TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Institut für Informatik der Technischen Universität München

A Decentralized Adaptive Architecture for Ubiquitous Augmented Reality Systems

Asa MacWilliams

TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Institut für Informatik der Technischen Universität München

A Decentralized Adaptive Architecture for Ubiquitous Augmented Reality Systems

Asa MacWilliams

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. Uwe Baumgarten
Prüfer der Dissertation:	1.	Univ.-Prof. Bernd Brügge, Ph. D.
	2.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy

Die Dissertation wurde am 9. Dezember 2004 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15. Juni 2005 angenommen.

Abstract

Ubiquitous augmented reality is an emerging human-computer interaction technology, arising from the convergence of augmented reality and ubiquitous computing. Augmented reality allows interaction with virtual objects spatially registered in the user's real environment, in order to provide information, facilitate collaboration and control machines. As the computing and interaction devices necessary for augmented reality become ubiquitously available, opportunities arise for improved interaction and new applications.

Building ubiquitous augmented reality systems presents three software engineering challenges. First, the system must cope with uncertainty regarding the software components; the users' mobility changes the availability of distributed devices. Second, during development, the system's desired behavior is ill-defined, as appropriate interaction metaphors are still being researched and users' preferences change. Third, the system must maintain near-real-time performance to create a convincing user experience.

This dissertation presents a new architectural style, the *adaptive service dependency architecture*, to address these challenges.

The architecture deals with component uncertainty, using middleware for decentralized service management to dynamically adapt the system's structure. It builds on existing architectural approaches, such as loosely coupled services, service discovery, reflection, and data flow architectures, but additionally considers the interdependencies between distributed services and uses them for system adaption.

With the development technique of *design at run time*, users and developers address ill-defined requirements by changing the system's behavior while it is running, facilitating the creation of new applications. This is based on existing agile development methods, but takes additional advantage of the architecture's adaptive and reflective properties.

Distributed middleware maintains the required performance by decoupling multimedia data flow from system reconfiguration. It combines existing communication and service discovery technologies, but additionally deals with the distributed coordination of networks of interdependent services.

Several prototype systems for real-world ubiquitous augmented reality applications have been built using the adaptive service dependency architecture, decentralized middleware, and design at run time. Feedback from users and developers, as well as performance tests, show the concepts' usefulness for building ubiquitous augmented reality systems.

Zusammenfassung

Ubiquitous Augmented Reality ist eine neue Form der Mensch-Maschine-Interaktion, die aus der Konvergenz aus Augmented Reality und Ubiquitous Computing entsteht. Augmented Reality erlaubt dem Benutzer die Interaktion mit räumlich registrierten virtuellen Objekten in seiner Umgebung, und stellt damit Informationen bereit, erleichtert Zusammenarbeit und steuert Maschinen. Aus der zunehmenden Verbreitung der Geräte, die für Augmented Reality nötig sind, ergeben sich Möglichkeiten für verbesserte Interaktion und neue Anwendungen.

Die Konstruktion von Ubiquitous-Augmented-Reality-Systemen stellt drei softwaretechnische Herausforderungen. Erstens besteht zur Laufzeit eine Unsicherheit bezüglich der Komponenten; durch Benutzermobilität schwankt die Verfügbarkeit verteilter Geräte. Zweitens ist die gewünschte Funktionalität des Systems während der Entwicklung unvollständig definiert, da Interaktionsmetaphern sich noch im Forschungsstadium befinden und Benutzervorlieben sich ändern. Drittens muss die Performanz den Echtzeitanforderungen für ein überzeugendes Benutzererlebnis genügen.

Diese Dissertation stellt einen neuen Softwarearchitekturstil namens *Adaptive Service Dependency Architecture* vor, der diesen Herausforderungen begegnet.

Die Architektur begegnet der Komponentenunsicherheit mittels Middleware zur dezentralen Diensteverwaltung, die dynamisch die Systemstruktur anpasst. Sie baut auf bestehenden Architekturansätzen wie loser Kopplung, Dienstsuche, Reflektion und Datenflussarchitekturen auf, berücksichtigt aber zusätzlich die wechselseitigen Abhängigkeiten verteilter Dienste zur Adaption.

Mit der Entwicklungstechnik *Design at Run Time* können Benutzer und Entwickler den unvollständigen Anforderungen begegnen und das System zur Laufzeit weiterentwickeln. Dies erlaubt das schnelle Erstellen von Prototypen und vereinfacht die Entwicklung neuer Anwendungen. Diese Technik baut auf bestehenden agilen Entwicklungsmethoden auf, benutzt aber zusätzlich die Adaptivität und Reflektivität der Architektur.

Verteilte Middleware erhält die Systemleistung aufrecht, indem multimediale Datenströme von der Systemadaption entkoppelt werden. Sie kombiniert bestehende Techniken zur Kommunikation und zur Dienstsuche, behandelt aber zusätzlich die verteilte Koordination wechselseitig abhängiger Dienste.

Mit dieser Kombination aus Architektur, Middleware und Entwicklungstechnik wurden einige prototypische Systeme für Ubiquitous-Augmented-Reality-Anwendungen entwickelt. Benutzer- und Entwicklerreaktionen sowie Leistungsmessungen zeigen die Nützlichkeit der Konzepte für die Entwicklung von Ubiquitous-Augmented-Reality-Systemen.

Preface

In the summer of 1999, I had the lucky opportunity to take part in a summer school, organized by the Technische Universität München, on the subject of augmented reality and wearable computers. Since then, I have been fascinated by the subject: a technology offering great potential, but also a multitude of research problems.

As a result of this summer school, a group of enthusiastic students, including myself, determined to build a software framework for mobile augmented reality. We set out to combine what we could learn from our group's professors on the subjects of software engineering (Bernd Brügge) and augmented reality (Gudrun Klinker). The resulting framework was named DWARF, and in my Diplomarbeit, I reported on the middleware of its first prototype.

The domain of ubiquitous augmented reality has proven challenging enough that several members of our group have continued to work in the area, and after four additional years, I am pleased to report the new results.

I have been fortunate to receive the support of many people in completing this dissertation. Thank you, Bernd and Gudrun, for your collaboration on the fields of software engineering and augmented reality, and your support for our interdisciplinary research group. Thank you, Thomas, for the vision of a distributed augmented reality framework. Thank you, Martin, Christian, Martin, and Marcus, for much hard work in our group and in the community. Thank you, Franz, Daniel, Otmar, Markus, Lothar, Dagmar, Felix, Christian, Fabian, Vinko, Korbinian, Niko, Peter, Maximilian, Gerrit, Ming-Ju, Marco, Jan-Gregor, Ming, and many more, for helping make DWARF a reality. Thank you, Dieter, Joe, Gerhard, Thomas, Tamer, Andreas, Christoph, Andi and Horst, for productive collaborations. Thank you, Timo, Allen, Hesam, Christoph, Claudia, Christian, Tao, Naoufel, Michael and Oliver, for feedback on my dissertation. Thank you, Helma, Monika and Uta, for excellent organizational support. And thank you, nameless reviewers, for constructive criticism of submitted papers.

Special thanks to my parents for much support and last-minute proofreading, and of course to Sabine, for unlimited patience and love.

And finally, to you, the reader: thank you for the time you are investing to read this dissertation—I hope it proves worthwhile!

For questions or feedback, please do not hesitate to contact me at Asa@MacWilliams.de.

Overview

Contents

List of Figures

Introduction

Ubiquitous augmented reality is the convergence of augmented reality and ubiquitous computing. This dissertation address the challenges of developing software for this emerging field.

Ubiquitous augmented reality is an emerging human-computer interaction technology, arising from the convergence of augmented reality and ubiquitous computing, that promises many novel applications. In the past years, the basic technology for ubiquitous augmented reality has become fairly mature, but important challenges remain. In my dissertation, I address the software engineering challenges.

I propose a new architectural style, the adaptive service dependency architecture, which, with its supporting middleware and the development technique it enables, is particularly suited to address these challenges.

This dissertation describes the domain of ubiquitous augmented reality and its challenges to software engineering, and shows how the adaptive service dependency architecture meets these challenges by combining existing architectural approaches and extending them. It presents architecture, middleware and development process in detail, using examples from prototype systems for real-world ubiquitous augmented reality applications. It then uses results from the implementation of these systems to evaluate the solution's usefulness in addressing the domain's challenges.

1.1 Ubiquitous Augmented Reality

Augmented reality and ubiquitous computing are two technologies that share the same vision: to augment the user's real environment by using advanced computing technology to provide an effective and natural interface to the virtual world of information. As the technologies mature, they are beginning to overlap. Their confluence is the domain for my research, and I call it *ubiquitous augmented reality*, or UAR.

Augmented reality (AR) is a rich interaction technology that provides users with spatially related information in the real world in real time [3]. An AR system tracks the position and orientation of objects, most notably the user's head, and superimposes virtual objects into the user's field of view (Figure 1.1 on the following page). AR systems often support multiple human-computer interaction techniques for input (multi-modality) and output (multi-media).

Ubiquitous computing aims to provide a "calm" interface to computers, making them invisible, yet omnipresent [141]. Users should be able to use computing technology in their

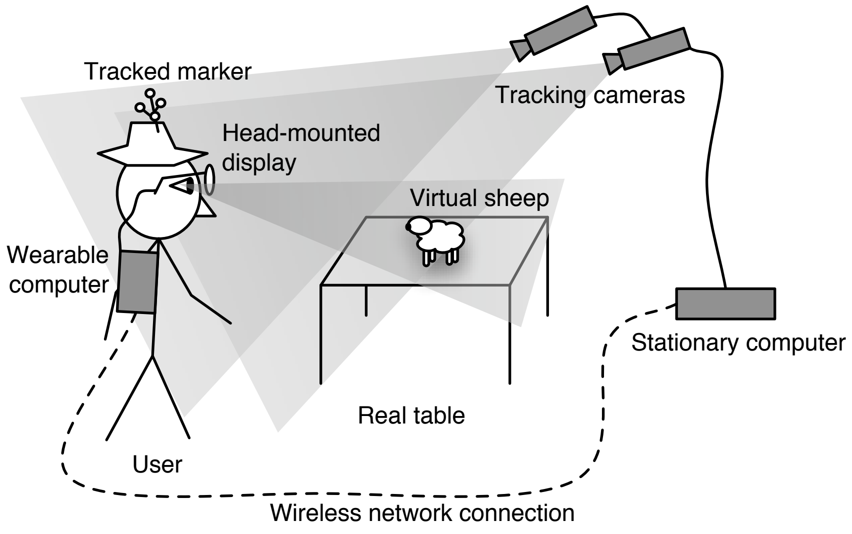


Figure 1.1: Schematic overview of a distributed augmented reality system.
The user's position is captured by a camera-based tracking system; a virtual three-dimensional object is shown in a head-mounted display. The system includes a wearable and a stationary computer.

environment without having to think about it as such. For this, devices must provide their services autonomously and shield the user from configuration tasks. Additionally, other users can bring along devices which themselves provide new services.

Ubiquitous computing is a broader field than augmented reality, and includes many different ways of augmenting the users' environment, with AR as one possible technology. As the computing and interaction hardware required for AR becomes smaller and more ubiquitously available, AR and ubiquitous computing can converge, creating systems that are ubiquitously available and allow interaction in the style of augmented reality. This allows new applications and improved interaction with existing applications.

In ubiquitous augmented reality applications, users are involved with tasks in the real world, so they cannot (or do not wish to) spend their time sitting in front of a desktop computer. This includes mobility: users move around in a fairly large, inhomogeneous area. At the same time, they wish to access or modify information associated with their current real-world situation, which is not otherwise immediately available.

A user interacts with a ubiquitous augmented reality system in different ways, sometimes immersively, and sometimes unobtrusively, especially when interacting with other people. This merges the immersive interaction of augmented reality with "calm" ubiquitous computing. An example is a large-scale hospital support system, in which doctors at times view large three-dimensional models superimposed on a patient's body using a head-mounted display, and at others, only glance at a medical record on a palmtop or a screen on the wall. Another

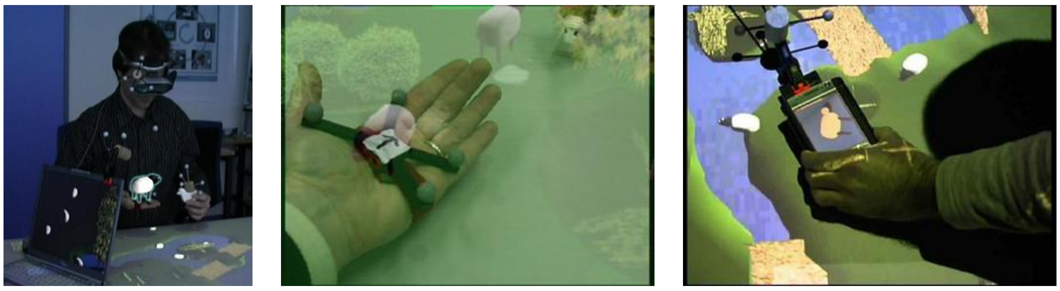


Figure 1.2: Augmented reality shepherding game.
Left, virtual sheep floating on hand is shown on head-mounted display and laptop. Center, view through head-mounted display. Right, virtual sheep is scooped off projector table onto palmtop. (Images from the SHEEP [80] project.)

example is a museum exhibit, in which different users use different interaction devices to explore the subject of an exhibition; or similarly, a multi-player game, where players cooperate using different devices (Figure 1.2).

Large-scale industrial applications are possible, such as prototype vehicle construction, production-line planning, and vehicle assembly in an automobile plant; here, a factory floor can be augmented with information regarding current or future production lines, and many users can interact with the system in complex ways (Figure 1.3 on the following page).

However, such large-scale, ubiquitous augmented reality systems have not been built and deployed in productive use yet.

1.2 Challenges in Software Development

Various difficult problems in building augmented reality systems, such as tracking the users' position and orientation, or rendering realistic three-dimensional scenes, have been the subject of research for some years now. It has become possible to build quite convincing augmented reality systems, as long as they are fairly static. Several research groups have even designed software toolkits or frameworks to build AR systems. Examples of such frameworks that have been used beyond the developers' own labs are *Studierstube* [119], a research framework for AR systems that supports synchronized views on several computers simultaneously; *AR Toolkit* [63], a library for detecting fiducial markers and rendering three-dimensional graphics over their video images; and *MR Platform* [136], a library for AR and MR (Mixed Reality) applications designed to work with custom head-mounted displays. The *ARVIKA* [37] project, a German research and industry partnership, promoted the spread of AR technology into industry, and developed its own software architecture.

Similarly, ubiquitous computing applications are becoming more numerous, but generally have a low degree of immersivity. Again, frameworks and toolkits for ubiquitous computing are

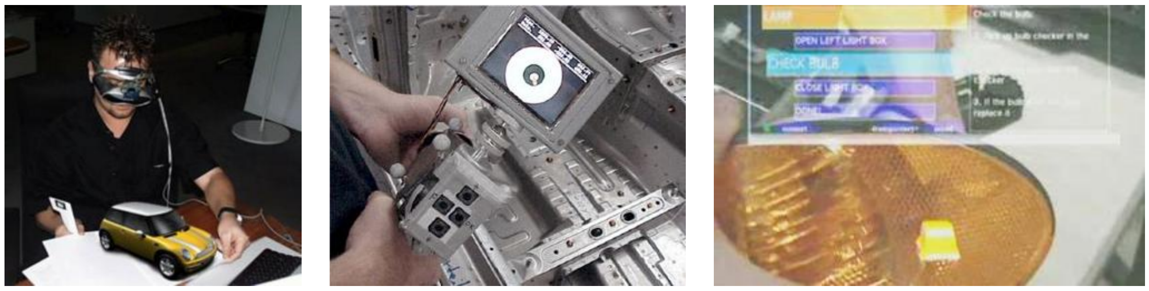


Figure 1.3: Ubiquitous augmented reality in automobile construction and maintenance. Left, designer regards virtual car model through head-mounted display. Center, small display on welding gun guides user to the correct 3D welding spot in the construction of a prototype vehicle. Right, mechanic's view through head-mounted display, showing step-by step repair instructions. (Images from the Fata Morgana [66], PAARTI [30] and TRAMP projects.)

becoming available. One example is the *Context Toolkit* [116], which supports the distributed evaluation of various forms of sensor data in order to build context-aware applications. Another is the *Sentient Computing* [91] project, which used a building-wide ultrasonic tracking system called the *Bat* to build location-dependent applications, even using rudimentary AR.

Research groups from both the area of augmented reality (e.g. the *Studierstube* project) and of ubiquitous computing (e.g. the *Sentient Computing* project) have begun to investigate the convergence towards ubiquitous augmented reality.

However, despite this progress, building a UAR system is, as yet, an unsolved problem. It poses both social challenges, such as user acceptance of wearable computing or privacy concerns, and technological challenges. This dissertation addresses the three main technological challenges inherent in developing the software.

First, a UAR system is distributed, combining stationary resources such as databases, cameras for position tracking and projection screens with mobile ones such as handhelds, wearable computers and head-mounted displays. The sets of devices that need to be combined change frequently as users move about. This creates a problem of *component uncertainty* that the software must cope with.

Second, during software development, it is hard to specify what the exact behavior of a ubiquitous augmented reality system should be. The users are involved in real-world tasks, and the computer system is of secondary interest. The technologies involved are new, and often difficult for users to imagine. Thus, the requirements are *ill-defined*.

Third and last, the system needs near-real-time *performance* to create the feeling of immersivity for the user. This makes developing the software even more difficult, as solutions to dynamically adapt a system to changing conditions and requirements, such as loose coupling and dynamic connections between components, can degrade performance.

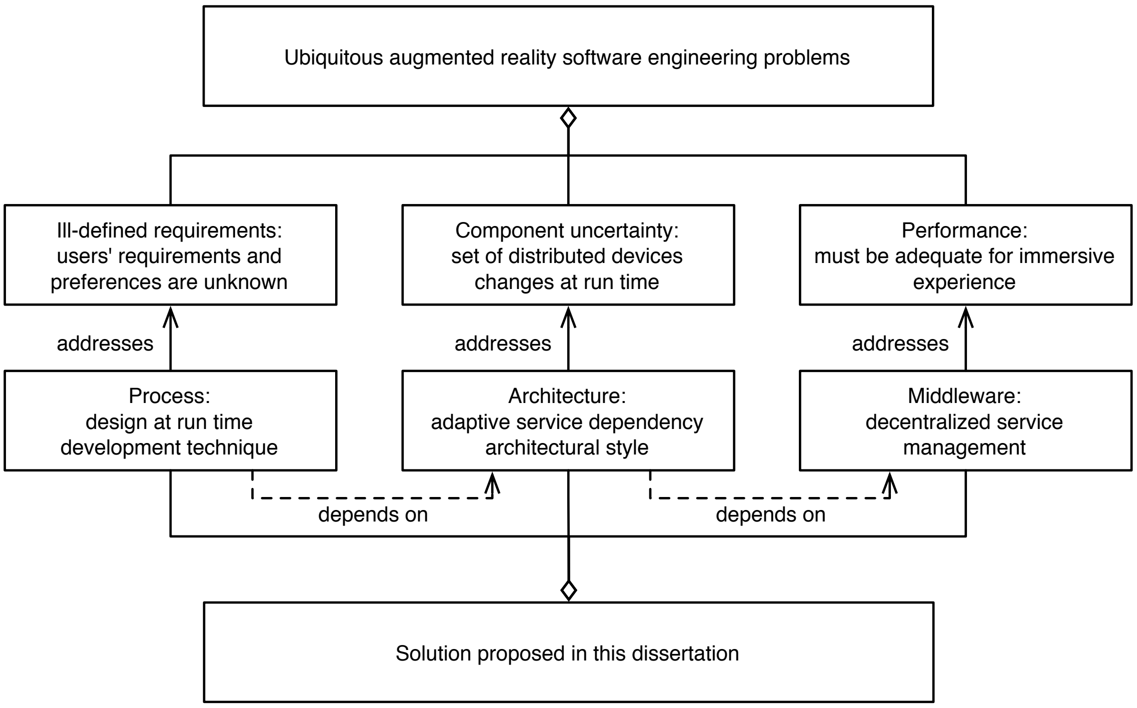

Ubiquitous augmented reality software engineering problems
Ill-defined requirements: users' requirements and preferences are unknown
Component uncertainty: set of distributed devices changes at run time
Performance: must be adequate for immersive experience
addresses
addresses
addresses
Process: design at run time development technique
Architecture: adaptive service dependency architectural style
Middleware: decentralized service management
depends on
depends on
Solution proposed in this dissertation

Figure 1.4: Architecture, middleware and process in proposed solution.
Each addresses a particular software engineering problem in ubiquitous augmented reality. The architecture requires supporting middleware, and the development process requires support from the architecture. (All diagrams are use UML notation.)

1.3 Proposed Solution

To address these three problems of component uncertainty, ill-defined requirements and performance, I propose to use a new style of *software architecture*, called the *adaptive service dependency architecture*; supporting *middleware* for decentralized service management; and a *development technique* for a dynamic development process (Figure 1.4).

The architecture (Chapter 5) consists of loosely coupled interdependent services that can adapt to changes, addressing the problem of uncertainty. For this, decentralized middleware (Chapter 6) is required, which dynamically changes the system's structure. The architecture enables a specific development technique (Chapter 7) called design at run time, which deals with ill-defined requirements by letting users and developers redesign the system's behavior while it is running. Finally, the middleware must maintain performance by decoupling communication protocols from system reconfiguration.

1.4 Research Hypothesis

My research hypothesis is as follows:

The adaptive service dependency architecture, consisting of loosely coupled distributed services, coordinated by decentralized middleware based on their interdependencies, facilitates the construction of ubiquitous augmented reality systems, by addressing the problems of component uncertainty, ill-defined requirements and near-real-time performance. This architecture is fast enough for augmented reality, even on small systems; and scales to large distributed ubiquitous computing applications. Its adaptability supports a cooperative, incremental development process at system run time. The architecture can be used to build systems for many different applications of ubiquitous augmented reality.

1.5 Approach

To support this claim, I combine two approaches: a constructive approach based on the problem domain and existing research results, and an explorative approach, showing the solution's feasibility by building several prototype systems.

Chapter 2 gives a first definition of the domain of ubiquitous augmented reality, based on previous definitions of augmented reality and of ubiquitous computing. It then presents several example applications and derives the main software engineering challenges. Chapter 3 shows related research projects and extracts common architectural approaches that can be used to address these challenges. Chapter 4 presents the solution, which combines several of these approaches and extends them, and shows several example prototype systems developed with this solution. Chapters 5, 6 and 7 present architecture, middleware and process in detail. Chapter 8 presents the results of building the software framework DWARF, and implementing several prototype systems, in the form of performance measurements and feedback from users and developers. Chapter 9 concludes the dissertation by critically examining how well the domain's challenges have been met, and showing limitations of this work and directions for future research.

1.6 Contribution

To the research communities of augmented reality and ubiquitous computing, this dissertation contributes an architectural style for future systems and applications; a design for decentralized service management middleware; and a technique for a dynamic development process. Within the DWARF research project, it has contributed a working run-time system and development tools for a software framework, allowing UAR researchers to build several systems and experiment with the concepts of ubiquitous augmented reality.

Beyond the domain of ubiquitous augmented reality, the results are applicable to any domain of software with similar characteristics: a rapidly changing environment, and evolving user requirements.

Developing Software for Ubiquitous Augmented Reality

Software for ubiquitous augmented reality must address component uncertainty, ill-defined requirements, and performance constraints, letting developers satisfy user interests.

This chapter defines the domain of ubiquitous augmented reality, based on existing definitions of augmented reality and ubiquitous computing. It discusses several example applications, and gives an overview of the required software components. It introduces the three major software development challenges of *component uncertainty*, *ill-defined requirements* and *performance constraints*, and shows the stakes held by users and developers in a ubiquitous augmented reality system.

2.1 A Definition of Ubiquitous Augmented Reality

The classic definition of augmented reality was given by Azuma in [3]: *augmented reality (AR)*

- combines real and virtual,
- is interactive in real time,
- and is registered in three dimensions.

As described by Weiser in [141], *ubiquitous computing*

- makes many computers available throughout the physical environment,
- makes them effectively invisible to the user,
- and preserves and augments the nuances of the real world.

As a combination of augmented reality and ubiquitous computing, I define ubiquitous augmented reality as follows: *ubiquitous augmented reality (UAR)*

- augments the real physical environment with virtual information,
- is interactive in real time,
- is spatially registered,
- is available throughout a large physical environment,
- and allows both immersive interaction and unobtrusive assistance.

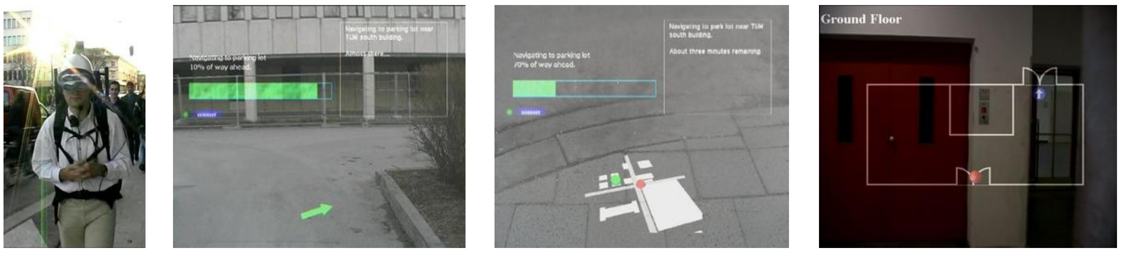

Navigating to parking lot
10% of way ahead.
Navigating to parking lot near TUM south building.
Almost there...
Navigating to parking lot
70% of way ahead.
Navigating to park lot near TUM south building.
About three minutes remaining
Ground Floor

Figure 2.1: Mobile augmented reality: indoor and outdoor pedestrian navigation. Left, “wearable” prototype system; center, outdoor view through head-mounted display when user is looking straight and when looking down; right, view during indoor navigation. (Images from the TRAMP and Pathfinder [5] projects.)

2.2 Example Applications

While there are many applications that can be implemented with augmented reality alone, or with ubiquitous computing alone, there are several that can benefit from both. These are the applications for ubiquitous augmented reality.

The following examples provide a representative overview of such applications. Each is presented as an informal description, followed by a check against the definition of UAR and particular challenges in implementing it.

Navigation

A user wishes to reach a specific location on foot. Navigational information is presented on several different devices, depending on the user’s preference. This can be head-mounted display (Figure 2.1), a palmtop computer, or a flat panel screen on the wall in a hallway. The display can include 2D and 3D maps, arrows [56], path indicators, images of way-points [44], and the position of other users [91].

Navigation fulfills the definition of UAR: it augments the physical environment (the area in which the user is navigating) with navigational information; it is interactive in real time, e.g. when a map in the head-mounted display adjusts to the user’s orientation; it is spatially registered, showing the user where he is; it covers a large geographic range; and the user can at times only glance at a direction indicator, and at others, consult a map in detail.

This application is challenging to implement if the navigation task has to support different environments, e.g. both indoors and outdoors. For different environments, different sensors are needed to determine the user’s current position and orientation. Similarly, different types of user interfaces are appropriate in different environments.

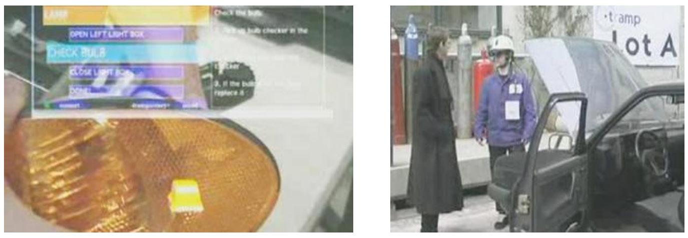


Figure 2.2: Augmented reality maintenance application. Left, mechanic's view though head-mounted display, showing step-by-step instructions. Right, mechanic with mock-up wearable computer. (Images from the TRAMP project.)

Maintenance

Maintenance of complex technical devices such as cars or aircraft is a promising field for ubiquitous augmented reality [38]. A mobile system assists the mechanic in diagnosing malfunctions and presenting appropriate step-by-step repair instructions (Figure 2.2).

To satisfy the definition of UAR, a maintenance application must be available throughout a large physical environment. Examples are deployment on a factory floor, with many machines to be repaired; a power plant; or maintenance of broken-down cars on the spot. The other points of the UAR definition are met even for stationary maintenance applications.

A challenge of this application is that the system must provide different forms of interaction to the user, in order to keep the user's hands free during the maintenance task itself, but also allow scrolling through a manual or data entry. Another challenge is the integration of the mobile system with diagnostic equipment and sensors within the machine itself.

Museum Guide

In a museum guide application, e.g. as described in [59], a visitor freely roams the rooms, looking at a number of exhibits. A UAR museum guide consists of small, wireless computing equipment with suitable multi-media display facilities, such as headphones, a wrist-worn PDA or a head-mounted display. The system presents information on the current exhibit, depending on the user's interests, location, and previously viewed exhibits. The information is presented using different interaction modes, such as audio, video, 2D or 3D graphics, or text. The visitor can request further details interactively, e.g. via a microphone, a dial, or by looking at a particular part of the exhibit for a while.

Like navigation, this application meets all UAR definition points. It is challenging from an interaction point of view, since the user interface in a museum must meet certain aesthetic requirements. An advantage is that it can be installed within the controlled environment of a museum exhibit. On the other hand, the application becomes quite complex if users are allowed to bring along their own computing devices, e.g. palmtops, and use them to interact with the exhibit as well.

Intelligent Campus

With a UAR intelligent campus system, students, faculty members and visitors use palmtop or laptop computers and public information displays to locate friends or colleagues, to find lecture halls, or look up schedules. The idea of an intelligent campus is a classic ubiquitous computing application [40, 45], but can benefit from AR, as well [91], letting users “look through walls” to see where colleagues are. Users collaborate at times, and take advantage of specially instrumented rooms for particularly immersive work.

The campus application meets all UAR definition points, and is quite challenging to implement. It requires users to be able to integrate their own devices into the system, it must use location sensors that work both indoors and outdoors, and different user interfaces must be provided for laptops and palmtops.

It also raises another challenge: ill-defined requirements and change over time. With many users of the system, its functionality will evolve significantly over time, as users think of new applications after the first versions of the system have been deployed.

Hospital

In a hospital, a ubiquitous augmented reality system could help doctors and nursing staff locate colleagues who are currently on other wards; keep track of the location of patients who are being sent from one treatment room to another; and visualize three-dimensional information, e.g. from CT scans. It could help patients keep track of their treatment schedule, and find their way around the hospital.

The hospital support application is similar to the intelligent campus, in that a major feature is displaying the location of people and helping them navigate. Additionally, however, it can take advantage of three-dimensional visualization, to aid doctors in diagnosis, and help explain medical conditions to patients and relatives.

In its challenges, this application is similar to the active campus; however, it naturally poses greater security and safety concerns.

Multi-Player Games

A commercially promising application is a multi-player game with several users operating in a shared environment. The game can range from a first-person shooter (two teams chase each other with virtual weapons [133]) to a collaborative simulated world (Figure 1.2 on page 3, [80]) or a distributed campus-wide or city-wide role-playing adventure [94]. The game system has to provide tracking of all users and coordinate their interaction.

A UAR game satisfies all UAR definition points, except perhaps that of unobtrusive assistance: some games are totally immersive, and distract the user from real-world tasks. The challenges in this application depend on the type of game: a first-person shooter requires fast, low-latency interaction in a distributed wireless network, whereas a distributed adventure game must support users with different sorts of interaction devices.

Team Action

Team action is an application where a heterogeneous team can share the capabilities of user-worn devices by offering their services to other users. Consider a fire brigade that has to fight a fire in an office building. Some firemen are outside the building observing the situation, others are inside and can provide more information from the front line. The team action system's task is to automatically collect data such as temperature and the presence of hazardous fumes with devices worn by the firefighters and provide this data to every participating person needing it. This is particularly useful to warn other team members of hazardous areas [60].

The team action application meets all UAR definition points, and is challenging from an interaction point of view. Users are distracted and under pressure; the system should be easy to use. Also, it poses a reliability challenge: Even if part of the system fails, (e.g. due to part of a building collapsing), the rest of the system should continue to work as best as possible. Furthermore, the system must support some level of self-organization, so that it can quickly be deployed in a new environment.

Factory Floor

The construction of complex machines was the original motivation for augmented reality: correctly placing wire bundles in an aircraft [25]. This remains a promising field in industry [38], especially in the construction of prototypes, where programming a robot would be prohibitively expensive, but unaided manual labor is too slow (Figure 1.3 on page 4). Ubiquitous AR has the potential to extend the range of construction activities, e.g. support an entire factory floor rather than the production of a single car. This includes support for different stages of construction, and for maintaining the assembly line itself.

A large-scale factory floor application satisfies all UAR definition points, and offers many challenges. Particularly, it requires the complex combination of many different location tracking sensors, and many different interaction techniques.

Collaborative Design

Virtual reality has been successfully employed in designing many products, such as automobiles [38], and augmented reality can be used in design as well. It is particularly suitable when exploring early stages of a design, or performing three-dimensional sketches. Ubiquitous augmented reality can extend the design process even further, by supporting collaborative groups of designers, architects and customers with different kinds of user interfaces in in a multi-room or large studio environment (Figure 2.3 on the following page). In a studio specially instrumented for 3D architectural modeling, a user with a mobile laptop can walk in and join the the modeling task.

Particular challenges in this area involve the design of the user interface, as is must satisfy high esthetic demands. Also, different input and output devices must be available to explore real and virtual models.

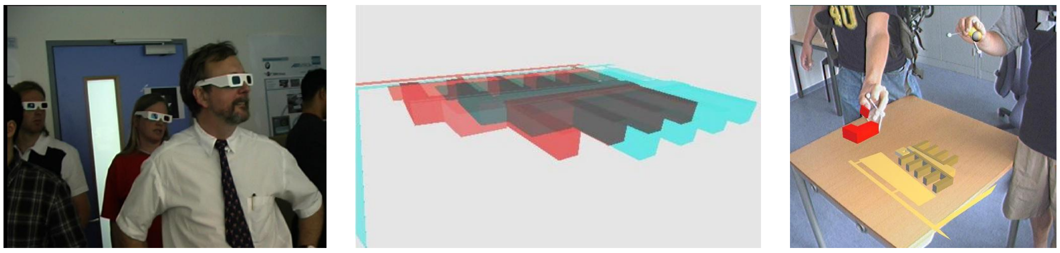


Figure 2.3: Augmented reality collaborative design and modeling.
Left, spectators watch modeling process with 3D glasses. Center, spectators' view. Right, manipulation of virtual building, shown as it appears to an observer through a head-mounted display. (Images from the ARCHIE project.)

All points of the UAR definition are satisfied for this application, if the area in which the design takes place is large (e.g. a building site). In that case, the deployment area becomes a challenge, as well.

Exploration

Exploring an unknown terrain and annotating it with information is another application for ubiquitous augmented reality. An example is archaeology. Archaeologists exploring a site can mark interesting areas with virtual notes, thereby creating maps for themselves and for colleagues. Similarly, an object which is removed from the site can be recorded in three dimensions first, so that subsequent investigators can see it in its original location. This application satisfies the UAR definition and is similar to collaborative design in a wide area; indeed, archaeologists can reconstruct historical buildings on the original sites, allowing tourists to see them there virtually [58].

The challenges in the exploration application involve instrumenting a wide area with different types of location sensors and combining different interaction devices, allowing hands-free operation.

2.3 Required Software Components

A ubiquitous augmented reality system must communicate with the user; observe the real physical environment, and be connected to the virtual world (Figure 2.4 on the next page).

In order to build a ubiquitous augmented reality system, many different hardware and software components are required. In this section, I briefly describe the basic software components of a UAR system, and which hardware they use.

A survey of several augmented reality systems [15] revealed that in spite of being quite different in detail, most existing AR systems share a common basic architectural structure. In addition, many basic components and subsystems can be found in many different systems,

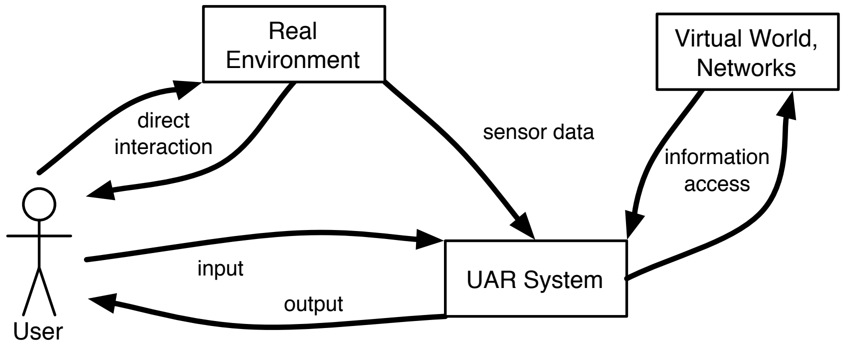


Figure 2.4: Ubiquitous augmented reality system links user with virtual world of information, in the real physical environment.
The arrows indicate data flow. The user interacts with the real world directly, and the UAR system allows him to interact with the virtual world. In order to augment the real world with virtual information, the system must observe the real world. (Figure adapted from [65].)

e.g. various trackers or a scene graph. Based upon this notion, a reference architecture for augmented reality systems was developed and presented in [109]. This reference model can be used to compare particular systems, or to construct new ones. The reference model is at the solution domain level of abstraction, meaning that it addresses neither lower-level issues such as the operating system and middleware nor higher-level issues such as support for a particular application domain.

In the following, the subsystems and components of this model (Figure 2.5 on the following page) are presented, showing how it extends from augmented reality to ubiquitous augmented reality. This dissertation does not include models or details on the functionality of any of these components. Rather, the components are used as examples to illustrate the adaptive service dependency architecture. Thus, the system design of a UAR system leads to requirements for the underlying architectural style and middleware.

Tracking

Tracking is a key element of UAR systems. It is used both in AR and in ubiquitous computing, although in different ways.

In order to show spatially registered AR information in a head-mounted or tablet display, the *pose* of the user's head, of the display devices and of real-world objects that are to be augmented must be tracked in real time. In most of today's AR systems, the pose of an object is represented in six degrees of freedom: three for the position in three dimensions, and three for rotation around each of the three axes. In AR, the pose information must be updated as frequently as the image is refreshed (24 Hz is considered a minimum acceptable rate, as used

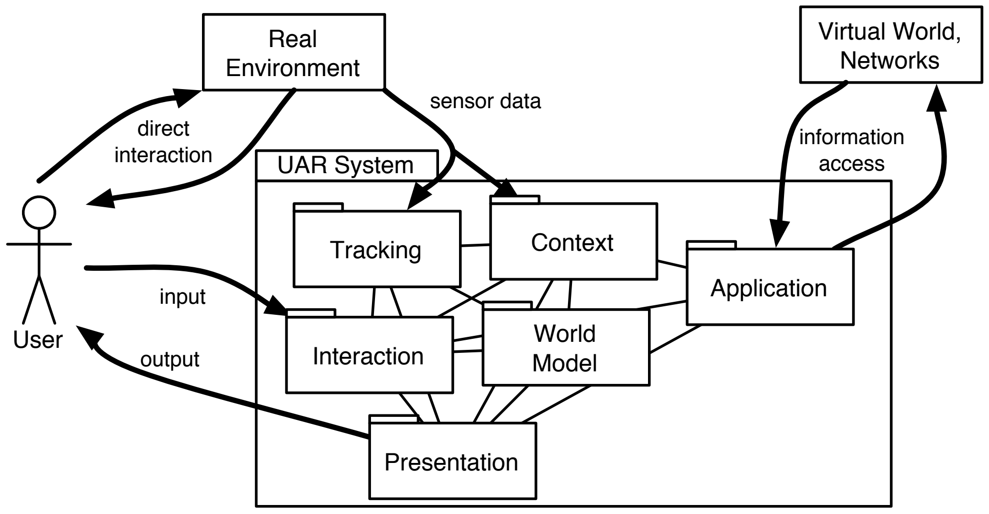


Figure 2.5: Required software components in a UAR system.
The arrows in indicate data flow. The software subsystems of a UAR system are generally divided according to different types of data flow in and out of the system.

in movies). It must be made available to the display devices as quickly as possible. Also, the accuracy of tracking must be quite high; ideally, augmentations on the display device should be positioned to within one pixel of accuracy.

In ubiquitous computing, tracking is also used to determine the pose of (multiple) users and objects. The accuracy and update rate requirements, however, are significantly lower than in AR. Often, it is enough to know which room a user is in, or which display device a user is closest to. However, the range of tracking is generally greater than in AR; trackers must be installed in a large environment, e.g. an entire building.

On the hardware side, tracking can be achieved with several techniques. There are optical, video-based trackers, magnetic trackers, inertial trackers, electromagnetic trackers, GPS, and combinations thereof, i.e. hybrid trackers. Tracking hardware can be carried with the user, be mounted on display devices or objects to be tracked, and installed as fixed infrastructure in the environment. Most tracking systems require some sort of information about the objects to be tracked, as well as the environment. For example, robust optical trackers require fiducial markers on the objects to be tracked, and wireless LAN trackers must know the position of the base stations.

The software for most AR and ubiquitous computing systems includes a dedicated *tracking* subsystem [38, 50, 111]. It is responsible for processing the sensor input coming from the trackers themselves, and the calculation of pose. Often, the calculation of the pose runs in parallel to the other system tasks. The tracking subsystem must also combine measurements from multiple trackers, which often use different types of sensors.

The tracking subsystem accesses the world model subsystem to retrieve information about the physical environment. For example, an optical tracker needs information about the features it should look for in the images. The result of the tracking process, the pose, is made available to the other subsystems, notably presentation for spatial augmentation, but also to the world model to update it, and to interaction to allow gesture input.

Tracking is crucial in the navigation application. For maintenance, it is required for exact registration of repair instructions, and the accuracy must be high, although the range may be limited. In team action as well as in games, several users must be tracked; although not as many as in the intelligent campus. In exploration, the environment that the tracking takes place in is not known and measured in advance; this can apply to team action as well.

Context

Beyond position and orientation, other contextual information is required in UAR applications. Context has been defined as everything that influences the interaction between user and system; and more narrowly as time, location, intent and user identity [27].

In AR systems, context such as the experience level of the user can be used to show different types of visualizations; the distraction level the user will tolerate can be used here as well.

In ubiquitous computing systems, context is similarly used to adjust the type of user interaction to the current situation. In the intelligent environments of *context-aware computing*, context is used to trigger actions, such as adjusting the lighting level of a room.

Context is measured using a variety of hardware sensors, such as accelerometers, RFID tags, light sensors, temperature sensors or microphones, and also those used for tracking. This information is generally used to infer information about the user's situation, e.g. whether the user is in a meeting or working at his desk.

Many AR and ubiquitous computing software architectures include a dedicated *context* subsystem [38, 27, 60]. The context subsystem processes the raw sensor data, aggregates it, makes inferences, and makes the higher-level information available to other subsystems.

Note that there is an overlap between the context and the tracking subsystems. As seen in Figure 2.5 on the preceding page, both subsystems process incoming sensor data and make them available to other subsystems. The user's pose can be regarded as a special case of context information, and indeed, some architectures, especially in ubiquitous computing, process pose within the context subsystem and have no special tracking subsystem. However, pose plays a unique role in both AR and ubiquitous computing systems, and must be delivered to the presentation system much more quickly than other context information. Therefore, the software architectures for many systems, both in AR [38] and in ubiquitous computing [50] have separated the two subsystems.

An example for the use of context is in the maintenance application, where the level of assistance (step-by-step instructions or a simple overview) should be adjusted to the mechanic's experience level. In team action, contextual information (such as temperature in a firefighting application) must be rapidly distributed between mobile users. An intelligent campus or hospital system can take advantage of contextual information to determine whether a user can currently be interrupted, to receive a telephone message.

World Model

A ubiquitous augmented reality system augments the real world with information from the "virtual world". For this, the system needs a spatial model of the physical environment around the user. This then can be enriched with spatially registered information to form a model of the virtual world.

In AR, the required model of the world consists essentially of three-dimensional geometry. This is then used for rendering three-dimensional scenes, but also to determine the position of markers, e.g. for optical tracking.

In ubiquitous computing, the model of the world is also geometric, but often limited to two dimensions, such as a floor plan of an office building.

Many current AR and ubiquitous computing systems include a *world model* subsystem [112]. This is not directly linked to sensors or I/O devices, but receives its information mainly from the application and tracking subsystems. A common approach is to use a repository architecture, letting other subsystems access information in the world model and subscribe to changes. Note, however, that the presentation and tracking subsystems generally communicate directly.

A world model is clearly required for navigation; here, it corresponds to an electronic map. In maintenance, the relevant model is a three-dimensional model of the machine to be maintained, including whole-part relationships, and perhaps dependencies between parts. In a museum guide, the world model includes a mixture of real and virtual: the real rooms and exhibits, and the virtual augmentations to be displayed in the exhibits. Similarly, the intelligent campus needs a world model with information on the location of other users, and temporal information such as lecture schedules or current events.

Presentation

In order to augment the real world, a ubiquitous augmented reality system must present information to the user, either optically or using other media.

In AR, the primary means of presentation are three-dimensional graphics. These can include models of real objects that are not present at the moment (e.g. missing parts of a machine) or not visible to the user (e.g. CT scans in medicine). They can also be virtual arrows indicating the direction of motion a user should take, or spatially registered text labels [10]. Additionally, several systems use sound for output.

Ubiquitous computing systems often use simple status indicators, sometimes embedded in unobtrusive ambient displays, e.g. to indicate that the user has received mail. Other means of presentation are two-dimensional graphics, such as area maps, and text messages.

Aside from common display hardware (desktop monitors, laptop and palmtop screens), UAR systems also use head-mounted displays and projection screens [80]. For audio output, headphones, loudspeakers, spatial audio systems and small speakers in mobile devices are used. Other types of media, e.g. haptic feedback, require specialized hardware.

