One scenario could be for example an application that displays additional information overlaid to the view of the user only when the accuracy is good enough not to distract the user, and otherwise switches to a different user interface.

Position Data. The Position Trackers provide means to extract position, *orientation* and *quality of service* parameters from all available tracker using a standard way without the need to know, which systems are actually used to get the data. For a 'stupid' system, this is only the location and orientation of the tracked object; for a more advanced system this includes all the mentioned accuracy and reliability parameters *device abstraction*.

Additionally, position data from different objects that have a fixed pose relative to the tracked object should be used to determine the pose of the object. The process of finding the *relative pose* of fixed objects is called the *calibration* step.

Sensor Fusion and Dead Reckoning. In general, the same object may be tracked by different trackers. The tracking subsystem is responsible for combining these trackers output (sensor fusion) and post processing it to get the highest possible accuracy (*dead reckoning*).

If there is only a single tracker available, then the tracking subsystem should not introduce additional delay. For temporary unavailable trackers or for long tracker update intervals, intermediate values are computed by the tracking subsystem.

Dynamic Handover. When the user moves out from the range of a specific tracker into another ones, this has to be transparent for the application; new trackers are integrated seamlessly with the already used trackers.

Navigation. The tracking subsystem for DWARF is not a navigation system. It gives only the actual pose and orientation of the user; there is no way to extract other navigational data from the interfaces given to the system. It may be desirable some day to extend the DWARF system with another subsystem for navigation, that bases on the tracking component, but we decided to leave this to the application. The reason for this is that navigational tasks are most of the time highly application data dependent. For example, navigation routes for cars should be always on roads, while for example airplanes or helicopters can reach basically every point in three-dimensional space.

Subsystem Design

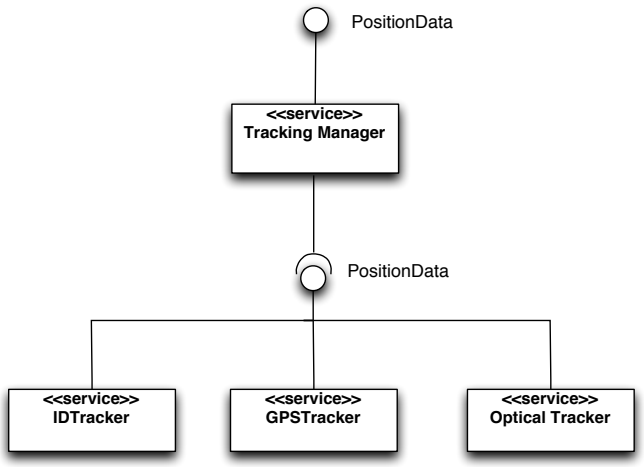


Figure B.8: Architecture of the Tracking subsystem

The tracking subsystem consist of two layers. It is based on a set of different low level trackers which provide by there own a guess of the users pose. These trackers can be either built using a single hardware device like Global Positioning System (GPS) or a gyroscope, or already be hybrid trackers combining two different kinds of tracker hardware in close interaction. The tracking devices provide position data either in all six dimensions or in position or orientation only. Based on those guesses the tracking manager combines the outputs of the trackers and tries to get better results using some

kind of prediction algorithms. This hierarchy can be extended to more subsequent layers; the tracking manager also makes dynamic handover between different trackers possible. On the top of this hierarchy sits the application that uses the processed data.

Each external tracking device that is used in the tracking subsystem gives a new module. All those modules can be distributed over several networked computers. The dynamic combination of different trackers as well as filtering and identification of bad trackers is done in a new module that we call Tracking Manager. The communication between the modules is done using the Service Manager, therefore there is no need for the single modules to know of each other. This enables us to switch the modules on and off as we like and as the current situation requires.

System Integration

Figure B.9 shows the specification of the Tracking subsystem. It consists of the three Services Tracking Manager, GPS Tracker, and ID Tracker. The Tracking Manager Service requires Position Data and a World Model and provides improved Position Data, GPS Tracker and ID Tracker require a World Model and provide Position Data.

Service Description

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="TrackingManager" startOnDemand="false"
       stopOnNoUse="false" startCommand="">
    <need name="world" type="WorldModel"
           predicate="hopcount>=0"
               minInstances="0" maxInstances="4711">
        <connector protocol="CorbaObjImporter"/>
    </need>
    <need name="theposition" type="PositionData" predicate=""
           minInstances="0" maxInstances="4711">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <ability name="myposition" type="PositionData">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
</service>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="GPSTracker" startOnDemand="false"
       stopOnNoUse="false" startCommand="">
    <ability name="position" type="PositionData">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
```

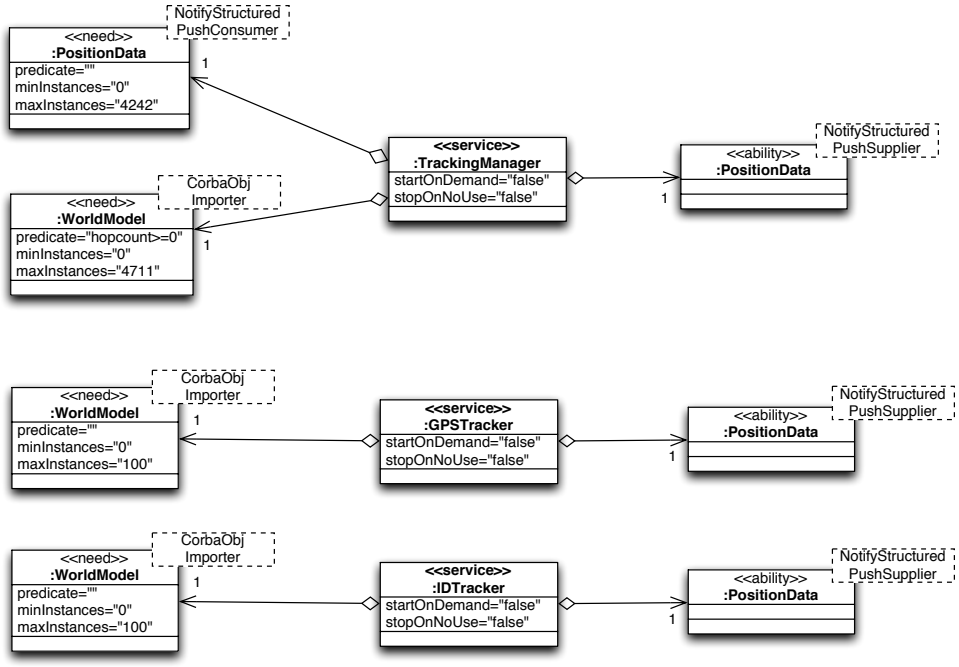


Figure B.9: Tracking Manager and the Position Tracker Service specifications