Many current AR and ubiquitous computing software architectures include a dedicated *presentation* subsystem or layer. This is responsible for driving the output devices and presenting appropriate information from the application and world model. The presentation is influenced by information from the context subsystem, e.g. by selecting an appropriate level of detail for the user; and the spatial registration is achieved by receiving the correct pose from tracking. Often, different software components are used for the different types of media (e.g. a 3D rendering component and a sound player component); but sometimes, these are integrated into one common component [119]. An important challenge is to combine multiple output devices, e.g. multiple displays in a room, into a coherent user experience.

In the navigation application, 2D and 3D maps and arrows, as well as labels and images of landmarks can be presented on head-mounted displays, PDAs or screens in the environment. A maintenance application typically displays animated arrows or machine parts indicating how the user should move a part or tool. A museum application requires artistically pleasing 3D graphics, and could benefit from the use of spatial audio. The campus and hospital applications would use small portable displays, such as palmtops, which display information when the user wants it, and perhaps beep to get the user's attention. They would also use large projection screens in the environment, as well as stationary terminals for retrieving detailed information. In collaborative design, a particular challenge is to combine multiple displays and ensure all users have a consistent view of the object being designed.

Interaction

In AR, some interaction metaphors can be borrowed from VR. Thus, the user can interact with the virtual world using hand gestures, by moving a tracked pen or selecting menu items from a virtual tablet [119]. For hands-free input, some systems use voice recognition.

Ubiquitous computing systems use more indirect forms of interaction, since the computer is supposed to become invisible; the user's mere presence in a room causes the system to react. Many ubiquitous computing systems do allow conscious and direct forms of interaction, such as pressing simple 2D buttons on a touch screen or moving tangible objects on a table.

Finding appropriate UAR interaction metaphors that work in different situations (immersive, as in AR, and unobtrusive, as in ubiquitous computing) is still a challenging research topic [117].

The most common hardware for interaction in AR systems is a tracked physical object, e.g. a pen or a tablet; often, such an object has several buttons attached to it, which communicate wirelessly with the rest of the system. Space mice allow similar interaction, mainly based on orientation. Other useful input devices are touch screens and microphones.

An *interaction* subsystem is an integral part of many current AR and ubiquitous computing systems; it processes input from input devices, interprets and combines it. For input using gestures, the interaction subsystem uses data from tracking. Input may be interpreted in different ways, depending on the context from the context subsystem. The interaction subsystem uses the presentation subsystem to give the user feedback when input has been recognized, and when it hasn't. As in the presentation subsystem, a key challenge is combining multi-modal inputs from different devices, and perhaps by different users.

There is an overlap between tracking, context and interaction subsystems, as each processes input from sensors. However, tracking plays a special role in UAR systems and is separated from the others; and generally, sensors for interaction and context can be divided by whether the user is consciously giving a command to the system or whether the system is observing the user in the background.

The interaction and presentation subsystems are linked quite closely when the same physical device is used for input and output, e.g. a touch screen.

The interaction in applications such as navigation is quite minimal; the user should be able to enter a target destination before actually navigating; afterwards, he only needs to indicate blocked roads, e.g. using a touch screen. In games, the interaction metaphors are quite sophisticated, sometimes borrowing from magic, such as using a magic wand and a speech command [80]. In team action, the interaction must be very simple to use, even in a stressful situation [60]. In maintenance and in a factory, the user must be able to interact with the system while keeping his hands free, e.g. using speech.

Application

A UAR system is linked to the virtual world of information and computer networks in order to retrieve the information to be presented to the user. This is highly dependent on the application itself, and thus can be placed in a separate *application* subsystem.

Many experimental AR and ubiquitous computing systems are built for only one application, and do not include a dedicated application subsystem. However, other systems that use a common framework or infrastructure for different applications do include specific place holders in which application developers can include application logic [119] and links to external software [38].

Examples for the application subsystem include access to parts databases, CAD diagrams and workflow instructions in the maintenance and factory applications.

2.4 Software Development Challenges

In developing ubiquitous augmented reality systems, several problems must be considered, which pose challenges to software development. These are the challenges that are addressed in this dissertation. For each problem, the *forces* from which it arises are described here.

Note that there are many other problems that are not addressed by this dissertation. In particular, each of the subsystems described in Section 2.3 poses unique challenges, e.g. in developing appropriate abstractions for user input or modeling different types of tracking sensors. These problems, however, are fairly independent of the software architecture of a UAR system as a whole; and thus are out of the scope of this dissertation.

2.4.1 Component Uncertainty

The first problem is the *component uncertainty* that arises, at system run time, when distributed components with changing availability must cooperate in a changing context without knowing each other fully.

This problem arises from several forces which are immanent to the domain of ubiquitous augmented reality.

Interdependent distributed devices A ubiquitous augmented reality system is inherently *distributed*, combining stationary resources such as databases, cameras for position tracking and projection screens with mobile resources such as handhelds, wearable computers and head-mounted displays. Thus, a UAR system must combine distributed hardware and software components.

The components are *interdependent*: for their correct functionality, components depend on other components, which in turn depend on others and so on. For example, a three-dimensional rendering component in the presentation subsystem depends on up-to-date pose data from one or more trackers, which depend on descriptions of fiducial markers from the world model.

This means that a component does not only depend on other components directly, but, transitively, depends on many other components which it may not be aware of.

Changing availability of devices In ubiquitous computing, and hence in UAR, the set of devices that are available to form the system changes frequently as users move about. Some devices may simply be out of a mobile computers' wireless network range, or the network quality may be insufficient for the desired type of communication.

Even if we assumed unlimited wireless network coverage, the available network quality is not sufficient for all types of inter-component communication all the time. Furthermore, many devices such as trackers only have a limited physical range of operation; a stationary optical tracker can only operate within one room, since it cannot see through walls.

This means that the architecture of a UAR system cannot be designed to rely on a fixed set of hardware devices; it must *adapt* itself to the available devices.

In general, the system cannot rely on any fixed infrastructure. For example, when two mobile users meet outside, no fixed infrastructure is available, and their computers must form an ad hoc network. An extreme case of this is the team action application, where a mobile team must be able to cooperate in an unknown environment, and must bring all their required infrastructure with them.

Changing context influences choice of components The set of hardware devices used in an UAR system, and hence the set of software components, depends on the users' context.

For example, in the maintenance application, an experienced mechanic may simply wish to glance at the screen of a palmtop fixed to the machine he is working on, whereas an inexperienced mechanic may require detailed instructions in a head-mounted display. Thus, depending on context, the same information should be presented on different output devices.

The user's context not only influences the choice of components, but also their communication structure and configuration. For example, when a user enters a room with a stationary optical tracking system, the tracker should be reconfigured to track that user as well. Or, in the intelligent campus, when a user leaves his office, an application on his desktop system could "follow" him on his palmtop.

This means that even once the set of available hardware devices is known, the structure of the communicating software components must still *adapt* itself to the user's context, either automatically or upon explicit user commands. This requires support from the software architecture.

Incomplete knowledge of components   In a system that is deployed over a wide area, mobile users will bring new devices and software components into contact with other devices and components that are unknown to the components on their computers.

Thus, a user's mobile computer is confronted with previously unknown software components on other computers, which must be integrated at run time. For example, in the museum and campus applications, users should be able to bring along their own palmtops and use them with the rest of the system.

This means that each software component must be able to operate with incomplete knowledge of the other components. The software architecture must be designed so that each component knows just as much as necessary about the others, and can operate without global knowledge.

Towards a solution   To address these challenges, UAR systems need a *software architecture* that can adapt the set of communicating software components, and their communication structure, to both the availability of devices and to the user's context, in a decentralized fashion. Such a solution is proposed in Chapter 5.

2.4.2 Ill-defined Requirements

As a second problem, it is hard to specify in advance what the exact behavior of a ubiquitous augmented reality system should be. This goes beyond system run time; it applies to the entire development cycle of the system.

This problem arises from several forces. Many users are involved; they are performing real-world tasks, and the computer system is of secondary interest. The technologies involved are new, and users have little experience with them. Thus, the requirements are *ill-defined*, especially before the system is built and in place.

New and changing technology   The possibilities of ubiquitous augmented reality systems are just beginning to be explored. Many usability issues remain to be solved, especially in applications such as team action, where users are involved in real-world tasks and wish to be supported, not distracted. Promising advances are being made in the area of tracking, especially markerless optical tracking. Hardware is changing rapidly, e.g. in the area of head-mounted displays.

This means that at the current state of research, it is important to be able to build experimental systems quickly, and to improve on them as new results become available.

Ideally, a UAR prototyping environment would make an initial setup for a certain application simple to implement, in order to try out new hardware and new human-computer interaction techniques.

Many people involved Due to its wide range of deployment and the collaborative applications, many different people are involved in using, but also in building a ubiquitous augmented reality system.

Not only is the number of people large, but they come from many disciplines: user interface design, human-computer interaction, 3D graphics, sensor analysis, image processing, software development, and application domains such as medicine.

This makes communication between all involved parties about the requirements and behavior of an UAR system difficult before it is built and in place.

New applications In a decentralized system such as the campus application, many users are involved. Thus it is impossible to describe, a priori, all possible applications the users will find. Users will combine their devices and software components in new ways, creating new individual sub-applications.

As users find new applications, developers have to add functionality. Thus the system's functionality is never “finished.” If a system is in widespread use, extensions will have to be made while it is running.

During this cycle of identifying new applications and implementing them, entire new classes of applications for a ubiquitous augmented reality system may emerge. Thus, design decisions made when the system was first deployed may no longer be appropriate.

Towards a solution To address these problems, the software architecture must be extensible, allowing a system that has been deployed to be modified afterwards. Furthermore, users and developers should be supported by a *development process* that allows them to incrementally improve a deployed system. Chapter 7 describes a technique for such a process, which supports changing a running system.

2.4.3 Performance Constraints

Augmented reality has a much higher degree of interactivity and immersivity than ubiquitous computing, and this poses strict performance requirements. Ubiquitous computing systems, however, have a higher degree of distribution than augmented reality systems. Thus, in ubiquitous augmented reality, the tight performance requirements must be met despite a higher degree of distribution and scalability.

This is the third problem. In a sense, it is not as fundamental as the other two; rather, it acts as a constraint on the possible system designs.

Immersivity Augmented reality poses stringent requirements on the data flow throughout the system to gain the desired immersion of the user in the augmented environment. To render a three-dimensional scene in a head-mounted display, accurate and timely tracking data on the user's head position and orientation are necessary.

Thus, tracking information must be delivered to presentation components in near real time to be useful. A three-dimensional scene in a head-mounted display must be correctly updated within approximately 30 ms of the user's head movement to prevent motion sickness.

Many communication types Besides pose information, many other data types flow through the system. For example, optical trackers use a stream of video images from a camera in order to determine the position of objects in the camera's view and the camera itself. If the camera is on a different computer than the one that is performing the image processing, the video image must be sent across the network.

Thus, the system must let components communicate using a variety of different multimedia protocols and formats.

Scalability In the hospital and campus applications, potentially hundreds, if not thousands of users would be using a single ubiquitous augmented reality system. Of course, they tend to collaborate in groups and generally do not all interact with each other simultaneously. Still, the system must be designed so that it scales to a large number of simultaneous users, each of whom is using several computing devices.

Towards a solution To support communication in a distributed UAR system, a common software *infrastructure* or *middleware* is required. It must support communication using different protocols, and scale to a large number of computers. At the same time, it must be flexible enough to support an adaptive and extensible architecture; a difficult tradeoff against performance. Chapter 6 describes such middleware.

2.5 Stakeholders

In the model of a ubiquitous augmented reality system of Figure 2.4 on page 13, the only type of person involved is a *user*. However, many other people are involved in the development of the system, and are subsumed into the *developer* shown in Figure 2.6 on the next page. The following list of stakeholders describes different types of developers separately.

User The user of a UAR system is not primarily concerned with the system itself, but focused on a real-world task. This is an important fact to remember when designing a UAR system.

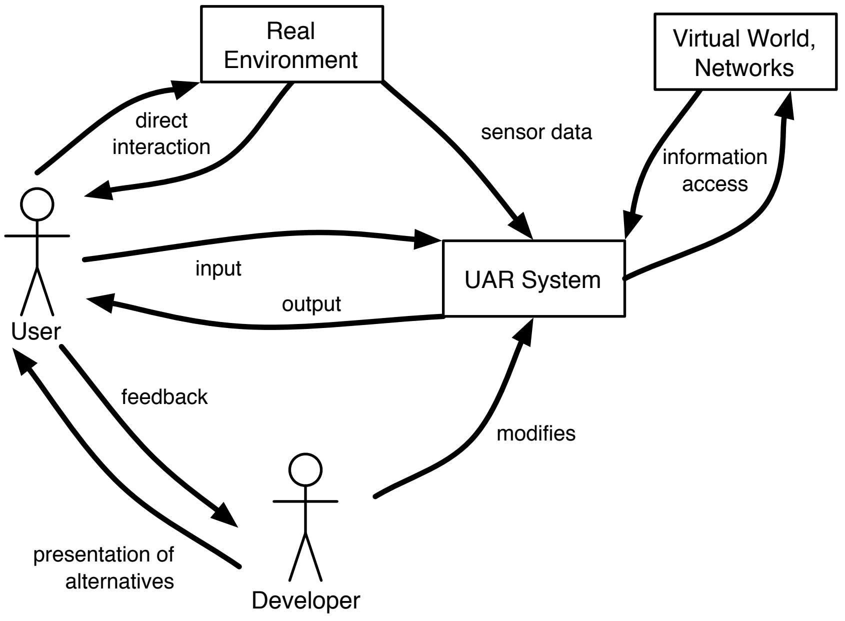


Figure 2.6: Ubiquitous augmented reality system links user, developer, real physical environment and virtual world of information.
Compare to Figure 2.4. Besides users, developers are involved in a ubiquitous augmented reality system.

Aside from that point, the user is concerned that the system be easy to use and to understand, powerful, fast, reliable, and easily adaptable to new circumstances.

Application Developer An application developer uses existing components, arranges them appropriately, configures them and adds application-specific functionality. Application developers are concerned that the components of the system be reusable and easily adaptable to the application's needs; and benefit from development tools and a prototyping environment.

User Interface Designer A user interface designer works with the application developer to design an appropriate UAR interface for a given application. Like the application developer, he benefits from a prototyping environment. However, in contrast to the application developer, he may have little programming experience, and thus the software components must be easily configurable using appropriate authoring tools. At the same time, in order to create an esthetically pleasing application, the designer must have a fine level of control over the system's presentation and interaction subsystems.

Component Developer The component developer wishes to focus on the task at hand of solving a particularly difficult subproblem of ubiquitous augmented reality. Thus, although he requires other components to test his own component, he does not want to be bothered with details of the other components, nor with those of communication. Thus, a component developer profits from high reusability and robustness of other components, and from an easily usable, fast and reliable middleware platform. In order to test is component with others, the component developer can benefit from development tools.

Furthermore, the component developer would like to avoid the burden of adaptability as much as possible—he cannot specify all possible configurations of the component in advance. Thus, he benefits from an architecture and supporting middleware that provide mechanisms for adaptivity.

Maintainer A maintainer, or tester, has to deal with an installed ubiquitous augmented reality system, and extend it or diagnose problems. Thus, he particularly needs monitoring, debugging and management tools, as well as versioning and feedback systems for the components.

Infrastructure Developer The infrastructure developer provides middleware and development tools to the other developers, and is interested in making these both as useful and as stable as possible. He is not concerned with the particular functionality of any of the components.

Conflicts and Tradeoffs Although all developers and users are working towards a common goal, conflicts can arise; for example, a component developer wishes to try a new and more powerful algorithm in his component, whereas the application developer is interested in the component being robust and well packaged.

Both the software architecture and the development process can influence these tradeoffs, and must consider them carefully.

Related Work

This work is influenced by research in distributed software architectures, adaptive middleware, context-aware computing, ubiquitous computing, and augmented reality; and builds on previous work of myself and of colleagues.

Research groups in the fields of augmented reality and ubiquitous computing have had to deal with the problem of how to develop software for their field, and several have put significant effort into developing reusable software architectures.

This chapter describes these research projects, and several from the domain of mobile multimedia, as these are also relevant to software architectures for ubiquitous augmented reality. The systems share several architectural approaches.

Several active areas of software engineering, especially middleware and component technology, can also help address the architectural challenges of ubiquitous augmented reality.

A particularly relevant area of previous work for this dissertation comes from the DWARF research project.

3.1 Augmented Reality Research Projects

The presented research projects are divided into three areas: augmented reality, ubiquitous computing and mobile multimedia. Note that the division into research areas is somewhat forced at times, since several projects address research questions from more than one area.

This section first presents those projects that address *augmented reality* as their main focus, as well as those from *mixed reality* and *virtual reality*. The projects are ordered alphabetically, and each is presented in the same brief format, consisting of general project information, and a description of the architecture.

Much of the material in this and the following two sections has been published in greater detail, in two survey reports: one on software architectures for augmented reality [15] and one on infrastructures for interactive distributed systems [34].

3.1.1 AR Toolkit

Group Human Interface Technology laboratory (HITlab), University of Washington
Key publications The working principles and results about tracking accuracy are described in [62]. Further publications are listed on the AR Toolkit home page.

Research goals To solve the recurring problem of position and orientation tracking in video-based and see-through AR to enable developers to write their own (potentially collaborative) AR applications; to provide a toolbox for easy prototyping of AR applications; to demonstrate possible uses of collaborative AR.
System description The AR Toolkit is basically a specialized computer vision library which analyzes the video stream of a camera, recognizes markers with their pose and ID, and provides marker position and orientation in a data structure suitable for OpenGL-based rendering.
Interesting aspects The AR Toolkit has a large user community, a robust recognition, and uses cheap printed markers.

3.1.2 ARVIKA – Augmented Reality for Development, Production and Servicing

Group The ARVIKA project was a joint academic/industrial project sponsored by the German federal ministry for education and research. It involved many partners from basic research, technology manufacturers and various application domains, e.g. automotive and machine maintenance.
Key publications [15, 38]
Research goals To bring together research and industry and promote the industrial acceptance of augmented reality, particularly in development, production and service of complex technical products; with one architecture for high-end applications in product development, and one for mobile applications in production and service environments.
System description 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, either on Windows 2000/NT, Linux, or SGI IRIX. On top of the operating system are the device integration interface *IDEAL*, the *AR browser*, and the tracking component. On top of these AR-specific components is the application-specific software. IDEAL allows network-wide access to various tracking and interaction devices. A common interface, the *low level device interface (LLDI)*, provides an abstract model of different devices.

The *mobile web-based solution* is based on documents; augmented reality content is one type of media among others (HTML, PDF, or CAD data). The client-side mobile platform provides the typical AR functionality with localization, tracking, graphics, and integration of interaction devices. The client uses Microsoft Internet Explorer 5 on Windows platforms as a thin client. The *AR Browser* is a local ActiveX component for tracking and 3D visualization. When AR scenes are referenced in web pages, the AR Browser is started and displays the registered scene.

The server side handles information management and enterprise integration. It is based on the Apache Tomcat application server. Each component is realized following the Java Beans specification. For autonomous mobile use, the server-side components can also be deployed on the client platform.
Interesting aspects Despite the use of different hardware and operating system platforms for the stationary and mobile systems, all software libraries concerning rendering, tracking and device interface can be reused from a single source tree. Thus, the *AR Browser* runs both

as a standalone high-end application and as an ActiveX component. The architecture for the ARVIKA mobile system is one of the only systems to consider the integration of data from enterprise systems.

3.1.3 DART – Designer's Augmented Reality Toolkit

Group AEL (Augmented Environments Laboratory), Georgia Institute of Technology
Key publications [39]
Research goals To enable designers to rapidly develop and test their AR experiences in the deployment environment, with a focus on supporting early stages of design.
System description The DART system is based on Macromedia Director. It uses familiar Director paradigms of a score, sprites and behaviors to allow a user to create complex AR applications. DART also provides low-level support for the management of trackers, sensors, and camera.
Interesting aspects DART is a multimedia programming environment built as a collection of extensions to the Macromedia Director.

3.1.4 EMMIE – Environment Management for Multiuser Information Environments

Group CGUI lab, Columbia University, New York
Key publications The whole system has been described in [19] and the underlying distributed graphics library was presented in detail in [76].
Research goals Development of a prototypical environment manager for a collaborative AR environment in analogy to the window manager of a desktop GUI.
System description EMMIE provided a straight translation of the WIMP GUI into the 3D space of a shared augmented reality environment. Users were equipped with a 3D pointing device with which they could manipulate 3D icons representing data files and 3D widgets, such as ring menus or so-called privacy lamps [18]. All objects were implemented as 3D objects in a shared 3D scene graph. The objects were replicated in each client to allow efficient rendering and local modification, and although one side had to create each object first, it was then replicated on all clients without any central server. The data files, however, were held in a file system shared among all clients in order to avoid excessive data duplication.
Interesting aspects EMMIE allowed interaction between the physical and the virtual worlds, such as drag and drop: a 3D data icon only seen in the user's head-worn display could be dropped onto a physical laptop and the corresponding document would appear on the laptop's screen. Conversely, data files could be dragged onto a special icon on the laptop screen, representing a wormhole into the virtual space, and would then appear as 3D icons next to the laptop.

3.1.5 ImageTclAR

Group Media Entertainment Technology Laboratory, Michigan State University
Key publications [102]

Research goals To support different types of developers in building AR systems: novice developers with little knowledge of AR; application developers who wish to reuse components; and advanced AR system developers.
System description The ImageTclAR system is based on ImageTcl, an image processing library for Tcl/Tk. It consists of several C++ components, and infrastructure to build glue logic in Tcl. The C++ components are configured and composed using Tcl. Application logic is written in Tcl as well. This hybrid approach allows novice developers to build applications quickly, and lets experienced developers build powerful components.
Interesting aspects ImageTclAR includes several development tools: an interactive component creation utility to easily define new data types or components; a build utility; and a graph editor to generate Tcl script code defining the data flow between C++ components.

3.1.6 Studierstube

Group Technical Universities of Vienna and Graz
Key publications [111, 119]
Research goals To explore three-dimensional interaction and new media in a general work environment, where a variety of tasks are carried out simultaneously; to find 3D interaction metaphors as powerful as the desktop metaphor for 2D.
System description Studierstube is based on *OpenInventor*, a powerful scene graph library. It takes advantage of the capabilities of the scene graph wherever possible. Studierstube runs as a *workspace* executable. Several users, each with his own workspace, can collaborate, using a shared scene graph. The scene graph synchronization is handled by a custom reliable multicast protocol. Workspaces find each other using a central *session manager*. For tracking and user input, Studierstube uses *OpenTracker*, an XML-configurable data flow framework that handles input from several tracking sources. For interaction, Studierstube provides the *personal interaction panel*, a handheld 2D virtual menu, as well as a 3D window manager, allowing users to switch between applications.
Interesting aspects Studierstube is one of the longest-standing AR frameworks. A very large number of applications have been built using it.

3.1.7 Tinmith

Group University of South Australia
Key publications [106]
Research goals To develop a stable infrastructure for mixed reality applications, with high performance even on resource-constrained mobile platforms as the major design goal.
System description The Tinmith software architecture is based on data flow from sensors, through the application logic, to rendering. Data flows via callbacks between lightweight C++ objects, which reside in a custom-implemented, in-memory hierarchical *object store*. Object classes are defined using C++ headers. Additional macros are parsed by a specialized compiler to generate code for serialization and callbacks. The objects are organized in a hierarchical

object store, which is modeled on the Unix file system. The object store provides serialization and deserialization features. Callbacks and listeners form a directed data flow graph which is independent of the object store hierarchy. Tinmith provides a complete library of components which can be joined together to write complex applications, e.g. outdoor AR modeling.
Interesting aspects The entire system is optimized for performance and runs stably on low-powered hardware. As a simple remote debugging tool, the object store can export all data as an NFS server.

3.1.8 VRIB – Virtual Reality Interaktionsbaukasten

Group TU Ilmenau and DaimlerChrysler R&D
Key publications [121]
Research goals To develop a toolkit *(Virtual Reality Interaktionsbaukasten)* for the rapid combination of hardware and software components in virtual and mixed reality, with both hardware (reusable sensor components, electronics, wiring boards etc.) and a software architecture, called *Vario*.
System description The Vario architecture is based on the ideas of modularization and data flow graphs. Hence, the basic reusable components form the nodes of a data flow graph, communicating over a network. The components are managed by a run-time infrastructure consisting of three main units. The *central system manager (CSM)* exists exactly once and is the central point of configuration and control. A run-time configuration user interface controls the CSM. On each host in the network, there is one *daemon*, which communicates with the CSM, and can start new processes in which components run. In each process, there is one *process manager*, which communicates with the daemon, and starts and configures components within a process. Communication resources are managed at the appropriate level (e.g. daemon sets up local inter-process communication).
Interesting aspects To aid development of new components, a builder tool generates stubs for components from XML descriptions. VRIB is an ambitious project, aiming to improve both hardware and software development for augmented reality.

3.2 Ubiquitous Computing Research Projects

The projects presented in this section deal with *ubiquitous computing*, *pervasive computing*, *intelligent environments* and *context-aware computing*. The projects are presented in the same format as the augmented reality projects of the preceding section.

3.2.1 ActiveCampus

Group Department of Computer Science and Engineering, University of California, San Diego
Key publications [45]
Research goals To simultaneously support extensibility and tight integration in a context-aware infrastructure for campus-wide ubiquitous computing using PDAs. On the one hand,

components must be tightly integrated to present a convincing user experience; on the other hand, they should be only loosely coupled, to allow extensibility.

System description ActiveCampus uses a central server for all components except data acquisition and user interaction, in order to ease administration and to minimize requirements placed on mobile devices. Additionally, centralization provides greater freedom to organize (server-side) components according to extensibility concerns. Sensors and display devices communicate with the server using SOAP. The server is a web server running PHP, backed by an SQL database.

The architecture contains five layers. At the top is the *device* layer, where mobile devices connect to the ActiveCampus server. Next is the *environment proxy* which abstracts raw sensor data. A *situation modeling* layer aggregates context from several sensors, whereas an *entity modeling* layer refines context over time. At the bottom is the *data* layer, which handles persistent storage.

Data regarding entities (users, buildings etc.) is stored in a *normal form* in the database. For example, each user has a unique numeric identifier. All other information (name, picture) are associated with that identifier. Similarly, all position information is represented in a normal Cartesian form.

Services are decoupled from another using *introspection*: each service has a method which returns whether the service may be invoked upon a certain entity. Thus, for example, a buddy service can easily be integrated with an e-mail service, so that users can send mail to buddies they were chatting with.

Interesting aspects The ActiveCampus system has been experimentally deployed in a large environment.

3.2.2 Aura – Distraction-free Ubiquitous Computing

Group Carnegie Mellon University, and industrial partners

Key publications [123, 124]

Research goals To provide the user with a digital “halo” of computing and information while trying to satisfy two competing goals: to maximize the use of available resources while minimize the distraction of the user.

System description The core part of the system is a *task manager* called *prism*, which tries to minimize the distraction of the user in the following four cases: the user moves to another environment; the environment changes; the task changes; and the context changes. The prism has the following components available and communicates with them: one or more context observer, with possibly varying degrees of sophistication, an environment manager as gateway to environment and file access, and several service suppliers. Furthermore, the prism can also communicate with other prisms to allow the seamless relocation of the user and its tasks to another environment.

Interesting aspects Aura uses a task-centered approach with platform-independent description and migration of the tasks.

3.2.3 BEACH – Basic Environment for Active Collaboration with Hypermedia

Group AMBIENTE group, Fraunhofer IPSI
Key publications [130, 131, 132]
Research goals For *AMBIENTE*, to investigate human-centered technologies for workplace interaction environments; for BEACH, to support synchronous collaboration in work environments, especially meetings.
System description The system has a grid architecture with four horizontal layers of abstraction (core layer, model layer, generic layer, task layer) and five vertical slices of basic concerns (interaction model, environment model, user-interface model, application model, data model). A third dimension is the degree of coupling between the components in the architecture. This approach offers flexibility needed for heterogenous devices and the inclusion of new or future devices. At the core of the architecture is a *shared object space* which supports the development of higher-level functionality.
Interesting aspects BEACH allows synchronous collaboration by building on shared states of objects and application.

3.2.4 Context Toolkit

Group Previously Georgia Institute of Technology, now UC Berkeley
Key publications The whole system is described and put into perspective in a comprehensive journal article [27].
Research goals To provide a formalism and framework for building context-aware applications.
System description The Context Toolkit was developed as the implementation of a formalism for describing context in its various levels of abstractions. It is a textbook example for this kind of formalization. Context can be data from single context widgets (basically sensor drivers), but also context from several widgets aggregated by context aggregators, and finally translated by context interpreters (such as lookup tables). The toolkit also provides a lookup and subscription mechanism for applications to use this context.
Interesting aspects Very strong formalization of the term context which imposes a clear structure and encourages a systematic approach to context awareness.

3.2.5 DynAMITE – Dynamic Multimodal IT Ensembles

Group Fraunhofer IGD, in cooperation with Loewe and the European Media Laboratory
Key publications [51, 52]
Research goals To support the user through dynamic ad-hoc device ensembles. A main goal is the development of a suite of algorithms and protocols as a potential new standard.
System description The system aims at enabling the spontaneous interaction of heterogenous devices and software components from different vendors in order to analyze the user's interaction, interpret his goals, and to realize them. The work is based on previous projects at Fraunhofer ITG, especially *Embassi* and *Soda-Pop*.

Interesting aspects DynAMITE uses ontologies to enable automatic component collaboration. It has a dataflow-based approach using memoryless channels and a publish/subscribe mechanism with treshholds.

3.2.6 FLUIDUM – FLexible User Interfaces for Distributed Ubiquitous Machinery

Group Fluidum project, previously Saarland University, now Munich University
Key publications [32, 33]
Research goals For *Fluidum*, to investigate new interaction techniques and interaction metaphors for instrumented environments, while attempting to formulate an interaction standard for those environments. For Fluid Manager, to investigate general principles of component communication in instrumented environments. The focus is on device access and device-application communication; the solutions found here however could be reused for similar problems, e.g. handling of distributed services in instrumented environments.
System description The Fluid Manager provides the software infrastructure for instrumented environments in different scales. Its primary focus is the management of dynamic addition and removal of devices and services. The system provides a matchmaking mechanism between applications and devices, and a uniform programming interface to the devices for the application programmer, which is independent of hardware, network or driver details.
Interesting aspects The Fluid Manager has a device classification using property lists instead of taxonomy. It uses a hybrid component communication approach (both centralized and decentralized).

3.2.7 Gaia – Active Spaces for Ubiquitous Computing

Group University of Illinois at Urbana-Champaign
Key publications [115]
Research goals To investigate Active Spaces, where data and applications are associated with the user.
System description Gaia supports mobile, user-centric active space applications. It is built as a *distributed object system*. The kernel consists of a component management core for component creation, destruction and upload, with currently seven services built on top of it (context service, context file system, component repository, event manager, presence service, space repository, and security service).
Interesting aspects Gaia uses a MPCC (model, presentation, controller, coordinator) pattern, which is an extension of the MVC (model, view, controller) [17] pattern. Gaia has its own scripting language LuaOrb [20], based on Lua [57].

3.2.8 Limbo

Group Distributed Multimedia Research Group, Lancaster University
Key publications [26]

Research goals To provide better support for adaptive mobile applications by using the tuple space paradigm [42] for communication.
System description The platform is based on the Linda tuple space model [12] and built on top of the MOST platform. It includes a number of significant extensions to Linda, mainly to address the specific requirements of mobile environments. Key extensions are: multiple tuple spaces, explicit tuple-type hierarchy with support for dynamic sub-typing, explicit QoS attributes in tuples, and a number of system agents that provide services for QoS monitoring, creation of new tuple spaces, and propagation of tuples between tuple spaces.
Interesting aspects Limbo is the first mobile application infrastructure using tuple spaces with some interesting extensions of the original tuple space concept.

3.2.9 Nexus

Group Nexus SFB, University of Stuttgart
Key publications A good overview is provided in [93].
Research goals To provide a structure and a software framework for large scale location dependent services.
System description The Nexus platform provides a structure for location based services (LBS) similar to the one of the world wide web. Nexus servers contain localized information. Nexus clients query information knowing their own location. Queries are modified in a federation layer and sub-queries are forwarded to the appropriate servers, and sub-results are combined again to form the overall result. *Area Service Registers* keep track of available localized information.
Interesting aspects By taking an approach similar to the structure of the web, the conceptual model will be intuitive for many developers, and scalability potentially corresponds to that of the web.

3.2.10 PIMA – Platform-Independent Model for Applications

Group IBM T. J. Watson Research Center
Key publications [4]
Research goals To develop design time, load time, and run time infrastructure support for pervasive computing, with devices as portals into an application/data space; applications as a means by which a user performs a task; and the computing environment as the user's information-enhanced physical surroundings.
System description In [4], the authors propose a system model for pervasive computing (which was not implemented at that time). This model is described by design time, load time and run time. At design time, applications are modeled as abstract tasks, which require certain services. The developer is encouraged to focus on a particular task. Interaction elements are specified abstractly, capturing user intent rather than a specific representation. Required services are described in an abstract service description language. At load time, devices are

specified by their capabilities, and the infrastructure matches this with applications' requirements. The system discovers and composes the system in order to perform the desired tasks. This involves dynamic discovery, pruning of hostable functions and adaptation of the presentation. At run time, the system handles redistribution, disconnection and failure recovery, when parts of the system change or fail.
Interesting aspects The model proposed for PIMA is very broad in its application; implementing these concepts will probably require much future research.

3.2.11 Metaglue

Group Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology
Key publications The first paper [22] and the initial thesis [105] about Metaglue.
Research goals To develop interactive environments in which computers communicate with humans on the human's terms, not on the computer's.
System description The MIT project *Oxygen* aims to provide mechanisms and technologies for the seamless integration of computing into our daily environments. It unites a large number of researchers (up to 250) with various interests, approaches and methodologies. It should be seen as a pool of ideas and concepts all contributing to the common goal of pervasiveness of computing, rather than a single software infrastructure which could be applied elsewhere. Nevertheless, there are system layers with a potential for reuse, such as the *pebbles* approach and the GOALS and Metaglue frameworks.
Interesting aspects There was a very large scale research effort integrating specialists from various fields. This should make for high quality research in the detail solutions.

3.2.12 Sentient Computing

Group AT&T Research UK, Cambridge University Lab for Communications Engineering
Key publications [50, 91]
Research goals To explore both the possibilities and the practical and social issues of ubiquitous computing and augmented reality, based on the Active Bat ultrasonic, wireless building-wide tracking system.
System description The Active Bat system uses small ultrasonic devices called *Bats* which are carried by users and attached to devices. It achieves a position accuracy of a few centimeters, but does not accurately measure orientation. The software architecture used within this project is called SPIRIT. The *Bat* system is controlled by a central server, which polls the Bats wirelessly, each in its own time slot. A scheduler can dynamically assign priorities to Bats that are moving frequently or which require a higher update rate, e.g. for a head-mounted display. The position of tracked objects is gathered on a central server. Object state, including position, is stored persistently in a relational database. Position information is directly forwarded to interested clients, without going through the database, so as to maintain interactivity. Clients then display information to the user, such as the location of colleagues. Mobile AR clients have been built,

including a Laptop with HMD (with additional inertial tracking) and a PDA (which acts as a remote X terminal for an AR application running on the server). However, most users do not use a mobile computer as a client, but use only the Bat itself; the Bat has a few buttons for input and can beep for output. Paper signs on the wall act as virtual buttons; by holding the Bat up to the sign and clicking the button on the Bat, the user activates a desired function.

Interesting aspects The Active Bat system was (and probably still is) the largest tracking system of its accuracy.

3.3 Distributed Multimedia Research Projects

These research projects deal with distributed multimedia, especially in changing environments and in mobile computing. They tend to put a greater emphasis on performance than the ubiquitous computing projects of the preceding section.

3.3.1 QoSDream – Quality of Service for Dynamically Reconfigurable and Adaptive Multimedia

Group Laboratory for Communications Engineering, Cambridge University

Key publications [89]

Research goals To develop a middleware framework for the construction and management of context-aware multimedia applications, with a uniform high-level data flow model; application-defined quality of service constraints; a sensor-independent spatial model of the world; event filtering and abstraction; and persistence.

System description QoSDream includes a *location service*, which processes sensor data and handles location information; an *event service*, based on the CORBA notification service, for passing events to applications and between applications; a *distributed object database* for storage of persistent and static information; and a *distributed multimedia service* for management of data flow between components. Recently, the open source *Framework for Location Aware Modeling (FLAME)* has been developed, which extends the location related aspects of QoSDream, but without the multimedia components.

The core of QoSDream is the location service. *Federators*, or *technology adapters* in FLAME, process incoming sensor data. Federators are available for different sensing technologies, such as the *Active Bat* system, or for active badges. The federators generate events which are sent to the *spatial relation manager*. The spatial relation manager organizes these events into *regions*, using information from technology-specific *location modules*. For example, a person wearing an active badge has an associated region depending on the accuracy of the sensing technology, and a “visibility region” which is the area in front of the person.

Event adapters detect overlaps between the regions. They can detect, for example, when the visibility region of a person overlaps with that of a computer screen. The event adapters send *overlap events* to the applications, which can react accordingly. Static location information, such as the location of rooms and walls, are stored in an object-oriented database.

Interesting aspects Although originally intended to deal with general multimedia applications, both QoSDream and FLAME now focus on the management of spatial information.

3.3.2 NMM – Network-integrated Multimedia Middleware

Group Computer Graphics Lab, Saarland University
Key publications [81, 82, 83]
Research goals To design and develop multimedia middleware for (mainly) Linux, which considers the network as an integral part and enables the intelligent use of devices distributed across a network. On the kernel side this means building an open, integrating architecture which can include heterogenous systems. On the service side it means investigating principles of session sharing and hand-over.
System description Within NMM, all hardware devices and software components are represented by so called *nodes*. A *node* has properties that include its input and output ports. The system distinguishes between six different types of nodes: *source*, *sink*, *filter*, *converter*, *multiplexer*, and *demultiplexer*. The NMM architecture uses a uniform message system for all communication. There are two types of messages: *multimedia data* and *events*. Object-oriented interfaces allow control of objects by simply invoking methods. External listener objects can register to be notified when certain events occur at a node. At run time, supported events and interfaces can be queried by the application.
Interesting aspects NMM has a meta-architecture which integrates heterogenous systems independent of underlying technology and thus enables new forms of middleware services.

3.3.3 UbiCom – Ubiquitous Communications

Group Delft University of Technology
Key publications [73, 104]
Research goals To specify and develop resource-constrained wearable systems for mobile multimedia communications, using a system approach with negotiated quality of service.
System description The architecture of all UbiCom systems is heavily influenced by quality of service and resource constraints. A QoS management mechanism called *Adaptive Resource Contracts (ARC)* is used to describe QoS requirements between components, evaluate tradeoff curves, and optimize according to application-defined criteria. ARC is decentralized, in that it assumes that no component has system-wide knowledge. UbiCom uses a client/server design, whereby a mobile AR client performs some, but not all, tracking and rendering locally. This allows an optimized tradeoff between latency, rendering quality and resource use. For example, the server performs polygon reduction of complex virtual objects, and generates bitmapped "imposters" of distant objects, which the mobile client can display quickly. As the user approaches a virtual object, the higher-quality representation is chosen. On the wearable system, the software components are distributed on a set of hardware modules. Each of them has an SA 1100 computing running Linux and function-specific code. The modules are *positioning*,

rendering and display, *video and application*, *wireless connection*, and *interconnect*. Each module uses function-specific hardware components, e.g. a GPS receiver, or a camera.

Interesting aspects UbiCom follows a system approach, building custom software on custom hardware. This gives the developers full control of the design space.

3.4 Common Approaches

The research projects of Section 3.1, 3.2 and 3.3 use several common architectural approaches, which address the problems posed in building software for UAR systems. This section presents some of them briefly and relates them to the adaptive service dependency architecture.

This section presents the *architectural approaches* informally, which is sufficient to give an overview. Many could be described as *patterns*, at greater length. Indeed, there are many more software patterns that can be identified in the domain of augmented reality alone [79, 110]. Similar work is underway to compile patterns for ubiquitous computing [74].

3.4.1 Distributed Components

Acknowledging the fact that both AR and ubiquitous computing systems consist of interdependent distributed devices, many researchers have used software architectures that consist of distributed components.

The granularity of the components, their homogeneity, and the rules of their interaction, are quite different in different systems. In some systems, the components are fairly symmetric in size and in their interaction. For example, in Studierstube, each mobile user has an *workspace* component that stores a version of a shared scene graph; then replicate changes to the scene graph between one another. In others, there is a central server component that coordinates all others, which may be small mobile devices. In most cases, there is a central component that coordinates the others. This simplifies development by providing a place to store overall application logic. ActiveCampus, ARVIKA and VRIB use such server-based architectures. In Aura, a server-based distributed file system handles data storage.

Some architectures use coordinating components, but do not limit them to a central server. For example, Gaia uses distributed components, called *services*, and follows the model/presentation/controller/coordinator pattern; the *coordinator* coordinates the other components. In Aura, each mobile user has several managing components. These include a *task manager*, a *context observer* and an *environment manager*. Together, these coordinate the other distributed application components.

The adaptive service dependency architecture (Chapter 5) uses distributed components as well, but is entirely decentralized, with no single coordinating component. This improves flexibility and fault tolerance, but has implications for manageability.

3.4.2 Loose Coupling

Using distributed components does not, by itself, address the problem of component uncertainty; the availability of devices changes, and changing context changes the optimal communication structure between the components.

If the distributed components of a system are tightly coupled, i.e. if they rely on each other explicitly, the structure of the system cannot change easily. To avoid this, many architectures use loose coupling: components rely only on the existence of appropriate other components, but make no assumptions about their identity. This allows components to have incomplete knowledge about each other: a component assumes that another component implements a certain *interface*, but knows nothing about how this interface is implemented. The idea of loose coupling is used in the concept of *polymorphism* in object-oriented programming, as described in [14] or [84].

Loose coupling not only allows the exchange of components at run time, but also makes the reuse of components in new applications easier, simplifying the development process. ImageTclAR is a good example of a system that uses this approach. This principle is used in service-oriented architecture, as well.

One method of accomplishing loose coupling between distributed devices is to coordinate them all via a central instance, e.g. a server, as described above. In ActiveCampus, the mobile devices do not know anything about each other; they communicate only with the central server. QoSDream and the Sentient Computing project use a similar approach. Naturally, this approach can form a bottleneck in communications, as all data must go through the server.

Another approach is to use a distributed, shared data repository. The components do not communicate with one another directly, but only with the shared repository. BEACH uses this approach with a shared object space. Aura uses a shared file system. EMMIE uses a shared scene graph, similarly to Studierstube. Context Fabric and Limbo use distributed *tuple spaces* for communication.

In a third approach, middleware connects components together, and they then communicate directly. This allows faster communication, as connections can be set up in advance. OpenTracker, the Context Toolkit, NMM and FLUIDUM are examples of this data flow architecture approach, as described below.

The adaptive service dependency architecture follows this third approach as well.

3.4.3 Middleware and Component Descriptions

If loose coupling between components is to be achieved as described above, the architecture needs supporting middleware or infrastructure that knows about the components and coordinates them. For this, several groups enhance their component models with *component descriptions* that are externalized from the component functionality itself.

For example, PIMA proposes an application model in which interaction elements are specified abstractly, capturing user intent rather than a specific representation. Devices are specified by their capabilities, and the infrastructure matches this with applications' requirements. In

Gaia, abstract application descriptions specify the components involved in an application. The Gaia infrastructure finds and connects these components. Similarly, Oxygen separates the description of the functionality from the of components that implement the features. Aura represents represents user intent as a coalition of abstract services, coordinated by middleware.

The use of component descriptions and middleware does not necessarily mean that the components have to be physically distributed. For example, in ActiveCampus, the central server hosts several *services*, which are application components. Using introspection, the middleware can determine whether two services are compatible or not. In Studierstube, reusable scene graph nodes are described in a scene graph language.

Component descriptions are particularly important if the infrastructure is supposed to optimize quality of service. In UbiCom and NMM, the components are described using quality-of-service parameters which the middleware can dynamically monitor, and adapt the components appropriately.

The adaptive service dependency architecture also uses component descriptions and an active distributed middleware to connect the components. It additionally addresses the problem of interdependent components.

3.4.4 Service Discovery

One function that such middleware can offer is service discovery. This addresses the changing availability of devices by querying the available networks for new components and devices that have become available in the environment. Such components that are found in the environment are often called *services*.

FLUIDUM and NMM use service discovery; the concept described in PIMA does, as well.

When using service discovery, components only have incomplete knowledge of other components; they know what they require of other components, but not which other components provide that functionality. This is a facet of loose coupling.

The adaptive service dependency architecture uses service discovery as an integral part of the distributed middleware.

3.4.5 Quality of Service

Loosely coupled components still depend on one another. One component thus depends on another component functioning correctly. Beyond that, however, it also depends on the other component's quality of service, such as the update rate or the accuracy of a sensor, or the delay of a network transmission.

Several systems model quality of service and use it, within the middleware, for component selection or for optimization. NMM monitors quality of service to optimize communication. UbiCom models quality of service in great detail and selects components accordingly. The adaptive service dependency architecture similarly uses quality of service as a criterion for component selection.

3.4.6 Connectors

When combining distributed components using middleware, several systems use the concept of *connectors*. Connectors improve loose coupling by separating the communication between two components from the components themselves. Connectors encapsulate the communication protocol, allowing the middleware to support many communication types. They transparently handle tasks such as multiplexing, binary format translation and network communication.

Different systems use this concept in different fashions. In Aura, components can only communicate via connectors. In NMM, there are specialized components in the data flow network for multiplexing or network communication.

In the adaptive service dependency architecture, connectors are used explicitly as part of the middleware, and support alternate communication types between components.

3.4.7 Data Flow Architectures

The performance of augmented reality and distributed multimedia systems is a significant issue in designing their architecture. For immersivity in an augmented reality system, data must flow, quickly, from tracking sensors through the system and finally to the displays. Several groups have chosen to design the systems around the *data flow* through the system.

Technically, the data flow may be implemented using callbacks, as in Tinmith or Studierstube's OpenTracker. Here, components of the system are designed as *sources* and *sinks* and connected as nodes in a graph. Other systems use existing middleware that allows network transparency. For example, QoSDream uses CORBA as a platform for data flow between components. A data flow network can also be designed to support quality-of-service optimization, such as in NMM.

Data flow architectures are often combined with loose coupling. The nodes of a data flow graph correspond to components which do not know each other directly; the application, or the run time infrastructure, sets up edges between these nodes. ImageTclAR uses this approach, as do OpenTracker, VRIB and Tinmith.

The adaptive service dependency architecture can similarly be used to set up data flow graphs, although this is only one possible application.

3.4.8 Event-Based Communication

Many systems, especially data flow architectures, use event-based communication, although with different types of protocols. Event-based communication contributes to loose coupling.

Studierstube uses a low-level multicast protocol for tracking data and for synchronizing scene graphs. Systems that do not include AR, such as the Context Toolkit, also use event-based communication, but with more flexible, high-level protocols, e.g. based on XML. This improves loose coupling between components.

Some systems combine event-based communication with a *publisher/subscriber* architecture, where interested components can subscribe to certain types of events that are produced by event sources. QoSDream and DynAMITE are examples of such systems.

The adaptive service dependency architecture can be used to implement a publish/subscribe mechanism. The current implementation within the DWARF framework uses CORBA middleware for communication, which allows the same flexibility as XML, but is fast enough for AR. In the *COVRA* project [64], Kim et al. showed that CORBA is viable for virtual reality applications, which pose the same stringent performance requirements as augmented reality.

3.4.9 Latency Layering

The performance requirements of augmented reality are not the same for all types of events that can occur during use of a system. For example, the image shown in a head-mounted display must be updated very quickly (within a few milliseconds) when the user turns his head. When the user enters a room, the information on the display should be updated to reflect that as well, but a latency of half a second is quite acceptable here. Finally, the position of another user in a campus application can be updated several times per minute.

Taking advantage of this fact, several systems use an approach that is called *latency layering* [104] in UbiCom. Some of the data flow through the system, usually tracking data for the user's head, is optimized to reduce latency. Thus, it flows directly from the tracking to the rendering component, without going via a central server or coordination component. Often, a specialized communication protocol is used for tracking data, such as Studierstube's OpenTracker or ARVIKA's IDEAL format.

Other data, such as a change to the contents of the rendered scene, take a more indirect route, through application components, or use a less specialized and more extensible communication protocol. In the Sentient Computing and QoSDream projects, a central server post-processes the streams of tracking data, aggregating it to higher-level information which can then be stored persistently, as it is often not required immediately.

The adaptive service dependency architecture can similarly take advantage of different communication protocols as described here, although it is not limited to tracking data. The implementation uses CORBA notification service events and other protocols, such as shared memory for local high-bandwidth communication, and CORBA method invocations for tightly coupled reliable communication.

3.4.10 Adaption by Component Selection

To address the problem of component uncertainty, especially as it arises from the forces of changing availability and changing context, several systems use descriptions of which components to select in which situation, and then reconfigure the system when that situation arises.

Aura, PIMA, Gaia and Oxygen model user activities as tasks. In each step of a task, the user needs certain interaction devices, which are specified abstractly. Middleware then selects appropriate available components.

The adaptive service dependency architecture also uses adaption by selecting resources, but also addresses the issue of configuring them. Rather than specifying certain situations in advance, several dimensions of context (user, room, application, role, etc.) are used to independently determine possible system configurations.

3.4.11 Context Refinement

In adaption by component selection, above, many systems take advantage of sensor input, e.g. from tracking systems, but also separate low-level and high-level context. This reduces the problem of component uncertainty somewhat, as the range of possible contexts that the system can adapt to is smaller than the range of all possible sensor readings.

The Context Toolkit uses *context widgets* to gather sensor data and provide it to the system in an abstracted form. *Context aggregators* and *context interpreters* operate on the low-level context information from the context widgets to provide higher-level, refined context information. The ActiveCampus system uses an *environment proxy* to abstract raw sensor data. A *situation modeling* layer aggregates context from several sensors, whereas an *entity modeling* layer refines context over time.

This approach is often combined with latency layering; low-level sensor data flows through the system quickly, and high-level context data flows more slowly.

Similarly, the adaptive service dependency architecture uses components that gather context data, and others that aggregate them. Position-related context data, from tracking systems, is also used for augmented reality user interfaces. The refined, high-level context data is used to control system adaption.

3.4.12 Prototyping

To address the problem of ill-defined requirements involved in building a UAR system, several research projects have designed tools for the construction of prototype UAR systems.

For example, DART is designed around an existing multimedia development tool, and ImageTclAR and VRIB provide tools for designing new components and composing existing ones. This allows prototypes to be built quickly, letting many people cooperate in building a system, and letting developers and users explore changing technology when it becomes available.

The development technique of design at run time (Chapter 7) similarly takes advantage of prototyping to explore the design space of UAR systems. It even goes further, extending an existing system at run time.

3.5 Related Tools and Technologies

The solution presented in the next chapters uses ideas from various software engineering tools and technologies.

These technologies have nothing to do specifically with the domain of ubiquitous augmented reality. However, they can help the problems described in Section 2.4. Here, they are presented very briefly, as there is ample literature available for each.

3.5.1 Service-Oriented Architecture

The motivation behind service oriented architectures, similarly to web services, is the desire to integrate different IT systems, purchased from different manufacturers, in a business environment. This allows both *enterprise application integration (EIA)* within one business, e.g. integrating production systems with purchasing systems, and also automated *business–to–business (B2B)* communication, e.g. for supply chain management. This type of integration normally has to deal with all sorts of compatibility problems.

The programming paradigm behind web services is that of cooperating, but *autonomous* distributed systems; the individual components or *services* are not centrally controlled. This paradigm matches the problem of interdependent distributed devices which occurs in ubiquitous augmented reality. Services are separate software components which may run on different computers. They communicate by way of middleware, sometimes called a *service bus*.

Individual services are loosely coupled. This means that a service does not assume anything about the identity or implementation of another service it communicates with. However, it does assume that the other service implements a commonly understood *interface*. Thus, service implementations can be exchanged after deployment, and the new services will still be able to interact with each other. As described above, loose coupling is a principle that is employed in several existing augmented reality and ubiquitous computing systems.

Using a service-oriented architecture only makes sense together with a service-oriented development process. Services must be developed in such a way that they actually can be reused in different environments. Simply using web service technology does not stop a system from becoming monolithic and difficult to maintain.

This combination of software architecture, middleware and development process which support each other is a major theme in the solution of Chapters 4 to 7.

3.5.2 Web Services

The term *web services* refers to an entire bundle of emerging industry standards for describing service interfaces and properties of implementations, for service registration and location, for service composition and communication between services. Web services can be used to implement a service-oriented architecture. [70] presents a good overview of web services.

Web services communicate using the *Simple Object Access Protocol (SOAP)*. This is a standardized XML-based encoding for method requests and replies. SOAP requests and replies, in turn, are transmitted via other network protocols, such as HTTP or SMTP. Programmer implementing a web service generally would not program the XML encoding and decoding (marshaling) themselves, but rather use stubs and proxies generated by the middleware. SOAP is to web services what IIOP is to CORBA objects.

The interfaces that a web service supports are specified in the *Web Service Description Language (WSDL)*. Some interfaces are industry standards; others may be defined within an organization. WSDL defines the SOAP operations that a web service supports, and the data formats it understands. Development tools generate proxies and stubs as part of the run-time middleware, which convert between SOAP and the implementation language of the web service.

Each service is identified by a *universal resource indicator (URI)* with a network address and a communication protocol that the service can be contacted by. Services can be registered in a *registry*, which again may be publicly accessible or only used within an organization. Web service clients can access such a registry in order to find an appropriate web service that implements a particular interface which they wish to use. For this, web services and web service clients use the standardized *Uniform Description, Discovery and Integration (UDDI)* API.

3.5.3 CORBA

The *Common Object Request Broker Architecture (CORBA)* is a middleware standard developed by the Object Management Group [96] and which has been continuously improved since the early 1990s. Its objective is similar to that of web services: to create a family of standards for interoperability between software developed by different vendors, running on different platforms. CORBA can be used to implement a service-oriented architecture.

The basic unit of a CORBA application is an *object*, which combines functionality and data, and frequently represents something in the real world. The functionality of objects are accessed via *interfaces*, which are specified in CORBA *Interface Definition Language (IDL)*. Interface definitions are translated by an *IDL compiler* into *stubs* and *skeletons* in various programming languages. These act as object wrappers for accessing remote objects.

Programmers access the methods of a remote object just as those of an object in the local process space. The stubs use a library, the *Object Request Broker (ORB)*, which translates local method calls to a standard network format. Another ORB, on the server side, un-marshals the parameters, and calls the implementing object's method using a skeleton.

Language bindings define a standard mapping from IDL to many different implementation languages. The standard network format is the *Internet Inter-ORB Protocol (IIOP)*. A lot of literature is available on CORBA, for example [53] and [101].

The *CORBA component model* is a feature of newer CORBA standards, which describes *components* as well as interfaces. Components can support many different interfaces simultaneously, and also require interfaces of other components. These required and provided interfaces are specified in *Component Description Language (CDL)*, which is similar to IDL.

This approach is well suited to model the interdependent distributed devices in UAR systems, and the adaptive service dependency architecture uses a similar approach.

3.5.4 Design by Contract

In design by contract [84], preconditions for the use of software resources are made explicit, as well as postconditions that hold after use, if the preconditions have been met. These preconditions and postconditions are specified explicitly, usually taking advantage of special programming language features. The preconditions and postcondition create a *contract* between the caller of a method and the object implementing that method.

Design by contract is used to promote loose coupling between software components; one component implementation can be exchanged for another as long as it satisfies the appropriate contract. It can also be used as the basis for a formal verification of software correctness.

In the adaptive service dependency architecture, the middleware manages dependencies between components, which could be modeled as simple multilateral contracts.

3.5.5 Extreme Programming

Extreme programming [9] is a method of software development that is designed around customer satisfaction. It allows developers to respond to changing customer requirements, even late in the software development life cycle. Thus, it can help address the problem of ill-defined requirements found in UAR systems. This type of development process is called an *agile process*.

This approach also emphasizes team work, whereby customers, developers and managers work together to solve the software development problem. This is similar to the many people involved in building a UAR system.

The technique of design at run time (Chapter 7) uses ideas from extreme programming.

3.5.6 Scrum

Scrum [122] is an incremental, iterative agile process for developing software (and other products). It focuses on self-organizing teams called *Scrum teams*, which select tasks for their 30-day development *sprints* from a *project backlog* which is updated after each sprint. In each Scrum team, every work day commences with a *daily Scrum*, a short meeting in which all developers report on their current progress and remaining work. The team maintains a *sprint backlog* for the tasks of its sprint. The role of management in Scrum is to enable the teams to perform their work, keeping obstacles out of the way as much as possible.

Scrum can reuse techniques such as extreme programming. It does not make any assumptions about the software being developed. In particular, it does not force developers to keep documentation and models consistent with code; it can works on the existing system as the main artifact. Thus, it is a good match for design at run time.

3.6 The DWARF Project

The *Distributed Wearable Augmented Reality Framework (DWARF)* research project [87] has been exploring the possibilities of distributed augmented reality since 2000. It has investigated technical issues ranging from distributed tracking to multi-modal user interfaces, and the software engineering issues of ubiquitous augmented reality.

The implementation of an open source software framework—also called DWARF—has led to a large number of experimental systems. Most of the concepts proposed in this dissertation were implemented and tested in this framework.

3.6.1 DWARF 2001

A first version of DWARF was implemented in 2000 and 2001; the entire framework is described in [5] and [109]. This section describes the features and limitations of this first version, DWARF 2001.

The basic idea behind DWARF was that of AR-ready intelligent environments [69]; a mobile augmented reality system takes advantages of stationary devices in a rich environment. This was the first step towards the concept of ubiquitous augmented reality.

The target domain of DWARF 2001 was mobile augmented reality. The *Pathfinder* application essentially consisted of a navigation scenario; a pedestrian user navigated both outdoors and indoors.

DWARF 2001 used distributed components, which was fairly new to augmented reality at that time. It used middleware and component descriptions to assemble the components into a coherent system, as described in [77]. The components of DWARF 2001 were also designed to be loosely coupled, although this was not always carried through in practice. It also used event-based communication, and was designed to use service discovery, although that was not actually implemented.

3.6.2 Limitations of DWARF 2001

Despite its features, which made it a success at the time, DWARF 2001 had several important limitations.

First, the development process was not considered when developing DWARF 2001. Thus, it did not include any development tools, and was difficult for developers to learn. This was a crucial limitation when facing the problem of ill-defined requirements, which must be addressed with the development process. There was also no clear model for applications. Thus, building new applications was more difficult than it should have been.

Second, the adaptivity was limited. The architecture was designed to use service discovery to bootstrap a mobile augmented reality system from available components. However, it could not adapt the system to environmental changes once it was running—crucial in addressing the problem of component uncertainty. The distributed components were called services, but they were not actually configurable or adaptable. Thus, the system could not adapt to changing context or changing availability. The middleware was fully autonomous, and did not require user interaction. However, it also did not *allow* user interaction, even when that would have been desirable, such as when adapting the system to changing context.

Finally, in 2001, DWARF was a framework that was used to build exactly one system. At the time, no valuable claims about its use to develop further systems could be made.

Part of the work for this dissertation has been to address these limitations, as described in the following chapters.

Overview of Proposed Solution

I propose to build software for ubiquitous augmented reality systems using the adaptive service dependency architecture, middleware for decentralized service management, and the development technique of design at run time.

This chapter gives an overview of the architecture, supporting middleware, and development process which I propose in order to address the challenges of developing software for ubiquitous augmented reality.

The adaptive service dependency architecture (Chapter 5) consists of of distributed *services* that are interdependent, but loosely coupled. The dependencies between services are externalized into service descriptions, where they are modeled as needs and abilities. The behavior of the system depends on the arising structure of cooperation and communication between services.

Middleware, in the form of decentralized service managers (Chapter 6), adapts the system's structure to environmental changes.

In design at run time (Chapter 7), users and developers cooperate in incrementally improving a running system, either synchronously or asynchronously.

Four prototype systems that use the adaptive service dependency architecture, decentralized service management, and design at run time, are used as examples in the following chapters.

4.1 Feedback Loops for AR, Adaption, and Development

In a ubiquitous augmented reality system, there are three feedback loops. The first is the tightly coupled *AR feedback loop* between a user's actions and the system's response; this is part of every AR system. For example, when the user moves his head, the image in the head-mounted display adjusts within a few milliseconds to maintain the impression of a virtual object registered in three dimensions. Data flow in this loop must be fast, to support near-real-time performance. In Figure 4.1 on the following page, this feedback loop is shown, together with two other, larger, feedback loops.

The second is the *adaption loop*. To deal with the problem of component uncertainty that arises from the users' mobility and the changing availability of distributed devices, the middleware changes the system's structure in response to environmental changes. This changes the set of running services and their communication structure, as well as individual services'

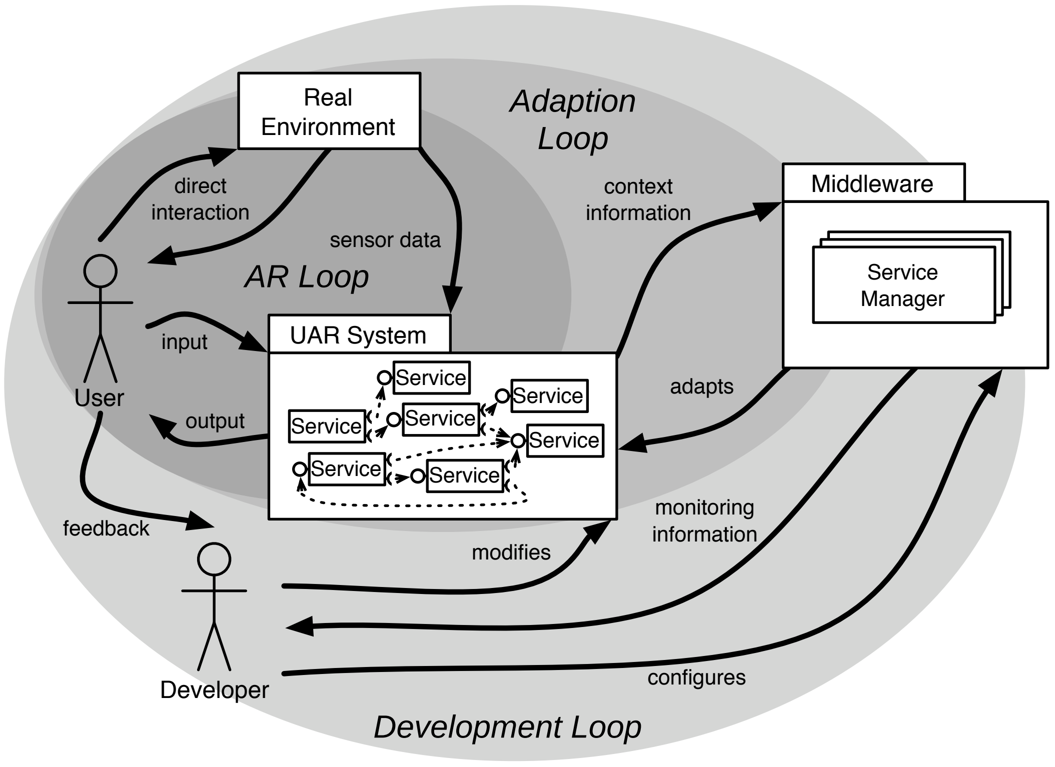


Figure 4.1: Three feedback loops in proposed solution.
The *AR loop* is immediate feedback between user and system, e.g. when the user moves his head, or a tangible interaction object. The *adaption loop* is adaption of the system behavior at run time, e.g. when the user enters a new room. The *development loop* is incremental extension of the running system by users and developers. The arrows indicate data flow.

behavior. This in turn influences the information available to the middleware for reconfiguring the system. This loop is the core of the system's adaptability.

The third feedback loop is the *development loop*. To deal with ill-defined requirements, users and developers cooperate in the process of *design at run time*. Monitoring tools gather information on the system's structure, as well as user context.

Since the system tolerates pieces being removed and added at run time, developers can reconfigure (or reprogram) individual services while the system is running, as well as reconfigure the system's structure. Development cycles become extremely short, and users and developers can cooperatively improve a live system—a process that is particularly useful in improving the usability of a ubiquitous augmented reality system. In addition, the system itself can gather feedback from the user and present it to developers, who then can update the system off-line.

This creates a feedback loop between users, developers and the system itself, allowing the system to evolve over time.

These three feedback loops operate at different time scales: in the AR loop, the system should react within approximately 10 milliseconds; in the adaption loop, reconfiguration of the system within about 1 second is acceptable; and in the development loop, making a change to the system and evaluating it can take from minutes to hours or several days. The use of different time scales is similar to the idea of latency layering.

4.2 Services and Service Managers on Distributed Hardware

In the adaptive service dependency architecture, the basic building blocks of the software for a ubiquitous augmented reality system are distributed services, running on mobile and stationary computers (Figure 4.2 on the next page). Each service has a well-defined functionality that is understandable in the same fashion to both the user and the developer.

The adaptive service dependency architecture allows these services to be composed into a complete augmented reality system. Beyond that, it contains mechanisms for parameterizing services and describing their interdependencies. A service's dependencies on other services are modeled as needs; the functionality that a service offers to other services and to the user are called abilities. Needs depend on abilities; however, these dependencies are not hard-coded, but use loose coupling and service discovery. This allows the services and the system structure to be reconfigured according to changes in the environment or desires of the user.

The distributed services require a supporting middleware infrastructure, and this consists of distributed *service managers*. One service manager runs on each mobile or stationary computer, and together, they coordinate the services and adapt the system to environmental changes.

4.3 Architecture, Architectural Style, and Frameworks

The adaptive service dependency architecture is more than just an architecture: it is an *architectural style* [21], like *pipes-and-filters* or *model-view-controller*. Different systems can be built using this architectural style, and each has an architecture of its own. For example, the simple system of Figure 4.2 has a simple architecture that consists of five services and their interdependencies, and this architecture is an instance of the adaptive service dependency architectural style. To reach the level of an established *architectural pattern* [17], the adaptive service dependency architecture still must be instantiated and implemented in many more systems.

An architectural style can be used as the basis for a *framework*: a set of classes or components providing a general solution that can be refined to build an application [14]. For example, the *model-view-controller* architectural style has been used for many user interface frameworks.

In [109], Reicher points out that a full-fledged framework for augmented reality must consist of three layers, with a sub-framework in each: the *architectural style framework*, the *solution domain framework*, and the *inter-application framework* (Figure 4.3 on page 51).

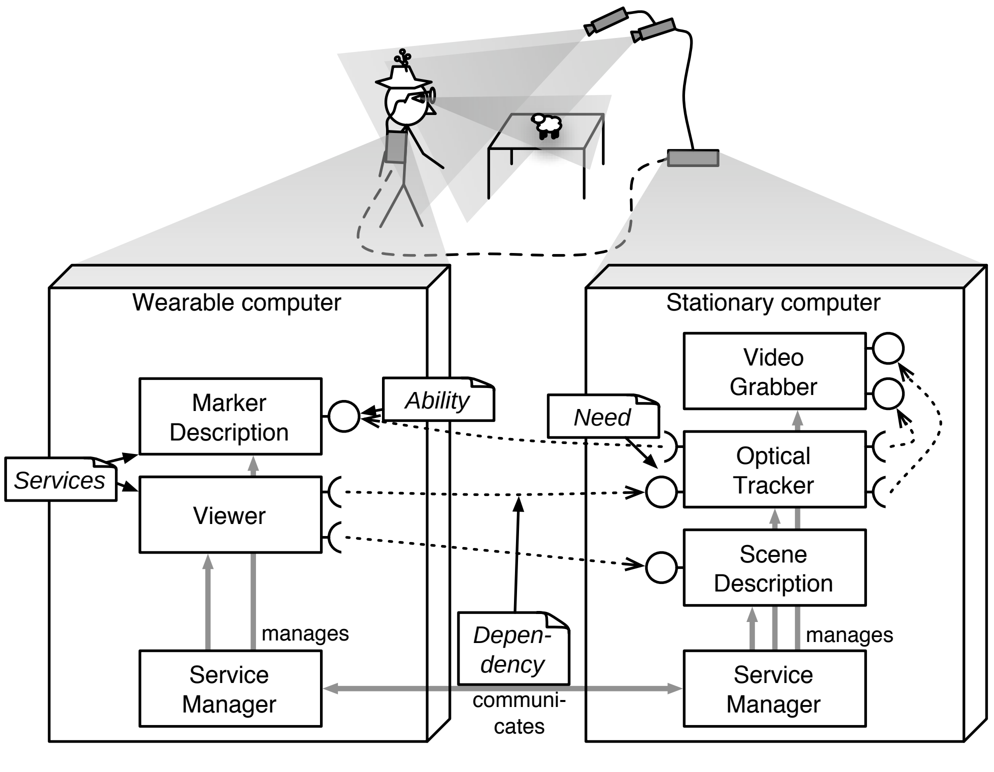


Figure 4.2: Services with and needs abilities, managed by service managers, on distributed hardware of a simple UAR system.
The user carries an optical marker on his hat, described in the Marker Description service. This is used by the room's optical tracking system (the Video Grabber and Optical Tracker services) to determine the position of the user's head. The user's Viewer service uses this position to display the scene from the Scene Description service. The services are connected and coordinated by distributed service managers.

The adaptive service dependency architectural style is the basis for the three-layer DWARF framework. The architectural style framework of DWARF consists of the architecture *meta-model* and the middleware implementation (the service managers). The meta-model is an abstract model of services, needs, abilities and more. It differs from the model of a particular architecture built with the adaptive service dependency architectural style; the architecture model consists of several services, whereas the architecture meta-model describes the structure of the services themselves, and is a model of the architectural style.

Throughout the following chapters, the term *adaptive service dependency architecture* is used for simplicity; however, *adaptive service dependency architectural style* could be used as well.

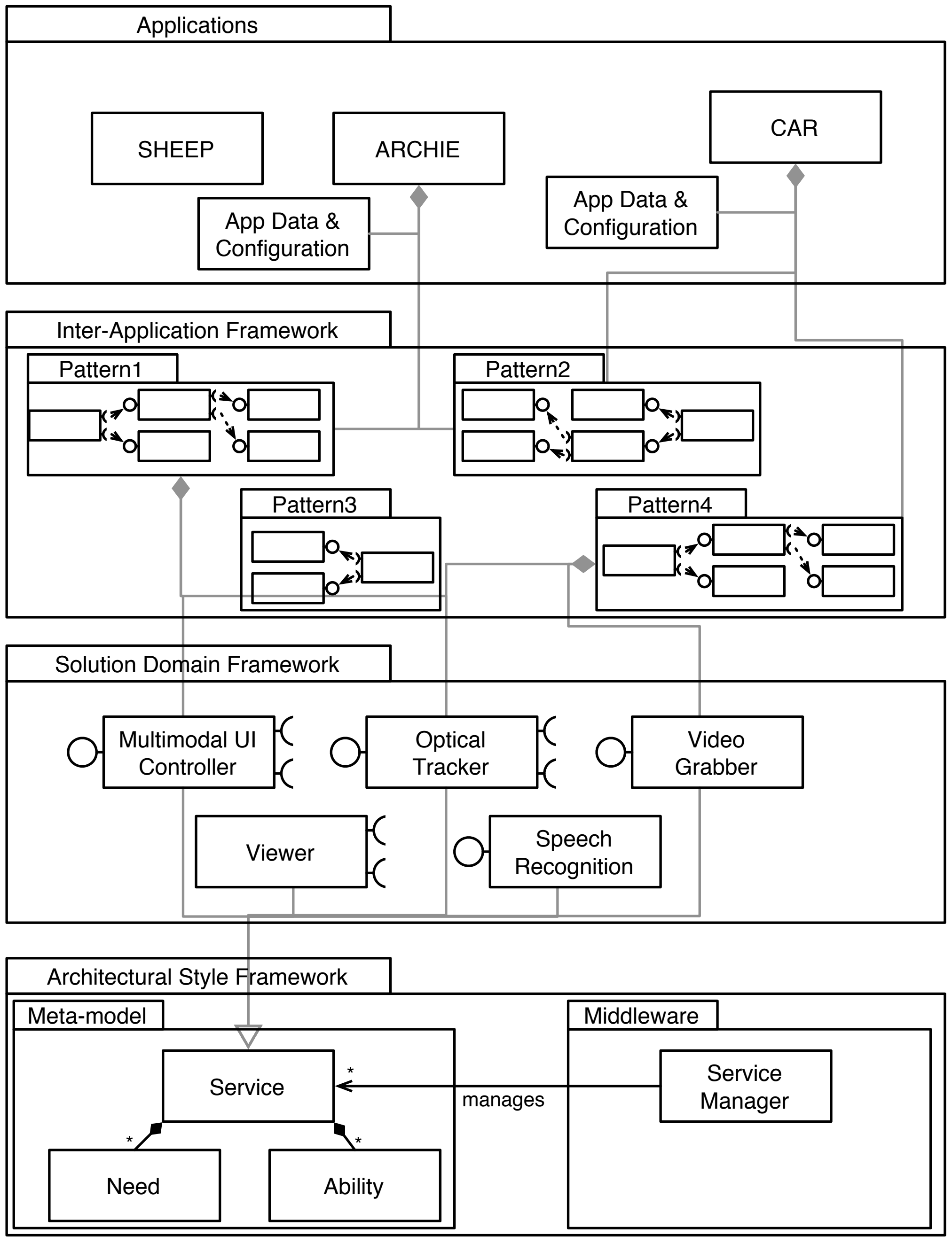


Figure 4.3: Building frameworks on top of the adaptive service dependency architecture. The adaptive service dependency architecture is the basis of the *architectural style framework*. This can be used to build a *solution domain framework* of services for UAR systems, which are then part of an *inter-application framework* for specific subproblems and application domains of UAR. These patterns are used to build UAR applications. (Figure adapted from [109].

4.4 Effects on Software Development Challenges

Together, adaptive service dependency architecture, decentralized service management and design at run time address all of the software development challenges described in Section 2.4, by combining and extending existing architectural approaches from Section 3.4.

4.4.1 Component Uncertainty

The problem of component uncertainty is addressed by the adaptive service dependency architecture, and its supporting service managers. The architecture is designed following the forces that lead to this problem.

Interdependent distributed devices The services in the system are not simply an unordered collection of distributed components. Rather, their interdependencies, modeled as needs and abilities are an integral part of the architecture.

Changing availability of devices Services are loosely coupled, so one service does not rely on a particular implementation of another service. Rather, it has the need for another service to provide an ability of the appropriate type. This is similar to the concept of *polymorphism* in object-oriented programming.

This means that when one service is not available, another—with the same type of ability—can be used instead. For example, in the team action application, different team members can use each other's devices when they come in range of an ad hoc wireless network. Service discovery is used to find available devices, and service are described by their quality of service.

Furthermore, there is no central coordinating component, improving fault tolerance. The middleware is entirely decentralized.

Changing context influences choice of components The architecture employs context refinement, using sensor data to determine changing context. According to this context, the middleware reconfigures the connection structure between services, employing adaption by component selection. Since the architecture uses component descriptions, the middleware can evaluate descriptions of the services to determine which service should be used in which context.

Incomplete knowledge of components The loose coupling of services allows them to be developed with incomplete knowledge of each other. In specifying the needs of a service, a developer is encouraged to describe exactly what the service depends on, not more, and not less. Thus, the service description available at run time is sufficient to select appropriate other matching services. This allows services that have been developed independently of one another to interoperate, as long as they use standardized interfaces.

4.4.2 Ill-defined Requirements

The problem of ill-defined requirements is addressed by the development technique of design at run time. This process takes advantage of the architecture's flexibility, and again is designed based on the forces that lead to this problem.

New and changing technology Design at run time makes prototyping easy; it allows developers and users to combine existing services in new ways quickly. This allows users and developers to explore the possibilities of ubiquitous augmented reality.

Many people involved The idea of design at run time is centered around the idea of developers and users cooperating. This pays tribute to the fact that many different people with many different backgrounds must contribute to make UAR a success. It also follows the idea of agile processes such as Scrum.

New applications When users discover new possibilities for using a ubiquitous augmented reality system, they can either reconfigure the system themselves, building new applications. Or they can cooperate with developers to build a prototype of the new application, which is then finished off-line. Or, they can record the wish for a new feature or new application using the UAR system itself; the system then packages this request in a convenient form for the developer.

Developers, in turn, can add functionality to the running system, letting them test it quickly and get rapid feedback from users.

4.4.3 Performance Constraints

The problem of performance is essentially a constraint on the design space for software architecture and middleware. Especially the middleware must address certain issues that are inherent in ubiquitous augmented reality.

Immersivity To maintain near-real-time performance required for immersive applications, the middleware can take advantage of latency layering; it can connect tracking services directly to rendering services without having to feed the data through other services or to a central server. This follows the approach of data flow architectures.

Furthermore, the use of different communication protocols allows specialized protocols to be used for data that must be delivered quickly.

Many communication types The middleware uses connectors which handle different communication protocols, so that services can communicate using the most efficient protocol in different situations. One common protocol is event-based communication.

Also, service developers can choose the communication protocol of their choice, to make development easier.

Scalability The service discovery of the distributed middleware takes advantage of services' context to discover matching services. For example, the middleware components on computers in the same physical room communicate more closely than those in different rooms. This allows the middleware to scale to large numbers of distributed hosts.

Furthermore, using latency layering and context refinement, the architecture makes it easy to design hierarchies of services that communicate intensively over a short range and less so over a long range. For example, in a campus scenario, the exact head position of a user in another building is usually not important; knowing the room that he is in will suffice.

4.5 Effects for the Stakeholders

Architecture, middleware and process affect the interests of all stakeholders, as described in Section 2.5, of a UAR system.

For the user, the architecture bundles software with hardware in units which are easily understandable, following the *tool metaphor* [6]. This makes the system as a whole more understandable. The process of collaboratively improving a running system together with developers is very beneficial to the user; new applications can be developed more quickly, and existing ones can be custom-tailored to the user's task. On the other hand, this also means new responsibility for users: they become actively involved in development of the system, something they may not be comfortable with.

In the adaptive service dependency architecture, applications are built by combining existing services, configuring them appropriately, and describing application logic in special *application services*. This is a natural approach for application developers.

However, in developing UAR applications, the developer must consider the adaptivity of the architecture. He cannot simply select a fixed set of existing services, but rather must design the application services so that they can adapt to changes in other services. The adaptive architecture and middleware make this task easier, but they do not shield the application developer form thinking about the problem.

In developing prototypes of applications, the application developer can afford to relax the adaptivity requirement somewhat; the prototype may be tailored to a particular hardware configuration at first. This makes prototyping easier, especially when combined with the development tools described in Section 7.2.

User interface designers face a difficult task in ubiquitous computing systems: they must design their interfaces in such a way that they can run on a variety of different devices. There is still much research to be done in this area. However, Section 4.6.2 shows an architectural approach that makes this task easier.

The component developer, or service developer's task is greatly simplified by the architecture and middleware: he can focus on solving the task at hand and does not have to worry about the internals of other services or the communication mechanisms. However, he does have to design a service rather than a component; this means that he has to make the service adaptable to varying contexts. The middleware offers support for this, but again, it does not shield the developer from thinking about the problem.

Maintaining any installed system is no easy task, and UAR is no exception. However, the fact that the architecture and middleware allow services to be removed from and added to a running system makes the maintainer's job somewhat easier. Also, Section 7.2 describes several management tools that can help manage the system and diagnose problems.

The infrastructure developer carries a large responsibility, since the architecture is centered around the middleware. If the middleware fails, the entire system fails. On the other hand, the decentralized adaptive architecture can help in building middleware that can adapt in changing circumstances.

4.6 Prototype Systems

This section describes four experimental systems that were built between 2002 and 2004. For each, it shows the project goals and the application scenario, and the overall system architecture. Each of the systems was realized using the adaptive service dependency architecture, as implemented in DWARF, and the SHEEP, ARCHIE and CAR projects also used design at run time. The following chapters use examples from these four projects.

4.6.1 SHEEP

The main goal of the *Shared Environment Entertainment Pasture (SHEEP)* project was to test the feasibility of distributed augmented reality systems using the adaptive service dependency architecture. SHEEP was presented to the international AR research community at the demonstration session of ISMAR 2002 in Darmstadt [80, 118].

The demonstration system used a multiplayer shepherding game to explore the possibilities of mutimodal, multiuser interaction with wearable computing in an intelligent environment. The game is centered around a table with a projected pastoral landscape (Figure 4.4 on the next page). Players can use different intuitive interaction technologies offered by the mobile and stationary computers.

For example, when a player puts a tangible plastic sheep on the table, the virtual sheep recognize it as a member of the herd and move towards it. In order to add or remove sheep, the user points at the table with a magic wand and says "insert" or "die." With a palmtop computer, a player can pick up a sheep and move it to another part of the pasture. Using a see-through head-mounted display or a laptop, the user can view the landscape three-dimensionally, pick up virtual sheep and color them using virtual color bars.

The architecture of SHEEP is shown in Figure 4.5 on page 57. The basic software components of SHEEP are DWARF services. The services can be divided into the subsystems *tracking*, *sheep simulation*, *visualization* and *interaction*. Many of the DWARF services form adapters to connect to third-party software (shown in gray), e.g. for tracking, speech recognition or 3D rendering.

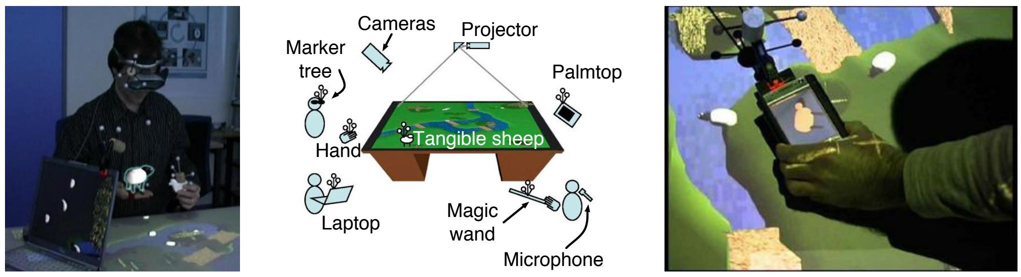


Figure 4.4: SHEEP, the Shared Environment Entertainment Pasture.
Left, virtual sheep floating on hand is shown on head-mounted display and laptop. Center, game setup centered around a projector table. Right, virtual sheep is scooped off projector table onto palmtop. (Figures from [80].)

The same service can have one or more instances running in the network. For example, tracking services are running as a single instance, sending positional data to all interested components. Other services, such as user interface controllers or 3D viewers for scenes described in the *Virtual Reality Markup Language (VRML)*, are provided in many instances. For example, a separate user interface controller is available for each user, and each display has its own VRML viewer. Finally, any number of sheep services can be running on the machines in the network.

The data flow through the system is straightforward. A commercial optical tracking system, *DTrack*, manufactured by *ART* [2], tracks the position of retro-reflective markers on the plastic sheep, head-mounted display, palmtop, magic wand and other objects. The ART Tracker service generates PoseData from the tracking system and sends it to the Calibration service. This adjusts the pose data to calibrated coordinates, and sends it to the rest of the system. The Viewer services receive pose data, e.g. for the user's head and hand position, and use them to render the 3D landscape correctly.

At the same time, the Sheep services simulate sheep-like motion, generating PoseData as well. The sheep services receive pose data from each other, and also from the tracking system for the tangible plastic sheep. The viewer services receive the pose data as well, and render grazing sheep on the landscape.

For interaction, the Collision Detection service compares incoming PoseData from the user's hand, the magic wand, the table, and of the virtual sheep, generating Collision events, e.g. when a user picks up a virtual sheep or touches the table with the wand. The UI Controller services receive these events, as well as speech recognition events from the Speech Adapter. Using Petri nets to model user interaction, with transitions such as "scoop sheep" and "put down sheep," the UI Controller sends appropriate SheepData events to the sheep (which stop moving when they are picked up) and to the viewers (which display a floating sheep on the user's hand, rather than a grazing sheep on the meadow). The UI controllers also send events to the Sound Player service to provide audio feedback to the user.

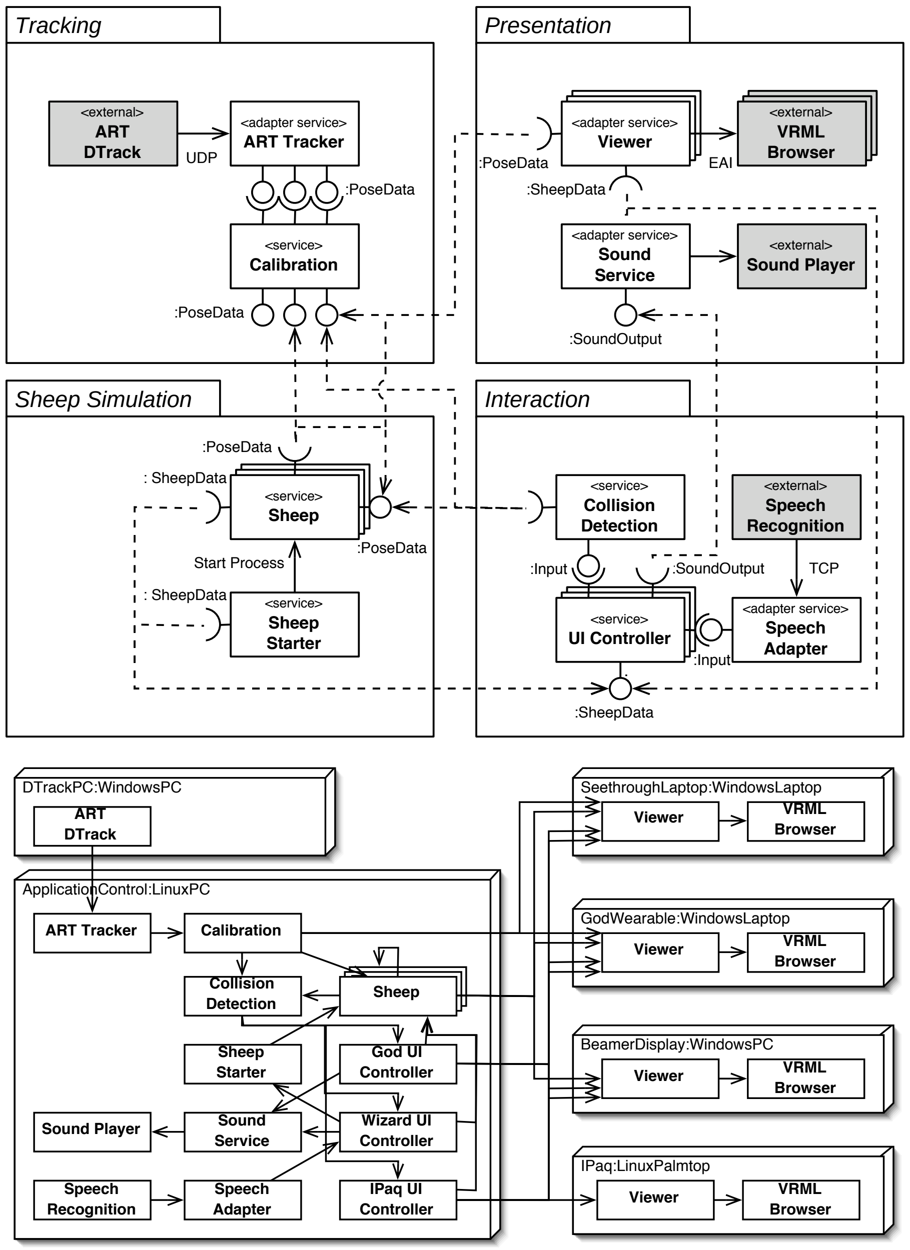


Figure 4.5: The architecture of SHEEP.
Top, services and subsystems. Services communicate via their abilities (shown as circles) and needs (shown as semicircles); third-party components are shown in gray. Bottom, deployment. The boxes are services and external components; The arrows indicate data flow. The system consists of three stationary and three mobile computers. (Figures from [80].)

The data flow between services follows the services' dependencies, modeled as *needs* and *abilities*. For example, the viewer service has a need for pose data, which the tracking service can provide with its abilities. Distributed middleware, consisting of *service managers*, perform service discovery and set up the data flow.

The services are distributed on several machines, following the a tool metaphor [6], bundling software with hardware in units that are easily understandable to the user. For example, the palmtop system performs the complete 3D rendering locally, rather than retrieving an externally rendered video image from a server. The services of SHEEP are described in more detail in [36, 54, 80].

4.6.2 ARCHIE

The *Augmented Reality Collaborative Home Improvement Environment (ARCHIE)* project's goal was to extend the concepts of the adaptive service dependency architecture towards greater mobility and adaptivity, building a multiuser, collaborative system that operates in several different rooms. ARCHIE is described in detail in [54, 71, 75, 126, 134, 142, 144].

The scenario for ARCHIE dealt with the collaborative design of buildings, involving architects, engineers and customers. The collaboration took part in several rooms, allowing users with mobile computing equipment to roam between rooms and use different stationary services installed there. Thus, ARCHIE was one of the first applications to satisfy all points of the UAR definition (Section 2.1).

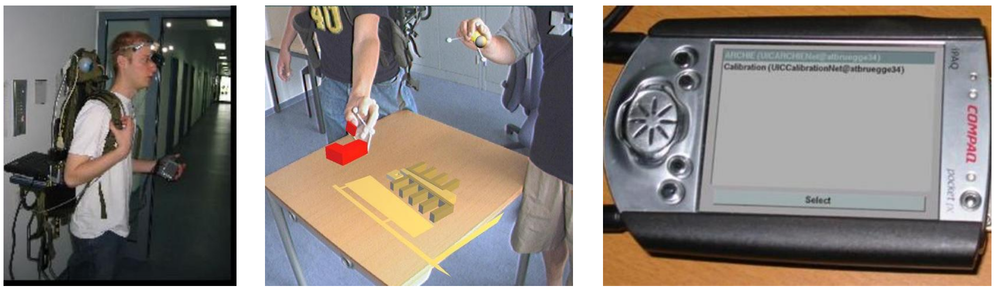


Figure 4.6: ARCHIE, the Augmented Reality Collaborative Home Improvement Environment. Left, prototype wearable system, with touchglove input device. Center, two users collaboratively work on a model of the TUM computer science building. Right, palmtop system used to run the selector service. (Figures from [126].)

An overview of ARCHIE's architecture is shown in Figure 4.7 on the next page. As with SHEEP, the basic elements are DWARF services, which communicate via their needs and abilities. There are three notable differences, however.