```
    <need name="world" type="WorldModel"
        minInstances="0" maxInstances="100">
        <connector protocol="CorbaObjImporter"/>
    </need>
</service>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="IDTracker" startOnDemand="false"
        stopOnNoUse="false" startCommand="">
    <ability name="position" type="PositionData">
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
    <need name="world" type="WorldModel"
        minInstances="0" maxInstances="4711">
        <connector protocol="CorbaObjImporter"/>
    </need>
</service>
```

B.7 Optical Tracker

Use Cases

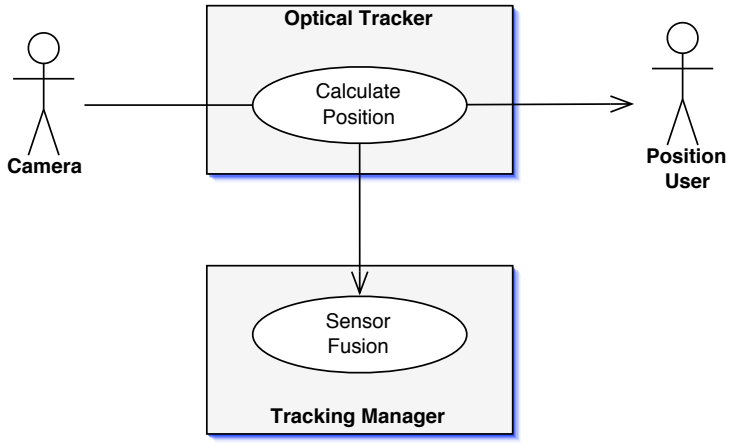


Figure B.10: Use case of the Optical Tracker Service.

Calculate Position The OpticalTracker Service calculates the current pose from video image, forwards the pose to the TrackingManager and the PositionUser. The UIEngine is such a PositionUser.

Use case name:	**CalculatePosition**
Participating actor:	Initiated by `OpticalTracker`
	Communicates with `TrackingManager`
Flow of Events:	1. *Entry condition:* A PositionUser needs the current pose.
	2. The OpticalTracker does image processing on the current video image from the camera.
	3. The calculated pose from the video image is sent to the TrackingManager.
	4. *Exit condition:* The pose is sent to the PositionUser.

Functional Requirements

Pose determination The optical tracker determines the pose (translation and orientation) of a video camera. It uses a live video stream from this camera in order

to detect some artificial landmarks (*fiducials*) in every video image. The real-world position of these fiducials has to be given.

Due to the nature of its task, the optical tracker does not provide any user interface. The optical tracking service does not provide any output capabilities beyond debugging purposes. All access to the optical tracker (except starting it) is done via software interfaces. To ensure the optical tracking being encapsulated from any visualization tasks, even debugging programs for rapid visualization outside the DWARF framework should communicate using the standard DWARF tracking API.

Determination of intrinsic video camera parameters The following (intrinsic) parameters of the video camera have to be determined in an offline procedure:

1. The focal length multiplied by the pixel size in u and v direction of the image frame: α_u and α_v
2. The center point C of the image frame: $C = (u_0, v_0)$
3. The angle between the u and v axis of the image frame: θ

During runtime, the optical tracker gives regular updates of the camera's pose. This data is sent via the DWARF event architecture to other DWARF components using the data.

Dynamic reconfiguration To ensure the use of the tracker in a large environment, there have to be mechanisms that allow dynamic reconfiguration of the Tracking Service's model of the real world. These mechanisms have to be based on data stored in the DWARF World Model Service. The Tracking Service should register for events indicating a change of the user's global position. For every such incoming event, the World Model should be queried for the existence and probably location of fiducials that can be tracked.

Subsystem Design

In subsequent sections, we will give detailed explanations to every subcomponent of the Optical Tracker Service. Each subcomponent is responsible for a particular task in the image processing flow.

Image Acquisition The task of this component is simple: get the image of a video camera attached to the computer and transfer the data in 24-bit RGB format to a memory buffer. However, getting the image data requires in-depth knowledge of several operating system dependent Application Programmer Interface (API). Altogether, three different methods to acquire video data were implemented: from IEEE 1394 digital cameras, USB webcams, and a file video data reader.

Processing the Image Data The processing of the image data acquired from one of the modules described above can be divided in three parts: *fiducial detection*, *optical flow tracking* and *absolute pose estimation*. In addition, an implementation of the relative pose estimation algorithm has been provided, although it is not called in the current tracker implementation.

Fiducial Detection: ARToolKit The Augmented Reality Toolkit [73] has been used to detect the artificial markers mounted in the areas of interest.

The ARToolKit code has been slightly modified to fit our needs, the modifications are marked as such in the source code. For every marker the toolkit detects, a structure `ARMarkerInfo` is put in a shared memory area. These structure holds the two-dimensional information of the detected markers along with information like a confidence value and the marker's direction.

During start-up and if the user enters new rooms, the fiducial detection component reads in marker data from the World Model and stores it in an array of marker data structures.

Optical Flow Tracking For the optical flow tracking a pyramid implementation of the Lucas Kanade optical flow algorithm described in [20] was used. One of the driving forces for this decision was the availability of this algorithm in the Intel OpenCV library [63], leading to minimized implementation efforts.

Once new correlations are established, the corresponding absolute pose is estimated via a function call and distributed to other DWARF components.

Collaboration Between Optical Flow Tracker and Fiducial Detection The tracker we developed is thought of as a proof of concept, not a fully functional implementation.

We simply tested the effect of post-calculating 2D–3D correspondences by means of optical flow without worrying about delay in general or delay being constant or minimized. We did not implement the check for still valid 2D points after each update from the fiducial detector. Instead, we forced the ARToolKit tracker to update its values at an update rate that was ten times slower than the video camera's frame rate. In consequence, the intermediate nine image frames between the fiducial detector's updates were tracked solely using optical flow.

Communication between the two trackers was done as usual using shared memory and semaphores for access control.

Absolute Pose Estimation The absolute pose estimator is called directly from the optical flow tracker and puts the resulting six-dimensional pose in a shared memory area for further processing.

It relies heavily on matrix operations like simple multiplications or the Singular Value Decomposition. These operations have been encapsulated in an object oriented library that facilitates the handling significantly.

Displaying the Results Although the DWARF system has its own user interface engine to display the results of the tracking subsystem, it is preferable to have a separate display component that allows testing of the tracker without the whole DWARF system running. In addition, such a component offers an easy possibility to perform video see-through augmented reality by using shared memory to access the camera image.

This component is implemented as a separate process that has to run on the same computer as the tracker. The two processes communicate via shared memory. Obviously, the interprocess communication is highly platform dependent. To ensure a maximum portability besides this, we decided to use *OpenGL* [170] as the API to draw the images. First, the camera image is taken and drawn as background image, afterwards the viewpoint of the OpenGL scene is set and some simple objects like cylinders are drawn at well-known locations to create a testing environment.

As mentioned above, the displaying task is usually performed by the DWARF User Interface Engine subsystem. This engine internally uses VRML [66] to display correctly aligned virtual objects.

Service Integration

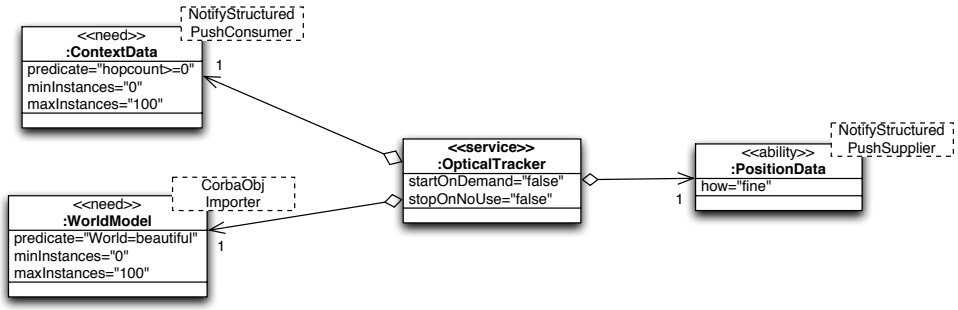


Figure B.11: Optical Tracking Service specification

Figure B.11 shows the specification of the Optical Tracker Service. It requires Context Data and a World Model and provides Position Data.

Basically, there are only two ways of communication between the Optical Tracker and the other DWARF Services:

Sending Tracking Data Every time the Optical Tracker has estimated the camera's six-dimensional pose, it encapsulates this pose with some other information such as the time for which the pose is valid or the delay that occurred during the processing of the image frame and sends it to the DWARF event bus.

The encapsulation is done using a CORBA structured event of type `Any` that contains a single object of type `PositionEvent`. Note that the six-dimensional pose is given in VRML coordinates (i.e. Rodriguez coordinates) as well as a homogeneous matrix.

Obtaining Information About the Current Environment To get up to date information about the currently available features to track, the Optical Tracker registers for so-called `RoomChangedEvents` during start-up. Every time the other components of the DWARF Tracking subsystem realize that the user has entered a new room, such an event is sent. After receiving it, the Optical Tracker checks for and loads the marker data of the new environment as described in section B.7.

Service Description

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="OpticalTracker" startOnDemand="false"
    stopOnNoUse="false" startCommand="">
    <need name="RoomChangedEvent" type="ContextData"
           predicate="hopcount>=0"
               minInstances="0" maxInstances="100">
        <connector protocol="NotifyStructuredPushConsumer"/>
    </need>
    <need name="World" type="WorldModel" predicate="World=beautiful"
           minInstances="1" maxInstances="100">
        <connector protocol="CorbaObjImporter"/>
    </need>
    <ability name="PositionEventSender" type="PositionData">
        <attribute name="how" value="fine"/>
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
</service>
```