First, ARCHIE includes several services that store and provide access to the building model, the Model and Model Server services. The viewers access this model concurrently. Second, the system supports different actions in different rooms: the same viewer can be used to model

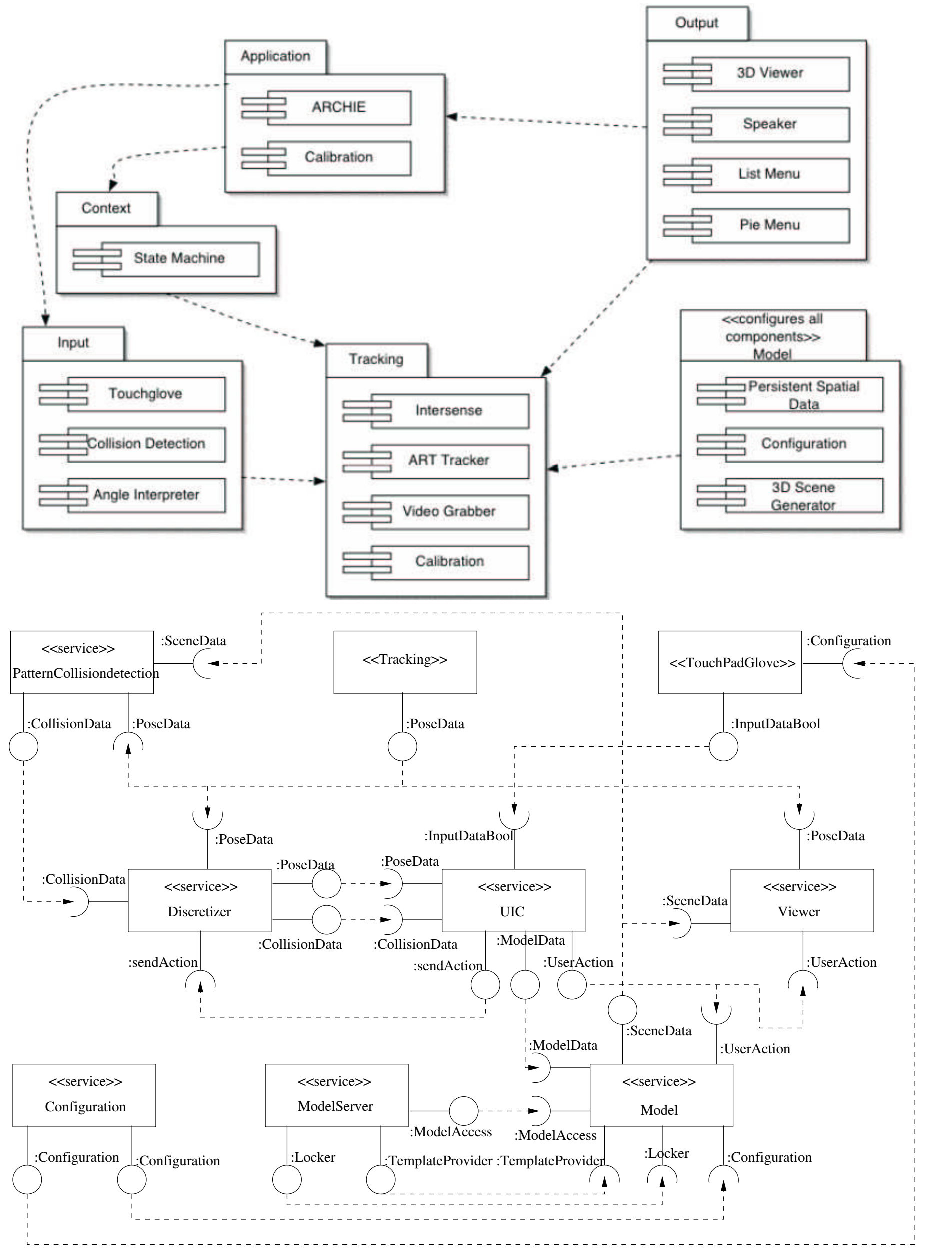


Figure 4.7: The architecture of ARCHIE.
Top, all services in the ARCHIE demonstration system, shown as components, organized by subsystem. The arrows indicate dependencies. Bottom, the services participating in the stationary modeling task. Here, the arrows indicate data flow. (Figures from [134].)

buildings (in the modeling room), but also to navigate through a real building (e.g. in order to find the modeling room) or to calibrate the head-mounted display. To model these different situations the user can be in, the services have *context attributes*. Depending on the values of these attributes, the services are connected differently. Third, the user can dynamically choose between different interaction devices, using a Selector service running on a palmtop to indicate his preferences. Thus, the service managers do not combine services completely automatically, but allow the user to choose.

The ARCHIE project additionally developed tools for live usability evaluations [72]. Using these tools, shown in Figure 7.2 on page 126, a usability engineer can view graphs of the user's performance with different experimental interaction devices, and record the user's comments and wishes. This allows rapid iterations of improvement for prototype interactive systems.

4.6.3 Ubitrack

The *ubiquitous tracking (Ubitrack)* project is an ongoing collaboration between the TU Wien and the TU München [11, 90, 92, 108, 127, 137, 139], with the goal of dynamically combining different position and orientation tracking sensors to enhance both the range and the accuracy of existing tracking technology, allowing ubiquitous augmented reality applications.

A small example has been built to illustrate the Ubitrack concepts and provide a suitable scenario for testing implementations. A stationary tracker, combined with a mobile camera delivering images to an optical tracker, effectively allows the system to "see around corners". Figure 4.8 on the next page shows a camera tracked by an ART tracking system, a schematic of the hardware setup, and a graph describing the corresponding spatial relationships. A ceiling-mounted projector P displays a pastoral landscape, populated by a sheep, on a table. The projector is calibrated such that it shares the same coordinate system as the ART tracker A; therefore, a static relationship exists between P and A. A mobile user is equipped with an AR Toolkit camera C on which an ART target T is mounted, resulting in another static relationship described by another edge in the graph. The camera is attached to a mobile computer with a wireless network interface. The middleware is running on the lab computers and on the mobile computer.

When a user enters the room, the two Ubitrack systems connect, and the description of an AR Toolkit marker is transferred to the user's mobile computer enabling the camera to track it. As long as the marker remains in the view of the camera, and the ART target attached to the camera is tracked by the ART system, then the virtual sheep can still be tracked in the coordinate frame of the ART system, even if it is physically out of range. Consequently, when the marker is moved the image of the sheep displayed by the projector can be seen to move accordingly. If the sheep is moved into a pen outside the range of the projector, it can still be viewed using some other tracked or fixed display.

Spatial relationships can be represented by a graph [16], in which objects are nodes, and spatial relationships between objects are directed edges. Each edge represents the spatial transformation between objects.

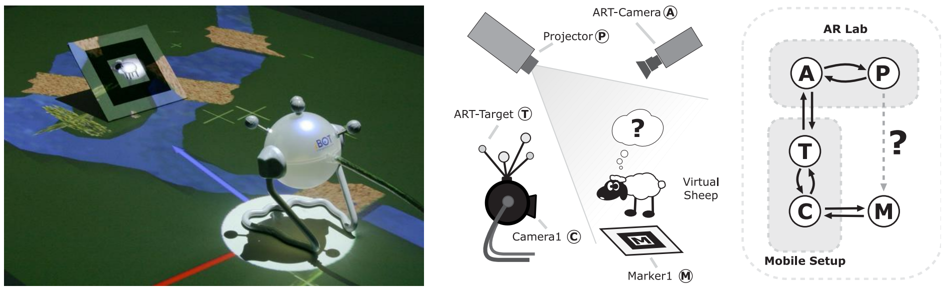


Figure 4.8: An example application of ubiquitous tracking.
Left, an AR Toolkit marker tracked by a mobile camera attached to an ART target. A virtual sheep is projected onto the AR Toolkit marker relative to the ART coordinate system. Right, the example setup and its spatial relationship graph. (Figures from [90].)

Trackers take measurements of the spatial relationship between themselves and other objects or *locatables*. The quality of a measurement is described using a set of *attributes* which includes properties such as latency, confidence values, or a standard deviation. Measurements add edges to the graph or update attributes in existing edges.

At runtime, the Ubitrack framework builds a data flow graph that is distinct from the spatial relationship (SR) graph. This data flow graph dynamically combines tracker data, so that spatial relationships requested by an application can be delivered. The data flow graphs are designed to minimize communication overhead, while still being able to globally infer new measurements through dynamic recombination. The configuration strategy is informed by qualitative high-level measures derived from measurement attributes rather than the value of each raw measurement taken individually. It can be seen that the topology of the SR graph changes much less frequently than the underlying raw measurement data, especially for high frequency trackers.

The Ubitrack architecture (Figure 4.9 on the following page) consists of three layers; in each layer, functionality is mapped onto DWARF services.

The *sensor layer* incorporates all hardware devices and software components that deliver raw spatial data. A generic *sensor API* provides an abstraction from the specific hardware devices while providing sufficient information to make full use of existing AR trackers. Trackers are DWARF services, with abilities of type PoseData.

The *Ubitrack layer* represents the formal model described earlier: aggregating tracking data, inferring knowledge of spatial relationships, and building runtime data flow graphs from the abstract spatial relationship graph. A distributed middleware component, the *Ubitrack Middleware Agent (UMA)*, extends the DWARF service manager and searches the spatial relationship graph for required relationships. Both nodes and edges of the SR graph are modeled as DWARF services. *Inference components* combine data from trackers and are realized as individual services each with a single ability for sending combined measurements to applications, and an

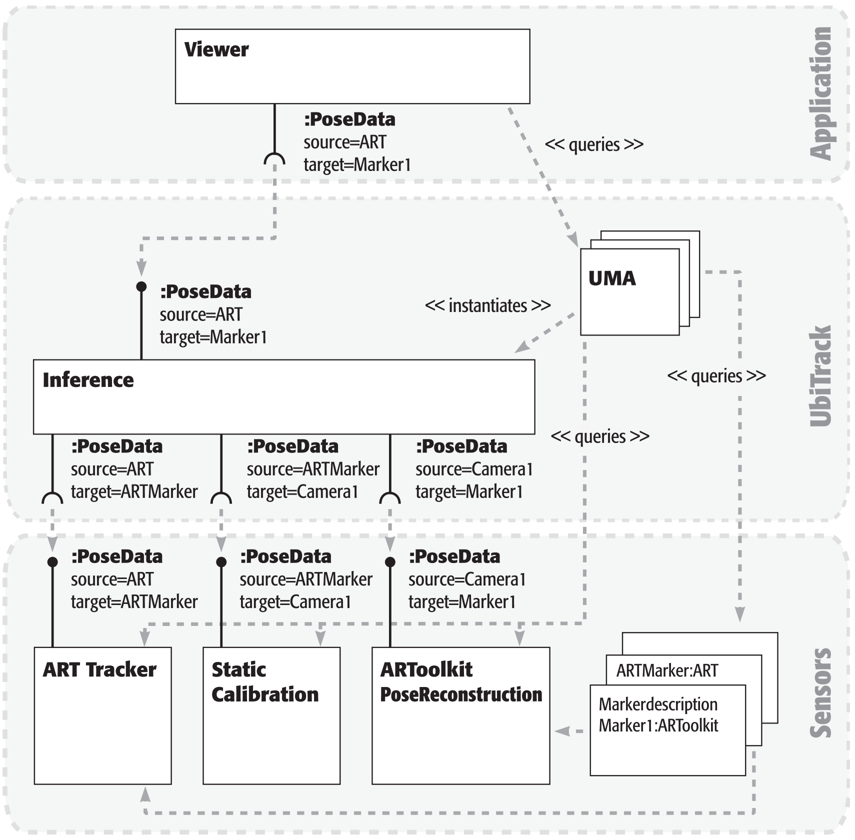


Figure 4.9: The architecture of Ubitrack.
DWARF services are structured in the three layer Ubitrack model. Arrows indicate dependencies from needs to abilities. (Figure from [90].)

arbitrary number of needs. The needs and abilities are automatically set by the UMA in order to connect tracking services that form the chosen path.

The *application layer* contains components that need spatial information, such as interaction controllers or displays. Ubitrack queries are modeled as DWARF needs, of type PoseData, with attributes that specify the desired source and target nodes within the SR graph.

4.6.4 CAR

The *Car Augmented Reality (CAR)* project of spring 2004 was a cooperation with a partner from the automobile industry. The goal of CAR was to create a collaboration platform for computer scientists (UI programmers) and non-technicians (human factors specialists, psychologists). The platform allows collaborative design of visualizations and interaction metaphors to be

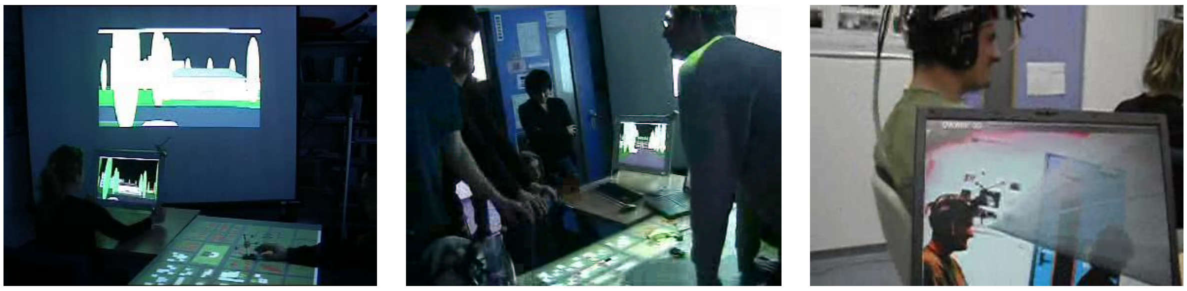


Figure 4.10: The Car Augmented Reality (CAR) project.
Left, a user sits in the simulated car environment viewing the projection screen (simulating the windshield) and the laptop (simulating the view through a heads-up display), while a tangible car on the map table controls the position. Center, several users discuss the current prototype system. Right, visualization of a user's field of view for ergonomic evaluation, as determined by a combined head and eye tracking system.

used in the next-generation cars with heads-up displays. It focused on two scenarios: parking assistance and a tourist guide. CAR is described in detail in [28, 41, 55, 95, 128].

On the technical level, CAR incorporated techniques like layout of information on multiple displays and active user interfaces with eye tracking. A core element of the CAR project was the idea of design at run time: designers, psychologists and computer scientists could collaboratively improve prototypical user interfaces.

Figure 4.10 shows the collaboration and prototyping environment of CAR. In one part of the room is a mockup of an automobile interior, with a simulated landscape shown on a projection screen where the windshield would be. A mobile laptop can be placed in various positions around the driver, showing the effect of a heads-up display there. Beside the car mockup is a projection table with a tangible toy car on a projected city landscape. This toy car can be moved around, like in the SHEEP system; here, however, the position of the toy car corresponds to the driver's position in the car mockup. Thus, one user can move the toy car around to change positions, and another user can sit in the simulated car and give feedback to of UI's behavior.

The parameters of the user interface in the simulated heads-up display can be adjusted at run time, using interactive tools. For example, the magnification factor of a virtual map in the windshield can be specified as a function of the car's distance to the destination; a UI designer can adjust this function at run time, using tools such as those shown in Figure 7.7 on page 130.

Developers can also modify the behavior of the system by configuring components and changing the data flow between services. For this, CAR uses the *DWARF Interactive Visualization Environment (DIVE)*, a monitoring and configuration tool that displays the structure of a running DWARF system as a dynamically generated graph [41, 107]. The services in one running instance of CAR, displayed in DIVE, are shown in Figure 4.11 on the next page.

The basic architecture is similar to that of SHEEP and ARCHIE. A difference is the addition

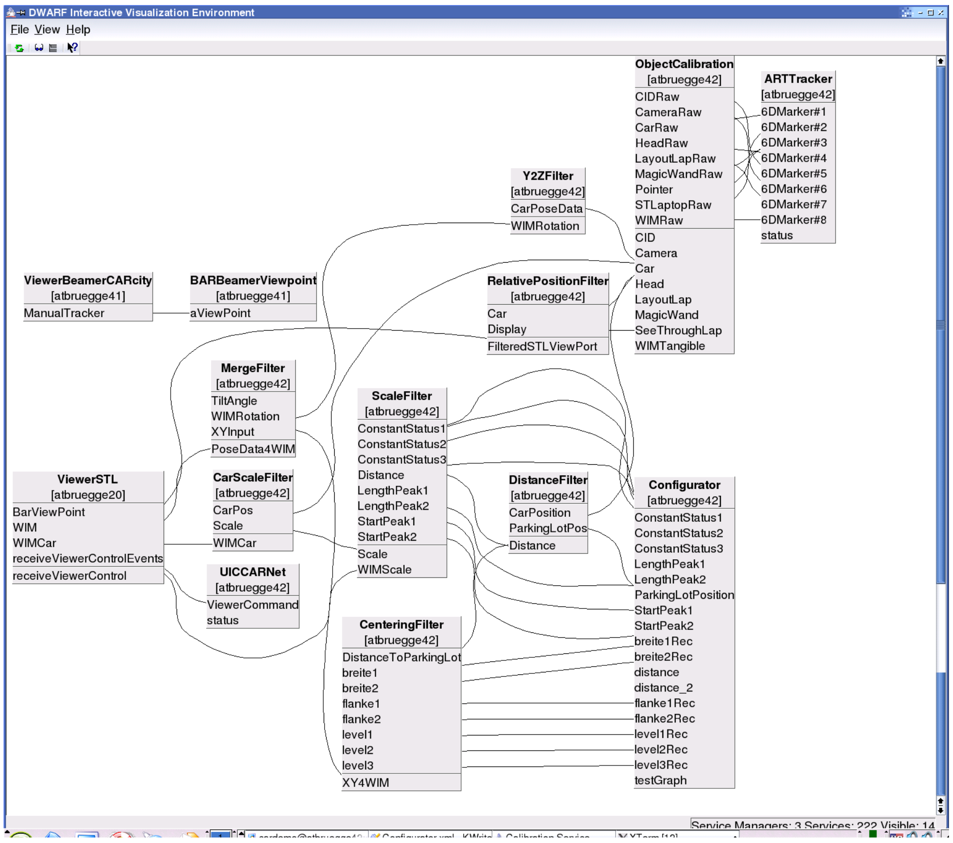


Figure 4.11: The architecture of CAR.
Screenshot of the visualization tool DIVE used in DWARF, showing the services of CAR, their needs and abilities, and the connections between them.

of several Filter services, such as the Relative Position Filter or Scale Filter services. These form a data flow network which can be configured at run time. The visualization tool showing the services in Figure 4.11 can be used to reconnect the services in a different configuration, letting developers change the system's behavior at run time.

The Configurator service provides the current configuration of different parameters of the user interface under development. There are actually two versions of this service: one provides static values that are stored in a configuration file, whereas another allows developers to tune the configuration parameters manually. To the rest of the system, this makes no difference. This design allows static setups to be built and deployed which nevertheless can be easily reconfigured with appropriate configuration tools.

The Adaptive Service Dependency Architecture

A new architectural style, the adaptive service dependency architecture, consists of adaptively cooperating decentralized services on mobile and stationary computers.

This chapter describes the *adaptive service dependency architecture*, an architectural style that can be used in building ubiquitous augmented reality systems. The architecture's adaptivity is based on the dependencies between services, which are modeled as needs and abilities. These dependencies are influenced by a service's context, and in turn determine the context for other services.

The adaptive service dependency architectural style can be used a basis for a UAR framework, given implementations of the services and patterns for combining them.

5.1 Architectural Elements

The elements of the adaptive service dependency architectural style are shown in Figure 5.1 on the following page, forming a meta-model of UAR architectures.

The term meta-model signifies that this is a model of an architectural style that can be used to build concrete architectures. The architecture of a particular UAR system, such as ARCHIE, SHEEP or CAR, has a *model* which consists of services, which are modeled in the meta-model.

The following sections present these architectural elements in detail, and show examples of them in concrete UAR systems. They also show a notation for services, needs and abilities, based on UML 2.0 [46, 47].

5.2 Distributed Services and Service Managers

The most basic concept of the adaptive service dependency architecture is that of a service, which is managed by a service manager.

The functionality of a UAR system is decomposed into services. Each service has an implementation that runs on a mobile or stationary computer. The services of each computer are managed by a service manager running on that computer.

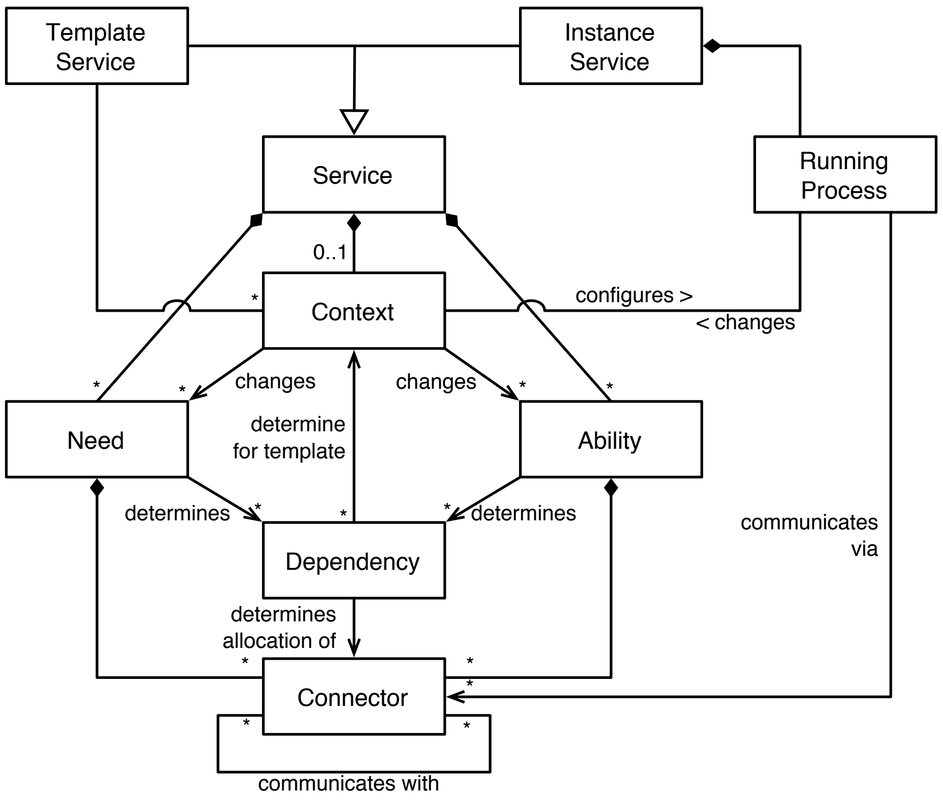


Figure 5.1: Simplified meta-model of the adaptive service dependency architecture.
The basic elements are services, which have needs and abilities. Dependencies of needs upon abilities are determined at run time and influenced by a service's context. A service may be a template for other services, operating in separate contexts determined by the template service's dependencies. An instance service's functionality is implemented by a running process, which communicates with other services via connectors. The connectors are allocated and connected based on the services' dependencies. A running instance service is configured by its context, and may change it, influencing other services.

5.2.1 Reflective Architecture

The adaptive service dependency architecture is *reflective*. Each service manager maintains *descriptions* of its services, and makes these descriptions available to other service managers. At the same time, the service manager makes the descriptions available to the services themselves. Thus, services may modify their own service descriptions, as well as those of other services (Figure 5.2).

This follows the *reflection* pattern [17], which allows a system to access its own system model, and by modifying that, to modify itself. Reflection is thus one of the main mechanisms for adaption in the adaptive service dependency architecture.

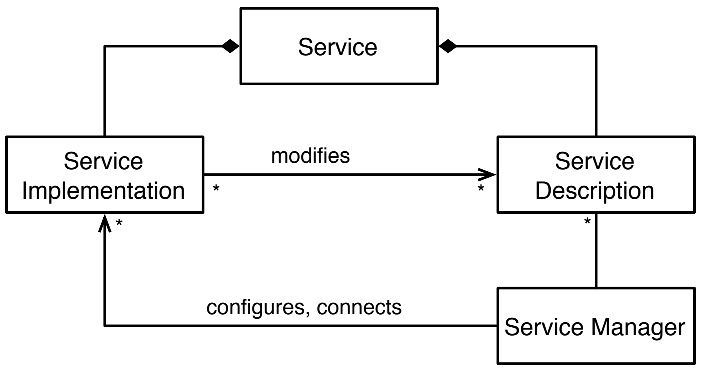


Figure 5.2: Service descriptions and service implementations.
A service manager maintains descriptions of running services. It uses them to configure and connect services. Services, however, can modify their own service descriptions, as well as the descriptions of other services. This creates a reflective system.

5.2.2 Services

Ubiquitous augmented reality systems use many interdependent distributed devices. Accordingly, the adaptive service dependency architecture uses the approach of distributed components. For reasons described below, the components are called *services*.

A service is a software component that performs a specific task and runs on a mobile or stationary computer. Its functionality must be clear enough to be understandable to end users. This lets users understand the overall software architecture and the decomposition of the system's functionality, without having to understand the services' implementation. Keeping the architecture understandable to the user is an important feature of the architecture, as users will combine services in different ways after they are developed and deployed. This follows the metaphor of a *tool*, as shown in Figure 5.3 on the following page—the user can chose the appropriate pieces of hardware for the task at hand.

The user may carry several small computers with him, each of which acts as a tool that can be used to perform a specific set of tasks. Each computer has one or more services running on it, which are associated with these tasks. Similarly, stationary computers have services running on them which are associated with stationary devices, or with the room that the computer is in. In [6], these computers are called *modules*. In [109], they are called *copliances*, for *cooperating appliances*. The following discussion uses the term *computer* for simplicity, or *network host* when discussing distributed algorithms.

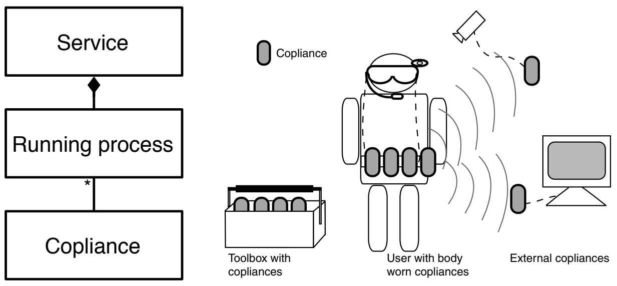


Figure 5.3: Services bundled into hardware copliances.
Left, meta-model; Right, example. A mechanic carries several *copliances*, small computers with a given functionality, following the metaphor of a tool. Additional copliances in the environment communicate wirelessly with the body-worn copliances and allow him to use external devices as well. (Figure adapted from [6].)

The simplest kind of service corresponds directly to a hardware device for input or output. For example, Video Grabber service processes the images from a video camera, and makes it available to the rest of the system. Or, a Viewer service renders three-dimensional images to a display. These services are easily understandable to users, as they are associated with a tangible hardware device.

Another simple type of service is responsible for storing information. Here, the hardware device the service is associated with is the computer it is running on. For example, a user's palmtop computer can have a Preference service that stores his personal preferences. Or, a stationary server in a room can store a description of the room's three-dimensional geometry, so that a mobile user's wearable system can correctly render three-dimensional models in the room. Data storage services are also fairly easy for the user to understand, as they correspond to physical storage mediums.

Other services are not associated with a particular input or output device, but perform a significant amount of processing. For example, an Optical Tracker service uses the video image from a video grabber service to determine the relative pose between the camera and objects in view. Or, a UI Controller service coordinates multi-modal input and output. These processing services can run on different computers; generally, they run on the same computer as other services they interact with. For example, the optical tracker service would generally run on the same computer as the video grabber service of the camera whose image it processes. These processing services are more difficult for the user to understand, although they are natural for software developers. Thus, special care should be taken when designing processing services that their functionality is clear.

Why “services”? At this architectural level, services might just as well be called “components”. I have chosen the name “service” for three reasons.

First, it implies loose coupling between services: one service does not depend on the other’s implementation. This is the same concept that is used in service-oriented architecture. The term “service” emphasizes that the most important feature of a software component is what it provides to other components.

Second, a mobile user with a wearable computer can take advantage of services that are provided by computers in the environment. For this, the wearable computer performs service discovery. Here, the term “service” implies transient use of a software component.

Third, the term “service” also implies adaptivity: a service adapts to its users. This concept has not been described as an integral part of the concept of services before.

5.2.3 Service Managers

Each computer that has services on it also contains a *service manager*, a middleware component that manages the services. Because of the changing availability of devices, there is no central software component, not even for managing the distributed services. Each service manager cooperates with the others in the network to set up connections between services, as shown in Figure 4.2 on page 50.

A key concept of the architecture is that of a *service description*. This is a description of the service, at a technical level, which is processed by the middleware, following the commonly used approach of component descriptions. A service description is stored persistently, e.g. in XML format, and contains all relevant information that the middleware needs to manage a service. This includes information to start and stop it, to load it into memory, to adapt it to different contexts, and to manage communication with other services. A service description is all the middleware knows about a service; the internal workings of the service are hidden. Thus, a service description contains all information about a service that is relevant to the architecture of the entire system.

Service descriptions are can be parameterized, so that the middleware can change services’ behavior at run time based on the users’ desires and on the availability of other services. This externalization of service descriptions lets the middleware connect and coordinate services, reducing effort for the service developer.

Together, the service managers manage the structure of a UAR system and adapt it to changing context. In Figure 5.4 on the following page, this role is described as a use case model. The simplest use case of the service managers is to manage a single service. The second is to manage communication between two services. The third, to adapt a service to its changing context (which is determined by other services), and the fourth and most complex use case, to manage the adaption of an entire graph of collaborating services.

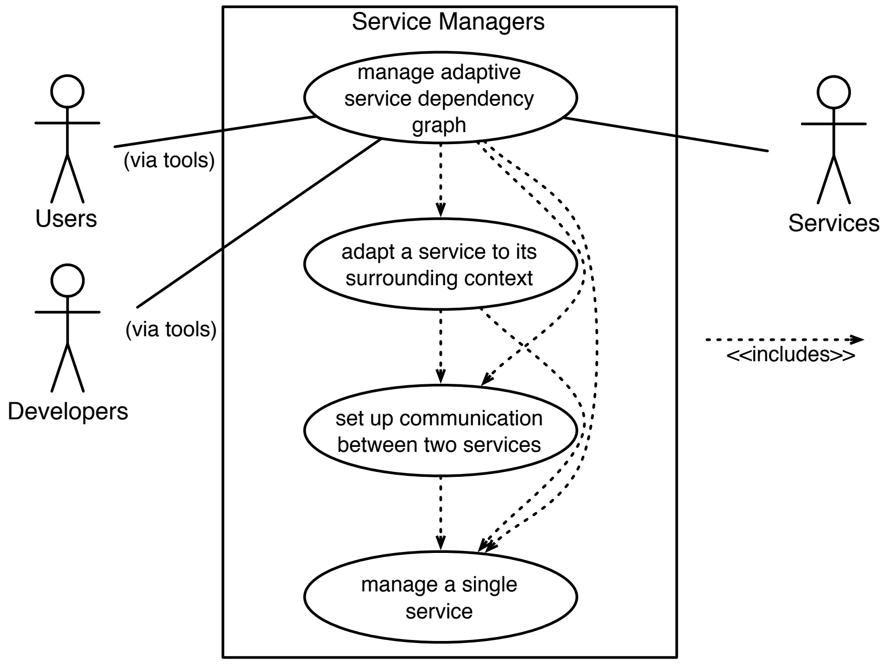

Service Managers
manage adaptive service dependency graph
adapt a service to its surrounding context
set up communication between two services
manage a single service
Users
(via tools)
Developers
(via tools)
Services
<<includes>>

Figure 5.4: High-level use cases of the service managers in the adaptive service dependency architecture.
The main use case of the service manages is to manage the adaptive graph of services and their dependencies. This, in turn, includes other use cases: adapting a service to its context, setting up communication between a pair of services, and managing individual services. These use cases build upon each other.

In Chapter 6, this use case model is used to describe the functionality of the service managers in detail. The next sections of this chapter follow the use case model, as well: Section 5.3 describes the management of two communicating services, Section 5.4 describes how a service is affected by its context of other services, and Section 5.5 describes an entire graph of services, forming an adaptive system.

In a concrete system built as an instance of the adaptive service dependency *architectural style*, the use cases of the service managers can be seen as *boundary conditions* [14]: they involve system startup, shutdown and reconfiguration. Indeed, in UAR systems, this type of boundary condition is very frequent; the system must adapt itself every time a user enters a room or picks up a new device.

5.3 Dependencies and Communication Between Services

Dependencies between services are modeled as *needs* and *abilities*; the dependencies are established dynamically and lead to communication between services.

5.3.1 Needs and Abilities

Given the interdependent distributed devices of ubiquitous augmented reality systems, services may depend on other services. For example, an optical tracker service needs a video image to process, which it may get from a video grabber service. This kind of dependency is modeled in the concept of *needs* and *abilities*.

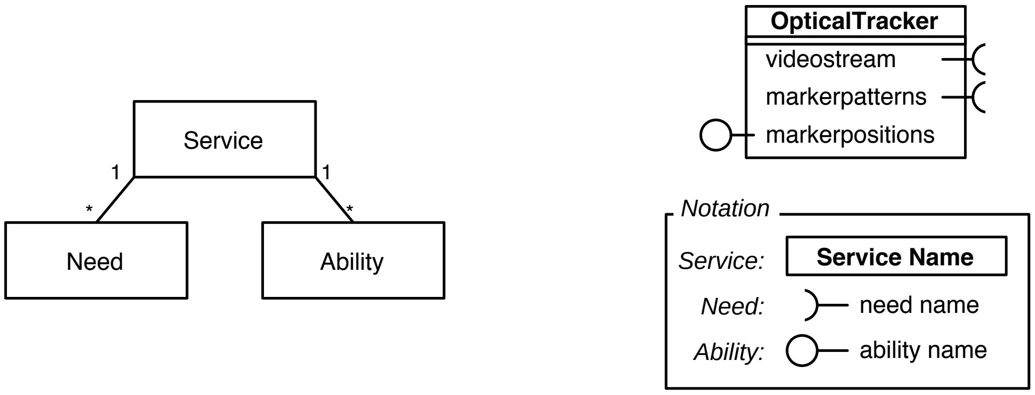


Figure 5.5: Services have needs and abilities.
Left, meta-model. Right, example service and notation.

An *ability* is the functionality provided by a service. For example, an Optical Tracker service has the ability to provide the relative position between a camera whose images it processes and objects in the scene. A Viewer service has the ability to display information to the user.

A *need* is the functionality that a service is dependent on. For example, an Optical Tracker service has the need for a video sequence to analyze, and it may also need descriptions of the optical properties of objects it is supposed to find in the scene. A Viewer service has the need for a geometrical description of the scene to display, and the viewpoint to render it from.

One service's ability may satisfy another's need. For example, the ability "provide relative position" of the Optical Tracker satisfies the need "viewpoint" of the Viewer service. This creates a dependency between the Viewer and the Optical Tracker service (Figure 5.6 on the next page).

When a service's need is satisfied by an ability of another service, it does not know anything about the other service's dependencies. Thus, although there is a transitive dependency relationship between services, transitive dependencies are not modeled as needs and abilities.

An ability can satisfy many needs at the same time; for example, the Video Grabber service could broadcast its image to several different optical trackers, or an external monitor. Similarly, a need can use several different abilities simultaneously; for example, the Viewer service may

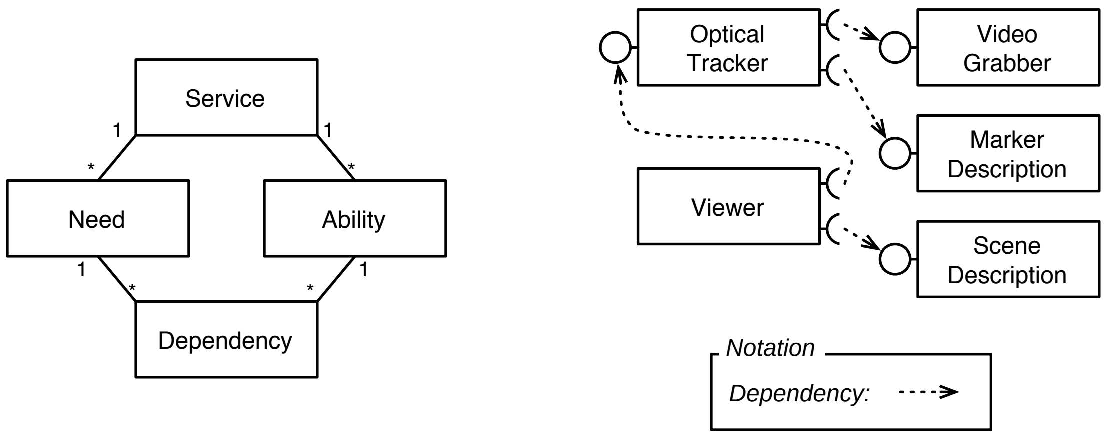


Figure 5.6: Dependencies between services, modeled as needs and abilities.
Left, meta-model. Right, example, with dependencies shown as dashed arrows, and notation. A Viewer service depends on a geometric scene description and on an Optical Tracker service. The optical tracker service depends on a video grabber service and the description of markers it should find in an image. This forms a simple dependency graph.

receive estimates of the viewpoint position from different trackers. A need may require a certain minimum number of abilities that it can use before its is satisfied.

For one service's ability to satisfy another's need, the services must communicate. Needs and abilities can thus be used to describe data flow architectures. The main direction of data flow is from the ability to the need. For example, the tracking data flows from the tracker to the viewer service. Thus, the two relations "depends on" and "sends data to" between services are inverse, as shown in Figure 5.7 on the facing page. This is not a strict rule, however; it is quite possible for data to flow in two directions when one service depends upon another. For example, if the Viewer service retrieves the the description of which scene to display from the Scene Description service using a request-response protocol, data flows in two directions. However, the main direction of data flow is from the Scene Description to the Viewer.

Sometimes, it is not immediately clear to developers which of two services depends on the other. One guideline can be the data flow, as described above. Another good guideline is the "uses" relation as described in [21]: there, a component *uses* a second component when its *correctness* depends on the result of the second component. Of course, this means that the the "correctness" of a service needs to be well-defined. It is also quite possible for two services to depend upon each other; each then has a need that can be satisfied by an ability of the other. These two dependency relations may imply different types of data flow, however.

There are two additional types of dependencies that are worth considering, which do not imply inter-service communication. First, a service may depend on a piece of hardware; for example, the video grabber service depends on a connected camera, which is not a service. It

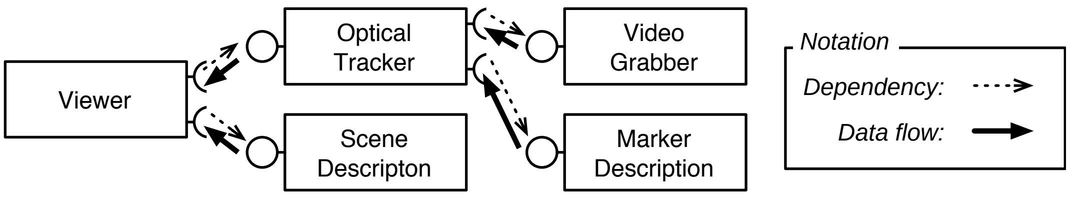


Figure 5.7: Dependencies versus data flow between services.
The example of Figure 5.6 is shown again, with data flow indicated, along with dependencies. Here, the data flow relation is the inverse of the dependency relation.

is still possible to model this dependency as a need of the video grabber service. However, this is only useful if there is another "camera" service which serves as a placeholder for the camera, and provides an appropriate ability.

Second, a service's ability may satisfy a user. For example, the 3D rendering service can show a three-dimensional scene to the user; this may be exactly what he wants. Again, it is possible to model this as an ability of the rendering service, and to create a placeholder "user" service that has a need for visual input. Both cases are shown in Figure 5.8.

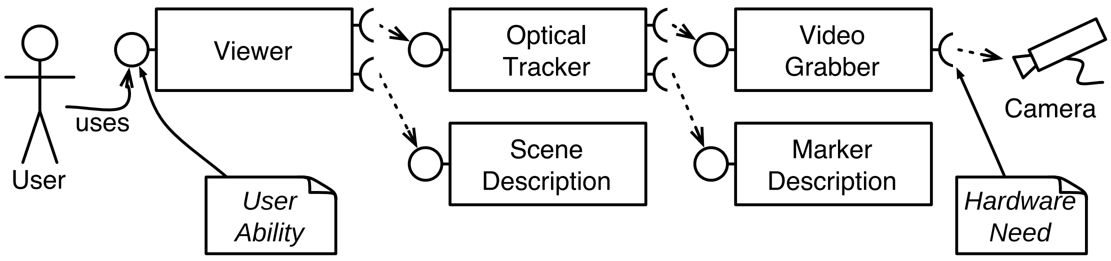


Figure 5.8: Non-communicating needs and abilities.
A service may depend on a hardware device; similarly, a user may use a service's ability. Such special needs and abilities are called *hardware needs* and *user abilities*.

5.3.2 Types for Needs and Abilities

Services are loosely coupled. This means that when a service has a need that depends on an ability of another service, it makes no assumptions about the identity or implementation of that other service. Instead, it has only incomplete knowledge about the other service, just enough to ensure that the ability can, in fact, satisfy the need.

For this, within each need description, there is an abstract specification of required features of potential abilities; and within each ability description, there is information that limits the potential needs that can use that ability. At system run time, the middleware uses the need and ability descriptions to perform service discovery. This allows matching needs and abilities to find each other, so that the services can cooperate. Service discovery takes place both within each computer on the network, allowing the services on a single computer to find each other, and also between network hosts, allowing different computers to cooperate.

Using service discovery allows the system to adapt to devices' changing availability. In the team action scenario, one user's wearable computer may have sophisticated sensors, which are made available to other users via a *sensor* service. A second user's wearable computer, without those sensors, has a *danger alert* service, which has a need for sensor information. When the two users are within range of their ad hoc wireless network, the services on the mobile systems can discover each other, and they can cooperate. When they are out of range, the danger alert service must rely on other, less sophisticated sensors, on the user's own computer.

Another example shows that even with wireless network coverage, services may go out of range. Assume a user is wearing a head-mounted display showing an image from a 3D rendering service running on his wearable computer. The rendering service has a need for tracking information, which is satisfied by an ability of a stationary tracker in the room. When the user leaves the room, he is out of the range of the room's tracker. However, his rendering service can use a (perhaps less accurate) tracking system in the hallway.

Needs and abilities both have *types*. A need can only use abilities that have the same type. For example, in Figure 5.9, the renderer service has a need of type PoseData, which can be satisfied by the optical tracker.

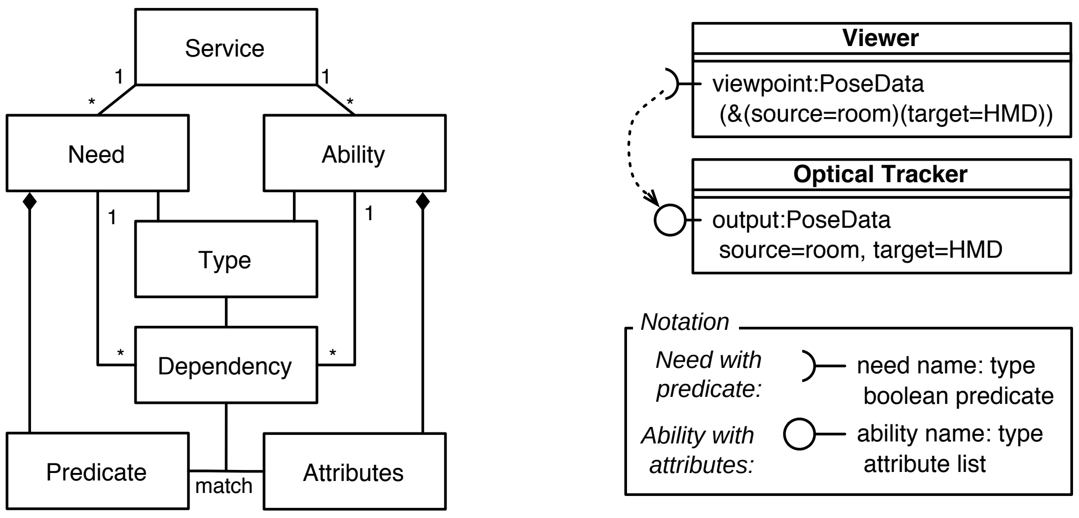


Figure 5.9: Types, attributes and predicates.
Left, meta-model. Right, example and notation. The renderer service's viewpoint need and the tracker service's output ability have the same type, PoseData. Furthermore, the need's predicate evaluates to true, given the ability's source and target attributes. This allows the need to use the ability, creating a possible dependency.

Choosing an appropriate type name for a need or ability is an important architectural decision. All developers have to agree upon the type names for services to interoperate. Users will not normally see the type names, but if they do, they should make sense. As a guideline, a type name should be chosen that fits naturally with the term *need* or *ability* in spoken language. For example, "the renderer needs pose data," and "the tracker has the ability to provide pose data". In this case, the Data in the type name also implies that information is to be sent.

If services written by different parties are supposed to interoperate, there must be a standardized catalog of need and ability types, as well as standard interfaces and standard data formats. This is also needed if the architecture is to be used to build a general UAR software framework, as described in Section 5.6. Standardization involves many difficult issues, such as detailed specifications, incomplete implementations, versioning, vendor-specific extensions, compatibility tests, and so on. These are all issues which naturally arise in any architecture that combines software from different vendors, e.g. in any service-oriented architecture.

Needs and abilities may have names, as well; in the example of Figure 5.9 on the facing page, the renderer service's viewpoint need is shown. However, this name is for internal use by the service only, it is not used in service discovery. Only type, attributes, and predicates are relevant for service discovery. Need and ability names are still important, as they can be useful in understanding, monitoring and debugging the system.

5.3.3 Attributes and Predicates

Simple types are not enough to distinguish between the many abilities running on different devices. For a more detailed description, an ability has *attributes*. In the example in Figure 5.9, the optical tracker service's output ability has two attributes: source, with the value Room, and target, with the value HMD. This signifies that the output ability provides data describing the relative pose of a head-mounted display to the room's coordinate system.

Thus, attributes can be used to distinguish between different services, or between different abilities of one service. For example, a tracker might currently be tracking the user's head and hand. It would then have one ability with the attributes source=Room and target=HMD, and one with source=Room and target=Hand.

Similarly, needs have *predicates*, which are boolean expressions over attributes. In the example, the renderer service's viewpoint need has the predicate (&(source=Room)(target=HMD)), which evaluates to true for the optical tracker's output ability. Predicates, like types, limit the abilities that can satisfy a need. Only when a predicate evaluates to true over the attributes of an ability, can the need use that ability.

As for types, choosing appropriate attribute names is important. Attribute names should be understandable to users, so they can reconfigure the system easily. Furthermore, attribute names and values must be standardized, just as need and ability types must be.

Attributes may have types, not to be confused with ability types. The example above uses a simple character string type, which serves essentially as a unique identifier. These are not unique enough for a real application; Room would be a singularly bad choice for the value of target. A hierarchical identifier such as de.tum.in.01.07.059 might be more suitable, indicating a room number in an organization identified by it's internet domain name.

Many other types are possible, ranging from basic (boolean, integer, floating point, character, string) to composite types (arrays, structures, sets). In fact, composite data types for the domain of UAR can be used here, just as such data is exchanged between services. For example, the attribute pose, of type PoseData, indicates where the computer is located that a service is running on. Different implementations of the adaptive service dependency architecture may choose different allowable types for attributes. For example, the current implementation within DWARF only allows basic types for attributes. Again, some form of standardization is required for attribute types and values.

Another implementation decision is whether attributes may have only one value, or several values simultaneously. For example, a video grabber service's video output ability might have a resolution attribute with several different values, one for each supported resolution. On the other hand, it might simply have a different ability for each resolution, each with a different value for its resolution attribute.

As a convenience for developers, services, not only abilities, may also have attributes. This simply means that each ability of the service "inherits" the attribute and its value. For example, an optical tracker that is tracking several objects relative to itself can have the attribute source=Camera, which is valid for all its abilities; each ability then has an additional attribute such as target=Head or target=Hand.

An important implementation decision in the adaptive service dependency architecture is the expressive power of the syntax. For example, predicates can include wild cards, such as (room=de.tum.in.01.07.*) to indicate any room within a certain part of a building; or regular expressions. Also, if implementation supports complex attribute types, it must allow predicates to specify expressions on these types. For example, (pose1-pose2<5) could indicate that two objects are within 5 meters of one another. The syntax used for boolean expressions in this chapter is a prefix notation with strict use of parentheses, from the *Lightweight Directory Access Protocol (LDAP)*. This choice is arbitrary; any similarly expressive syntax could be used here.

Attributes and predicates of a service's needs and abilities do not have to remain static. While it is running, a service may change its attributes, or those of its abilities. For example, an optical tracker service may change its accuracy attribute when light conditions change.

A service's attributes may also be changed by another service. For example, all the services on a wearable computer may change their room attribute when the user enters a new room. This assumes that there is a tracking service somewhere that can detect that the user has entered a new room, and which room that is.

Similarly, a service may change predicates of its needs at run time. For example, a service that only wishes to be connected to other services in the same room should update all predicates to (room=newroom) when the mobile computer it is running on enters newroom. This technique addresses the appropriate choice of devices in a changing context.

A services' attributes may change quite frequently. For example, the accuracy attribute, above, changes with every measurement, which can be tens or hundreds of times per second. Re-evaluating all predicates depending on these attributes at that rate can lead to a performance problem, as described in Section 6.6.

However, in most cases, attribute changes do not actually have to be evaluated that quickly. It may be quite enough for a tracker to update its accuracy attribute only once per second, or when it changes by more than a certain value. This way, the changed attribute may be used for other services' service discovery, but the service discovery does not actually have to be performed hundreds of times per second.

Furthermore, although the accuracy of a measurement is actually represented as a complex type (e.g. a covariance matrix), it can be useful to map it to a more simple one (e.g. a real number), allowing developers of other services to specify simpler predicates. These two optimizations follow the ideas of context refinement and latency layering.

5.3.4 Penalties

Both ability types and predicates can narrow down the possible abilities for a given need. However, in many cases, a number of alternatives still remains to choose from. To differentiate between possible abilities, needs may specify *penalties* in addition to predicates. These specify a sorting order for abilities based on quality of service.

Similarly to a predicate, a penalty is an expression over a set of attributes. However, it does not evaluate to a boolean value, but to a positive real number. The lower the number, the better the match of the corresponding ability. For example, in Figure 5.10 on the next page, the output ability of the optical tracker service has an attribute confidence, a numeric value between 0 and 1, indicating the probability that the tracked object has been correctly identified. Also, it has an attribute frequency, indicating the number of measurements per second. Some of these attributes are constant (e.g. frequency), others change over time (e.g. confidence), as they depend on individual measurements. Other attributes are possible as well, e.g. lag, indicating the average age of a measurement in seconds. The renderer service's viewpoint need may now specify the penalty (10/frequency) + (1/confidence)). If two different trackers are available, this would favor the one with the higher frequency and the greater confidence.

The weighting of these different factors is very important in determining which service to choose. Is a higher update rate more important, or a greater confidence? Such a decision is highly dependent on the particular application. As for predicates, the syntax here is arbitrary; any other would do as well. An important implementation decision, however, is the expressive power. A more advanced syntax could specify complex arithmetic operations, and appropriate conversions of complex types (such as 6D PoseData)to numeric values).

5.3.5 Distances

The penalty that one service assigns to another is not always absolute, depending only on the attributes of the ability. Rather, it may depend on the needing service's attributes as well. This is generally the case when two services should be "close" to one another, in some dimension.

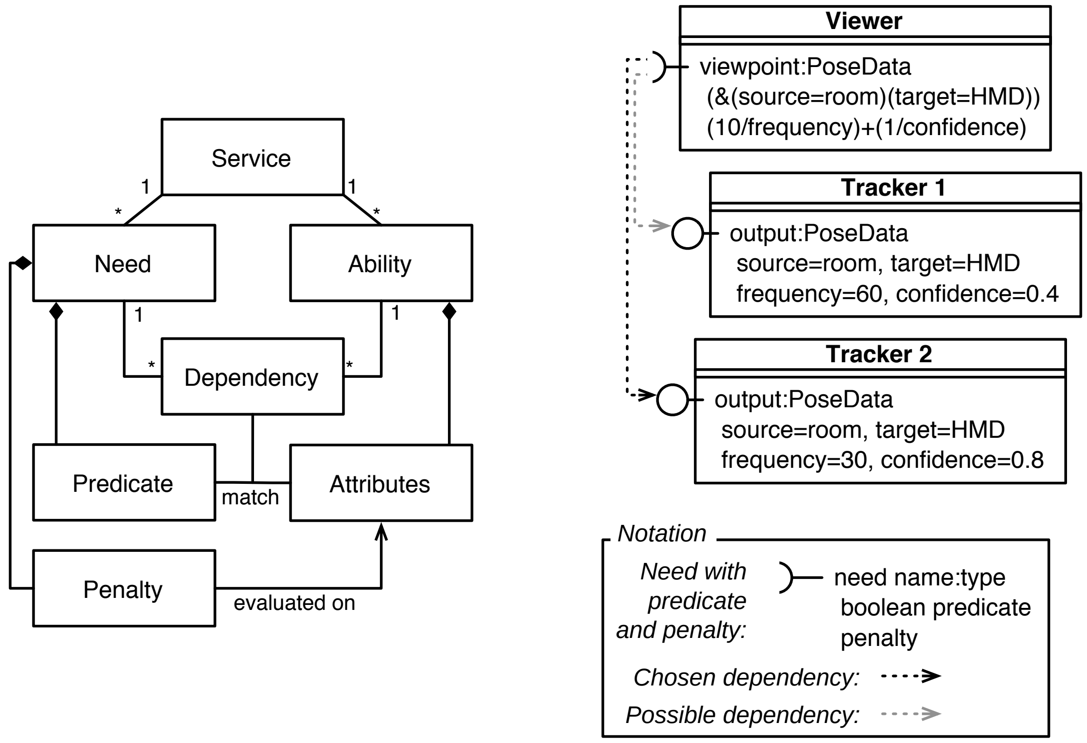


Figure 5.10: Penalties over attributes.
Left, meta-model; right, example and notation. A Viewer service's need specifies a penalty over attributes, indicating a quality-of-service preference. This leads to the choice of Tracker 2 (penalty 1.6) over Tracker 1 (penalty 2.7).

For example, consider a UI Controller service (Section 4.6.2) that needs to give acoustic feedback when the user performs an input gesture, as in the SHEEP system. For this, it has a need of type SoundOutput, which a stationary *sound player* service, attached to a loudspeaker in the corner of the room, has an ability for. Now, there may be a different sound player service in each room; or there may even be several sound player services within the same room. Which one should the user interface controller choose? One obvious solution is the one which is physically closest to the user at the moment.

To model this kind of preference, a penalty may include *distances*. In the example, the user interface controller may specify a penalty expression (d(room)). For each ability of a sound player service, this evaluates to the distance between its room attribute and the user interface controller's own room attribute. Similarly, the need can specify the penalty (d(position)), or a more complex expression such as ((3*d(room))+d(position)+(1/volume)). This last expression prefers sound player services that are in the same room, near the user, and loud.

To build expressions using distances, basic *distance functions* are needed for each attribute type. For two values of type T with a value range of T, the distance function d_T calculates a positive integer:

$$d_T : T \times T \to \mathbb{R}_0^+$$

Thus, if the attribute position is of type *Position*, d(position) is calculated as $d_{Position}(p_a, p_n)$ for each ability's position attribute p_a and the needing service's own position attribute p_n.

Some obvious distance functions are:

$$d_{Bool}(a,b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$$

$$d_{Position}\left(\left(\begin{smallmatrix} x_1 \\ y_1 \\ z_1 \end{smallmatrix}\right), \left(\begin{smallmatrix} x_2 \\ y_2 \\ z_2 \end{smallmatrix}\right) \right) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Other distance functions might be based on heuristics. For example, the following function computes a distance between two strings, assuming that strings are similar if they share a common prefix. This might be appropriate for dotted-numeric IP addresses, or for room numbering schemes such as building/floor/hallway/room. However, this particular distance function would obviously fail for internet domain name entries.

$$d_{String}(s,t) = \frac{1 + max(|s|, |t|)}{1 + |\text{longest common prefix}(s,t)|}$$

For distance functions to be useful, the services must have appropriate attributes. The user interface controller example assumed that the position attribute of the user interface controller is that of the wearable computer it is running on. However, if the user interface controller is running on a stationary server in the room on behalf of a mobile user, its position would remain constant, being that of the server. In this case, the user interface controller must have an additional userposition attribute which is updated, by an external tracker, to the user's position; and it must specify a penalty of (d(position,userposition)) rather than simply (d(position)).

Distance expressions are obviously useful in predicates, as well as in penalties. For example, setting a predicate of (d(room)=0) would limit communication partners to those in the same room. However, this makes the mechanisms for service discovery more complex.

5.3.6 Connectors, Protocols and Communication Resources

Communication always follows the dependency relationship defined by needs and abilities, as shown in Figure 5.7 on page 73. Only when one service's ability can satisfy another service's need, can the services communicate.

It is worth pointing out that this is an important architectural restriction. If there is no dependency relationship between two services, they cannot communicate. Many architectures do not have this kind of restriction. For example, if components communicate via a tuple space, there are no explicit dependencies between them.

At first sight, this restriction even appears to violate the principle of loose coupling—why should one service have to know which other service it is communicating with? Here, it is

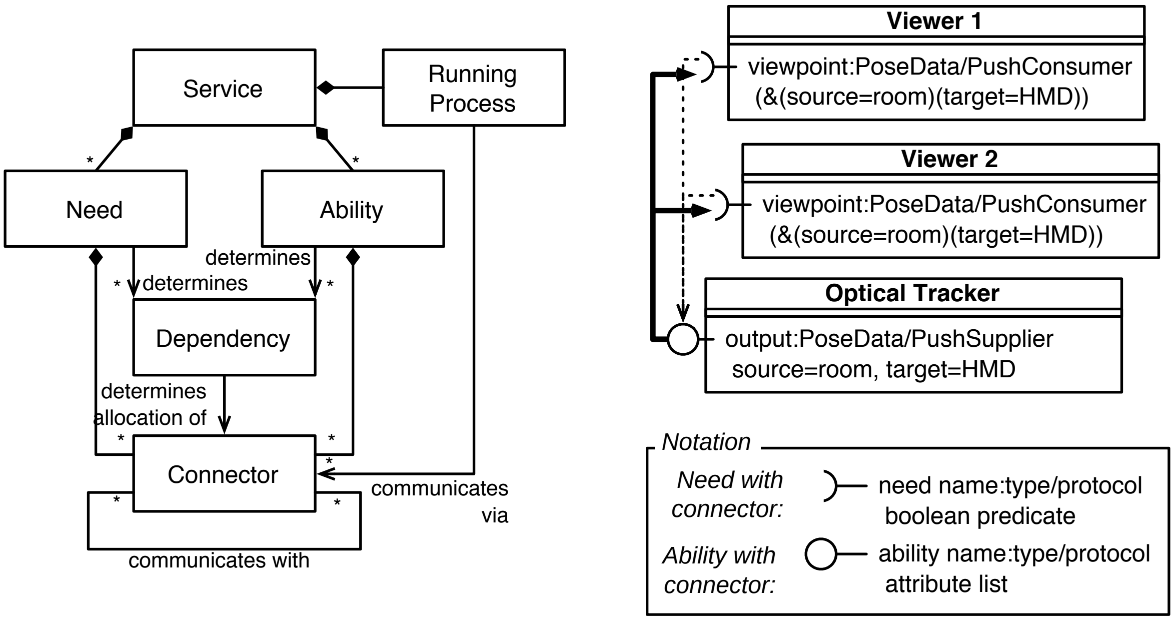

Service
Running Process
Need
Ability
determines
determines
Dependency
determines allocation of
Connector
communicates via
communicates with
*
Viewer 1
viewpoint:PoseData/PushConsumer (&(source=room)(target=HMD))
Viewer 2
viewpoint:PoseData/PushConsumer (&(source=room)(target=HMD))
Optical Tracker
output:PoseData/PushSupplier source=room, target=HMD
Notation
Need with connector:
need name:type/protocol boolean predicate
Ability with connector:
ability name:type/protocol attribute list

Figure 5.11: Connectors and communication between services.
Left, meta-model; right, example and notation. Connectors manage communication between services. For each connection, there are two connectors involved: one for the need and one for the ability. Connectors are typed by protocol. Here, the Optical Tracker sends data using a PushSupplier connector, and the two Viewer services receive data using the PushConsumer connector. The data flow follows the dependencies between needs and ability.

important to note that the architecture achieves loose coupling using a different mechanism. Dependencies between needs and abilities are based on types, attributes and predicates. The communication relationship then simply builds on the loosely coupled dependencies.

An advantage of this restriction is that service developers are forced to model the dependencies of their services upon other services, if they wish to communicate with them. This ensures that in a running system, it is always possible to determine which services are communicating with which, and which services depend upon each other. Of course, this advantage disappears if developers use transitive dependencies. If two services depend on a common tuple space service, and communicate with it, they may communicate with each other, indirectly, via the tuple space. This then creates a hidden dependency, which developers must consider.

To support the many communication types in a UAR system, needs and abilities communicate via connectors. A *connector* is part of the middleware and encapsulates communication resources and protocols. For example, in Figure 5.11, the tracker and renderer services communicate via push-style *events*. The tracker supplies events, and the renderer consumes events. The events are transported between the services via a notification channel.

In setting up communication, two connectors are involved: one for the need, and one for the ability. In Figure 5.11, there is a PushSupplier connector for the Optical Tracker's output ability, and a PushConsumer connector for the Viewer's viewpoint need. The two connectors cooperate to set up communication.

Note that the term connector is used differently in other work. In [21], the *components and connectors* architectural view type uses only one connector between components, not two. There, connectors only encapsulate communication; they do not set it up. In the adaptive service dependency architecture, connectors are involved in actively setting up the communication. Since the two services are independent, each has a dedicated connector.

The connectors for needs and abilities have connectors are specified in the service descriptions. A connector is specified by its *protocol*, such as PushSupplier. Only connectors of matching protocols can communicate. For example, PushSupplier and PushConsumer match, but ShmemWriter and PushSupplier do not.

Connectors for some communication protocols can handle multiplexing: one connector can set up communication with several other connectors. In Figure 5.11 on the facing page, the tracker communicates with two renderers. This sets up many individual one-to-one communication relationships. However, optimization is possible here; the PushSupplier connector can set up a multicast channel to send events to several consumers on different network hosts simultaneously.

A need or ability can have several connectors. For example, the optical tracker could receive video data by shared memory or a streaming protocol, or even by both at the same time (if it were to process several video streams at once).

Different connectors are preferable in different circumstances. To indicate this, connector descriptions (for a need) can specify penalties and predicates, just as for the need itself. A connector's penalty is added to that of the need, and the connectors predicate is combined with that of the need using a logical *and* operator. For example, if the video grabber is on the same host as the optical tracker, shared memory communication is preferable over a streaming protocol. To model this, the a ShmemReader connector adds a penalty of 1, whereas the RSTP-Receiver adds a penalty of 2. The ShmemReader connector also adds (d(host)=0), to indicate that the partner must be on the same host.

Some connectors have quality-of-service parameters that can be adjusted. These are specified as attributes within the connector description. For example, the PushConsumer connector has a queue attribute, which indicates whether incoming events should be buffered if the service cannot process events as quickly as they arrive.

A special type of connector allows conversion between a service's attributes and data exchanged by other communication protocols such as events. These *attribute change connectors* are handled entirely within the middleware. For example, consider the user interface controller service described above. Its userpose attribute should reflect the current position of the user, as measured by a tracking service. For this, the user interface controller has a PoseData need with a AttributePushConsumer connector. This connector receives pose data events from a tracker service and sets the service's userpose attribute accordingly. Thus, the user interface controller service can take advantage of the user's position for service discovery, without having to communicate with tracker services directly.

5.4 Context Adaption by Instantiating Service Templates

A single service can adapt itself to its *context*, which consists of the available communication partners; beyond this, a service can exist in multiple contexts at the same time. This uses the abstraction of *templates*.

5.4.1 Templates

While some services exist only once per host, there are others that can exist multiple times. These different kinds of services are called *instance* and *template* services. This is similar (although not identical) to the distinction between classes and objects in object-oriented programming languages.

Instance services are services that exist only once per computer. Examples for these are device services, which correspond directly to a particular piece of hardware. For example, there is only one video grabber service per physical camera. Others are data storage services, which store a particular set of information.

Template services are services that can have an arbitrary number of instances running simultaneously on the same computer. These are generally processing services, which take input from other services and generate appropriate output. Although there may be several instances of such a template service, the developer only has to create a single service description.

A simple example for a template service is the pose inference service shown in Figure 5.12 on the next page. Assume a setup of two "cascaded" trackers: tracker A computes the pose of B relative to A; and tracker B computes the pose of C relative to B. This is a typical setup from the Ubitrack project. The pose inference service now receives input from both trackers and combines it, computing the pose of C relative to A.

Based on the available communication partners, the middleware instantiates template services and creates distinct instance services. These instances are then distinguished by their attributes. Figure 5.13 on page 84 shows how different instances of the pose inference service can combine data from different trackers.

Similarly to template services, services can have *template abilities*. For example, an optical tracker can have a template ability for sending tracking data. For each marker description that the tracker's marker need finds, the tracker can track a new object in the scene. For each tracked object, the middleware will then instantiate a new ability.

Template services may have template abilities as well. In this case, the template service must be instantiated first; then, the template abilities of the resulting instance services are instantiated. Figure 5.14 on page 85 shows an example.

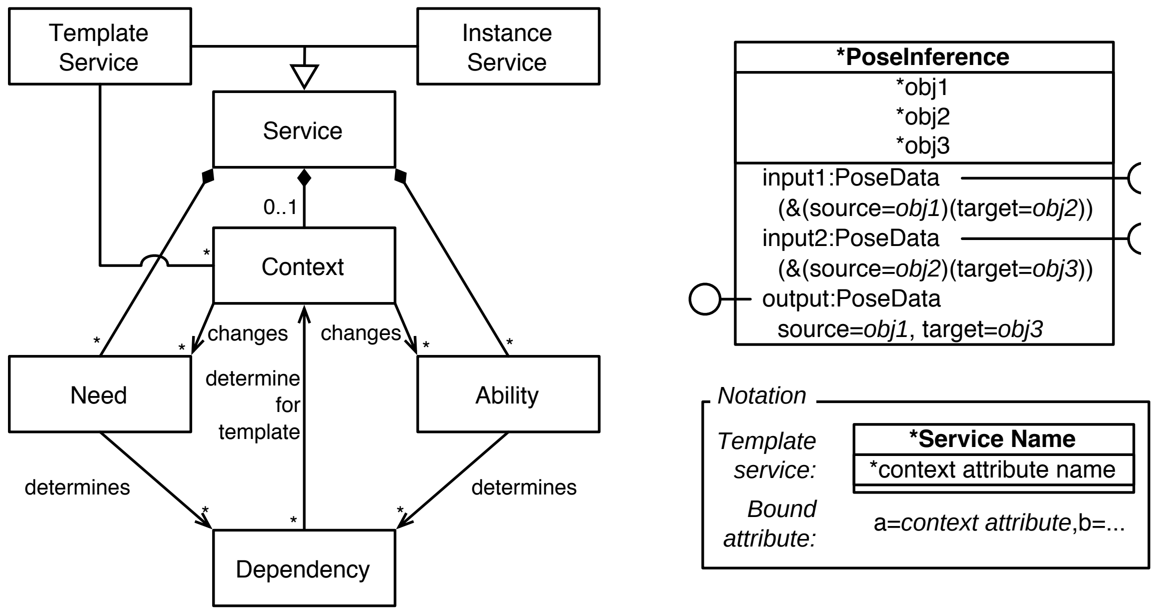


Figure 5.12: A template service that processes data.
Left, meta-model; right, example and notation. The pose inference service combines the input from two cascaded trackers. The objects obj1 and obj3 between which it computes the relative pose depends on the trackers it is connected to.

5.4.2 Contexts

Instantiated template services, needs and abilities are distinguished by their *context attributes*. Together, these attributes form the *context* of an instantiated service. A service's context is defined by its communication partners, i.e. by other services.

In Figure 5.12, the pose inference service has the context attributes obj1, obj2 and obj3. When the template service is instantiated, these context attributes are bound to specific values in the attributes or predicates of other services. For example, in Figure 5.13 on the next page, the context attributes obj1, obj2 and obj3 of the PoseInferenceADH service have been bound to the values A, D and H. These values are propagated into the needs' predicates and the abilities' attributes. In the input1 need, the predicate has become (&(source=A)(target=D)), matching the source and target attributes of the TrackerA service's outputAD ability. Similarly, the output ability now has the attributes source=A and target=H.

The rules for these bindings are specified in the template service's service description. In instantiating a template service, the middleware must ensure consistency of bound attributes. For example, the value for obj2 must match both the target attribute of input1's communication partner and the source attribute of input2's.

The middleware takes advantage of the service discovery mechanisms to determine the contexts in which to instantiate services. Template services can depend on instance services and

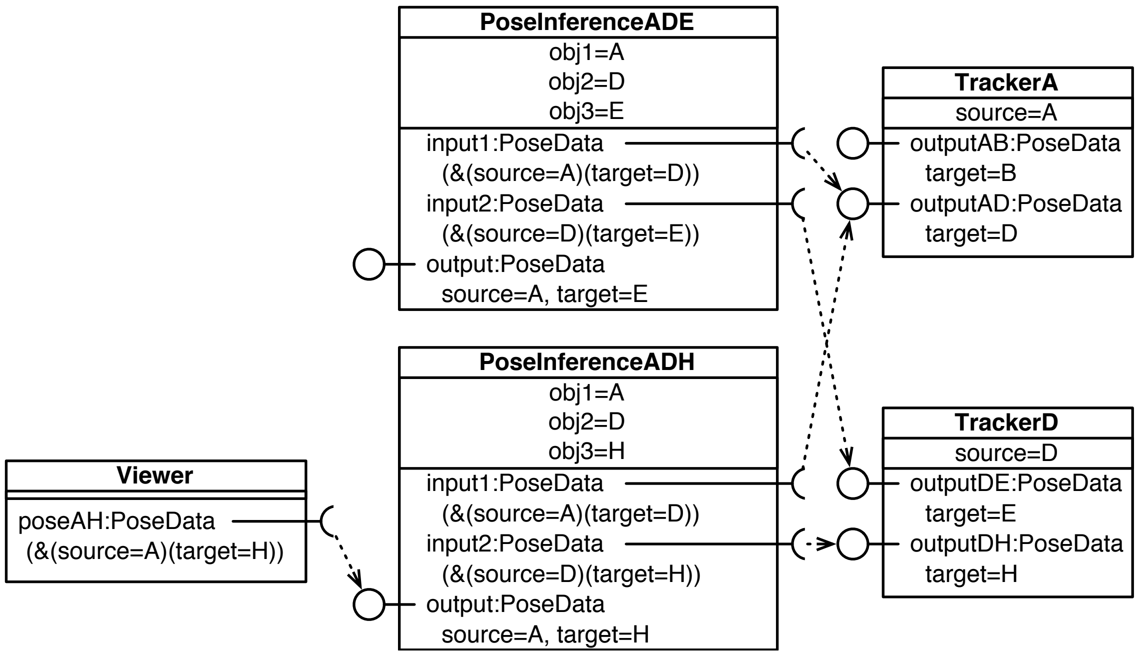


Figure 5.13: A template service instantiated in different contexts.
The pose inference service of Figure 5.12 is instantiated for different combinations of trackers.

vice versa. As shown in Figure 5.15 on page 86, a template service's needs and abilities match a large number of template services. Based on those service's attributes and predicates, the middleware can create contexts for instantiation.

An important application of templates and contexts is to provide services with the appropriate configuration information. For example, a stationary optical tracker must be reconfigured when unknown optical markers enter the room. Configuration data is stored in separate configuration services. These are special data storage services that provide abilities to the services that need to be configured. For example, a marker description service on a user's wearable computer tells the stationary optical tracking system what the optical marker on the user's head looks like. The different configurations are distinguished using a well-defined set of attributes, such as room, user or application. This allows template services (and template abilities) to be instantiated in different contexts depending on the different configurations that are available. When a user enters a room with his preferences on a wearable computer, new, personalized, instances of application services become available in the room, as described in [78].

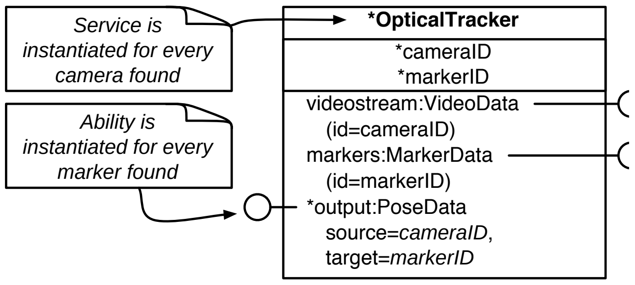

Service is instantiated for every camera found
Ability is instantiated for every marker found
*OpticalTracker
*cameraID
*markerID
videostream:VideoData (id=cameraID)
markers:MarkerData (id=markerID)
*output:PoseData source=cameraID, target=markerID

Figure 5.14: A template service with a template ability.
The optical tracker service is a template; a new instance service is created for each video stream. Then, for each marker description found, a new ability is instantiated.

5.5 Adaptive Service Dependency Graphs

A graph of services and their dependencies can adapt to changing context by combining the mechanisms described in the previous sections.

5.5.1 Instance Graph

Based on their needs and abilities, instance services form a graph, called the *instance graph*. the nodes are the services themselves, and the edges are possible matches, from need to ability. An example is shown in Figure 5.16 on page 87.

In a ubiquitous augmented reality system, the instance graph can become quite large, both in terms of the number of nodes and the number of edges. However, not all services must be running at all times, and not all possible matches between needs and abilities must actually lead to communication relationships. Thus, there are two subgraphs of the instance graph, which the middleware must compute and maintain.

For a given need, there are many abilities of other services that can satisfy it, based on type and predicate. However, usually only one is actually necessary, and this is selected based on the penalty specification of the need. This forms a subgraph of the instance graph, the *selected instance subgraph*. It contains the same nodes as the instance graph, but fewer edges.

Similarly, not all services must actually be running at all times. For example, a tracker service does not have to be running if there is no other service to consume its output. This forms another subgraph, the *running instance subgraph*. This is a subgraph of the selected instance subgraph, containing only those nodes of services that are actually running. Of course, if one service of the selected instance subgraph is running, and thus a node within the running instance subgraph, all other nodes that it has edges pointing to must be within the running subgraph as well. The running instance subgraph can be formed recursively: When an ability of one service is requested, the service starts, and its need requests the abilities of other services.

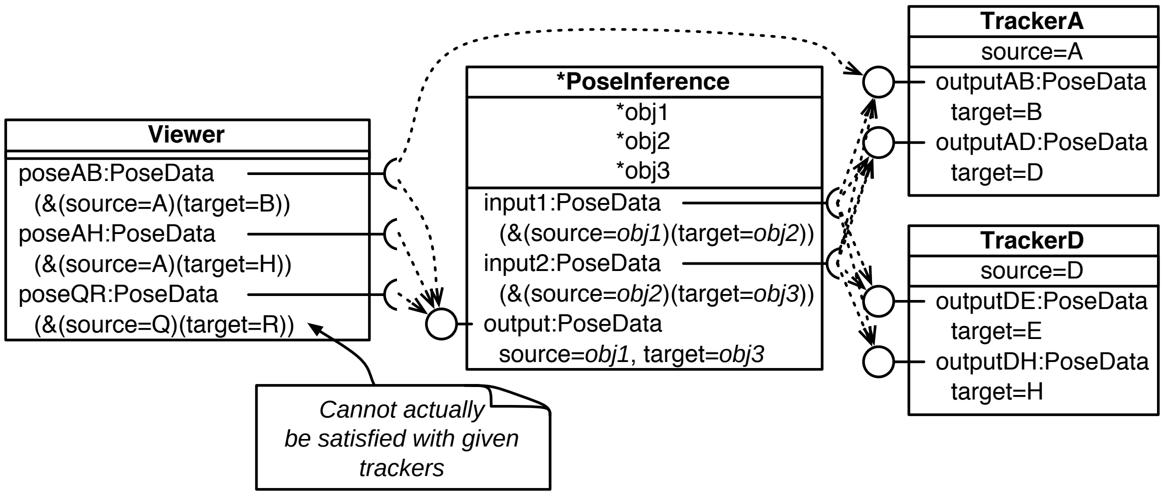


Figure 5.15: Contexts are determined from service discovery.
The template service's need can use abilities of several other services. Similarly, the template service's ability can be used by needs of several other services. Based on these, the middleware creates contexts to instantiate the template.

For the recursive formation of the running instance subgraph, at least one initial running service is required—otherwise no services will start at all, and the running instance subgraph will remain empty. This first service can either be started manually by the user, or automatically by the system. Starting a service manually can be quite simple—services are bundled into simple hardware modules like tools, and starting a service may just mean turning on a piece of hardware.

For static applications, such as laboratory demonstrations, it may even be desirable to store a complete running instance subgraph, and restore it on command, without any adaptivity at all. This lets developers try out and compare different static setups. For this, the middleware simply needs the descriptions of all services within the running instance subgraph, and the connections between them.

In an instance graph, there are no template services, or services with template needs or abilities. Thus, the number of nodes in the instance graph remains constant, unless new services are added by the discovery of new hardware devices. Also, all attributes and predicates are fully bound; there are no references to unbound context attributes.

5.5.2 Template-Instance Graph

The template services introduced in Section 5.4 form a graph, as well, with services as nodes and potential dependencies as edges. However, this *template graph*, containing only template services, is not very useful by itself. Template services cannot run on their own; they must be

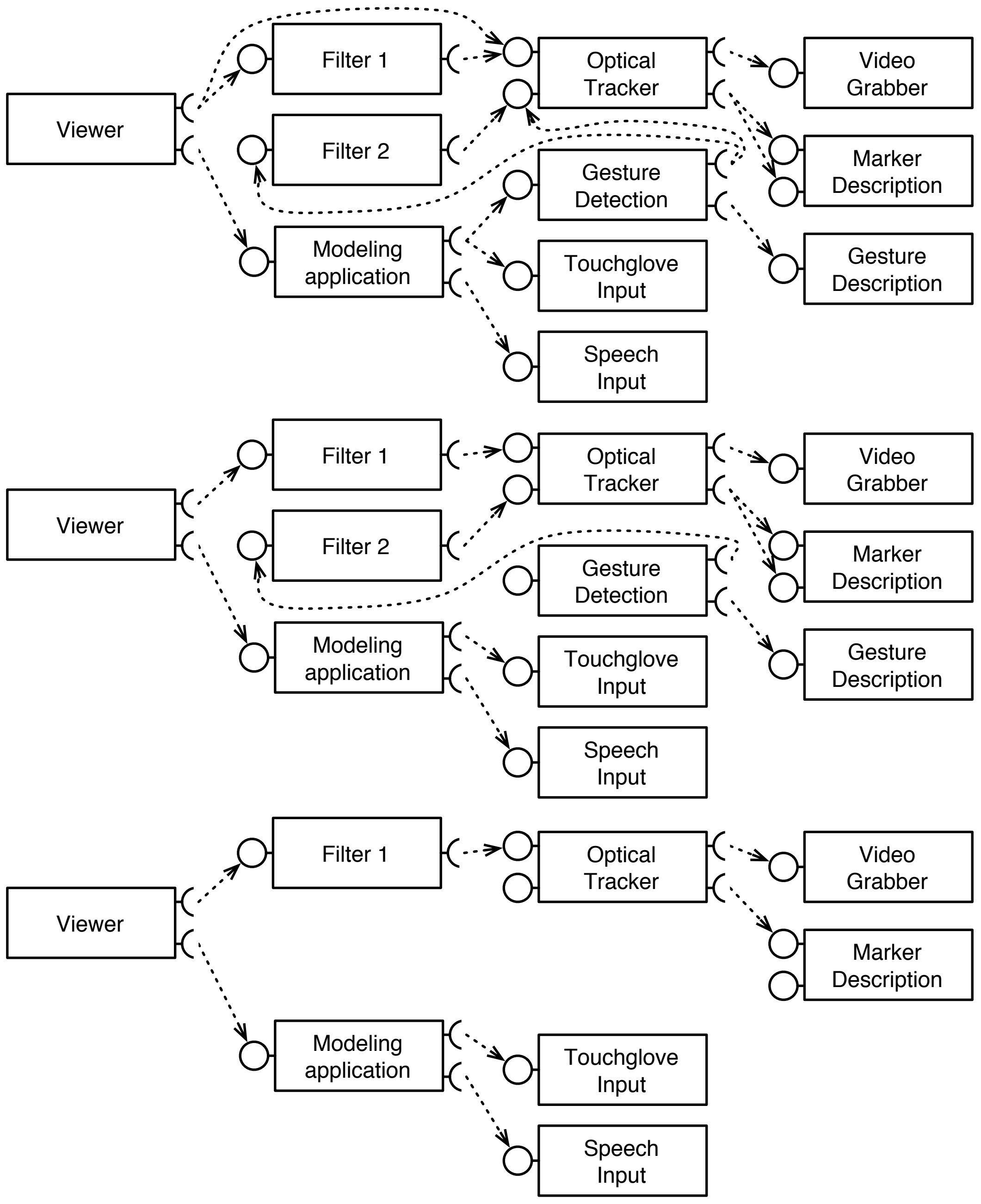


Figure 5.16: Instance graph of services, selected and running subgraphs.
The example shows a multi-modal UAR modeling application; the user can interact with the modeling system using gestures, a touchpad, or speech. Top, instance graph with all services and their dependencies. Middle, selected subgraph; the unprocessed tracking data is inferior to that from the filter services, and the user has chosen the touchglove input over gesture input. Bottom, the running subgraph.

instantiated first. For a template to be instantiated, its context attributes must be bound to concrete values. These values come from instance services (which may, in turn, be instantiated from templates). Thus, at least one instance service is required for instantiation of services from the template graph.

Together, the template graph and the instance graph form the *template-instance graph*. This contains instance services as well as template services, and also allows dependencies between them. An example is in Figure 5.17 on the facing page.

A template-instance graph can grow inductively. When the middleware instantiates template services, it adds nodes to the graph. These new instance services may lead to the instantiation of more instance services. Particularly, different instances of the same template service may depend on each other and cooperate. Figure 5.17 shows how different instances of the pose inference service can cooperate to combine data from different trackers.

As the template-instance graph grows, so does the instance graph of the previous section: the instance graph is simply the template-instance graph without any of the template services. Similarly, the selected instance subgraph, and the running service subgraph can grow as well. The running instance graph thus adapts to changes in the environment. When new services are added to the system, because new hardware has been found, or when attributes of existing services change, e.g. when a user enters a room, the template-instance graph can change significantly.

The middleware can construct the template-instance graph and the selected instance subgraph even for services that are not actually running, since it has access to the service descriptions. This creates a graph of potentially interconnected services. Only when one of the services starts up, are the others started, recursively.

This is an important optimization, for two reasons. First, not all services need to be started at all times, conserving computing resources. Second, the middleware can perform the distributed computation of the "optimal" template-instance subgraph in the background, without interrupting running services. Thus, the system remains interactive, while becoming adaptive at the same time.

Several ubiquitous computing systems employ adaption by component selection: in different contexts, the middleware selects different components. The approach used in the adaptive service dependency architecture goes beyond that. It allows services to be instantiated in different contexts and recursively composed into large-scale systems. This allows more adaptive and more flexible systems, but also makes them more complex.

The template-instance graph can grow very large, due to combinatorial complexity. A context may contain several context attributes, and the number of nodes in the template-instance graph can grow to include all possible combinations of all context attributes' values. It may even grow infinitely, with instantiated templates inducing other instantiated templates without end. Of course, the middleware cannot possibly construct such a graph, nor would it be useful. Appropriate measures must be taken to limit boundless growth of the graph. These are discussed in Chapter 6.

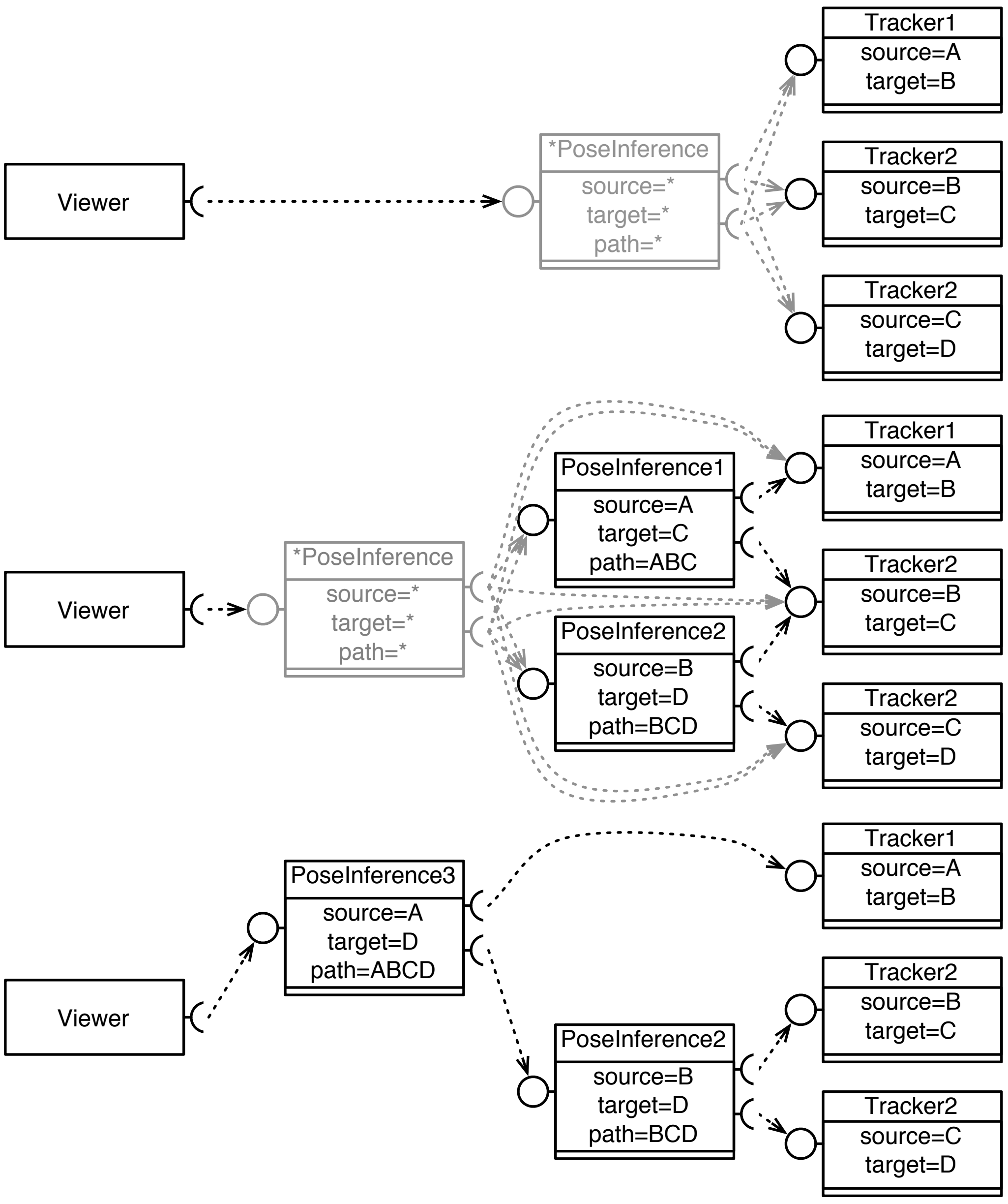


Figure 5.17: A graph of template and instance services.
The template pose inference service of Figure 5.12 is instantiated multiple times, processing data from other inference services. Top and center, two steps in the growth of the template-instance graph, with the template service shown in gray. Bottom, a running instance subgraph of pose inference services and a viewer service.

5.6 Instantiating the Adaptive Service Dependency Architecture in a Framework

As described in Section 4.3, the adaptive service dependency architectural style can be used as the foundation of a framework for ubiquitous augmented reality systems. Such a framework is then an instance of the adaptive service dependency architecture, and can be used to build concrete UAR systems.

This section describes the necessary ingredients for an implementation of the adaptive service dependency architecture within a complete UAR framework, and show how many of these are implemented within the current version of DWARF.

Domain restriction A framework need not address the full range of ubiquitous augmented reality; it may prove beneficial to address only a sub-domain. For example, [109] describes the *Minimal Mobile Maintenance Augmented Reality Framework (M^{3}ARF)*, which is specialized for the domain of maintenance of machinery in AR. Another example is the Ubitrack framework (Section 4.6.3), which addresses solely the domain of tracking in ubiquitous computing environments.

DWARF addresses the entire domain of ubiquitous augmented reality, but has, so far, been used primarily for the design of research prototypes.

Hardware and operating system platform In the UbiCom project, custom hardware for wearable computers was developed, using embedded Linux. The Tinmith project developed a wearable system based on standard laptop PC hardware, with certain additional I/O components. In both cases, the software could be highly optimized for the hardware.

Similarly, the choice of operating system is important: by limiting oneself to one operating system, all the advanced features of that platform can be used. However, this kind of specialization comes at the cost of decreased portability.

DWARF runs under Linux, Mac OS and Windows on standard PC and laptop hardware, as well as on palmtops supporting Linux.

Middleware platform The adaptive service dependency architecture requires strong middleware support, as is explained in detail in Chapter 6. Thus, a framework based on it must include an implementation of this middleware. This, in turn, may be based on a standard middleware platform, such as a CORBA implementation. A middleware platform may require run-time infrastructure, as well. For example, a middleware platform based on web services may require a UDDI server for service discovery.

The DWARF middleware is based on CORBA and uses SLP for service discovery.

Language choice The implementation language is an important consideration, as it may involve a tradeoff between ease of development and run-time efficiency. A strong middleware platform may allow the use of several different languages simultaneously.

With the use of CORBA, DWARF supports many languages. Currently, C++, Java and Python are used.

Mapping of architectural elements The elements of the adaptive service dependency architecture must be mapped onto corresponding elements of the underlying platforms: operating system, middleware and programming language. This involves important resource allocation decisions. For example, a service may be mapped onto a standalone running process, a loadable library, a self-contained computer, a web service, or a single class in the implementation language.

In DWARF, services of the adaptive service dependency architecture are mapped onto *DWARF services*, which are implemented as CORBA objects, standalone processes and implementation language classes.

Implementation restrictions The concepts described in the previous sections may only be partially implemented, or may be implemented with restrictions. For example, the expressive power of predicates and penalties may be limited to simple keyword matching, or may include complex mathematical expressions. Attributes may or may not support complex types. The choice of an underlying middleware platform can influence this restriction—for example, each service location technology has its own predicate syntax.

The current implementation in DWARF has several restrictions:

- Attributes may only have simple types (strings, numbers, booleans), not complex ones. Typing is implicit, based on the attributes' value.
- Predicates only support simple boolean expressions.
- Support for quality-of-service attributes in connectors is quite limited.
- Support for penalties and distances is currently only experimental.

Reusable service implementations The "meat" of a framework is the implementation of services that can be reused in different applications. A ubiquitous augmented reality framework should include services from all necessary subsystems (see Section 2.3): tracking, context, world model, presentation and interaction. If the framework targets a certain application domain, it should include services for the application subsystem, as well.

The service implementations must meet high standards regarding reusability, as users will wish to recombine them into new applications while the system is running. Thus, the design of the framework should include a model of the *variability* of systems built with it, defining which applications a service will be reused in.

DWARF includes reusable services from all subsystems of ubiquitous augmented reality.

Standardized types, attributes, interfaces and protocols For services to interoperate, several aspects of them must be standardized among developers. This includes types for needs and abilities, with corresponding interfaces and data types, attribute types and corresponding naming schemes (e.g. for room numbers on a campus), and communication protocols.

Coordinating these standards becomes more and more complex as the number of developers grows. In a small group, a simple group consensus may be enough, but if several organizations are involved, they may require standardization body or naming authority. A starting point is a collection of design patterns for augmented reality [79, 86, 110].

Interfaces in DWARF are defined in CORBA IDL. Types for abilities and needs, as well as attribute names and meanings, are documented informally on the internal development web site. Communication protocols are specified in CORBA IDL where appropriate, and otherwise using documented implementation examples.