B.8 World Model

Use Cases

The scenarios described above allow us to derive use cases specifying further the flow of events of the World Model service. There are only two possible actors that can initiate a use case: the user of a DWARF system (`User`) or other components of the system

(`External`). Figure B.12 describes the relationship between the use cases described below.

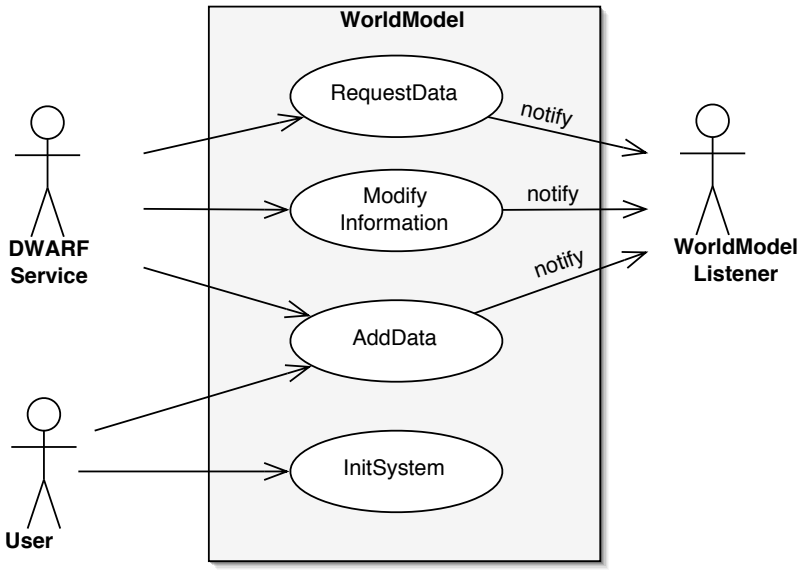


Figure B.12: Use cases describing the World Model's behaviour

RequestData

Use case name:	**RequestData**
Participating actor:	Initiated by `DWARFService` Communicates with `TaskflowListener`
Flow of Events:	1. *Entry condition:* A DWARF service needs information stored in the World Model. 2. The external service sends an information request regarding a certain object to the World Model. 3. If this object does not exist, the World Model notifies the external service. Otherwise, the requested information is returned. 4. *Exit condition:* The DWARF service processes the information.

ModifyInformation

Use case name:	**ModifyInformation**

Participating actor: Initiated by `User or DWARFService`
Communicates with `TaskflowListener`

Flow of Events:
1. *Entry condition:* Another DWARF service has detected a change in the user's environment.
2. The DWARF service either calls the World Model directly or sends an event to the DWARF system bus in order to insert the new information into the World Model database.
3. The World Model changes its data and uses `NotifyDWARF` to inform the other DWARF services.
4. *Exit condition:* A consistent state of information in the overall system is maintained.

AddData

Use case name: **AddData**

Participating actor: Initiated by `User or DWARFService`
Communicates with `TaskflowListener`

Flow of Events:
1. *Entry condition:* Either the user or another DWARF service gets a bunch of new information about the user's environment.
2. The caller invokes the World Model with this information.
3. The World Model adds the given data to its internal memory and uses `NotifyDWARF` to inform the other DWARF services.
4. *Exit condition:* A consistent state of information in the overall system is maintained.

InitSystem

Use case name: **InitSystem**

Participating actor: Initiated by `User`
Communicates with

Flow of Events:
1. *Entry condition:* The user wants to start a DWARF system.
2. The user starts the World Model either with initial data or in an empty state.
3. The World Model initializes itself, registers to the DWARF system bus and reads the data given by the user.
4. *Exit condition:* The World Model is in a valid state and can be used by other DWARF components.

Functional Requirements

The World Model stores information about the real and virtual objects that may be important for the user's interaction with the AR system.

As for most databases, there is no direct user interaction with the World Model. This service itself does not provide any output capabilities beyond debugging purposes. All access to the World Model is done via software interfaces.

Every real or virtual object has an associated position and orientation, its *pose*. To facilitate the development of applications, the World Model has to provide means to compute the pose of one object relative to an arbitrary other.

The objects stored in the World Model are highly variable. To handle this variability in a wide range of possible applications, it has to be possible to store an arbitrary amount of arbitrary information associated with each object.

In addition, it must be possible to change the World Model's content dynamically. As several DWARF services may access the World Model at the same time, there shall be mechanisms that allow consistent multi-threaded access. After every change to the World Model's content, all services wishing to do so must be notified by an efficient event mechanism about the details of this change.

Finally, the World Model has to register for events that indicate the change of an object's position or orientation. Every such change has to be processed by the World Model and, if necessary, stored in the internal data structure.

The DWARF system is designed to be able to work in a large variety of different settings. In consequence, it may occur that more than one World Model service is present. This situation has to be handled.

Subsystem Design

We fixed the central data structure to be a tree. In this section, we will think about the overall software organization of the World Model service.

As we can see in figure B.13, the structure is simple. We have one central object, the `World Model`. This object handles the initialization of the service, the CORBA communication and some high-level functionality as loading new information out of files. The actual data is stored in a set of `Thing` objects organized in a tree data structure. Each `Thing` object has exactly one parent `Thing` and an arbitrary amount of children. To facilitate the loading of files containing World Model entries, we provide an `XML Parser` object that encapsulates all necessary tasks for reading files as described in section B.8.

Persistent Data Management In the current state of development, the World Model holds all data in memory at runtime. However, it may be necessary to add a large amount of data at some point in time, e.g. during start-up or at a situation similar to the Information Download scenario.

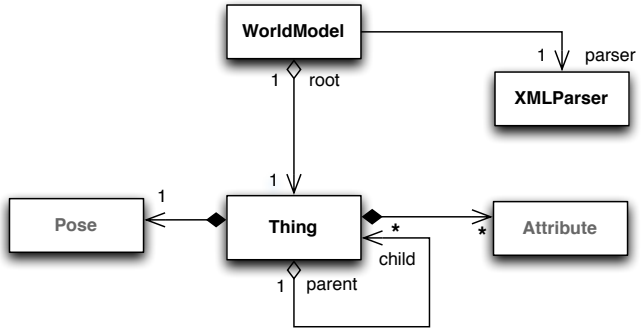


Figure B.13: Subsystem Decomposition of the World Model Service

It seems reasonable to create a possibility to hand a file of arbitrary size to the World Model that has the following properties:

1. Every type of information that can be stored in the World Model can be specified in the file as well.
2. The file should be readable by humans, it has to be possible to modify or create such a file with a simple text editor.
3. There has to be a possibility to add comment lines to the file.
4. It must not be possible to take the World Model in an inconsistent state by reading a malformed file.

A natural choice for these requirements is to use a variant of XML, the *eXtended Markup Language* [161]. With XML, it is possible to use a large variety of existing parsers that perform all error handling based upon a so-called Data Type Definition (DTD) that defines the syntax of well-formed documents describing content of the World Model.

System Integration

Figure B.14 shows the specification of the World Model Service. It requires Position Data. If it finds a Service that provides them it will provide a World Model and Thing Changed Events.