Developer tools Developers' productivity with the framework will improve dramatically if there are development tools to support them, as discussed in Chapter 7.

DWARF includes several development and testing tools, especially for system visualization and prototyping.

Implementation in other Frameworks It is worth pointing out that adaptive service dependency architecture could—in theory—be transferred to other existing component-oriented AR or UbiCom frameworks, e.g. Studierstube or Tinmith. This would involve several steps, none of which is trivial, but all of which are at least possible.

Decoupling of components First, the existing components must be decoupled if they are tightly coupled. In component-oriented systems, this is usually possible, with the use of appropriate abstractions, interfaces and communication middleware.

Externalization and management of component descriptions Component descriptions must be externalized from the components themselves, so that they can be managed by external middleware. Appropriate managing middleware components must be developed, as well.

Turn components into services Finally, the components must be turned into services, i.e. made adaptive. This means that components must change their behavior when their descriptions or their communication partners change.

Of course, this only serves as a very rough guideline, and many more steps are necessary in detail. Another possible approach is to integrate existing frameworks, such as described for Studierstube and DWARF in [7].

Middleware for Decentralized Service Management

The architecture's adaptivity is supported by reflective middleware, consisting of distributed service managers.

The adaptive service dependency architecture requires middleware to support it, and this middleware consists of decentralized service managers.

This chapter describes the service managers' required functionality, based on a detailed use case model, and discusses performance considerations.

6.1 Use Cases and Phases of Adaptive Service Management

In Section 5.2.3, the service manager's functionality, decentralized adaptive service management, was described in terms of high-level use cases. These high-level use cases can be further further be subdivided into lower-level use cases, as shown in Figure 6.1 on the following page. These lower-level use cases are called *phases*, since the concerned architectural entities (services, pairs of services, etc.) must go through each phase until the adaptive system is running, and when the system's context changes. Note that these phases are not exclusive, but overlapping; a service that has started up may still change its description.

Manage single services Services must go through the phases of description, loading and startup before they can operate.

Manage communication between two services A pair of communicating services goes through the phases of location, selection, session, connection and communication, until the two services can actually communicate.

Adapt a service to its context of other services An adaptive service, as defined by a service template, goes through the phases of binding, instantiation, induction and adaption, in order to form a context-adapted service instance.

Manage an adaptive service graph An entire graph of services goes through the phases of growth and coalescence, forming an adapted graph of services.

The middleware is designed around these phases. For each, it provides a certain default functionality, which can be customized by modifying service descriptions, and which can be overridden by services themselves.

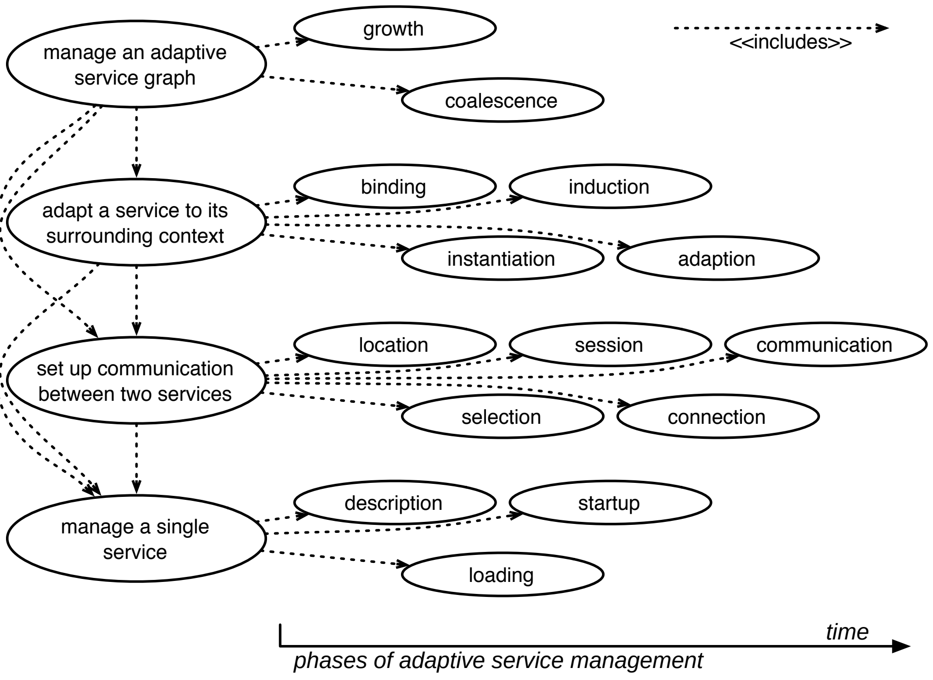


Figure 6.1: Detailed use cases for the service managers of the adaptive service dependency architecture.
The high-level use cases, left, include one another, as indicated by the dashed arrows. Each high-level use case is divided into further use cases, the *phases* of adaptive service management. These take place sequentially over time, although they can overlap.

The following sections use these phases to explain the interaction between services and the middleware. They concentrate on startup, creation, setup and so on, rather than on shutdown, deletion and cleanup. The middleware must handle these “cleaning up” activities as well, and they are grouped into the same phase. For example, the interfaces for the startup phase also contain functions relating to service shutdown. These shutdown functions are not described in detail, as they are essentially symmetric to the startup functions.

6.2 Managing a Single Service

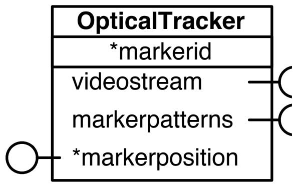
When considering the management of a single service, as described in Section 5.2, services are essentially components. They have service descriptions that are managed by a service manager on each network host. Before a service can operate, the service manager must support it in the phases of description, loading and startup.

6.2.1 Description

Each service, whether it is running or not, is described by a service description, which is maintained by the service manager.

Service descriptions must be stored persistently, so that the middleware can load them upon system startup. For this, an appropriate file format must be chosen, such as WSDL or CDL. The service descriptions should be stored on the same host that a service is installed on, so that the different hosts can operate autonomously. Given the persistently stored description of a service, a host's service manager can perform service discovery on its behalf and advertise it for discovery by other services (Section 6.3.1).

The syntax chosen for service descriptions is important, but may be chosen differently in different implementations. An important consideration is that the service description should have a syntax which developers can work with easily, as they must specify the description of services that they develop. If the service description syntax is complicated, developers will need tools to work with them. DWARF uses a simple XML format for persistent storage of service descriptions, as neither WSDL or CDL currently provide the necessary abstractions of needs and abilities. An example is shown in Figure 6.2. A component of the DWARF service manager called the *XML service describer* parses these XML service descriptions and loads them into memory.



```
<service name="OpticalTracker" startOnDemand="true"
  startCommand="/opt/dwarf/OpticalTracker">
   <attribute name="frequency" value="30"/>
   <attribute name="markerid" value="*"/>
   <need name="videostream" type="VideoStream">
      <connector protocol="Shmem"/>
   </need>
   <need name="markerpatterns" type="MarkerData"
      predicate="(id=$(markerid))">
      <connector protocol="ObjrefImport"/>
    </need>
    <ability name="markerpositions" type="PoseData"
      isTemplate="true">
      <attribute name="source" value="camera"/>
      <attribute name="target" value="marker$(markerid)"/>
      <connector protocol="PushSupplier"/>
   </ability>
</service>
```

Figure 6.2: Example service description. Simplified UML and XML description of a marker-based optical tracker service with a template ability.

A persistently stored service description is static; its set of needs and abilities is fixed, as are its attributes and predicates. In certain circumstances, it may be desirable to create a service description programmatically. Thus, the service manager must provide an externally accessible interface that lets other programs (especially services) create new service descriptions. This is especially useful in order to support development tools. For example, the monitoring tool DIVE can show events sent by a given service's ability. For this, it creates a new service description for a small monitoring service, which has a need matching the event-sending ability of the service to monitor. This way, the monitoring service receives the events just as any other services would, and can display them to the developer.

The programmatic creation of service descriptions is also useful for extensions to the middleware. For example, in the Ubitrack project, the Ubitrack management agent UMA creates descriptions of pose inference services and configures them. The DWARF service manger provides the DescribeServices interface to allow these modifications.

Once a service is running, i.e. after the startup phase, is should be able to change its own service description. This includes changing its attributes to reflect its status or changes to its quality of service, but also adding or removing needs and abilities. An example is the DWARF Viewer, which renders three-dimensional scenes [54]. The viewer service has an ability that lets other services modify the displayed scene, for example by adding new tracked objects to it. When this happens, the viewer must somehow know where to display the new object. For that purpose, it develops a new need for pose data for the newly inserted object (Figure 6.3 on the facing page).

A service description is modifiable by other services, especially development tools and middleware extensions. Again, DIVE is an example; it allows developers to modify attributes and predicates of running services' needs and abilities (Figure 6.4 on the next page) and thereby reconfigure a running system.

When a service description is changed by another service, a developer tool or by the middleware, the middleware notifies the service of this change, so that the service can change its behavior appropriately. For this, DWARF provides an AttributesChanged interface that services can implement.

After service descriptions have been modified by services, developer tools or the middleware, services (and developers) can save them persistently again. This allows services to store configuration data when they are shut down, and developers to save parts of the system state.

Development tools can retrieve a list of services from the service manager on a given host, in order to display them to the developer. Extensions to the middleware require this functionality as well. Thus, the service manager provides an interface to enumerate service descriptions, and query for service descriptions with certain attributes.

Just as service descriptions can be created programmatically, they can be deleted programmatically, as well. Especially when the middleware creates service descriptions automatically from templates, instance service descriptions that are no longer needed must be cleaned up afterwards.

6.2.2 Loading

In the loading phase, the implementation of a service is loaded into memory and registered with the service description in the middleware.

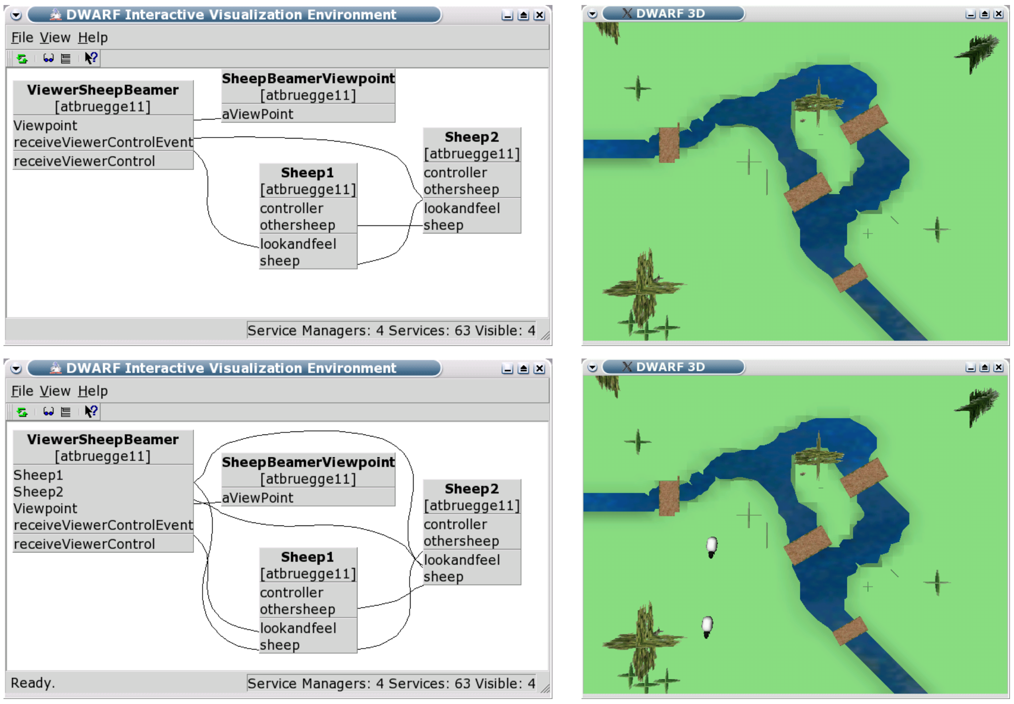


Figure 6.3: A running service develops a new need.
Here, the DWARF Viewer service develops two new needs for pose data after two sheep have been inserted into the scene. Left connected services shown in the development tool DIVE; right, rendered scene. Top, before the sheep have been inserted; bottom, after the sheep have been inserted.

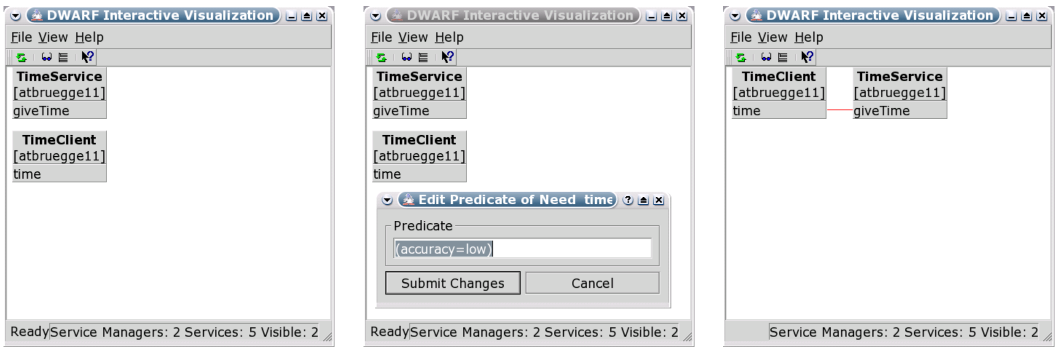

DWARF Interactive Visualization
File View Help
TimeService [atbruegge11]
giveTime
TimeClient [atbruegge11]
time
Ready
Service Managers: 2 Services: 5 Visible: 2
DWARF Interactive Visualization
File View Help
TimeService [atbruegge11]
giveTime
TimeClient [atbruegge11]
time
Edit Predicate of Need time
Predicate
(accuracy=low)
Submit Changes
Cancel
Ready
Service Managers: 2 Services: 5 Visible: 2
DWARF Interactive Visualization
File View Help
TimeClient [atbruegge11]
time
TimeService [atbruegge11]
giveTime
Service Managers: 2 Services: 5 Visible: 2

Figure 6.4: Changing a service description with a development tool.
Services shown in DIVE. Left, two services that are not connected because predicate and attributes do not match. Center, a dialog for changing predicates. Right, the services are connected.

Creation of a new process One simple mechanism for loading a service is for the service manager to create a new operating system process which runs the executable for the service's implementation. Which executable to run must be specified in the service description. This approach is used in DWARF. On desktop systems, the new process is loaded in a terminal window.

Loading services in separate processes has several advantages: services can easily be implemented in many different programming languages and use different libraries, as they are completely independent of one another. Also, a developer can load a service manually, simply by starting a process, making debugging individual services easier. Finally, if one service crashes or deadlocks, it can be terminated by the operating system, and other services will not be affected. There are also disadvantages, however: if each service runs in a separate process, communication between services even on one host will always involve interprocess communication, which is more expensive in terms of processing time than local method invocations in a single process.

Loading within an existing process To circumvent this last difficulty, an alternative is to load several services into the same process, creating *colocated services*. This is particularly easy for several services of the same template service, sharing a common implementation in the form of a C++ or Java class. In that case, a single executable process can contain the service implementation, and loading a new service is as simple as instantiating an new object of the service's implementation class. The DWARF middleware supports this mechanism: a *loader service* (or other process) can register its SvcLoad interface with the service manager for one or more service descriptions. When the middleware needs to load one of those services, it calls the appropriate method of the SvcLoad interface, and the new service can be instantiated within a running process (see Figure 6.5 on the facing page).

Another approach to loading services into a running process is to load code dynamically. This is language specific, but supported by many languages, such as dynamically loadable libraries for C++, class files for Java, or modules for Python. If a loader service is implemented in each supported language, then services can still be implemented in many different languages, but be loaded into only a few different processes at run time (as many processes as languages).

Yet a third approach is to enforce a common binary or format for all supported languages, so that all services are available as dynamic libraries in the same format. COM [23] uses this approach, defining a common binary format for loadable libraries; .NET [24] uses it as well, using a *common intermediate format (CIF)* for libraries which is compiled with a just-in-time compiler.

Registering with the service manager Once a service is loaded, it must register itself with the service manager, establishing a communication link between the service implementation and the service description. One possible implementation for this is to have the service loading mechanism, which is invoked by the service manager, return a reference to the service's implementation. However, this makes it more difficult for developers to load a service manually, e.g.

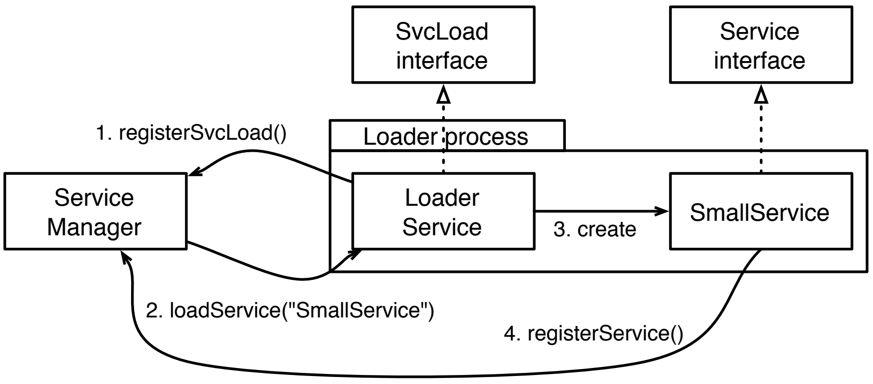


Figure 6.5: Loading a service into a running process.
The Loader service registers its SvcLoad interface for a service description. When that service is supposed to be loaded, the middleware delegates this to the loader service. The service then registers itself with the service manager.

for debugging—all services must be loaded by the middleware. A more flexible implementation therefore is for the service manager to provide a registration interface, which the newly loaded service registers itself with. This is the approach used by DWARF (with the RegisterServices interface), as shown in Figure 6.5.

In order to register with the service manager, the service must have a way of contacting it. This can be solved either by using a local naming service or a well-known initial reference. In DWARF, the service manager is contacted by a well-defined CORBA location string, consisting of a standard TCP port number and an object identifier. This initial reference is compiled into every service in the form of a helper library which initializes the CORBA communication and contacts the service manager [114].

Communication between service and service manager At this point, bidirectional communication between the service and the service manager must be set up. In DWARF, the communication is via CORBA, as this allows interprocess, language-transparent communication, and services can be implemented in different languages and as separate processes. Note that the communication between a service and the service manager still remains local to a single network host (unless the service manager is running on a remote network host).

The communication link between the service manager and the service's implementation is important in all the later phases. During registration, a service should also let the service manager know which default behavior of the middleware it wishes to override. In DWARF, services can implement a number of different *callback interfaces*. This allows them to be notified of certain events, and to override default behavior of the service manager. For example, if a service implements the SvcSelection interface, the service manager will call this interface when it must select one of several possible abilities to match to a need. If the service does not implement this interface, the service manager implements a default selection strategy. This follows the extension interface design pattern [17].

When the communication between the service and the service manager is set up, a keep-alive message lets service manager detect if a service implementation is still running. In DWARF, this is implemented by the getStatus method, which every service must implement, and which must simply return an arbitrary string (which can also be helpful to developers in monitoring the system). When the service manager detects that a service is no longer running, it unloads the service, terminating the process, if necessary.

Unloading services Unloading a service is symmetric to loading it: it involves either terminating a process, unloading a dynamic library, or deallocating an object in memory. The only difference is that a service can generally unload itself, but obviously not load itself. In DWARF, unloading is handled by the same SvcLoad interface as loading.

6.2.3 Startup

When a service is loaded into memory, that does not necessarily mean it should start running. This does not happen until the startup phase.

A service can start running when all its needs are satisfied, and when at least one of its abilities is requested by another service or by the user—or when a developer wishes to test a service manually. Conversely, a service should stop when none of its abilities are requested anymore, or when its needs can no longer be satisfied. The middleware can detect these conditions and notify the service that it should start or stop.

The service should therefore implement callback interfaces for startup and shutdown. In DWARF, these are part of the SvcStartup interface. Within the startup callback, there are three important tasks the service should perform. The first is initialization: the service may wish to configure itself according to attributes in its service description, perhaps even creating new needs or abilities depending on the service description. At this point, the service can register newly created abilities or needs with the service manager. The second task is to allocate system resources that the service needs. For example, a video grabber service may need to access the operating system's *FireWire* driver or a video capture card. Third, if the service has a main processing loop, it should start this within a separate thread of execution.

6.2.4 Deployment and Other Missing Phases

The discussion above started with the description phase: a service which is installed on the system is described. However, there is obviously a point in time when a service is not installed yet. The process of installation can be modeled with an additional "deployment" phase, in which the middleware downloads executable code and installs it on the computer, or assists the administrator in installing a piece of software.

However, this phase has been intentionally left out of the discussion, as it is (currently) out of the middleware's scope.

6.2.5 Order of Phases

A service must go through all of the phases described above in order to start running. The order is the same as the order they have been presented in: first, a service is described, then loaded, then started. When it shuts down, the phases are gone through in reverse: the service stops, is unloaded, and finally the description is deleted.

However, there are exceptions to this rule:

- If a service is implemented as a standalone process which is started manually, and it generates its own service description, then the loading phase starts before the description phase.
- If a service crashes, or its process is terminated by the operating system, then the service is unloaded without going back through the startup phase.

Also note that some phases "finish", whereas others do not. For example, the loading phase completes before the startup phase can start. On the other hand, the description phase is never completely finished, as a service can modify its own service description while it is running. However, there is a point at which the description phase is "complete enough" for the service to proceed into the loading phase. In DWARF, there is a special method in the DescribeServices interface, activateServiceDescription, which indicates that a service is sufficiently described to proceed to the next phases.

6.3 Setting Up Communication Between Two Services

A pair of communicating services goes through the phases of location, selection, session, connection and communication, until the two services can actually communicate. This allows a data flow relationship between services to be created based on dynamically discovered dependency relationships.

6.3.1 Location

The service manager on each host in the network performs service discovery for services available on other hosts. The terms *service discovery* and *service location* are synonymous; thus, the *location* phase deals with service discovery.

Existing service location mechanisms There are many existing standards for service location, such as of SLP [48], Jini [140], mDNS [85], to name a few. Thus, an implementation of the middleware can reuse such an existing technology. DWARF currently uses OpenSLP [100], an implementation of the *Service Location Protocol (SLP)* [48]; this has the advantage of built-in support for boolean predicates.

Most service location technologies use a concept of service *offers* and service *requests*; this maps to abilities and needs. In DWARF, the need and ability types, as well as the communication protocol, are mapped onto SLP service types, and the *location*, or network address, of an ability is mapped to a URL. Thus, the TimeService running on host atbruegge11, which provides an ability named giveTime of type TimeOfDay using the protocol ObjrefExporter, is mapped to the URL

service:TimeOfDay.ObjrefExporter://atbruegge11/TimeService/giveTime/ObjrefExporter

and this URL matches SLP request of the type for

service:TimeOfDay.ObjrefExporter

which is generated for a corresponding need.

To map needs to offers and abilities to requests, the service manager has a component called the *locator*. This communicates with the underlying service location libraries.

In DWARF, the *SLP locator* creates a request for each connector of a need and an offer for each connector of an ability. It registers the offers with the SLP daemon running on the local host, and periodically sends multicast queries to the network for each request.

Other implementations of the locator are possible, using different service location mechanisms. For example, for debugging purposes, DWARF also includes a *simple locator* which simply compares needs and abilities on the same host, and does not perform any network communication at all.

Service location technologies can operate in different modes: either peer-to-peer mode, without relying on central infrastructure, or using directory servers. Using a directory server has the advantage of allowing centralized management and reducing network communication; relying on a directory server has the disadvantage that one must always be available [103]. SLP can run in both modes automatically; if a directory server is present, it will be used; if not, the hosts running SLP use network broadcasts to find each other.

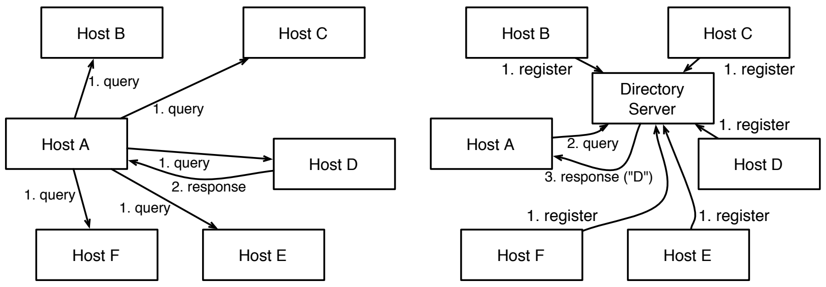


Figure 6.6: Peer-to-peer location and use of a directory server.
Left, the nodes of a network perform service discovery by broadcast; right, the network nodes perform discovery via the directory server. In the example, host A queries for a resource that host D can provide.

Some service location mechanisms support *announcements*, i.e. a host can make an announcement on the network when it has a new offer or when attributes change. For protocols that do not support announcements, such as SLP, the locator must use polling, i.e. periodically re-query the network for new offers. Obviously, this is inefficient. As a workaround, the DWARF SLP locators send each other "query hint" notifications when significant changes have occurred, triggering a re-query for the type of service for which a change has occurred.

The locators must be able to deal with several different network interfaces, and with mobile computers that roam between different wireless networks. This must be transparent to the rest of the system.

The locator is responsible for the evaluation of needs' predicates; it detects matches between needs' predicates and abilities' attributes. It also must check for need/ability type and connector type matches. If the underlying communication protocol does not support predicates (e.g. mDNS), the locator must implement the predicate-handling logic itself.

Context-based locator federations Locators can form a *locator federation*; within the federation, locators notify each others of changes and exchange announcements. in DWARF, the locators all form one simple federation, sending each other query hints.

A more advanced technique is to form federations based on context. Here, all locators that are physically in the same room, that are running the same application, or that are otherwise "close" in some dimension of context, are federated, and communicate more intensely within the federation than with other locators outside of the federation.

Assuming that services that are "close" in context are more likely to want to connect to one another, this forms a more efficient mechanism for service location. This technique is described in [35] and is similar to the idea of dynamically allocating directory servers [103].

To decide how to form federations, the locators can take advantage of the distance definitions used in penalties and predicates. By forming the federations dynamically, the locators can adapt to the need and ability descriptions on behalf of which they are acting. Locators can be part of several federations at the same time, if federations are based on several context dimensions.

Queries and and announcements are replicated between federations, although slowly. This means that when a service's attribute changes, other interested services that are in the same or neighboring room will be notified quickly, whereas services that are farther away will have to wait several seconds or minutes until the update is propagated. This optimization allows services that are running on machines near a user to react quickly to context changes (e.g. the user enters a room), without incurring the performance overhead of broadcasting changes to the entire network.

Several different kinds of locators can work together, forming a hierarchy. For example, a *local locator* can quickly detect matches between services on a single host. An SLP locator can detect other service managers in the network, and exchange context attributes between them.

A federation locator can replicate queries and offers in a federation, which is formed by the results of the SLP locator. Finally, within each federation, the federation locators (which may be implemented as services) can elect an inter-federation locator, which is responsible for replicating data between different federations. The inter-federation locators form a higher-level federation among themselves.

This procedure can obviously continue on upwards; at some level, the inter-federation locators could also use a peer-to-peer network technique such as Gnutella [43] to discover each other, or simply revert to a managed network infrastructure such as DNS.

Stored static locator results The result of the location phase is a list of matches between needs and abilities. This list can also be stored statically and restored again, in order to create static setups, e.g. for testing purposes. For this, a simple *static locator* would be appropriate, which reads matches from a file. At the same time, developer tools would be needed to create this file, e.g. from an interactive system deployment diagram.

6.3.2 Selection

The selection phase is only relevant for needs, not for abilities. In this phase, the service with a need selects which ability of another service it wishes to use.

For a given need, the result of the location phase is a list of potential matches: abilities of other services that have a matching type, a matching connector protocol and attributes that satisfy the need's predicate. Based on this information, the middleware must decide which of those abilities to select and set up communication to.

A need description must specify how many abilities it wishes to use. The two most frequent cases are exactly one or arbitrarily many. However, a need might also require exactly 2 partner abilities. In DWARF, the allowable range for the number partners is specified by the maxInstances and minInstances parameters.

One strategy for selecting partners for a given need is to evaluate the penalty for each, and choose the n required partners with the smallest penalty, where n is the number of desired partners. In DWARF, general penalties are currently not supported. However, there is a default penalty, which penalizes services on remote hosts, running under different user accounts, and ones that have not started yet. This gives preference to local services and ones that are already running.

In some cases, it may not be possible to describe the preferred partners in terms of a penalty specification. Then, an alternate strategy is to let the service with the need decide itself; this follows the *client knows best* pattern described in [103]. In DWARF, this is realized by the service implementing the SvcSelection interface.

A similar strategy is to delegate the selection to a third service, which presents a menu to the user and lets the user select. This follows the *people know best* pattern described in [1]. In DWARF, this is realized by the Selector service, which has an ability of type Selection implementing the SvcSelection interface. This uses a simple user interface (e.g. on a palmtop) to present the user with a list of alternatives. A service whose selection is to be delegated has a special need of type Selection in the service description, indicating that selection for this service's needs should be delegated to another service. An appropriate selector service, which displays a menu to the user and passes the selection back to the middleware, can be chosen using the standard middleware mechanisms. An example of this delegated selection is shown in Figure 6.7. This process of *delegated selection*, as well as the selector service, are described in [78] and [126].

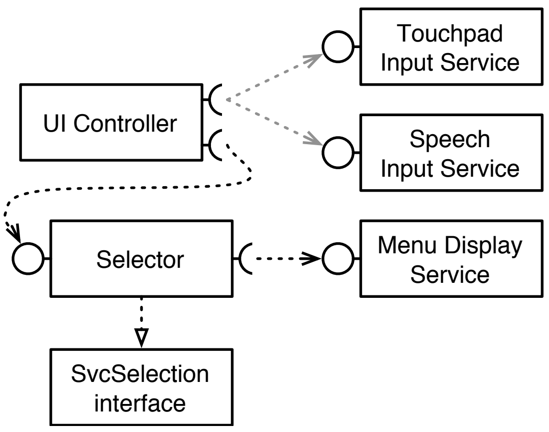


Figure 6.7: Delegated selection using the Selector service.
A UI Controller service cannot choose between the touchpad and speech input services (dependencies shown in gray). The selection is delegated to the Selector service, which uses the MenuDisplay service (Figure 4.6) to present the user with a list of choices. To find the Selector service, the UI Controller has a need of type Selection.

When the attributes of available abilities change, the selection should be re-evaluated. This means re-evaluating penalties, presenting a menu to the user again, and so on. Thus, the service managers on different hosts must inform each other, on a regular basis, of attribute changes that could lead to a change of selection. In DWARF, the service manager responsible for a service's need subscribes to AttributesChanged notifications for all potential abilities, and calls the SvcSelection interface when it receives a notification from a remote service manager that one the abilities' attributes have changed.

6.3.3 Session

When a need has selected an ability, the pair enters the *session* phase. Here, the services negotiate terms of communication, and set up internal data structures to prepare for communication.

In this phase, the services have access to each others' attributes, and can use these for detailed compatibility checks, or even to check security tokens. Services can also perform resource management or accounting, e.g. to bill for use of a service's ability. A service may accept or refuse communication based on the partner's attributes. None of the services in DWARF currently use these security-related features.

A service can also adapt its ability to the partner's need. For example, a Video Grabber service can adjust its frame rate to the maximum processing rate of an Optical Tracker service. If an ability is used by many other services simultaneously, it can use the session phase to adapt the quality of service it provides to provide maximum benefit to the other services.

A service allocate a separate object or internal data structure to handle communication with each partner; or it may communicate with all partners using a single object. In the session phase, a service can allocate an appropriate handler object, and register it with the middleware. The middleware then uses the service's newly allocated object for communication with the remote service.

In DWARF, services may implement the SvcSession interface, which is called by the middleware in the session phase. The newSession method may return an object reference to a newly allocated handler object, which is then used for communication. An example is shown in Figure 6.8.

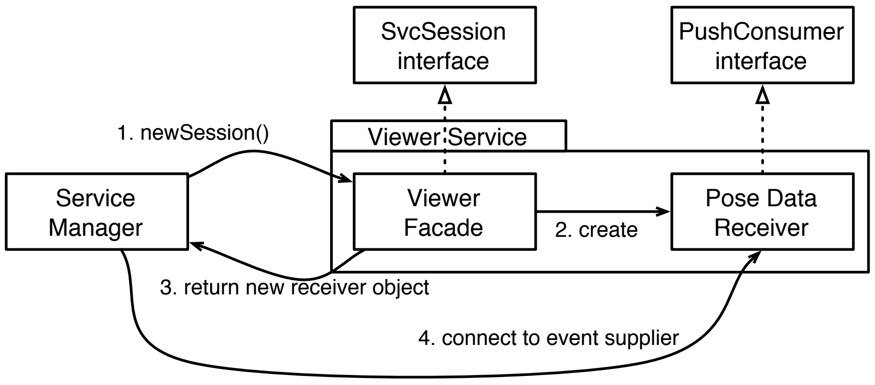


Figure 6.8: Services allocate handler objects for a session.
In this example, the Viewer service allocates a new Pose Data Receiver object for incoming pose events.

6.3.4 Connection

In the connection phase, the middleware allocates resources needed for communication between a pair of services, and exchange appropriate references.

For each communication protocol, the service must support an appropriate interface allowing the middleware to set up communication. These are entirely specific to the communication protocol. For example, in event-based communication (Figure 6.9 on the facing page), the service that sends (supplies) push-style events implements the interface SvcProtPushSupplier, which includes the method setConsumer. This allows the middleware to allocate a store-and-forward communication channel and connect the event supplier service to it. The consumer service does not have to do anything in the connection phase.

Communication resource allocation and protocol initialization is handled by connectors for different protocols. The service manager can obtain connectors from *connector factories*. A service manager must have access to a connector factory for each connector type, e.g. a connector factory for event-based communication and another for shared memory.

In DWARF, the connector factories are components that are compiled in to the service manager; there is one for each supported communication protocol. Currently supported protocols are CORBA remote method calls for tightly coupled services; CORBA notification service events for loosely coupled, event-based communication, and shared memory for high-bandwidth video data.

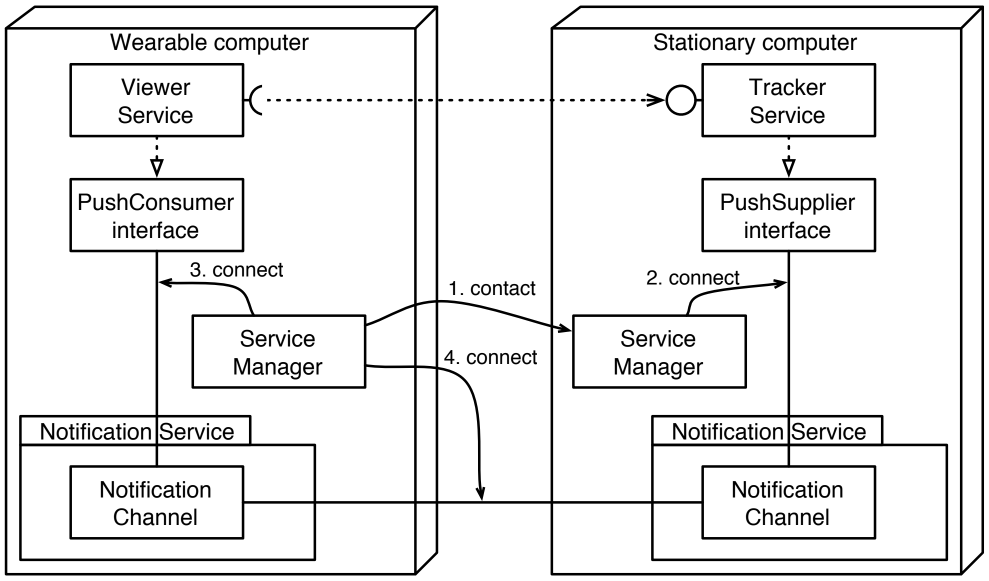


Figure 6.9: Two service managers cooperate in setting up communication between two services.
In this example, two services communicate using event-based communication. Two notification channels are created in the CORBA notification service daemons and connected to the communicating services.

When colocated services, i.e. two services that are running within the same process space, wish to communicate, the middleware should ensure that this communication does not incur inter-process communication. For example, in DWARF, event-style communication uses an implementation of the CORBA *notification service*, which runs as a separate process. (Note that the notification service is a *CORBA service*, not a DWARF service; the term “service” is overloaded.) Two communicating services in DWARF must communicate with the notification service using interprocess communication, even if they are colocated (Figure 6.9). The middleware can avoid this problem by using dedicated in-process connector factories for colocated services. Each loader service should thus include appropriate connector factories.

A special connector included in DWARF is the *null connector*—this is a connector that does nothing, and does not allow communication between services. It is included so that services can ensure each other's presence, even when they do not have to communicate directly. This is particularly useful when a service only needs another for configuration information, and the exchange of attributes in the session phase is enough.

The connection phase is finished when the communication resources for both services are set up, and the services are ready to communicate. At this point, the middleware should notify both services, using an appropriate callback that communication can start.

The connection phase is where existing *data flow frameworks*, such as NMM or OpenTracker, can be integrated into the middleware. Assuming such a data flow framework offers an interface for configuring data flow at run time, the middleware can set up appropriate resources for communication and connect them to the services.

6.3.5 Communication

Once the communication resources are set up, all the services must do is communicate. How they do this is, of course, protocol-specific.

For event-style communication, DWARF uses the CORBA notification service standard. Services exchange *structured events*, which allow inclusion of headers and strongly typed data, specified in IDL. Events are actually sent using CORBA remote method invocations, as shown in Figure 6.10 on the facing page. The events are stored and forwarded by the notification service. The corresponding connectors are called PushConsumer and PushSupplier.

The simplest form of communication in CORBA is simply that of invoking a remote method, specified in IDL. Compared to events, this has the advantage of supporting a return value; however, it increases the tightness of coupling between services. In DWARF, the ObjrefExporter and ObjrefImporter connectors allow services to communicate via method calls.

When a service sends an event, the middleware can dispatch this to many consumers. When a service invokes a method, the middleware cannot sensibly dispatch this to many implementations, since the return values are likely to be different. Thus, if a service has a need to invoke methods on remote abilities, it must manage a list of remote object references itself. In DWARF, a service can do this by implementing the SvcSession interface, as described in Section 6.3.3.

In communication via shared memory, one service writes to a buffer, and several others read from it, with synchronization using a semaphore. In DWARF, this is implemented using standard *System V IPC* libraries for inter-process communication.

For ubiquitous augmented reality applications, the communication methods used for communication in the tracker-renderer chain must have low latency, to prevent motion sickness for the user. This should be considered in the design of the communication phase.

This is also one of the main reasons for separating the communication phase from the other phases (e.g. selection, connection)—it allows the communication phase to use low-level operating system mechanisms such as semaphores, whereas the other phases are less time-critical and can be implemented using complex interprocess and network communication.

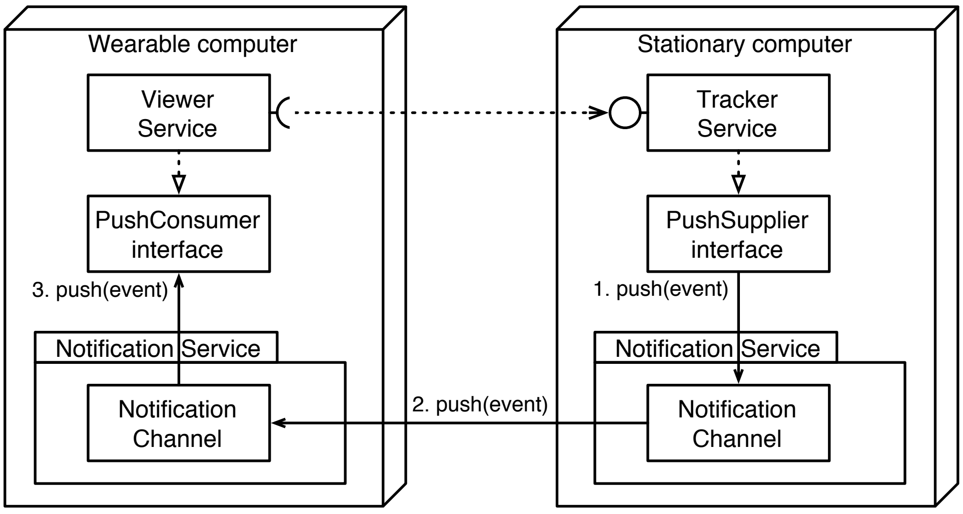


Figure 6.10: A tracker service sends events to a viewer service on a remote machine. The services communicate using push-style notification events, which are stored an forwarded by a CORBA notification service daemon. Pushing an event corresponds to calling the CORBA push_structured_event method, abbreviated as push() in the figure.

For implementation of the communication phase, the middleware provides helper libraries to the services. Thus, services do not have to call low-level semaphore-locking functions, but can call appropriate helper functions. Similar helper functions are available in DWARF, e.g. for the construction of structured events.

6.3.6 Order of Phases

For communication between two services, the order of the phases is fixed: location must precede selection, which must precede session, connection and communication. All phases except for connection and communication can be handled by default handlers in the middleware.

6.4 Adapting a Service to Its Context

An adaptive service goes through the phases of binding, instantiation, and induction, as an instance service is created from a template service.

6.4.1 Binding

In the binding phase, the middleware finds values for a template's context attributes from other services. The values for a context attribute come from the location phase of a pair of services.

In a template service, the binding process can start with a need or an ability.

When a template service's need finds a matching ability, this ability's attributes can be used to bind the template service's context attributes. Figure 6.11 shows a template service with the context attributes obj1, obj2 and obj3. The context attributes obj1 and obj2 are set from the matching abilitiy's attributes source and target.

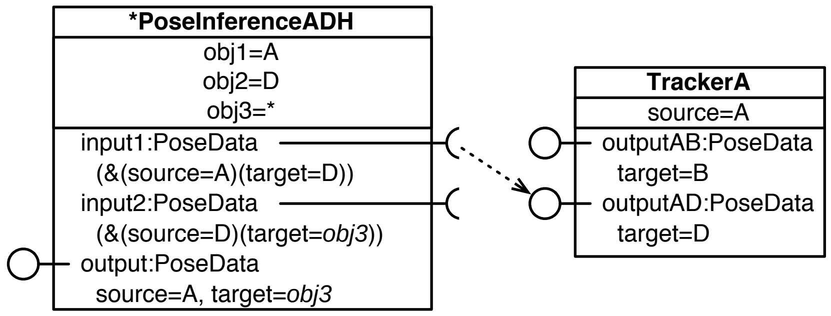


Figure 6.11: Binding within a need of a template service.
In the PoseInference service of Figure 5.12, the attributes obj1 and obj2 have been bound to the values A and D, from the TrackerA's outputAD ability. The resulting service is still a template service with the unbound context attribute *obj3*.

When a template service's ability matches a need, this need's predicate can be used to bind the template service's context attributes. Figure 6.12 shows the same template service as Figure 6.11. Here, the context attributes obj1 and obj3 are set from the matching needs' predicates, the values required for source and target.

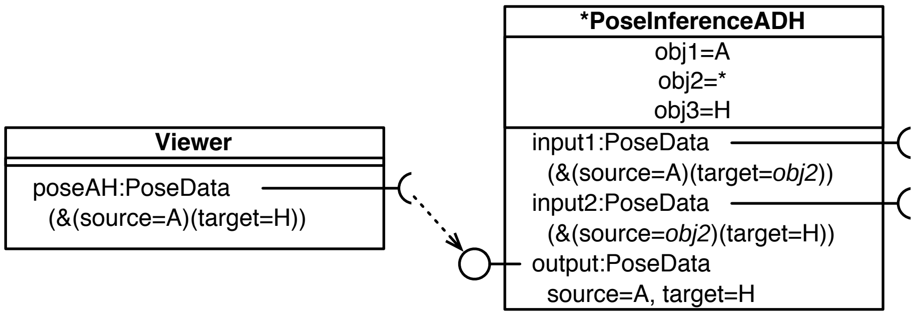


Figure 6.12: Binding within an ability of a template service.
The PoseInference service's attributes obj1 and obj3 have been bound to the values A and H, from the Viewer's poseAH need. The resulting service is still a template service with the unbound context attribute *obj2*.

The rules for binding are given in the service description. The middleware extracts the rules

```
<service name="PoseInference" isTemplate="true">
    <!-- context attributes are indicated by "*" -->
    <attribute name="obj1" value="*"/>
    <attribute name="obj2" value="*"/>
    <attribute name="obj3" value="*"/>
    <need name="in1" type="PoseData"
        <!-- binding and induction for obj1 and obj2 -->
        predicate="(&(source=$(obj1))(target=$(obj2))))">
        <connector protocol="PushConsumer"/>
    </need>
    <need name="in2" type="PoseData"
        <!-- binding and induction for obj2 and obj3 -->
        predicate="(&(source=$(obj2))(target=$(obj3)))">
        <connector protocol="PushConsumer"/>
    </need>
    <ability name="out" type="PoseData">
        <!-- binding and induction for obj1 and obj3 -->
        <attribute name="source" value="$(obj1)"/>
        <attribute name="target" value="$(obj3)"/>
        <!-- induction for confidence: product of input confidences -->
        <attribute name="confidence" value="$(*((in1.confidence)(in2.confidence)))"/>
        <attribute name="meanerror" value="$(errcat((in1.meanerror)(in2.meanerror)))"/>
        <connector protocol="PushSupplier"/>
    </ability>
</service>
```

Figure 6.13: Binding and inference specification in a service description.
Context attributes are indicated by the value "*". Binding rules are expressed using $(contextattribute) expressions in predicates and attribute values. Inference rules additionally may contain arithmetic expressions such as $(+(a)(b)).

for binding from the predicate and attribute specifications, which contain expressions of the form $(contextattribute). For example, the service description shown in Figure 6.13 contains the expression (source=$(obj2)) in input2's predicate.