Service Description

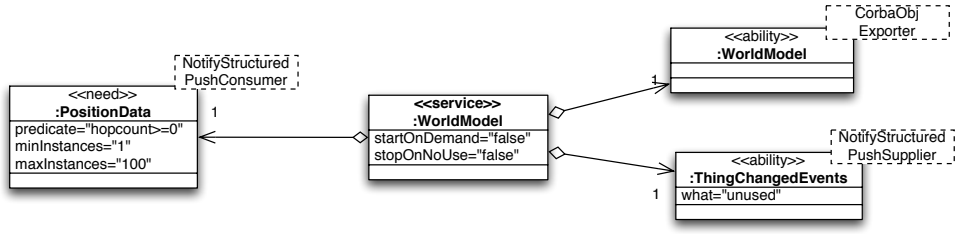


Figure B.14: World Model Service specification.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE service SYSTEM "service.dtd">
<service name="WorldModel" startOnDemand="false"
       stopOnNoUse="false" startCommand="">
    <need name="WMPositionData" type="PositionData"
           predicate="hopcount>=0" minInstances="1"
               maxInstances="100">
        <connector protocol="NotifyStructuredPushConsumer"/>
        <attribute name="bla" value="fasel"/>
    </need>
    <ability name="WorldModelInterface" type="WorldModel">
        <connector protocol="CorbaObjExporter"/>
    </ability>
    <ability name="ThingChangedEventSender" type="ThingChangedEvents">
        <attribute name="what" value="unused"/>
        <connector protocol="NotifyStructuredPushSupplier"/>
    </ability>
</service>
```

Bibliography

[1] *5DT Glove*. http://www.vrealities.com/5dtglove.html, 2003. 20

[2] M. Abrams, C. Phanouriou, A. Batongbacal, S. Williams, and J. Shuster, *UIML: An Appliance Independent XML User Interface Language*. http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html. 135

[3] advanced realtime tracking GmbH, *Infrared Optical Tracking System ARTtrack and DTrack*. http://www.ar-tracking.de/viewtopic.php?t=17, Sep. 2003. 4

[4] R. Allen and D. Garlan, *The Wright Architectural Specification Language*, Tech. Rep. CMU-CS-96-TBD, Carnegie Mellon University, 1996. 33

[5] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickvic, and S. Zimek, *Web Service Choreography Interface (WSCI) 1.0*, Tech. Rep. W3C Note 8 August 2002, World Wide Web Consortium, 2002. 46

[6] ARVIKA consortium, *Internet Presentation of the ARVIKA Project*. http://www.arvika.de, 2003. 19, 34

[7] R. T. Azuma, *A Survey of Augmented Reality*, Presence, 6 (1997), pp. 355–385. 2, 19, 49, 135, 137

[8] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski, *Challenges: an Application Model for Pervasive Computing*, Proceedings of 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000), (2000). 44

[9] W. Barfield and T. Caudell, *Fundamentals of Wearable Computers and Augumented Reality*, Lawrence Erlbaum Assoc, Dec. 2000. 20

[10] *Battlefield Augmented Reality System (BARS) - Website*. http://ait.nrl.navy.mil/vrlab/projects/BARS/BARS.html, 2002. 34

[11] L. Bass, D. Siewiorek, M. Bauer, R. Casciola, C. Kasabach, R. Martin, J. Siegel, A. Smailagic, and J. Stivoric, *Constructing Wearable Computers for Maintenance Applications*, in Fundamentals of Wearable Computers and Augmented Reality, Lawrence Erlbaum Associates, 2001, pp. 663–694. 20

[12] F. L. Bauer, *Meeting of the Study Group on Computer Science of the NATO Science Committee*, 1967. 1

[13] M. Bauer, *DWARF – Design and Prototypical Implementationof a Module for the Dynamic Combination of Different Position Trackers*, Master's thesis, Technische Universität München, Department of Computer Science, Feb. 2001. 3, 138

[14] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner, *Design of a Component-Based Augmented Reality Framework*, in Proceedings of ISAR 2001, New York, USA, 2001, IEEE Computer Society, pp. 124–133. 120

[15] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, C. Sandor, and M. Wagner, *An Architecture Concept for Ubiqutous Computing Aware Wearable Computers*, in Proceedings of the International Workshop on Smart Appliances and Wearable Computing (IWSAWC 2002), Vienna, AU, 2002. 149

[16] R. Behringer, C. Tam, J. McGee, S. Sundareswaran, and M. Vassiliou, *A Wearable Augmented Reality Testbed for Navigation and Control Built Solely with Commercial-Off-The-Shelf (COTS) Hardware*, in Proceedings of ISAR 2000, Munich, Oct. 2000, pp. 12–19. 42

[17] M. Billinghurst, H. Kato, and I. Poupyrev, *The MagicBook: Moving Seamlessly between Reality and Virtuality*, IEEE Computer Graphics and Applications, (2001). 34

[18] O. Bimber, M. L. Ecarnação, and D. Schmalstieg, *The Virtual Showcase as a new Platform for Augmented Reality Digital Storytelling*, in Proceedings of the 7th International Immersive Projection Technologies Workshop, J. Deisinger and A. Kunz, eds., 2003. 28

[19] *Bluetooth Special Interest Group Website*. `http://www.bluetooth.com`, 2001. 6, 29

[20] J.-Y. Bouguet, *Pyramidical Implementation of the Lucas Kanade Feature Tracker. Description of the algorithm*. Part of the Intel Computer Vision Library Documentation, 2000. 194

[21] P. Boulanger, J. Taylor, S. El-Hakim, and M. Rioux, *How to Virtualize Reality: An Application to the Re-creation of World Heritage Sites*, in Proceedings of VSMM98, vol. I, Gifu, Japan, Nov. 1998, International Society on Virtual Systems and Multimedia, pp. 39–45. 28

[22] B. Bruegge and A. H. Dutiot, *Object-Oriented Software Engineering – Using UML, Patterns, and Java*, Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, 2 ed., 2003. 1, 12, 172

[23] K. Burr and A. Dove, *Wearable Personal Computer System*. US patent, 2000-08-17. 43

[24] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A Systems of Patterns*, John Wiley & Sons, 1996. 14, 54, 66, 70, 74, 117

[25] A. Butz, T. Höllerer, S. Feiner, B. MacIntyre, and C. Beshers, *Enveloping Users and Computers in a Collaborative 3D Augmented Reality*, in Proceedings of IWAR '99 (Int. Workshop on Augmented Reality), San Francisco, CA, USA, Oct. 1999, pp. 35–44. 34, 69

[26] D. W. Carroll, *Wearable personal computer system*. US patent, Feb. 1996. 43

[27] D. W. Carroll, *Wearable personal computer system having flexible battery forming casing of the system*. US patent, Nov. 1996. 43

[28] W. Consortium, *DOM specification*. `http://www.w3.org/DOM/`. 182

[29] S. Corson and J. Macker, *Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations*. Request for Comments: 2501, Jan. 1999. 27

[30] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas, *The Design of a Configurable and Reconfigurable Middleware Platform*, IEEE Distributed Computing, (2001). 46

[31] I. Craig, *Blackboard Systems*, Ablex Publishing Corporation, Norwood, NJ, USA, 1995. 60

[32] J. Craig, *Flexible wearable computer*. US patent, 2000-08-22. 42

[33] D. Curtis, D. Mizell, P. Gruenbaum, and A. Janin, *Several Devils in the Details: Making an AR Application Work in the Airplane Factory*, in Augmented Reality - Placing Artificial Objects in Real Scenes, R. Behringer, G. Klinker, and D. W. Mizell, eds., A K Peters, Ltd., 1999. 4, 34

[34] A. D. Dey, *Providing Architectural Support for Building Context-Aware Applications*, PhD thesis, Georgia Institute of Technology, 2000. 7, 60

[35] *Dictionary.com/software engineering*. `http://dictionary.reference.com`, 2003. 1

[36] J. Dorsey and D. Siewiorek, *The Design of Wearable Systems: A Shift in Development Effort*, in Proceedings of the International Conference on Dependable Systems and Networks (DSN-2003), San Francisco, USA, Jun. 2003. 20, 43

[37] F. Eliassen, T. Plagemann, B. Hafskjold, T. Kristensen, H. O. Rafaelsen, and R. H. Macdonald, *QoS Management in the MULTE-ORB*, IEEE Distributed Systems Online, (2002). 46

[38] *GNU Emacs - GNU Project - Free Software Foundation (FSF)*, 2003. 28

[39] S. Feiner, B. MacIntyre, T. Höller, and T. Webster, *A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment*, in Proc. ISWC '97 (First Int. Symp. on Wearable Computers), Cambridge, MA, USA, Oct. 1997. 69

[40] S. Feiner, B. MacIntyre, and D. Seligmann, *Knowledge-based augmented reality*, Communications of the ACM, 36 (1993), pp. 52–62. 69