6.4.2 Instantiation

In the instantiation phase, the middleware instantiates one or more template services (or template abilities).

When a combination of context attributes has been found, the middleware can instantiate the template with this context. However, for performance reasons, it may be desirable not to instantiate all possible instances immediately, as the number of instances can easily grow exponentially. The choice of an appropriate instantiation strategy can significantly affect the middleware's performance.

One possible strategy is to instantiate a service in all contexts in which at least one partner service is already running. Another is to evaluate an appropriate distance function and instantiate services that are "close" to the user. In DWARF, the default strategy simply instantiates all possible services. Services, or extensions to the middleware, can specify more efficient instantiation strategies by implementing the SvcInstantiation interface.

The values for a context can depend on more than one need or ability; in the PoseInference example, there are three context attributes, two of which are determined by each need. To simplify the construction of contexts, the middleware allows *partial instantiation*. For example, a PoseInference service can be instantiated with obj1=A,obj2=C, but without a value for obj3. The resulting service is still a template service, as it still has one unbound context attribute, obj3. This is shown in Figure 6.11 on page 110. The template service can, in turn, be instantiated to a fully bound instance service.

Even though two instances may have different context attributes, they may still be functionally equivalent, and it would be redundant to instantiate both. For example, in Figure 6.14 on the facing page, there are two PoseInference services which calculate the pose of object D relative to object A, using the spatial relationships AB, BC and CD. The results of both services would be the same, although they are computed in a different order. To avoid this duplication, the description of a template service may specify *equivalence classes*. Each equivalence class has a *discriminator*, which evaluates to the same value for all members of the equivalence class. Within each equivalence class, only a limited number of services are instantiated.

Figure 6.15 on the next page shows a specification of two kinds of equivalence classes. The first has a discriminator that evaluates to the path through the spatial relationship graph, such as ABCD. The middleware can thus detect that PoseInference1 and PoseInference2 from Figure 6.14 are equivalent, and only instantiates one of the two possible services. The second equivalence class is based on the spatial relationship that a service computes; it evaluates to the label of the edge, such as AD. Here, the middleware will instantiate the two "best" services of each equivalence class: one which computes the most spatially accurate estimate, and one which gives the estimate with the highest confidence. This functionality allows the middleware to perform distributed optimization.

6.4.3 Induction

In the induction phase, the middleware sets a newly instantiated service's (or ability's) attributes and predicates based on the values of the context attributes. The rules for this are given in the service description.

In DWARF, induction rules are a superset of binding rules. For example, the expressions $(obj1), $(obj2) and $(obj3) in Figure 6.13 are replaced with the appropriate values of the context attributes in Figure 6.16 on page 114. For the confidence attribute, the expression $(*((in1.confidence)(in2.confidence))) evaluates to the product of the two input values' confidences. Similarly, the meanerr attribute is computed using a function called errcat.

The computation of attribute values in the induction phase can be quite complex. Since the service is not running yet, but the middleware must compute an estimate of what the service would compute if it were running. When the service is running, it can compute its attributes

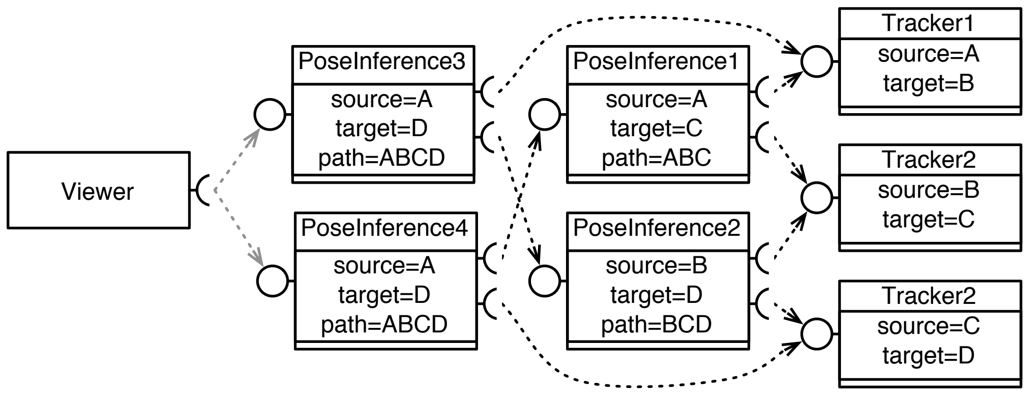


Figure 6.14: Two instances of a template service may be equivalent.
In this example, the PoseInference3 and PoseInference4 services are equivalent, as they compute the same path through the spatial relationship graph. Using the equivalence class definition of Figure 6.15 prevents the middleware from creating both instances.

```
<service name="PoseInference" isTemplate="true">
    <!-- context attributes are indicated by "*" -->
    <attribute name="obj1" value="*"/>
    <attribute name="obj2" value="*"/>
    <attribute name="obj3" value="*"/>
    <!-- determine the path in the spatial relationship graph -->
    <attribute name="via" value="$(in1.via)$(obj2)$(in2.via)"/>
    <attribute name="path" value="$(obj1)$(via)$(obj3)"/>
    <attribute name="edge" value="$(obj1)(obj3)"/>
    <!-- allow only one instantiation per spatial relationship path -->
    <equivalence name="samepath" discriminator="path"/>
    <!-- find most accurate and most confident instances for each edge -->
    <equivalence name="sameedge" discriminator="edge">
        <penalty value="(1/confidence)"/>
        <penalty value="(meanerror)"/>
    </equivalence>
    <!-- need in1, need in2 and ability out omitted -->
</service>
```

Figure 6.15: Limiting instantiation of templates.
The service description of Figure 6.13 has been extended to prevent a too large number of instantiations.

```
<service name="PoseInference-1" isTemplate="false">
    <!-- context attributes have been bound to concrete values -->
    <attribute name="obj1" value="A"/>
    <attribute name="obj2" value="C"/>
    <attribute name="obj3" value="D"/>
    <need name="in1" type="PoseData"
        <!-- bound values for obj1 and obj2 -->
        predicate="(&(source=A)(target=C))">
        <connector protocol="PushConsumer"/>
    </need>
    <need name="in2" type="PoseData"
        <!-- bound values for obj2 and obj3 -->
        predicate="(&(source=C)(target=D))">
        <connector protocol="PushConsumer"/>
    </need>
    <ability name="out" type="PoseData">
        <!-- from bound values of obj1 and obj3 -->
        <attribute name="source" value="A"/>
        <attribute name="target" value="D"/>
        <!-- from confidence values of in1 and in2; changes over time -->
        <attribute name="confidence" value="0.42"/>
        <attribute name="meanerror" value="0.05"/>
        <connector protocol="PushSupplier"/>
    </ability>
</service>
```

Figure 6.16: Service description of instantiated service.
The template service of Figure 6.13 has been instantiated.

itself. To allow the computation of complex attributes in the induction phase, the middleware supports the use of *computation functions* such as errcat, which concatenates two error distributions [108]. These error functions are implemented by extensions to the middleware, which are services themselves.

Induction can be used to compute the values of a service's attributes, attributes of a service's abilities, or the predicates and penalties of its needs. This makes instance services highly dependent on their context. In the expressions used for induction, partner services' attributes may be referenced. For example, in Figure 6.13 on page 111, the expression in1.confidence refers to the confidence attribute of the ability matching the need in1. Note that this assumes that each need is connected to only one ability.

6.4.4 Adaption

In the adaption phase, the middleware re-propagates partner's attribute changes into context attributes, needs and abilities. The adaption phase is limited to services which have already been instantiated. As an example, the confidence attribute in Figure 6.13 is re-evaluated as the values of the partner services change.

Although it would be possible to make the confidence a context attribute of the template service, this would have performance implications: whenever the confidence value of a tracker changed, the middleware would instantiate a new Pose Inference service, and delete the old one. It is more efficient to preserve the running service, but simply adapt its attributes. Once an instance service is started, it can change its attributes itself, taking over the adaption phase from the middleware. Until then, however, the middleware must continuously compute the values, using the same techniques as in the induction phase.

6.4.5 Order of Phases

The order of phases in adapting a service to its context can vary in different implementations:

Instantiation before induction: many services The middleware creates many different service instances, and computes their attribute values. This leads to a large number of instantiated services, many of which will never start. This strategy pre-computes many configurations, allowing it to react quickly to new requests, but care must be taken to avoid wasting resources. This is the default strategy of the DWARF service manager.

Induction before instantiation: distributed search In this strategy, the middleware performs a distributed search, performing binding and induction recursively on different template services. Only a small number of services are then instantiated. This strategy uses fewer resources if the system changes infrequently, but is less advantageous if it has to perform similar searches frequently, as partial results (in the form of instantiated services) are discarded. This is the strategy used by the Ubitrack middleware agent.

6.5 Managing an Adaptive Service Dependency Graph

An entire graph of services goes through the phases of growth and coalescence when it adapts to changes in the environment. Managing the adaptive service dependency graph is the highest-level use case of the middleware, and here, its functionality is essentially a combination of that of the lower use cases, but extended to many services.

6.5.1 Growth

In the growth phase, the template-instance graph grows, by the addition of edges between existing services and the instantiation of new services. The top two steps shown in Figure 5.17 on page 89 are part of the growth phase. As the template-instance graph grows, services and pairs of services go through various phases:

- single services are in the description phase,
- pairs of services are in the location phase, and

- services adapting to context are in the binding, induction and instantiation phases.

When a new service is instantiated, it becomes available for location by other (template or instance) services, which can connect to it or, in turn, create new instances. This can continue indefinitely, or peter out after a while. Figure 5.17 shows a template-instance in the growth, and later, in the coalescence phase.

The growth phase starts when new instance services become available through service location, since a new hardware device has been switched on or has come in range of the wireless network, or a service's attribute has changed due to a change in the environment. Both events can happen quite frequently while the system is running.

The growth phase stops when new instances have been created for all possible contexts, or the middleware has stopped creating new instances, due to resource constraints. Growth can also continue forever, if there is endless recursion in service descriptions. The middleware must take measures to prevent this.

Growth of the template-instance graph can proceed in two directions: from needs to abilities, or from abilities to needs. In *top-down growth*, an instance service with a need causes templates services to create instances with matching abilities. These instances have needs of their own, which cause other template services to create new instances. In *bottom-up growth*, an instance service with an ability causes templates services whose needs it matches to create new instances. These instances have new abilities of their own, which are matched by needs of other templates. A middleware implementation can combine both forms of growth; balancing them can be an important performance consideration. In DWARF, only bottom-up growth is currently supported.

The reverse process also occurs: when instances are no longer needed, or no longer match any other services, the middleware should delete them.

6.5.2 Coalescence

In the coalescence phase, services start, connect, communicate, and adapt to each other. By detecting startable services and starting them, the middleware forms the running instance subgraph from the newly grown template-instance graph. This is shown at the bottom of Figure 5.17 on page 89.

Again, this is a combination of phases from previous high-level use cases:

- single services go through the loading and startup phases,
- pairs of services go through the selection, session, connection and communication phases, and
- context-adaptive services are in the adaption phase.

An instance service is *startable* if it does not have any needs. An instance service is also startable if all of its needs are matched by abilities of other startable services. The middleware can evaluate this easily, creating an attribute startable for each service within the adaption phase (where attributes are set depending on connected services' attributes). This attribute can be used in the selection: a service is only selected if it is actually startable.

A problem arises when the startable attribute is calculated in a graph that contains cycles: those services that are part of the cycle will never become startable. There are two ways that the middleware can deal with this. First, the middleware can implement a distributed cycle-detection algorithm, and then mark all services in the cycle as startable if all needs pointing to services outside of the cycle can be satisfied by startable services.Second, the middleware can ignore this problem, and let the user start services manually, even if they are not marked as startable. It can also relax the restriction that services should only be selected when they are startable, and instead simply give services that are not startable a strong penalty in selection. This way, services with dependency cycles can be started eventually. This is the approach currently taken by DWARF.

Once a first service is started (e.g. a proxy service representing a user), the middleware loads and starts all other services that have been selected (by the middleware or by the user) for this service's needs. Recursively, these services cause other services to be started, until a network of cooperating services is running.

When a service shuts down, this can cause other services to shut down as well, either because they are no longer needed or because their needs can no longer be satisfied. This process also continues recursively, shutting down subgraphs that are no longer necessary or no longer viable.

6.5.3 Order of Phases

For fixed set of devices, and fixed environmental context, the branching and coalescence phases occur one after another. However, this is only the case in small, static systems. In general, phases can occur simultaneously, in different parts of the same service graph.

When a change occurs somewhere in the graph, its effects ripple throughout the service graph, transforming it as they proceed. When a new device appears, or an attribute changes, part of the graph enters the growth phase, and the coalescence phase afterwards. When new services start up, this can trigger attribute changes elsewhere in the graph. The distributed middleware has to start services, stop others, instantiate new subgraphs, adapt attributes, and discover new devices at the same time.

Adaption, manageability and chaos The constant restructuring of the service graph is the mechanism by which a ubiquitous augmented reality system can adapt to changes in the environment and the user's desires. However, with all these changes going on in the service graph, undesirable behavior known from chaos theory can emerge: oscillations (services subgraphs starting and stopping over and over again), degenerate attractors (infinite instantiation of templates), and unpredictable reactions (small differences in sensor input or timing leading to large differences in system behavior). All of this can be avoided by careful design; but developers must be aware of the fundamental difficulty.

6.6 Performance Considerations

This section briefly addresses some key performance considerations for the design of the middleware.

The most basic performance consideration in the middleware is that there are two different speeds that it must operate at: data flow between services must operate under near-real-time constraints (event latency should be below 10 milliseconds), whereas the distributed adaption of the UAR system only has to operate at interactive speeds (the system should react in less than one second). Furthermore, context and device availability changes are infrequent compared to changes of sensor readings. A mobile user does not change rooms very often; in contrast, trackers can easily send 100 events per second.

This assumption of two different speeds of change in the system is quite fundamental. This principle is similar to the idea of latency layering and leads to the different speeds of changes in the feedback loops shown in Figure 4.1 on page 48. It means that it is worthwhile for the middleware to set up data flow graphs in the background at fairly slow speeds, performing fairly infrequent reconfigurations and optimizations of the system. The real-time data flow then follows this graph. This leads to the decomposition of the middleware's implementation, and the separation of phases, e.g. discovery, connection and communication.

A basic scalability limitation comes from the problem of distributed service discovery: how do two computers at opposite ends of the world discover that they both have services that could cooperate, without flooding the internet with broadcasts? There are two basic approaches to solving this problem. One is to have a central directory service that all services register with. The other is to partition the network of services according to various forms of context and form locator federations.

This can be optimized by assuming that the distances in different context dimensions are correlated: if two services are physically far apart, it is unlikely that they belong to the same user or the same group of users. Similarly, if two computers have been used by the same person in the recent past, it is likely that they are physically close together. By creating dynamic locator federations based on different dimensions of context, services that are close together will be checked more frequently for matches, and those that are far apart will be checked infrequently. This approach optimizes the most frequent cases of service discovery without unduly penalizing the infrequent cases or overloading the network.

A third consideration concerns combinatorial complexity. A template service with a large context set can have very many possible instantiation contexts. For example, a simple inference service from the ubiquitous tracking example of Section 4.6.3 that combines the inputs of two different trackers can be instantiated for every combination of two trackers where one tracker tracks the other. To avoid instantiating millions of services and overloading the middleware, domain-specific strategies for the instantiation phase can be used, e.g. as has been done in the Ubitrack middleware agent [11, 90].

Network communication is another point for optimization. In a distributed, peer-to-peer system, the services and the service managers must exchange information on the state of each other and the system as a whole. Optimizing the protocols used for this exchange can be very beneficial. For example, the judicious use of multicast techniques can avoid needless repetition of sent network packets. Similarly, active change notifications are often better than polling for changes, and the frequency of change notifications should be limited. In the DWARF service manger, this is implemented in the ChangeNotifier class, which is responsible for sending notifications to AttributesChanged subscribers.

A final consideration for the efficiency of the middleware is the use of interprocess communication. Remote method invocations in CORBA use interprocess communication; this cannot be avoided if the services are on different machines. However, when two services are running on the same machine, a sensible optimization is to load them into the same process, using a loader service. This avoids excessive interprocess communication within the same machine.

Design At Run Time

In a continuous, incremental development process, users provide live feedback, enabling developers to improve the running system.

The adaptive service dependency architecture, together with its supporting middleware, allows an incremental, dynamic development technique called design at run time, described in this chapter. The core idea of design at run time is for users and developers to make changes to the system while it is running, improving it incrementally. This is necessary in a widely-deployed ubiquitous augmented reality scenario.

To support users and developers in this task, a number of development tools have been developed within the DWARF project. Design at run time is not a full-fledged software development process, nor has it been evaluated on a large scale. However, it has been useful in building several experimental systems.

7.1 A Development Technique for Ubiquitous Augmented Reality

Ubiquitous augmented reality systems suffer from the problem of ill-defined requirements: the requirements for a system are unclear from the beginning, and change over time.

This problem is partially addressed by the adaptive service dependency architecture: since the system is extensible, it can be deployed incrementally, and new services can be added as new requirements emerge.

However, the extensible architecture should be complemented by an incremental development process. Not only should the system be deployed incrementally, but it should be developed incrementally as well. Thus, parts of the system will be deployed—and running—before others are even developed. In consequence, I propose a development technique called *design at run time*: developers incrementally improve a system while users use it and provide feedback (Figure 7.1 on the following page). This follows the idea of ever shorter development cycles, as, for example, in extreme programming [9] or Scrum [122].

Below, two techniques that can be used in the process are described: jam sessions, where users and developers cooperate synchronously, and continuous extension, where they cooperate asynchronously. So far, jam sessions have only been tested on a small scale, within several projects using DWARF; and continuous extension is only a concept that has not been tested yet. Thus, both are described in less detail than the architecture or the middleware.

Also, design at run time is only a development technique, not a full *development process*. However, it could be incorporated in an agile development process such Scrum.

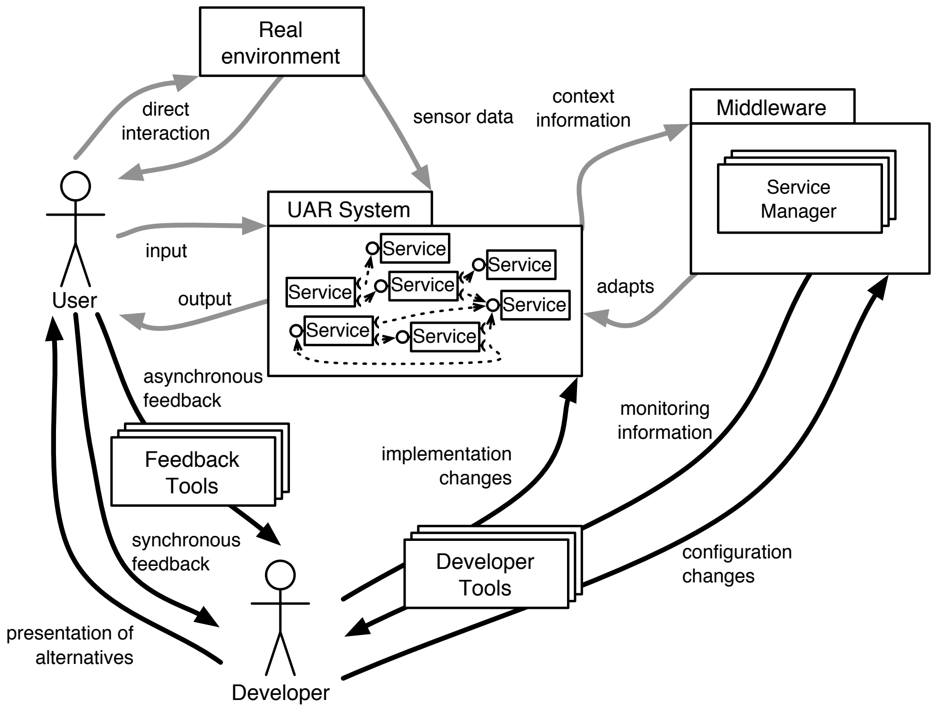


Figure 7.1: Design at run time.
Users and developers cooperate in improving the parts of a distributed system. For this, feedback and development tools are needed. The dark arrows are the development loop of Figure 4.1.

7.1.1 Jam Sessions

Inspired by the "on-site customer" from extreme programming [9], a *jam session* is a synchronous group activity of users and developers, cooperating in improving a running system, which is deployed within one room, or a small site consisting of several adjacent rooms.

A jam session takes from a few hours up to a whole work day. In developing a system, users and developers may wish to have several jam sessions consecutively. Projects such as TRAMP, SHEEP, ARCHIE and CAR have shown that this naturally tends to happen when the deadline for deployment or presentation of a system draws close.

Before a deadline, a jam session naturally is strenuous. However, beyond mere system integration tests, jam sessions can be used do generate ideas for new functionality, implement new features and test them. They are particularly valuable early in the development of a system, as they allow the early integration of components and the prototyping of user interfaces.

During development of a system, jam sessions do not have to be used exclusively; developers of difficult services may wish to spend a significant amount of time working alone or in pairs.

The basic unit of work in a jam session is the *feedback–develop cycle*. When a user finds a fault in a service or suggests an improvement, a developer either changes the service's configuration or changes its code, taking it off-line briefly, and restarting it while the rest of the system remains running. Conversely, a developer can implement several alternative versions or configurations of a service or a larger part of the system, and the user can test them and chose among them.

In a jam session, several of these develop–feedback cycles take place simultaneously, each involving at least one user and one developer. While one part of the system changes, the rest continues to run, and thus different parts of the system are improved simultaneously.

A jam session does not have to be centrally coordinated; it follows the idea of a jam session in music, where participants take the lead in turn. At times, one group of developers and users will have to defer a change to the system, because it would interfere with a test being performed by another group. This kind of group coordination requires a certain level of experience from developers and users.

Before a jam session, the goals should be agreed upon, as well as a time limit. The goals can be specific, such as "verify interaction scenario X", or vague, such as "explore interaction possibilities". After the jam session, users and developers meet and have a wrap-up discussion to reflect on the changes to the system and how well the goals have been achieved. This is similar to the process of Scrum.

A jam session requires support from the architecture and run-time infrastructure: it must allow components to be exchanged while the system is running. This is a feature that the adaptive service dependency architecture offers.

Jam sessions are particularly suitable for building prototypes of a system from existing components, and for evaluating and improving the usability of a system [71, 72]. Within a jam session, users may even discover ideas for new applications, allowing prototypes to be built on the spot. This is due to the fact that users and developers cooperate closely. A jam session is less suitable for the development of new algorithms within a specific component, or for performance tuning.

There is one significant difficulty in the application of jam sessions. Users and developers cooperating in a specific setting will quickly be able to develop and optimize prototypes for that particular setting. However, the system as a whole should remain adaptable. This means that developers must constantly try to keep the generality of their solutions in mind, rather than optimizing too closely for a particular setting.

A solution to this problem is to consciously try out different variations of the same scenario in a given jam session, and develop a solution that works well for all variants. Another is to have one jam session for prototyping first, and develop a solution for one particular setting, and then to have a second jam session which focuses on adaptivity.

7.1.2 Continuous Extension: Closing the Development Cycle

When a UAR system is deployed on a large scale, it is no longer possible for all users and developers to collaborate in the same room. At the same time, users discover problems with the system while they are involved in real-world tasks (such as maintenance or team action), where no developers are present. In contrast to jam sessions, the technique called *continuous extension* assumes that developers and users cooperate asynchronously, and that the system supports their collaboration. The basic idea remains that of concurrent develop–feedback cycles, except that these cycles are longer than in jam sessions.

Continuous extension assumes that a system has been (at least partially) deployed and is in use. When a user discovers a problem or an opportunity for improvement, he uses a feedback tool which is part of the system, and records a wish for improvement. The system stores this wish, together with the current user and system context—such as the state of services the user is using, the current graph of services, and the user's location. The system can also gather usage information automatically, by creating profiles of which services get used in which circumstances, whether the performance is adequate, or how long it takes the user to complete a task.

Using the information that the system has gathered, developers implement missing functionality, improve existing services, and fix bugs. This can happen off-line in a development laboratory. To test new and experimental features, they can even use jam sessions again, either with the same users who requested new functionality, or with selected pilot users. The new versions of services are then deployed to the hardware in the environment, and onto the users' mobile hardware. Assuming the services are backward-compatible, this can happen in the background, without users noticing. For this, the run-time infrastructure should include support for dynamic deployment.

Once the new services are deployed, users can take advantage of the improved functionality, and a new feedback–development cycle begins. As with jam sessions, many of these cycles are active at any given time, as users provide feedback asynchronously. Developers may wish to coordinate efforts, for example by performing new software roll-outs on a weekly schedule. In principle, many different developers can improve parts of the system in a decentralized fashion, without central coordination. However, as with jam sessions, developers and users should meet on a regular basis to discuss long-term changes to the system.

To the user, adaptability (as a feature of the architecture and middleware) and continuous extension (as a feature of the development process) are both mechanisms by which the system changes and improves; however, the time scale is different. When context changes or the user gives the system an appropriate command, it can adapt itself immediately—assuming it "knows" how. Assuming a speech recognition interface, the user can say, "show me a diagram of that engine", and a wall-sized display in the environment is reconfigured to show the diagram. On the other hand, the system may not be able to react immediately when the required functionality is missing. In this case, the system reacts by saying "sorry, I cannot do that yet", record the user's wish, and a developer implements the missing functionality. When the user tries the command again the next day, the system can fulfill it.

The concept of *end user programming* allows users to reprogram their ubiquitous computing systems with a simple visual interface. A simple example is a rule-based system: users define rules telling the system what to do in which circumstances. By contrast, in continuous extension, users do not program directly, but record requests for developers. This is based on the assumption that most users of a UAR system are not, by nature, programmers. Of course, continuous extension can be combined with end user programming; users can define simple rules and add simple functionality themselves, getting immediate results; and professional developers handle the more complex programming tasks.

Over a long time scale, continuous extension supports *emerging applications*: users discover new applications that can be built with deployed services, and developers improve services and build new services for these applications. Of course, this means that the design of old services can become outdated, necessitating re-factoring of individual services—or perhaps entire designs.

7.2 Development Tools

Development tools used in design at run time must support modifications to and evaluations of running systems. Thus, beyond classic development tools such as UML modelers and integrated development environments, additional tools are required—or existing tools need to be modified.

Ideally, developers should have a complete *tool suite* for jam sessions, allowing configuration, monitoring, authoring, development and debugging of services and systems. Figure 7.2 on the following page shows how several tools that have been developed in DWARF so far could be combined for usability evaluations. Similarly, a larger tool suite is needed for continuous extension, including user feedback tools, configuration management, bug tracking, rationale and traceability management.

Different tools are useful to different types of developers, as described in Section 2.5. This section presents tools needed for design at run time. Some have been developed in DWARF, others are presented as ideas.

7.2.1 System Monitoring and Reconfiguration

Infrastructures for distributed systems often include monitoring tools [61, 129], and these are especially important for visualizing the graph of distributed services in the adaptive service dependency architecture. Other important features for monitoring the system include observing the communication between services (such as the events they exchange) and viewing a service's configuration, i.e. its needs and abilities, as well as its attributes. A monitoring tool is especially useful to the application developer and maintainer, but also to the component developer.

An example of such as monitoring tool is the DIVE, the DWARF Interactive Visualization Environment, described in [41, 107]. DIVE takes advantage of the reflective features of the adaptive service dependency architecture. The DWARF service managers are modeled as services

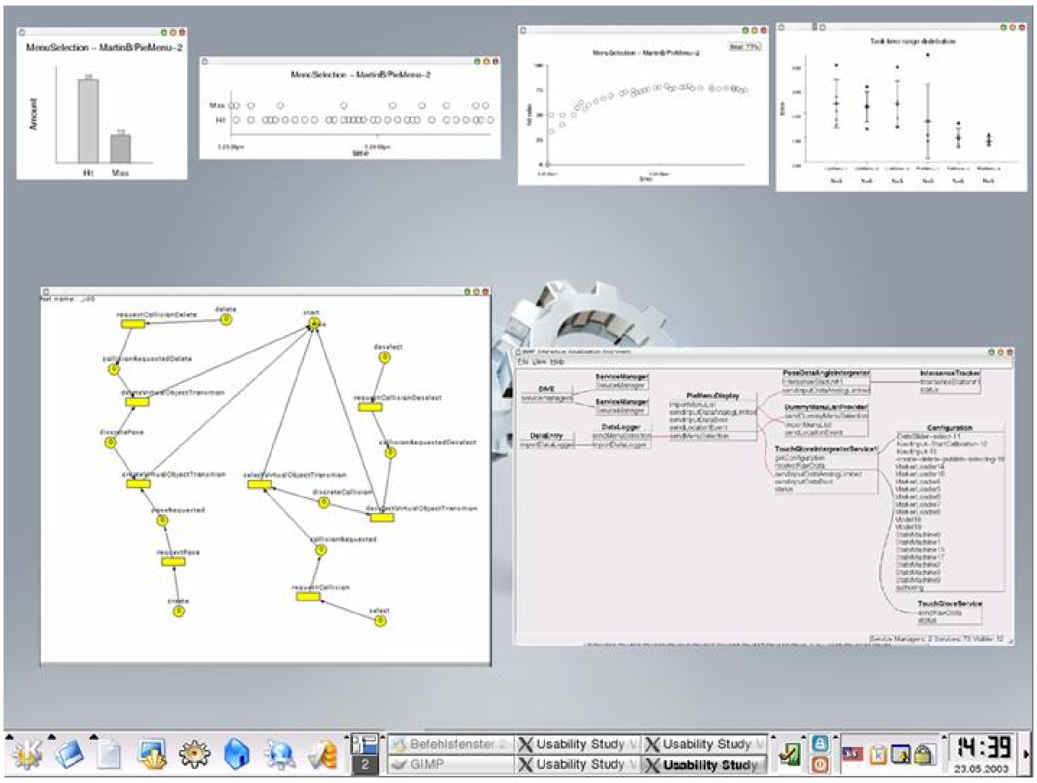
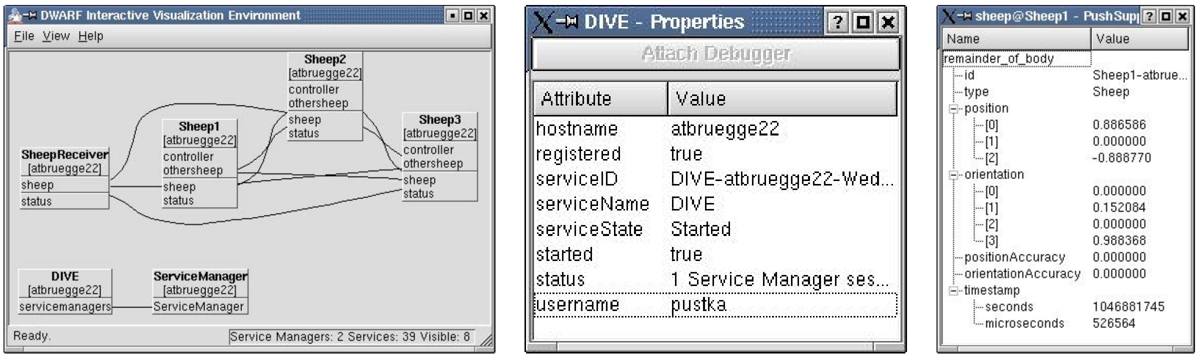


Figure 7.2: Mockup of a tool suite for run-time usability evaluation in jam sessions. Top row, graphs displaying the user's performance with a new input device, showing hit/miss ratios in selecting a given menu entry. Bottom left, the DWARF UI controller showing a Petri net with the current state of a multi-modal user interface, allowing the interface to be modified at run time. Bottom right, the dynamically updated graph of communicating services, displayed in DIVE. (Figure from [72].)

themselves, with an ability to provide information on the system state, and DIVE is realized as a DWARF service with a need for all service managers in the system. Screen shots of DIVE are shown in Figures 7.3 to 7.5 on pages 127–128.

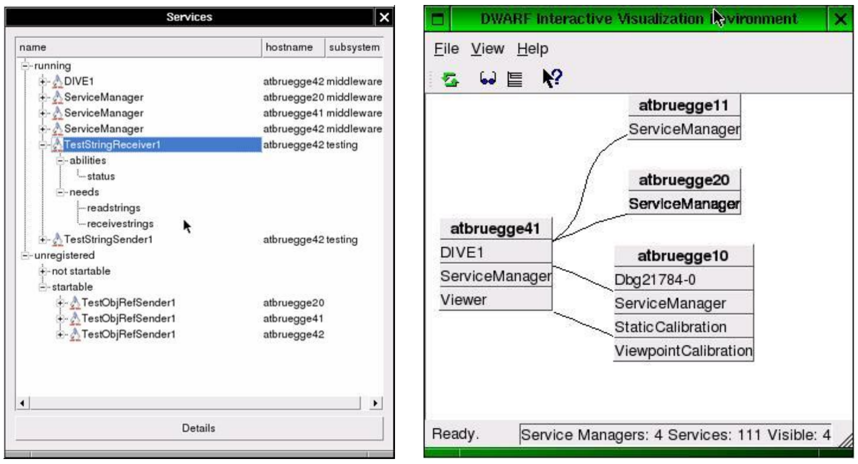
In design at run time, beyond merely monitoring the distributed system, developers must be able to reconfigure it on the fly. In DIVE, developers can change service descriptions of running services, editing abilities' attributes and needs' predicates (Figure 7.5 on page 128). This causes the middleware to reconfigure the graph of services. Developers must be able to save service descriptions persistently as well, so that the system configuration will be restored after a restart. Configuring the system as a whole is most important to the application developer.


DWARF Interactive Visualization Environment
File View Help
Sheep2 [atbruegge22] controller othersheep sheep status
Sheep1 [atbruegge22] controller othersheep sheep status
Sheep3 [atbruegge22] controller othersheep sheep status
SheepReceiver [atbruegge22] sheep status
DIVE [atbruegge22] servicemanagers
ServiceManager [atbruegge22] ServiceManager
Ready.
Service Managers: 2 Services: 39 Visible: 8
DIVE - Properties
Attach Debugger
sheep@Sheep1 - PushSup

Attribute	Value
hostname	atbruegge22
registered	true
serviceID	DIVE-atbruegge22-Wed...
serviceName	DIVE
serviceState	Started
started	true
status	1 Service Manager ses...
username	pustka

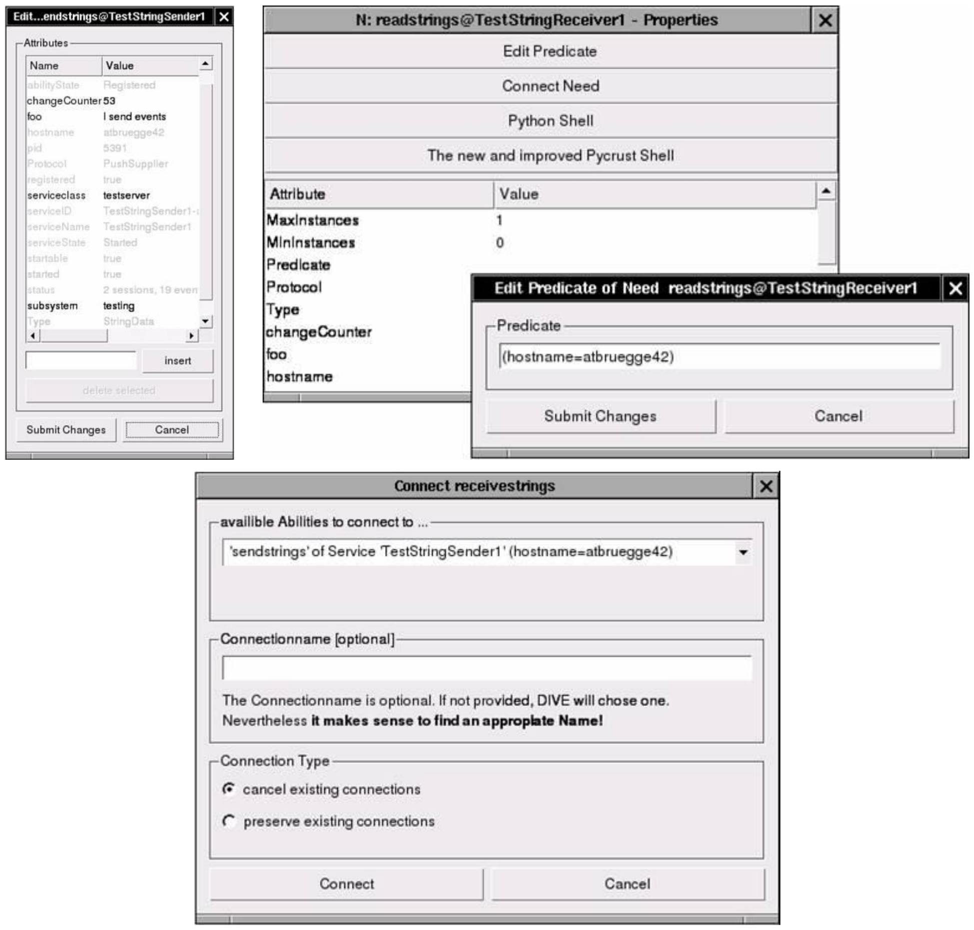
Name	Value
remainder_of_body	
id	Sheep1-atbrue...
type	Sheep
position	
[0]	0.886586
[1]	0.000000
[2]	-0.888770
orientation	
[0]	0.000000
[1]	0.152084
[2]	0.000000
[3]	0.988368
positionAccuracy	0.000000
orientationAccuracy	0.000000
timestamp	
seconds	1046881745
microseconds	526564

Figure 7.3: Monitoring with DIVE, the DWARF Interactive Visualization Environment.
Left, graph of communicating services (three Sheep and one SheepReceiver, as well as DIVE itself and the service manager). Center, a window showing the attributes of the DIVE service itself. Right, a window showing the contents of a *PoseData* event sent by a Sheep service. (Figures from [107].)


Services
Details
DWARF Interactive Visualization Environment
File View Help
atbruegge11 ServiceManager
atbruegge20 ServiceManager
atbruegge41 DIVE1 ServiceManager Viewer
atbruegge10 Dbg21784-0 ServiceManager StaticCalibration ViewpointCalibration
Ready.
Service Managers: 4 Services: 111 Visible: 4

name	hostname	subsystem
running		
DIVE1	atbruegge42	middleware
ServiceManager	atbruegge20	middleware
ServiceManager	atbruegge41	middleware
ServiceManager	atbruegge42	middleware
TestStringReceiver1	atbruegge42	testing
abilities		
status		
needs		
readstrings		
receivestrings		
TestStringSender1	atbruegge42	testing
unregistered		
not startable		
startable		
TestObjRefSender1	atbruegge20	
TestObjRefSender1	atbruegge41	
TestObjRefSender1	atbruegge42	

Figure 7.4: Alternate system visualizations with DIVE.
Left, tree view of services, grouped by state and network host. Right, graphical group view of services, grouped by network host. (Figures from [41].)



Edit...endstrings@TestStringSender1

Attributes

Name	Value
abilityState	Registered
changeCounter	53
foo	I send events
hostname	atbruegge42
pid	5391
Protocol	PushSupplier
registered	true
serviceclass	testserver
serviceID	TestStringSender1-
serviceName	TestStringSender1
serviceState	Started
startable	true
started	true
status	2 sessions, 19 even
subsystem	testing
Type	StringData

N: readstrings@TestStringReceiver1 - Properties

Attribute	Value
MaxInstances	1
MinInstances	0
Predicate	
Protocol	
Type	
changeCounter	
foo	
hostname	

Edit Predicate of Need readstrings@TestStringReceiver1

Predicate: (hostname=atbruegge42)

Connect receivestrings

availible Abilities to connect to ...: 'sendstrings' of Service 'TestStringSender1' (hostname=atbruegge42)

Connectionname [optional]

The Connectionname is optional. If not provided, DIVE will chose one. Nevertheless **it makes sense to find an approplate Name!**

Connection Type

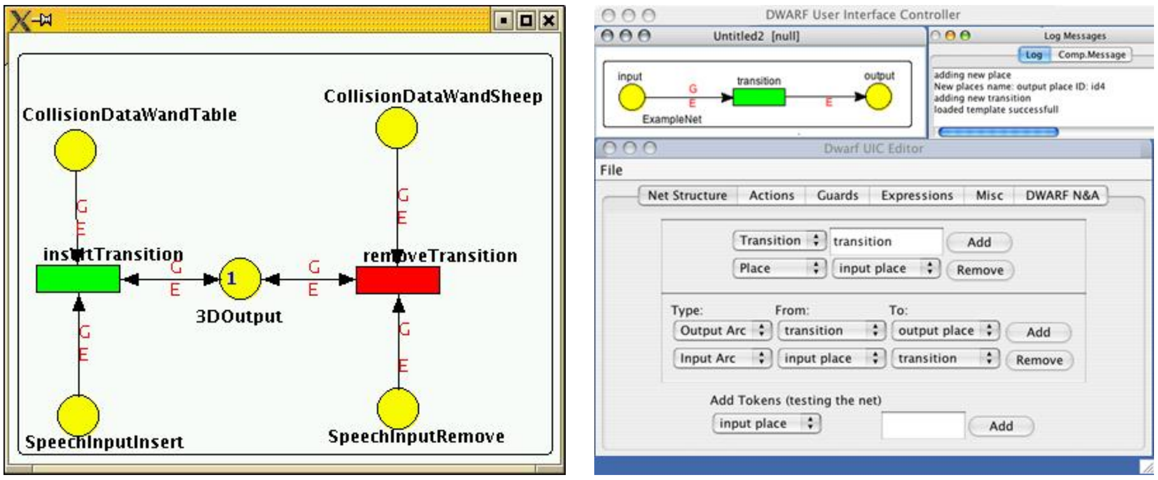
[✓ cancel existing connections]

preserve existing connections

Figure 7.5: Configuration with DIVE.
Top, dialog for editing attributes of an ability and dialog for editing predicates of a need. Bottom, dialog to manually connect a need to a specific ability, by modifying both the need's predicate and the ability's attributes. (Figures from [41].)

7.2.2 Authoring and Component Configuration

Both application developer and user interface designer need additional tools for designing applications and their interactive content. This requires many different tools—for example for 3D modeling. For UAR, new tools are necessary, e.g. to model complex multi-modal interaction, as in the case of the DWARF UI Controller (Figure 7.6). In contrast to classical authoring tools, for jam sessions, authoring tools must allow modifications to a running system.



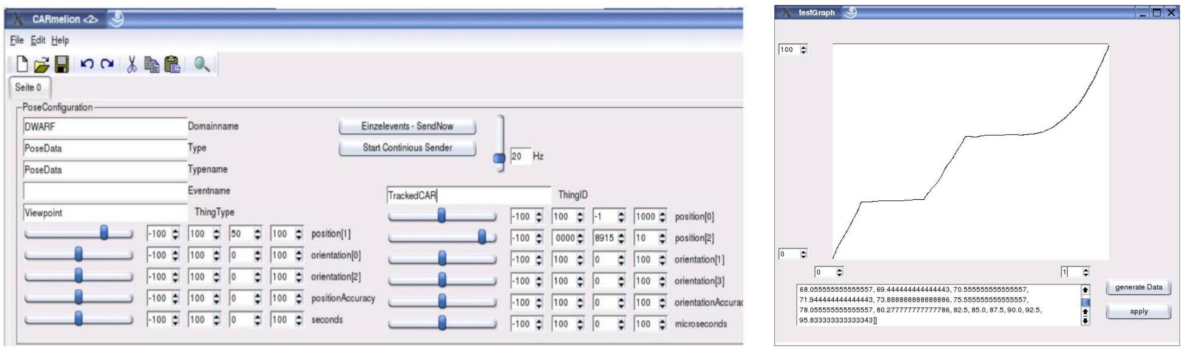
```
adding new place
New places name: output place ID: id4
adding new transition
loaded template successfull
```

Figure 7.6: Screenshots of the DWARF UI Controller for multi-modal interaction. Left, a Petri net modeling multi-modal interaction. Right, Petri net editor and dialog for adding and removing transitions and places. (Figures from [55].)

Part of the process of building an application is configuring individual services. For this, appropriate configuration tools are needed. An example of a configuration tool is *CARmelion* [28], developed in the CAR project. This service can send arbitrary events to a service that needs to be configured (Figure 7.7 on the next page).

Using DIVE and CARmelion, a developer can configure an arbitrary service by selecting "configure this service" in DIVE, and DIVE creates a service description for CARmelion so that CARmelion's ability matches the configuration need of the service to be configured. CARmelion then determines which type of events to send by inspecting its own service description and looking up the event type definition in an interface repository [28, 41].

A user interface designer can also use graphical tools to modify the user interface. For example, CARmelion also supports the graphical drawing of one-dimensional functions that can be used to adapt user feedback.