[41] W. Friedrich, *ARVIKA - Augmented Reality for Development, Production and Service*, in Invited talk at the International Symposium on Mixed and Augmented Reality, Darmstadt, Germany, Oct. 2002. 28

[42] W. Friedrich, D. Jahn, and L. Schmidt, *ARVIKA - Augmented Reality for Development, Production and Service*. Proceedings of the International Status Conference HCI, 2001. 32, 34, 53, 70

[43] W. Friedrich and W. Wohlgemuth, *ARVIKA - Augmented Reality for Development, Production and Service*, in The International Workshop on Potential Industrial Applications of Mixed and Augmented Reality, Tokyo, Japan, Oct. 2003. 16, 31, 34

[44] J. Fründ, J. Gausemeier, C. Matysczok, G. Mnich, and A. von Fircks, *AR-based Configuration and Information Retrieval of Household Appliances on Mobile Devices*, in Proceeding CHINZ03, 2003. 34

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995. 49, 54, 68, 70, 79, 173

[46] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, *Proactive Self-Tuning System for Ubiquitous Computing*, in Proceedings of the Large Scale Networks Conference, Arlington (VA), March 2001. 34

[47] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, *Project Aura: Towards Distraction-Free Pervasive Computing*, IEEE Pervasive Computing, special issue on"Integrated Pervasive Computing Environments", 1 (2002). 34, 44, 45

[48] J. H. J. Garrett and A. Smailagic, *Wearable computers for field inspectors: Delivering data and knowledge-based support in the field*, in AI in Structural Engineering, 1998, pp. 146–164. 42

[49] L. Gong, *Project JXTA: A Technology Overview*, tech. rep., Sun Microsystems, 2002. 45, 79

[50] S. D. Gribble, M. Welsh, J. R. von Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Y. Zhao, *The Ninja Architecture for Robust Internet-Scale Systems and Services*, Computer Networks, 35 (2001), pp. 473–497. 44, 46

[51] P. Gussmann, *The Workflow Editor for Mobile Industrial AR*, in The International Workshop on Potential Industrial Applications of Mixed and Augmented Reality, Tokyo, Japan, Oct. 2003. 39

[52] E. Guttman, C. Perkins, J. Veizades, and M. Day, *Service Location Protocol*. IETF, RFC 2608, June 1999. 27, 45

[53] Handykey Corporation, *Twiddler 2*. `http://www.handykey.com/site/twiddler2.html`, 2003. 20

[54] J. Hendler, T. Berners-Lee, and E. Miller, *Integrating Applications on the Semantic Web*, Journal of the Institute of Electrical Engineers of Japan, 122 (10) (2002). 45

[55] C. K. Hess, M. Román, and R. Campbell, *Building Applications for Ubiquitous Computing Environments*, in International Conference on Pervasive Computing (Pervasive 2002), Zurich, Switzerland, August 2002, pp. 16–29. 44

[56] M. Hiller, *Software Fault Tolerance Techniques from a Real-Time Systems Point of View: An Overview*, Tech. Rep. 98-16, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, Sweden, 1998. citeseer.nj.nec.com/hiller98software.html. 7

[57] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins, eds., *GPS – Theory and Practice*, Springer, Wien, New York, 4th ed., 1997. 141

[58] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Object Technology Series, Addison-Wesley Publishing Company, 2000. 10, 12

[59] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, and M. Schwehm, *Next Century Challenges: Nexus - An Open Global Infrastructurefor Spatial-Aware Applications*, in Proceedings of Mobicom '99, Seattle, Washington, USA, 1999. 62

[60] T. Höllerer, S. Feiner, T. Terauchi, G. Rashid, and D. Hallaway, *Exploring MARS: Developing Indoor and Outdoor User Interfaces to a Mobile Augmented Reality System*, Computers and Graphics, 6 (1999). 34

[61] *ImageTcl Multimedia Develpment System - Website*. `http://metlab.cse.msu.edu/imagetclar/`, 2003. 32, 34

[62] Infineon Technologies, *Infineon Wearable Technologies - Wearable Electronics*. `http://www.wearable-electronics.de/`, 2003. 24

[63] Intel Corporation, *Open Source Computer Vision Library*. `http://www.intel.com/research/mrl/research/cvlib/`, January 2001. Available for Linux and the Microsoft Windows Platform. 194

[64] International Telecommunications Union, *Specificaiton and Description Language (SDL)*, 2002. 148

[65] N. Ioannidis, *ARCHEOGUIDE*. `http://archeoguide.intranet.gr/project.htm`, 2002. 34

[66] ISO, *VRML97 International Standard*. `http://www.web3d.org/technicalinfo/specifications/ISO_IEC_14772-All/index.html`, 2003. 58, 62, 195

[67] *ITU-T X.901 — ISO/IEC 10746-1 ODP Reference Model Part 1. Overview*, 1995. 33

[68] C. M. Janik, *Flexible wearable computer*. US patent, Dec. 1996. 43

[69] M. Jenkins and S. Hussein, *Modular wearable computer*. US patent, 2000-10-25. 42

[70] *Jini*. `http://www.jini.org`, February 2001. 27

[71] E. L. Jorgensen, *DoD Classes of Electronic Technical Manuals*, tech. rep., Carderock Division, Naval Surface Warfare Center, Apr. 1994. 21

[72] G. Kan, *Gnutella*, in Peer-to-Peer: Harnessing the Power of Disruptive Technologies, Mar. 2001. 79

[73] H. Kato, M. Billinghurst, R. Blanding, and R. May, *ARToolKit PC version 2.11*, December 1999. `http://www.hitl.washington.edu/research/shared_space/download`. 3, 194

[74] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, and B. Serra, *people, places, things: web presence for the real world*, tech. rep., Hewlett-Packard Laboratories, 2003. 44, 45

[75] T. Kirste and S. Rapp, *Architecture for Multimodal Assistance Systems.* Proceedings of the International Status Conference HCI, Oct. 2001. 54

[76] G. Klinker and B. Brügge, *Einführung in die erweiterte Realität.* Lecture at the Technische Universität München, May 2000. 4

[77] G. Klinker, O. Creighton, A. Dutoit, R. Kobylinski, C. Vilsmeier, and B. Bruegge, *Augmented maintenance of powerplants: A prototyping case study of a mobile AR system*, in IEEE and ACM International Symposium on Aumgented Reality ISAR 2001, Oct. 2001. 4, 28, 139, 148

[78] G. Klinker, H. Najafi, T. Sielhorst, F. Sturm, F. Echtler, M. Isik, W. Wein, and C. Trübswetter, *FixIt: An Approach towards Assisting Workers in Diagnosing Machine Malfunctions*, in Submitted to the International Workshop on Exploring the Design and Engineering of Mixed Reality Systems, Funchal, Island of Madeira, Portugal, Jan. 2004. 146

[79] G. Klinker, T. Reicher, and B. Brügge, *Distributed User Tracking Concepts for Augmented Reality Applications*, in Proceedings of ISAR 2000, Munich, Oct. 2000, pp. 37–44. 8, 75, 145

[80] G. Klinker, D. Stricker, and D. Reiners, *Augmented Reality: A Balance Act between High Quality and Real-Time Constraints*, in Mixed Reality - Merging Real and Virtual Worlds (Proc. of the 1st International Symposium on Mixed Reality (ISMR'99)), Y. Ohta and H. Tamura, eds., Springer-Verlag, Mar. 1999. 145

[81] K. W. Kolence and P. J. Kiviat, *Software Unit Profiles and Kiviat Figures*, ACM SIGMETRICS, Performance Evaluation Review, 2 (1973), pp. 2–12. 30

[82] G. E. Krasner and S. T. Pope, *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*, tech. rep., ParcPlace Systems, Inc., Mountain View, USA, 1988. 44, 48

[83] Lart, *Project Home Page*. TU Delft, `http://www.lart.tudelft.nl`, 2001. 43

[84] F. Leymann, *Web Services Flow Language (WSFL 1.0)*, tech. rep., IBM Software Group, 2001. 46

[85] B. MacIntyre and S. Feiner, *Language-level support for exploratory programming of distributed virtual environments*, in Proc. UIST '96 (ACM Symp. on User Interface Software and Technology), Seattle, WA, USA, Nov. 1996, pp. 83–95. 69

[86] B. MacIntyre and M. Gandy, *Prototyping Applications with DART, The Designer's Augmented Reality Toolkit*, in International Workshop on Software Technology for Augmented Reality Systems (STARS 2003), Tokyo, Japan, Oct. 2003. 148

[87] A. MacWilliams, *DWARF – Using Ad-Hoc Services for Mobile Augmented Reality Systems*, Master's thesis, TU München, Department of Computer Science, Feb. 2000. 131

[88] A. MacWilliams, T. Reicher, and B. Brügge, *Decentralized Coordination of Distributed Interdependent Services*, in IEEE Distributed Systems Online – Middleware Work in Progress Papers, Rio de Janeiro, Brazil, June 2003. 147

[89] A. MacWilliams, C. Sandor, M. Wagner, and B. Bruegge, *A Component-Based Approach to Developing Mobile Maintenance Applications*. Report on the TRAMP student project, 2002. 146

[90] *Mobile Augmented Reality - Website*. `http://www.cs.columbia.edu/graphics/projects/mars/mars.html`, 1999. 34

[91] B. Meyer, *Applying Design by Contract*, IEEE Computer, 25 (1992), pp. 40–51. 33, 80

[92] F. Michahelles, *DWARF – Designing an Architecture for Context-Aware Service Selection and Execution*, Master's thesis, Universität München, Department of Computer Science, Feb. 2000. 131

[93] Microsoft Corporation, *Understanding Universal Plug and Play*. `http://upnp.org/download/UPNP_UnderstandingUPNP.doc`, Jun. 2000. 45

[94] Microsoft Corporation, *COM+ Component Model*. `http://www.microsoft.com/com`, 2003. 8, 45, 76

[95] Microsoft Corporation, *.NET Homepage*. `http://www.microsoft.com/net`, 2003. 45

[96] J. Miller and J. Mukerji, *MDA Guide Version 1.0*. `http://www.omg.org/cgi-bin/doc?mda-guide`, May 2003. 148

[97] MIT Laboratory for Computer Science, *MIT Project Oxygen: Overview*. `http://oxygen.lcs.mit.edu/Overview.html`, 2003. 44, 45

[98] MIThril, *Project Home Page*. Massachussets Institue of Technology, `http://www.media.mit.edu/wearables/mithril/`. 42, 43

[99] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics, 38 (1965). 5

[100] National Research Council Committee for the Electric Power of the Dismounted Soldier, National Research Council, *Energy-Efficient Technologies for the Dismounted Soldier*, National Academy Press, Washington, D.C., 1997. 24

[101] P. Naur and B. Randell, eds., *NATO Software Engineering Conference*, Garmisch, Germany, Oct. 1968. 1

[102] L. Nigay and J. Coutaz, *A design space for multimodal systems - concurrent processing and data fusion*, in INTERCHI '93 - Conference on Human Factors in Computing Systems, Amsterdam, 1993, Addison Wesley. 134

[103] Object Management Group, *CORBA 2.4.2 Specification*. formal/01-02-33, 2001. 76

[104] Object Management Group, *CORBA Components v3.0*. formal/02-06-65, June 2002. 45

[105] Object Management Group, *Real-Time CORBA Specification*, Tech. Rep. formal/02-08-02, Object Management Group, 2002. 46

[106] Object Management Group, *CORBA Trader Service Specification*. `http://www.omg.org`, 2003. 26, 45

[107] Object Management Group, *UML 2.0 Infrastructure Final Adopted Specifcation*, 2003. 112

[108] Object Management Group, *UML 2.0 Superstructure Final Adopted specification*, 2003. 32, 33, 51, 112

[109] Object Management Group (OMG), *Common Object Request Broker: Architecture and Specification, CORBA 2.6.1*. `http://www.omg.org/cgi-bin/doc?formal/02-05-08`, 2002. 8

[110] OpenSG Forum, *OpenSG Home*. `http://www.opensg.org/`, 2003. 58

[111] T. Oshima, *RV-Border Guards: A multiplayer entertainment in mixed reality space*, in Poster session of IEEE Internation Workshop on Augmented Reality, San Francisco, USA, 1999. 38

[112] C. Owen, A. Tang, and F. Xiao, *ImageTclAR: A Blended Script and Compiled Code Development Systems for Augmented Reality*, in International Workshop on Software Technology for Augmented Reality Systems (STARS 2003), Tokyo, Japan, Oct. 2003. 69

[113] W. Pasman and F. W. Jansen, *Distributed Low-latency Rendering for Mobile AR*, in Proceedings of the International Symposium on Augmented Reality, IEEE Computer Society, 2001, pp. 107–113. 34, 40, 89

[114] C. Perkins, *IP Mobility Support for IPv4*. IETF RFC 3344, Aug. 2002. 25

[115] W. Piekarski and B. Thomas, *An Object Oriented Software Architecture for 3D Mixed Reality Applications*, in Proceedings of the International Symposium on Mixed and Augmented Reality, Oct. 2003. 69

[116] W. Piekarski and B. H. Thomas, *The Tinmith System - Demonstrating New Techniques for Mobile Augmented Reality Modelling*, in 3rd Australasian User Interfaces Conference, Melbourne, Australia, Jan. 2002. 34

[117] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, L. Thiele, and G. Tröster, *The Case for Reconfigurable Hardware in Wearable Computing*, Personal and Ubiquitous Computing, 7 (2003), pp. 299–308. 149

[118] T. Reicher, *Augmented Reality in Entwicklung, Produktion und Service: Komponentenspezifikation Workflow Engine*, tech. rep., ARVIKA, 2000. 132, 172

[119] T. Reicher and A. MacWilliams, *Study on Software Architectures for Augmented Reality Systems, report for the ARVIKA consortium*, tech. rep., Technische Universität München, 2002. 17, 32, 34

[120] T. Reicher, A. MacWilliams, and B. Bruegge, *Towards a System of Patterns for Augmented Reality Systems*, in International Workshop on Software Technology for Augmented Reality Systems (STARS 2003), Tokyo, Japan, Oct. 2003. 146

[121] T. Reicher, A. MacWilliams, B. Brügge, and G. Klinker, *Results of a Study on Software Architectures for Augmented Reality Systems*, in Proceedings of the International Symposium on Mixed and Augmented Reality, Tokio, Japan, October 2003. 146

[122] D. Reiners, D. Stricker, G. Klinker, and S. Müller, *Augmented Reality for Construction Tasks: Doorlock Assembly*, in Proceedings of the International Workshop on Augmented Reality (IWAR '98), Nov. 1998. 4, 28

[123] G. Reitmayr and D. Schmalstieg, *OpenTracker – An Open Software Architecture for Reconfigurable Tracking Based on XML*, tech. rep., Vienna University of Technology, 2001. 138

[124] G. Reitmayr and D. Schmalstieg, *Data Management Strategies for Augmented Reality*, in International Workshop on Software Technology for Augmented Reality Systems (STARS 2003), Tokyo, Japan, Oct. 2003. 62, 148

[125] B. Rhodes and P. Maes, *Just-in-time information retrieval agents*, IBM Systems Journal special issue on the MIT Media Laboratory, 39 (2000), pp. 685–704. 27

[126] S. Riss, *DWARF – A XML based Task Flow Description Language for Augmented Reality Applications*, Master's thesis, Technische Universität München, Department of Computer Science, Feb. 2001. 133, 180

[127] C. Robertson and B. MacIntyre, *Adapting to Registration Error in an Intent-Based Augmentation System*, in ACM User Interface Software and Technology 2002 (UIST 2002), Paris, France, Oct. 2002. Presented as a poster. 34

[128] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt, *Gaia: A Middleware Infrastructure to Enable Active Spaces*, IEEE Pervasive Computing, Oct-Dec (2002), pp. 74–83. 44

[129] M. Román, D. Mickunas, F. Kon, and R. Campbell, *LegORB and Ubiquitous CORBA*, in Workshop on Reflective Middleware at Middleware 2000, New York, USA, 2000. 46

[130] C. Sandor, *DWARF – CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces*, Master's thesis, Technische Universität München, Department of Computer Science, Feb. 2001. 136

[131] C. Sandor, A. MacWilliams, M. Wagner, M. Bauer, and G. Klinker, *Herding Sheep: Live System Development for Distributed Augmented Reality*, in IEEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2003, Tokyo, Japan, 2003. 146

[132] C. Sandor and T. Reicher, *CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces*, in Proceedings of the European UIML conference, 2001. 124

[133] A. Savidis and C. Stephanidis, *Interacting with the Disappearing Computer: Interaction Style, Design Method, and Development Toolkit*, Tech. Rep. 317, ICS-FORTH, Heraklion, Crete, Greece, Dec. 2002. 43

[134] D. Schmalstieg and G. Hesina, *Distributed Applications for Collaborative Augmented Reality*, IEEE Virtual Reality, (2002). 69, 70

[135] B. Schmerl, *xAcme: CMU Acme Extensions to xArch*, tech. rep., Carnegie-Mellon-University, 2001. 33

[136] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*, Wiley, New York, NY, 2000. 49

[137] L. Schmidt, O. Oehme, S. Wiedenmaier, A. Beu, and P. Quaet-Faslem, *Usability Engineering für Benutzer. Interaktionskonzepte von Augmented-Reality-Systemen.*, it+ti - Informationstechnik und Technische Informatik, (2002), pp. 31–39. 28, 30

[138] *Siemens - SCR - Website*. `http://www.scr.siemens.com/2a.html`, 2002. 34

[139] M. Shaw and P. Clements, *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*, in Proceedings of The 21st Computer Software and Applications Conference, 1994. 11, 74

[140] M. Shaw and D. Garlan, *Software Architectur: Perspective on an Emerging Discipline*, Prentice Hall, Upper Saddle River, NJ, 1996. 60

[141] J. A. Stankovic and K. E. Ramamritham, *Tutorial: Hard Real-Time Systems*, IEEE Press, 1988. 7

[142] STAR consortium, *STAR website*. `http://www.realviz.com/STAR/`, 2002. 34, 39, 89

[143] P. Strauss and R. Carey, *An object-oriented 3D graphics toolkit*, Computer Graphics (SIGGRAPH 1992 Proceedings), (1992), pp. 341–349. 58, 60, 62

[144] D. Stricker, P. Dähne, F. Seibert, I. Christou, L. Almeida, and N. Ioannidis, *Design and Development Issues for ARCHEOGUIDE: An Augmented Reality-based Cultural Heritage On-site Guide*, in EuroImage ICAV 3D Conference in Augmented Virtual Environments and Three-dimensional Imaging, Mykonos, Greece, 2001. 28, 30

[145] *Studierstube Augmented Reality Project - Website*. `http://www.studierstube.org`, 2003. 32, 34

[146] Sun Microsystems, *JavaBeans*. `http://java.sun.com/products/javabeans/`, 2001. 45

[147] Sun Microsystems, *Sun Open Network Environment (ONE) Software Architecture*. `http://www.sun.com/products/sunone/wp-arch/`, 2001. 45

[148] Sun Microsystems, *Enterprise JavaBeans Specification, v2.3*. `http://java.sun.com/products/ejb/docs.html`, 2003. 45

[149] I. E. Sutherland, *A Head-Mounted Three-Dimensional Display*, in AFIPS Conference Proceedings, vol. 33, 1968, pp. 757–764. 20

[150] C. Szyperski, *Component Software - Beyond Object-Oriented Programming - Second Edition*, Addison-Wesley and ACM Press, 2002. 32

[151] S. Thatte, *XLANG - Web Services for Business Process Design*, tech. rep., Microsoft Corporation, 2001. 46

[152] *Tinmith - Website*. `http://www.tinmith.net`, 2002. 34

[153] *Ubiquitous Commmunications program, UbiCom home - Website*. `http://www.ubicom.tudelft.nl`, 2003. 34, 40

[154] S. Uchiyama, K. Takemoto, K. Satoh, H. Yamamoto, and H. Tamura, *MR Platform: A Basic Body on Which Mixed Reality Applications are Built*, in Proceedings of the International Symposium on Mixed and Augmented Reality, Darmstadt, Germany, 2002. 32, 34, 38

[155] US Department of Defense, *Requirements for Interactive Electronic Technical Manuals abd Associated Technical Information*, Nov. 1992. 133

[156] US Department of Defense, *General Content, Style, Format, and User Requirements for Interactive Electronic Technical Manuals*, Oct. 1995. 133

[157] US Department of Defense, *Performance Specification - Revisable Database for the Support of Interactive Electronic Technical Manuals*. `http://navycals.dt.navy.mil/ietm/`, Oct. 1995. 21

[158] US Department of Defense, *Revisable Database for the Support of Interactive Electronic Technical Manuals*, Oct. 1995. 133

[159] US Department of Defense, *Technical Manual - General Content, Style, Format and User Requirements for Interactive Electronic Technical Manuals*. `http://navycals.dt.navy.mil/ietm/`, Oct. 1995. 21

[160] ViA Inc., *Home Page*. `http://www.via-pc.com`. 42

[161] W3C XML Working Group, *The annotated XML specification*. `http://www.xml.com/axml/testaxml.htm`, January 2001. 200

[162] M. Wagner, *DWARF – Design, Prototypical Implementation and Testing of a Real-Time Optical Feature Tracker*, Master's thesis, Technische Universität München, Department of Computer Science, Feb. 2001. 139, 141

[163] M. Wahl, T. Howes, and S. Kille, *Lightweight Directory Access Protocol (v3)*. Request for Comments: 2251, Dec. 1997. 26

[164] J. Waldo, *Jini Architecture Overview*. Sun Microsystems, `http://java.sun.com/products/jini/whitepapers/architectureoverview.pdf`, February 2001. 33, 45

[165] N. Wanga, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz, *Total Quality of Service Provisioning in Middleware and Applications*, Elsevier Journal of Microprocessors and Microsystems, 26 (2003). 46

[166] S. Ward, C. Terman, and U. Saif, *Goal-Oriented System Semantics*, tech. rep., MIT Laboratory for Computer Science, Mar. 2002. 44, 46

[167] S. Ward, C. Terman, and U. Saif, *Pebbles: A Software Component System*, tech. rep., Massachusetts Institute of Technology, Mar. 2002. 44, 45

[168] J. Weidenhausen, *Reuse in the ARVIKA project*. private communication, 2002. 34

[169] M. Weiser, *The computer of the twenty-first century*, Scientific American, (1991), pp. 94–100. 21

[170] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL 1.2 Programming Guide*, Addison Wesley, Reading, MA, 3rd ed., 1999. 58, 62, 155, 195

[171] Xybernaut Corporation, *Home Page*. `http://www.xybernaut.com`. 42

[172] B. Zaun, *A Bluetooth Communication Service for DWARF*. SEP, March 2001. 132

Acronyms

API Application Programmer Interface (page 193)

AR Augmented Reality

ATM Asynchronous Transfer Mode

CAP Context-Aware Packet (page 131)

COM Component Object Model

CORBA Common Object Request Broker Architecture (page 32)

DA Directory Agent

DHCP Dynamic Host Configuration Protocol

DTD Data Type Definition (page 200)

GIS Geo Information System

DWARF Distributed Wearable Augmented Reality Framework

FPGA Field-Programmable Gate Arrays

GNU GNU is Not Unix

GPS Global Positioning System (page 189)

GPRS General Packet Radio Service

GSM Global System for Mobile Communications

HMD head-mounted display (page 2)

HTTP Hypertext Transport Protocol

IDL Interface Definition Language

IETM Interactive Electronic Technical Manuals (page 21)

IETF Internet Engineering Task Force

IP Internet Protocol

LAN Local Area Network (page 25)Local Area Network

LDAP Lightweight Directory Access Protocol (page 26)

MDA Model Driven Architecture (page 148)

ORB Object Request Broker

PC Personal Computer (page 23)

PDA Personal Digital Assistant (page 24)Personal Digital Assistant

POA Portable Object Adapter

RAD Requirements Analysis Document

RAM Random-Access Memory

RFC Request For Comment

RFID Radio Frequence ID (page 121)

RF Radio Frequency

RPC Remote Procedure Call

SA Service Agent

SCP Simple Control Protocol

SDD System Design Document

SLP Service Location Protocol (page 27)

SMB Service Message Block

STARS Sticky Technology for Augmented Reality Systems

TDL Taskflow Definition Language (page 133)

TCP Transmission Control Protocol

UDP User Datagram Packet

UML Unified Modelling Language (page 33)

UMTS Universal Mobile Telecommunications System

UPnP Universal Plug and Play

URL Uniform Resource Locator

VRML Virtual Reality Modelling Language (page 136)

VR Virtual Reality

WLAN Wireless Local Area Network

XML eXtensible Markup Language

Index