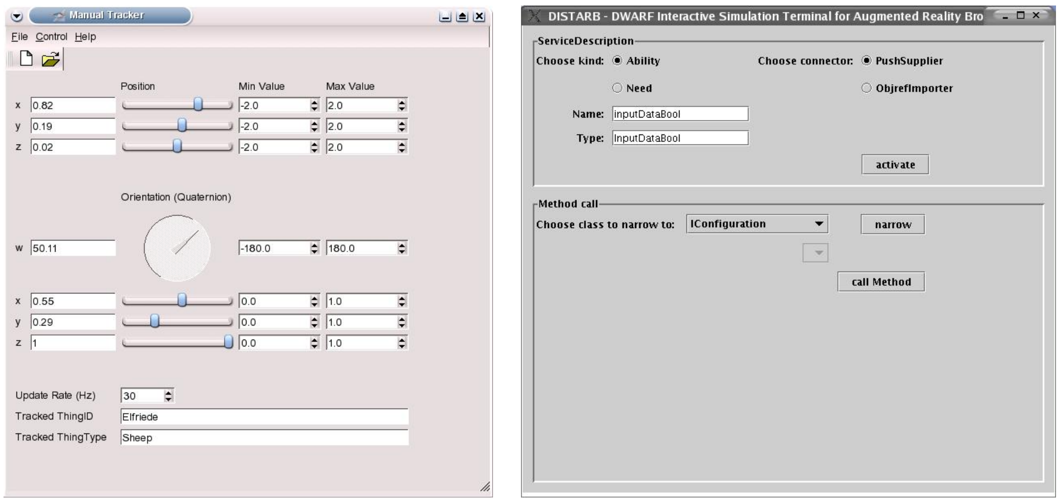
DWARF Domainname
PoseData Type
PoseData Typename
Eventname
Viewpoint ThingType
20 Hz
TrackedCAR ThingID

-100	100	50	100	position[1]
-100	100	0	100	orientation[0]
-100	100	0	100	orientation[2]
-100	100	0	100	positionAccuracy
-100	100	0	100	seconds
-100	100	-1	1000	position[0]
-100	0000	8915	10	position[2]
-100	100	0	100	orientation[1]
-100	100	0	100	orientation[3]
-100	100	0	100	orientationAccurac
-100	100	0	100	microseconds

100
0
0
1

```
68.055555555555557, 69.444444444444443, 70.555555555555557,
71.944444444444443, 73.888888888888886, 75.555555555555557,
78.055555555555557, 80.277777777777786, 82.5, 85.0, 87.5, 90.0, 92.5,
95.833333333333343]]
```

Figure 7.7: Screenshots of CARmelion, a service configuration tool in DWARF.
Left, a user interface for sending events that has been generated from a CORBA event definition. Right, a graph view to adapt the magnification of a 3D map in in response to the user's distance to his target. (Figures from [28].)



	Position	Min Value	Max Value
x	0.82	-2.0	2.0
y	0.19	-2.0	2.0
z	0.02	-2.0	2.0

Orientation (Quaternion)

w	50.11	-180.0	180.0
x	0.55	0.0	1.0
y	0.29	0.0	1.0
z	1	0.0	1.0

Update Rate (Hz) 30
Tracked ThingID Elfriede
Tracked ThingType Sheep

Choose kind: [✓ Ability] Need
Choose connector: [✓ PushSupplier] ObjrefImporter
Name: inputDataBool
Type: inputDataBool
Choose class to narrow to: IConfiguration

Figure 7.8: Screenshots of testing services: DISTARB and the Manual Tracker service.
Left, the DWARF Manual Tracker service, which simulates a tracker by generating pose data according to the slider positions the user sets. Right, the DISTARB testing service, with a dialog to invoke arbitrary CORBA method calls. (Figures from [80] and [134].)

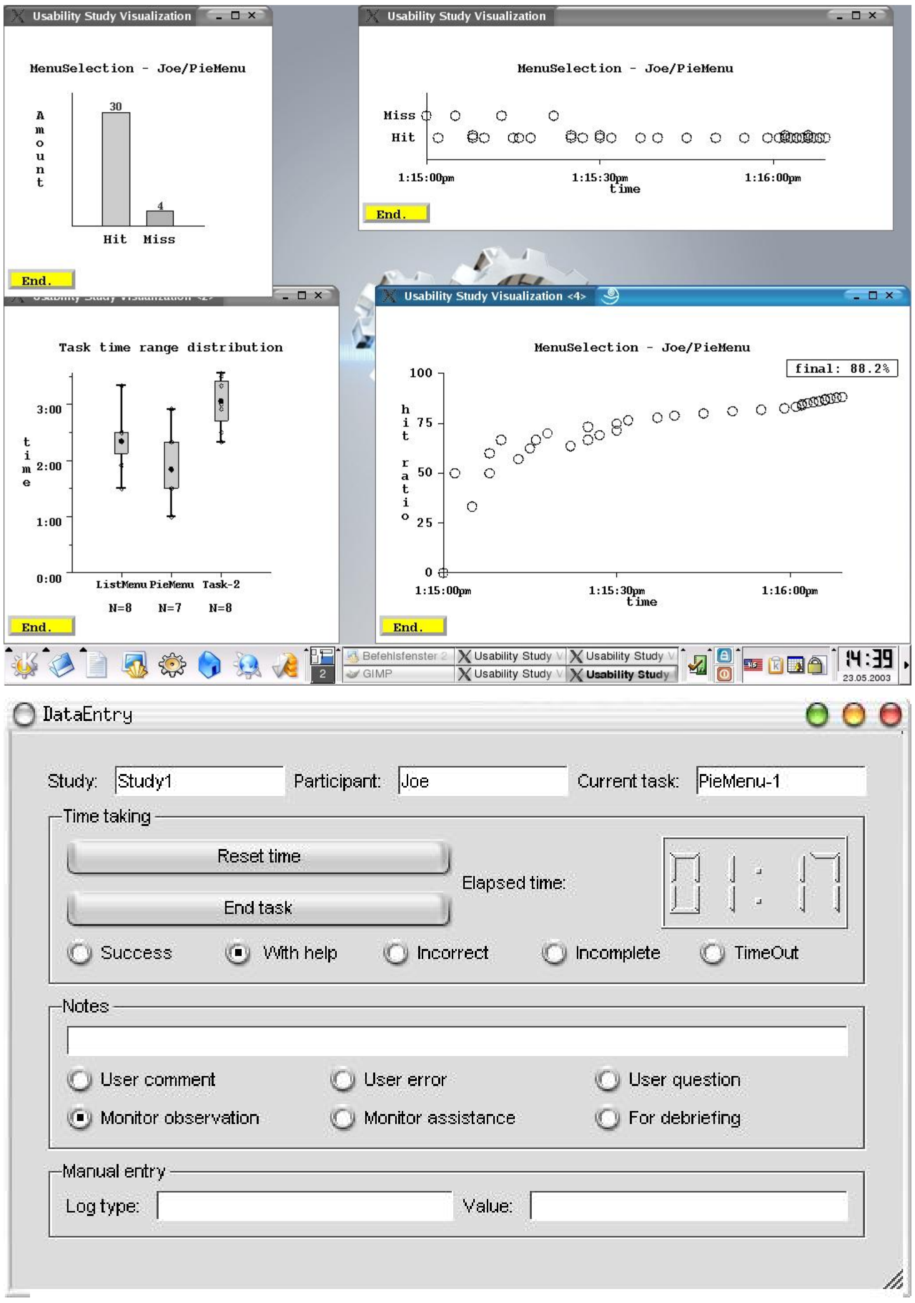
7.2.3 Testing, Evaluation and Feedback

A component developer wishing to test a single service needs tools that can simulate other services, in order to reproduce test scenarios. For example, DWARF includes several services that can be used as test drivers (Figure 7.8 on the facing page). DISTARB [134] lets developers manually send events and invoke methods; the manual tracker service simulates a tracker, by sending pose data that the developer can adjust using sliders.

For evaluating the usability of a system in jam sessions, usability testing tools that provide the usability tester with immediate feedback are needed. Several of these were developed in the ARCHIE project [72], and further development is the subject of ongoing research [117]. Figure 7.9 on the next page shows some of these tools; here, the usability tester is presented with an up-to-date graph of the user's performance (i.e. how often he correctly selected a menu entry using an experimental input device).

For continuous extension, users need some method of providing feedback to developers. These tools can be designed so that they fit naturally into the UAR environment. For example, if the application involves placing virtual annotations in a power plant, the user can use the annotation-placing functionality to add feature requests. If the system uses voice recognition, it can record the user's spoken wish as a sound file.

An important research issue is how to link user wishes to implemented services. For this, developers and users should have access to a single model that links requirements elicitation entities (user wishes, usage examples), requirements analysis entities (user tasks, use cases), system design entities (services, needs, abilities), and implementation entities (processes, event channels). For example, the *Requirements Analysis Tool (RAT)* and its web-based interface *REQuest* [143] were used in several projects using DWARF. Combining the rationale model of REQuest and RAT with the service model of the adaptive service dependency architecture would provide a powerful tool for continuous extension.



Study: Study1 Participant: Joe Current task: PieMenu-1

Success [✓ With help] Incorrect Incomplete TimeOut

User comment User error User question
[✓ Monitor observation] Monitor assistance For debriefing

Log type: Value:

Figure 7.9: Usability test tools, as used in the ARCHIE project.
Top, graphs updated in real time to reflect the hit/miss ratio of a tested input device. Bottom, a data entry dialog for the usability developer, allowing timing of experiments and entry of comments. (Figures from [71].)

Results

The architecture has been successfully applied in building the software framework DWARF, its supporting middleware, and many prototype systems.

The adaptive service dependency architecture, its middleware for decentralized service management, and the development technique of design at run time have been applied and implemented; this chapter describes the results.

Our software framework for ubiquitous augmented reality, DWARF, takes advantage of the architectural concepts, and thus has become the most flexible augmented reality framework available. An integral part of the framework is a powerful middleware implementation. For developers, it includes several tools and tutorials.

Several experimental systems have been built using the framework, and also using design at run time. This, together with several performance measurements of the middleware implementation, and feedback from users and developers, shows that the concepts are viable—at least to build experimental systems.

8.1 Implemented Framework

DWARF, the Distributed Wearable Augmented Reality Framework, is an instance of the adaptive service dependency architectural style.

Since 2000, approximately 130 computer science students at the Technische Universität München have worked with DWARF in the third or fourth year of their studies. This includes bachelor, Diplom and masters' theses, semester projects, and software engineering and AR lab courses. Additionally, 6 full-time graduate students (including myself) have worked on improving and extending DWARF. A small number of developers outside of TUM have also used DWARF in their research. In this process, DWARF has been used to build more than 10 different AR and UAR research prototypes, showing the architecture's viability.

Since the first ideas in 2000, DWARF has evolved significantly, although several concepts have stayed the same. The first prototype system to be built with DWARF was Pathfinder, described in [5] and [109]. This already included the basic concepts of services, needs and abilities, described in [77]. This version of the framework, DWARF 2001, was mostly conceptual. It included prototypical implementations of several basic services and of the service manager, but no development tools; and the concepts for adaptivity were severely limited.

Starting in spring of 2001, several research projects used DWARF to build experimental systems, and contributed to the framework. Thus, the concepts supporting the framework were improved and extended over time, mainly by graduate students, and many new services, including development tools, were implemented, mostly as student projects. Also, the middleware was extended and the framework was ported to several different operating systems. The framework evolved in the process of building several applications with it, such as TRAMP, SHEEP, ARCHIE, and CAR. In each of these group projects, the developers identified the necessity of new services, and extended existing ones. Major areas of development were the tracking services, including tracking device adapters and calibration; the viewer service, performing 3D rendering; and the interaction services, including input device adapters and multi-modal interaction.

DWARF, now evolved to *DWARF 2004*, has proven a powerful basis for implementing prototype AR and UAR systems, and a useful research platform. Besides its unique target domain of ubiquitous augmented reality, its major strengths are its flexibility, allowing new systems to be built quickly, and its ability to integrate many different types of hardware into one large system. These features have made DWARF attractive to other AR and UAR research groups. For example, there has been some effort to integrate Studierstube and DWARF [7].

The current version of DWARF includes many services, listed in Section A.2.

8.2 Implemented Middleware

Many of the middleware concepts described in Chapter 6 have been implemented within the DWARF middleware (Section A.1), which is one of the most basic elements of the framework. This middleware implementation has allowed the construction of flexible systems, and testing and improvement of the architectural concepts.

The DWARF middleware has evolved along with the rest of the framework. Most new architectural concepts required support from the middleware before they could be tested. The original prototype implementation in DWARF 2000 consisted only of a CORBA implementation, a rudimentary service manager and a CORBA notification service. Distributed service discovery was not implemented; thus, although the service manager had been designed to run on every computer in a DWARF network, the Pathfinder demonstration system in fact included only one service manager.

The first major improvement to the DWARF middleware, implemented within the TRAMP project, was the addition of the SLP locator, allowing service discovery between distributed service managers. The lessons learned from TRAMP led to a major restructuring in 2002. This included refactoring of several interfaces, the addition of interfaces for each phase, the implementation of the XML service describer, and on-demand loading of services using the default service loader. Furthermore, the middleware included the first support for template services. These new features were all tested within the SHEEP project.

Since the SHEEP project, the middleware interfaces have remained backwards-compatible, meaning that all implementations of DWARF services since 2002 can run unchanged with the newest middleware implementation from 2004. New interfaces have been added, however, particularly to support the phases of the context level, and an additional communication protocol (shared memory). Furthermore, the service manager has several new features allowing services to be notified of changes to service descriptions; this is particularly useful to monitoring tools such as DIVE. Most other implementation improvements to the middleware since 2002 addressed performance, scalability and stability.

Several important architectural concepts are not yet supported by the DWARF middleware, as is described in Section A.1. Attribute types are limited; predicates are limited to boolean expressions; penalties and distances are not supported; and binding and induction specifications are quite limited. Similarly, many middleware features that would allow greater scalability and higher performance have not been implemented, such as locator hierarchies or in-process connector factories. Since systems built with DWARF so far have not required these advanced features or this degree of performance and scalability, this has not (yet) been a problem. However, further research into UAR systems with DWARF will require further implementation of the DWARF middleware.

8.3 Developer Support

To support the process of design at run time, as described in Chapter 7, DWARF now includes several run-time developer tools. In addition, several tools have been implemented to increase developer productivity and ease the learning curve of a complex distributed software framework. These tools are motivated in Section 7.2.

The most important development tool for DWARF is DIVE, shown in Figure 7.3 on page 127; this lets developers monitor the distributed system. The necessity of DIVE was highlighted in the TRAMP project, a course with 50 students, who had great difficulties in developing a distributed system without visualization tools. As DIVE was developed in 2002, it proved very useful in the SHEEP project and has been used in all subsequent DWARF projects. Other important developer support tools include the DWARF UI Controller (Figure 7.6 on page 129), and several testing services such as the manual tracker and DISTARB (Figure 7.8 on page 130). These services are listed in Table A.11.

There are two types of services used for configuration in DWARF. First, services allowing developers to tune and test a system during design at run time. Second, services that are deployed in the environment, with configuration values that developers have adjusted, so that mobile devices are configured appropriately.

An important addition to DWARF in 2003 was support for the *Python* programming language, which allows simple scripting. Services can now be written in Python, and developers can write services with minimal effort—even interactively. This is particularly useful for design at run time.

To ease the learning curve, DWARF now includes several tutorials for the middleware, build environment, and most important services. Similarly, on-line documentation allows new developers to start working with the framework quickly. Still, when introducing new students to DWARF, several weeks are necessary for them to become productive with the framework.

8.4 Implemented Systems

This section briefly describes the most important systems that have been built with DWARF. The systems are presented in chronological order, with their application domain, the project scope, and lessons learned for architecture, middleware and process.

Pathfinder The first DWARF system, *Pathfinder*, was developed in the fall of 2000. It is described in [5] and [109]. The Pathfinder application consisted of a navigation scenario; a pedestrian user navigated both outdoors and indoors, on the former TUM campus in the center of Munich. In addition, the wearable system showed the user which of several printers his print job had been printed on.

The main purpose of the Pathfinder system was to demonstrate the feasibility of DWARF. It was a distributed augmented reality system combining services on a wearable system with stationary services in the environment. Pathfinder showed that the basic architectural concepts of services, needs and abilities were at least workable, and tested the first middleware implementation. It proved the value of using loose coupling, as the services of the final system—which were developed as separate student projects—could be integrated late in the project, and the demonstration scenario was adapted to fit the implementation status of the services.

The development of Pathfinder did not use any of the concepts of design at run time, nor were there any development tools for DWARF. The lack of these tools was not a significant problem, however, since the size of the project and of the system was fairly small.

Fata Morgana The *Fata Morgana* [67] project investigated the use of augmented reality in automobile exterior design. The goal was to present virtual mockups of new automobile bodies next to classical clay mockups.

In order render an image of sufficient quality for designers to evaluate subtle effects such as reflections on a curved surface, the Fata Morgana system integrated a high-quality commercial rendering system. DWARF was used to build a proof-of-concept prototype; a second prototype, using the customer's proprietary software, was built without DWARF.

The Fata Morgana system was the second system to be built with DWARF, and most of the original developers of DWARF were not involved directly in this project. The services developed for Pathfinder were not reused in Fata Morgana, with the exception of a world model service. However, the middleware from DWARF 2001, especially the service manager, was reused, and was ported from Linux and Windows to Mac OS X. Thus, the architecture and the middleware of DWARF 2001 proved to be reusable.

TRAMP The scenario of the *Traveling Repair and Maintenance Platform (TRAMP)* project was to guide a mechanic who is equipped with a wearable computer to a customer who has a car breakdown. The mechanic is then instructed by the system how to repair the car.

In terms of the number of developers (50), TRAMP was the largest project to use DWARF. This particularly highlighted the necessity of good training material for student developers, and of development tools that would allow many developers to simultaneously work on a distributed system. TRAMP was the source of many ideas that led to the concept of design at run time.

In TRAMP, several new services were integrated into the framework, some of which were subsequently reused (although some had to be improved in terms of stability first). In TRAMP, the middleware was extended with the SLP locator, allowing a fully decentralized system.

FixIt In the *FixIt* [29, 68] project, the scenario was to inspect a simple robot (in this case, built as a Fischer Technik model) and compare it with the virtual model of a simulated robot. By superimposing the virtual over the real robot, a mechanic can easily detect and diagnose malfunctions.

FixIt used the current version of DWARF after the TRAMP project, including several services and the middleware. Compared to Fata Morgana, DWARF was somewhat easier to learn by now, as simple helper libraries shielding developers from some of the more complex details of CORBA had been developed in TRAMP, and students had prepared training material.

On the architectural side, the basic DWARF architecture remained valid, although in a sense, it was "overkill" for the FixIt system, since there was no mobility involved. However, the distributed nature of DWARF made it comparatively easy to integrate the external robot interface.

SHEEP SHEEP, described in Section 4.6.1, was developed entirely with DWARF, and the project served as an opportunity for major redesign of several components. New tracking, presentation and interaction services were developed, the middleware interfaces were refactored, and the first development tools (DIVE, manual tracker) were introduced. The current implementation of the DWARF middleware remains compatible to the version used in SHEEP.

With these tools available, the idea of jam sessions arose naturally out of integration activities. All developers met in the lab to simultaneously improve the running system. Even when the demonstration system was being presented at the conference, we could make last-minute adjustments.

SHEEP was a major success for the DWARF project; in demonstration of the framework to other AR researchers, it was widely recognized as one of the most flexible and most highly distributed AR systems ever built.

PAARTI The *Practical Applications of Augmented Reality in Technical Integration (PAARTI)* project involved a cooperation with an industrial partner, BMW, in spring 2002 [30].

It dealt with the prototypical design and implementation of an "intelligent welding gun"—a tracked welding gun equipped with a display which helps welders navigate to find and shoot studs with high precision in experimental vehicles. The setup was tested by a number of welders at BMW. It showed significant time improvements over the traditional process of stud welding and was well accepted by the workers.

ARCHIE The ARCHIE project (Section 4.6.2) dealt with the collaborative design of buildings, involving architects, engineers and customers. The collaboration took part in several rooms, allowing users with mobile computing equipment to roam between rooms and use different stationary services installed there. Thus, ARCHIE was one of the first applications built with DWARF that fulfilled all of the UAR definition points, including the availability in a large physical environment, and it showed the suitability of the adaptive service dependency architecture in such an application.

Similarly to SHEEP, ARCHIE was an opportunity to improve DWARF significantly. For example, the Viewer service currently used in DWARF 2004 was developed within ARCHIE [54], and several advanced concepts of the middleware, such as the selector service, were designed and implemented [126]. ARCHIE was also the source of several ideas for the context and graph levels of the architecture of Chapter 5, and improved and elaborated upon the idea of jam sessions, adding usability evaluations at run time [71, 72]. ARCHIE also reused many services (and the middleware) that had been developed during the SHEEP project, showing the framework's reusability.

HEART Two different medical applications in heart surgery were part of the *Heart surgery Enhanced by Augmented Reality Techniques (HEART)* project, involving a cooperation with the TUM university hospital in summer 2003.

One application was the visualization of the optimal placement of ports for minimally invasive surgery [135]; the other was assistance in the placement of aortic stents [8]. HEART used DWARF as a platform for designing prototype systems that were then evaluated together with physicians. The services developed for ARCHIE were reused without any source code modification in HEART, showing the value of reusable services.

NAVI The *Navigation for the Visually Impaired (NAVI)* system was developed in fall 2003 [49, 113]. It was an unusual augmented reality application in that it uses audio and tactile output rather than visual augmentation. The input was designed in a way that the user knows, even without seeing anything, where in the menus he is and how he can get to where he wants to be. While the user navigates to his destination, the system informs him of interesting points (stores, etc.) on the way, using contextual data filtering to guess which information the user wants to know and which information isn't of interest.

NAVI was implemented with DWARF, showing the architecture's reusability even in an application of UAR that does not use visual augmentation.

CAR The CAR project (Section 4.6.4) created a collaboration for the design of visualizations and interaction metaphors to be used in the next-generation cars with heads-up displays.

CAR again contributed several services to DWARF, such as an improved user interface controller with a run-time prototyping interface (Figure 7.6 on page 129, [55]). Additionally a dynamically configurable set of filters (each having an appropriate UI for tuning parameters, Figure 7.7 on page 130) was developed, which can be easily instantiated and deployed with DIVE.

An important result of the CAR project was to show the viability of the idea of design at run time, letting designers, psychologists and computer scientists collaboratively improve prototypical user interfaces.

Ubitrack The Ubitrack project (Section 4.6.3) has the goal of dynamically combining different position and orientation tracking sensors to enhance both the range and the accuracy of existing tracking technology, allowing ubiquitous augmented reality applications.

Within Ubitrack, many of the DWARF tracking services were redesigned, and the middleware was extended with additional functionality for searching graphs of spatial relationships. The Ubitrack project extended several ideas of Chapter 5, showing the usefulness of dynamically adapting graphs of services.

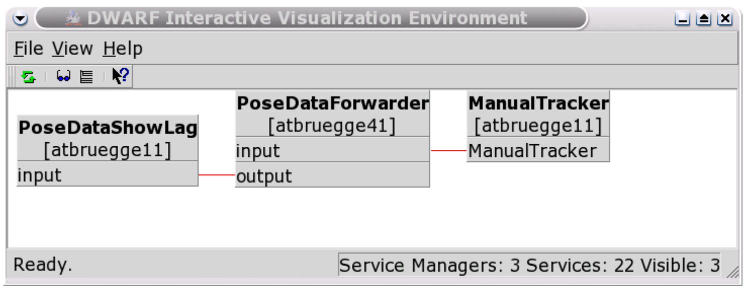
8.5 Performance Measurements

This section describes the results of several performance measurements performed with the current DWARF implementation. They show that the middleware's performance is adequate for building experimental UAR systems. Of course, there are several areas in which the performance needs to be improved before use in real-world systems. These are, however, mainly implementation issues and not fundamental architectural problems.

Event latency In contrast to many other AR systems which use proprietary UDP-based communication protocols, DWARF uses an implementation of the CORBA notification service. Using this "heavyweight" middleware for events that have to be delivered in near real time can, in principle, cause performance problems.

The first concern here is event latency: how long does it take for an event to be sent between two services running on two different machines? How does this affect the problem of performance?

To test this, one simple experiment (Figure 8.1 on the next page) used two networked computers, exchanging *PoseData* events at 30 Hz. A Manual Tracker service on the first computer sends events to a PoseData Forwarder service on the second, which forwards the events back immediately to a PoseData Show Lag service on the first computer. Thus, the event latency over two network hops can be measured on the first computer, avoiding timing inaccuracies due to unsynchronized system clocks. Figure 8.2 on the following page shows the results of this experiment, using both wired and wireless ethernet, as well as a variation where the services are running on the same machine. The laptops and workstations used for the experiment had CPU speeds of approximately 1 GHz.



PoseDataShowLag [atbruegge11] input

PoseDataForwarder [atbruegge41] input output

ManualTracker [atbruegge11] ManualTracker

Ready. Service Managers: 3 Services: 22 Visible: 3

Figure 8.1: Experimental setup to measure event latency.
Events are sent from the Manual Tracker service via a (possibly remote) PoseData Forwarder service to a PoseData Show Lag service on the same machine as the Manual Tracker.

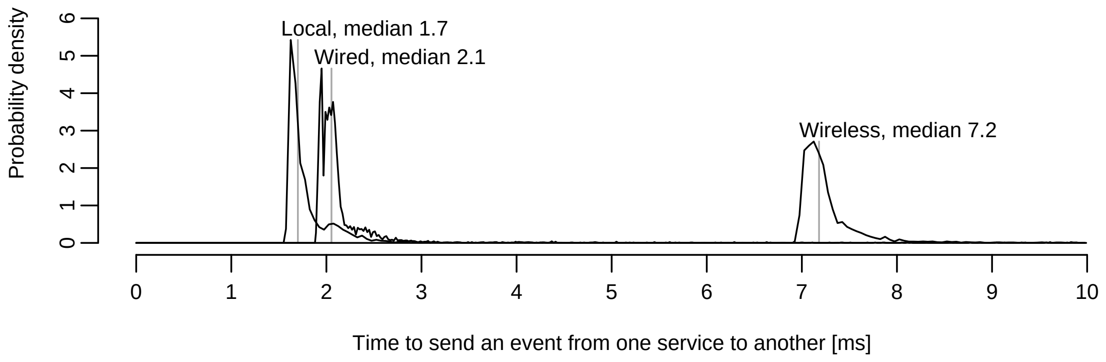


Figure 8.2: Event latency in several deployments.
Above, experimental setup: an event is sent from the Manual Tracker service via a (possibly remote) PoseData Forwarder service to a PoseData Show Lag service on the same machine as the Manual Tracker. Below, graph of average latency for one event hop when using local communication within a single machine, 100-megabit wired ethernet and 11-megabit wireless ethernet.

On average, sending an event incurs less than 2 milliseconds of latency that are due to the middleware. This is acceptable for the purposes of UAR, as long as events do not have to be sent along extremely long “chains” of services. In a typical SHEEP-type system, the tracking data takes two to three hops between services until it is received by the viewer service, adding up to a latency of 4 to 6 milliseconds. Note, however, that the distribution of event latencies extends far to the right in the diagram; some events have significantly higher latencies. Thus, the current implementation does not meet “hard” real time requirements. Also, with 2 ms of processing time per event hop, the middleware can process no more than 500 events per second on a 1GHz machine. This figure could probably be improved by reducing interprocess communication, using colocated services.

Memory usage on small systems Using “fat” middleware such as CORBA on thin platforms such as palmtops has worked quite well. We were able to use OmniORB on a Compaq iPaq with 32 MB of main memory, and middleware performance was no problem, especially compared to 3D rendering or video on a palmtop. Figure 8.3 shows the memory usage of the middleware as measured on the palmtop.

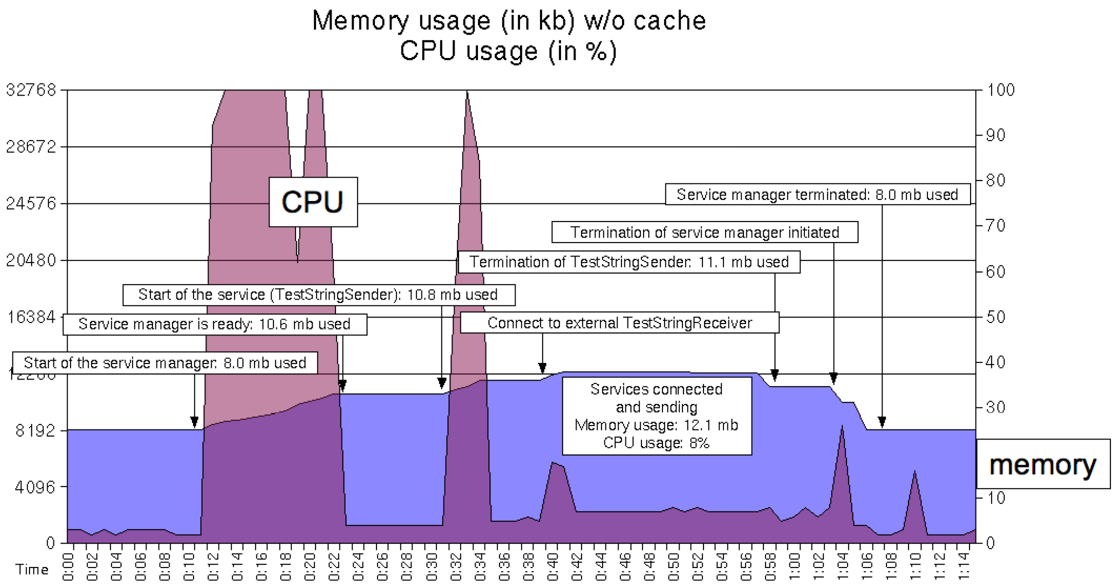


Figure 8.3: Memory usage of DWARF on a palmtop system.
One service and the service manager are started on a Compaq Ipaq with 32 MB of main memory. When the service is running and communicating with an external service, the middleware and the services use a total of 4 MB of memory. The peak in CPU usage is the startup process of the service manager. (Figure from [125].)

Distribution over many hosts, scalability Scalability is a major issue for ubiquitous computing, and thus also UAR, systems. The current implementation of our peer-to-peer middleware does not scale as well as it could, with additional implementation work.

Figure 8.4 shows the results of an experiment simulating a user entering a busy room of services. From 4 to 32 sheep simulation services are running on 2 or 4 computers. Each service communicates with each other service (an unusual case designed to stress the middleware), creating up to 1024 simultaneous connections between services. Now, a sheep simulation service on a separate computer changes its attributes and predicates so that it will be connected to the other services. The experiment measures the elapsed time until all connections are set up. This should not take more than a few seconds, since this is equivalent to a user entering a new room.

As the number of services in the network increases, the middleware takes longer to connect the services together. Note, however, that for the same number of services, distribution is beneficial: the setup with 32 services performs better when distributed over 4 hosts than when distributed over 2 hosts.

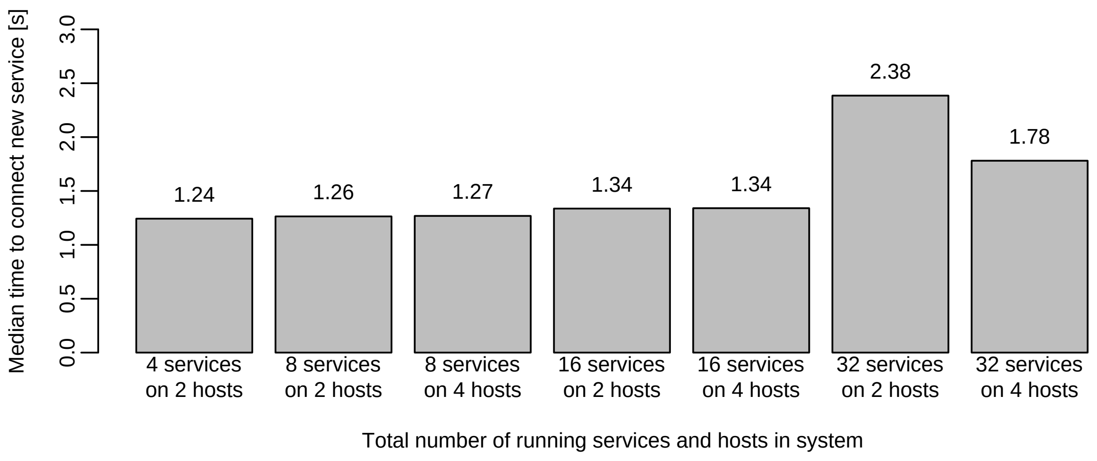


Figure 8.4: Scalability limits of current middleware implementation.
The plot shows the median time required to connect a new service to an existing network of services running on 2 or 4 machines.

8.6 User Feedback

This section shows feedback our research group has received from users of experimental UAR systems built with DWARF. This feedback was not collected in systematic user studies; rather, it is anecdotal evidence collected from personal communication. None of the systems built with DWARF so far are actually in real, productive use; they are all still at the stage of research prototypes. Thus, the “users” have mostly been application partners in research projects, and guests visiting our lab.

New applications In the SHEEP demonstration system, virtual sheep wander around a projection table, and can be influenced by moving a plastic sheep. Thus interaction is quite simple, and even small children can readily understand it.

Many users, once confronted with such new interaction concepts, immediately see potential applications in their area of expertise. When introduced to the SHEEP system, different users have pointed out that the sheep could be replaced with virtual cars, a power plant simulation, a simulation of chemical reactions, software design diagrams, and many other ideas. This indicates that once UAR technology is in widespread use, many new applications will emerge—and design at run time, particularly the cycle of continuous extension, will be extremely useful.

Users particularly liked the idea of being able to exchange ideas with developers and try out new ideas quickly; however, this has not been tested extensively.

Understandable system model The adaptive service dependency architecture is fairly easy for most users to understand at a basic level. The concept of distributed services, especially when visualized with DIVE, is readily understandable, as is the model of data flow, needs and abilities. This is crucially important, as end users must understand the basic structure of the system to reconfigure it at run time.

“Neat prototype—now, let’s build the real thing” Several projects that used DWARF to build prototype systems have since resulted in working AR (not necessarily UAR) systems installed with industrial partners. Here, the flexibility of DWARF was advantageous in requirements elicitation and building prototypes, but since the actual application is usually AR, not UAR, the flexibility of DWARF is not required in the final system.

While not the fastest augmented reality system available, DWARF’s performance has proven adequate to convey the experience of augmented reality to users, at least when using the system for a brief period of time. When the first productive system is deployed, performance optimizations will surely become necessary.

8.7 Developer Feedback

Finally, this section summarizes some of the feedback developers who used DWARF to build systems—or considered using DWARF—gave on the concepts, and on the implementation. Most developers who have used DWARF are computer science students, with widely varying programming expertise. Of course, developers also include graduate students performing research in the domain of UAR.

Implementation issues The most common complaint about DWARF was—and is—that it is too complicated. The learning curve for new developers is steep, since they have to deal with technologies such as CORBA, as well as complex libraries such as Open Inventor. For a new student, installing DWARF on his or her personal laptop is usually at least a full day's task.

Another common complaint is that the distributed system of self-organizing services is hard to manage, since the middleware adapts the system autonomously. To alleviate this problem, good management and development tools are needed, such as DIVE. Also, it is crucial that the middleware functions as it should, since the system as a whole will otherwise become unpredictable.

The last complaint regards the performance of systems built with DWARF: high event latencies when the system is overloaded can cause frustration to both users and developers, since the system is no longer usable for AR. Even a delay of a second is too much.

Value of concepts Most developers readily see the value of design at run time, especially for building prototype systems. It naturally fits the way groups of students work together in building demonstration systems, and encourages integration. However, both development tools and good ground rules for social interaction in design at run time are still issues to be investigated.

The concepts of services, needs, and abilities are usually readily understood by new developers, since they are already familiar with the concept of components. Similarly, the data flow model of AR systems is easy to convey to new developers. Conversely, the "newer" concepts of adaptability, distribution and growth of adaptive service graphs are still fairly hard for developers to understand; this may be because they have not been widely documented yet.

Developers also often use DWARF to integrate their favorite third-party software or hardware device; this is one of the strengths of the use of CORBA, since arbitrary pieces of software can easily be adapted.

"Larger system than we could have built with X" Researchers from other augmented reality groups have commented that DWARF is more flexible and allows a greater degree of distribution than the infrastructure they have developed. This is a promising indication that the concepts are well suited to building ubiquitous augmented reality systems in the future.

Conclusion

The architecture is suitable for building ubiquitous augmented reality systems—at least in a laboratory environment.

The previous chapters have described a concept for addressing the software development challenges of the new field of ubiquitous augmented reality, and shown that it is useful in building experimental systems. This chapter discusses the challenges of Chapter 2 again: to what degree have they been addressed, and what remains to be done?

Furthermore, it discusses the relevance and contribution of this dissertation to the domains of augmented reality, ubiquitous computing and software engineering, but also to other fields, and shows its fundamental limitations. Finally, it suggests some avenues of further research.

9.1 Addressing the Software Development Challenges

This section shows the value, but also the limitations of the proposed architecture, middleware and process. It evaluates the results of Chapter 8 in context of the software development challenges of Section 2.4: how well have these challenges been addressed?

9.1.1 Component Uncertainty

Interdependent distributed devices The dependencies and interdependencies between devices and software components of a UAR system have been modeled simply and sufficiently. In all systems developed with DWARF (Section 8.4), these dependencies could be modeled using the concepts of services, needs and abilities, which were well understood by both developers and users. Indeed, the dependency model is one of the unique features of DWARF.

Changing availability of devices Due to the loose coupling of services via typed needs and abilities, systems can adapt to changing availability of individual devices and services. This is inherent in the architectural model and supported by the middleware implementation. Several DWARF projects, such as Ubitrack, ARCHIE and SHEEP, have used and tested DWARF's capabilities of dealing with changing device availability.

However, the model of needs, abilities and dependencies only allows for binary availability: a device is either available or it is not. This model could be extended for sensors that have a measurement range that deteriorates towards the edges. [138] This would allow the middleware to set up multiple, weighted connections between services, and to change the weights dynamically.

Changing context influences choice of components The architecture's simple context model, based on attributes of services, has sufficed to build systems that can adapt to changing context within a limited scope, such as in the ARCHIE system. This has been tested at least for small-scale systems (e.g. several rooms), but not for larger ones; and the systems currently do not use any machine learning techniques to learn context-dependent behavior from the user.

Incomplete knowledge of components Services are loosely coupled, and DWARF services have been reused unchanged in different systems (see Section 8.1). There have not been any controlled experiments of black-box service testing, i.e. deploying services in the environment and testing their interoperability with newly developed ones.

Informal tests of backwards compatibility have shown that newly developed DWARF services can interoperate with older ones, e.g. components of different versions of the SHEEP system. However, changing or extending interfaces between services has degraded interoperability. Two examples are a new version of the ViewerControl interface, for controlling the 3D viewer service, and an extension of the crucial PoseData structure. Both changes led to minor incompatibilities, which could be fixed easily—but which could not have been fixed by a user in the field.

9.1.2 Ill-defined Requirements

New and changing technology Building prototypes has proven to be effective in demonstrating the potential of UAR, as spectators and potential users have confirmed (Section 8.6). As new types of hardware and new algorithms, e.g. for marker-less tracking, become available, the changing technology may make extensions to previously standardized service interfaces necessary, which could again cause compatibility problems.

Many people involved The concept of design at run time promotes cooperation between developers and users, between application specialists and different technological and scientific specialists. These concepts have been applied in the ARCHIE and CAR projects, and the users and developers gave positive feedback. However, whether this concept can be applied in the long term remains to be established in larger interdisciplinary projects.

New applications The concept of continuous extension, allowing users in the field to suggest system improvements, has not been tested at all, although potential users have responded positively to the idea.

9.1.3 Performance Constraints

Immersivity The performance of DWARF has been sufficient for the demonstration systems of Section 8.4. However, the current implementation's performance is not up to exacting real-time standards that would be required for extensive use in real applications.

Many communication types The concept of connectors allows the middleware to support many communication types, allowing a separation of real-time data flow from secure, transactional interaction. However, this has only been tested for a limited number of communication protocols in DWARF.

Scalability As with the performance, the scalability of DWARF has proven sufficient for all demonstration systems; however, these have never used more than 10 computers simultaneously. The implementation's scalability is currently insufficient for large-scale, e.g. campus-wide, applications; this could be improved by implementing the concept of context-dependent locator federations.

9.2 Relevance of Contribution

This dissertation offers contributions to the research areas of augmented reality and ubiquitous computing, but also to others.

To the field of augmented reality, this work contributes an analysis of software development challenges. It introduces and defines the new field of ubiquitous augmented reality, or UAR, which may become increasingly important to augmented reality research. Addressing the special challenges of UAR, it introduces an architectural style for adaptive, distributed augmented reality systems. It shows how a flexible software architecture can help AR system development, and it how frameworks and software patterns can be used to develop reusable AR systems [79, 109, 110]. Finally, by enabling the open source DWARF framework, this work makes reusable AR software available to the community.

To the field of ubiquitous computing, this work shows how the the use of augmented reality can enhance ubiquitous computing environments. This builds on collaboration with other researchers [34, 90]. The presented architecture can be used to build highly interactive and immersive ubiquitous computing systems.

To the field of software engineering, this work offers a new application domain, ubiquitous augmented reality, showing how established and new techniques such as component technology, service-oriented architectures and agile processes can be applied in this domain.

Beyond ubiquitous augmented reality, the proposed solution is, in principle, applicable to other domains with similar problems of uncertainty, ill-definition and performance. These include domains that are related to UAR, such as location-based services or mobile multimedia systems, and may include others. However, the applicability to these domains remains to be investigated.

9.3 Limitations

This section shows fundamental limitations of this work, in the problem addressed, the scientific approach, and the proposed solution. This goes beyond limits of the current implementation of DWARF.

Prototype systems vs. real-world use The concepts introduced in this dissertation have not been evaluated in productively used systems, but only in research prototypes. Although these systems have often been developed together with application domain experts and successfully demonstrated to potential users, only real evaluation in the field can prove the validity of the solution.

Many industrial applications do not actually require the degree of adaptivity that systems such as DWARF offer, nor do the users want them. Thus, in several projects with industry (Fata Morgana, PAARTI), DWARF was used to build prototype systems rapidly, but later, the DWARF middleware and individual services (e.g. for rendering) were replaced with proprietary components that were already in use in the industry partners' computing environment.

This shows that the adaptive service dependency architecture and design at run time are particularly suited to developing prototype solutions to unclear problems, and that the usefulness in the field remains to be established.

Model limitations: stateful services, service graphs The state of services in the adaptive service dependency architecture is not modeled beyond a service's attributes. This is appropriate for services of UAR systems that are oriented towards data flow and transient data. However, for complex services such as transactional databases, an extended model including service states would be required, e.g. as in FOCUS [13].

Also, in the current meta-model, graphs of services are not modeled as entities in their own right. This leads to a restriction of the operations the middleware can perform on graphs of services. An extension to the model here would greatly extend its expressive power, as it could then express concepts such as design patterns at run time.

Manageability of distributed, self-organizing systems Decentralized, distributed services can be difficult to manage, and to understand. Students are generally not taught to think in terms of decentralized distributed systems. Thus, the learning curve for DWARF is rather steep.

Another issue is the understandability of distributed systems with a large degree of component uncertainty: when a part of the distributed system becomes unavailable, there is some performance degradation. The adaptive service dependency architecture can take measures to make this degradation graceful, and to show users what has happened. However, users might not fully understand why the system's behavior has changed or wish to accept the degraded functionality. This is a fundamental problem that ubiquitous computing systems must address.

Scalability limits There are fundamental limitations to the scalability of service discovery: in a globally deployed system with hundreds of devices in each room, current service location techniques reach their limits. New approaches of context-based service discovery may help, but this also remains to be shown.

Local optima in evolutionary development The concept of design at run time, particularly the asynchronous collaboration of continuous development, leads to an evolutionarily growing system. This type of progress is prone to the development of local optima which are globally sub-optimal. As the system must remain backwards compatible, "revolutionary" jumps in design are discouraged.

Current limits of ubiquitous augmented reality Both augmented reality and ubiquitous computing use cutting-edge technology, in areas ranging from wearable computers to sensor networks to computer vision to photo-realistic rendering. Although technology is continuously progressing, it is currently still difficult to build commercially attractive ubiquitous augmented reality systems.

In [31], it is pointed out that an "experimental infrastructure" is something completely different from an "infrastructure for experimental systems," and that confusing the two will lead to unhappy developers. In UAR, both the systems and the infrastructure are currently experimental. This is a difficult problem, forcing developers and researchers to deal with many technical issues simultaneously.

Finally, although this dissertation has focussed on technological challenges, there are still many social challenges in ubiquitous augmented reality: will users wish to use a system that can make knowledge of their location continuously available to everyone else? Will users actually wear wearable computing equipment? Do people want their reality to be augmented? How dependent on computer technology do we wish to become? These questions must be resolved satisfactorily for UAR to be successful.

9.4 Future Work

Beyond implementation issues listed in Section A.1.7, there are several areas of future research work that could build on this dissertation. These includes architectural improvements, a full development process, a better domain analysis and betted experimental validation.

Architectural improvements For distributed graphs of services, the architectural model and the algorithms in the middleware are currently quite limited. More advanced algorithms could avoid system oscillations, e.g. when the user is in the doorway between two rooms; detect recursions and cyclic dependencies between services; and more generally manage resources intelligently, avoiding combinatorial explosion of service instantiations. First such algorithms for the domain of ubiquitous tracking are described in [137].

Another important issue is standardization and definition of interfaces. Data types and interfaces that are to be used between different organizations must be standardized. For this, domain-specific ontologies should be developed in cooperation with application-domain experts and other research groups.

Also, this dissertation has completely disregarded the issues of privacy and security, which will become important for real-world applications. Further research into this area will be necessary.

Development process Design at run time is only a development technique, not a full-featured development process. It would be interesting to extend it into a full development process within the field of agile development. Some necessary concepts for such a process can already be identified.

First, the process would be *entity-oriented* [14]; development tends to focus naturally on services. A service is a unit that a single developer can manage and understand easily; and furthermore, another developer can use it as-is for integration testing. Collaborative integration tests are common, with two developers testing their two communicating services.

As the framework evolves, there are recurring operations that developers apply to services. These would be worth formalizing for better tool support. An example is the “split” operation. A first optical tracking service included code to read camera data directly from a FireWire port, read marker descriptions from a configuration file, detect markers, and calculate 6D pose from them. By now, the video grabbing and the configuration of marker data have been split off into separate services; the video grabber service is now also used by other services.

To support the process, and even the techniques of continuous extension and jam sessions, tool suites are needed. This includes feedback tools, allowing users to record wishes for improvement in a particular context; services to capture user context and system state, including history; and feedback management tools for the development organization. Similarly, there is currently a lack of tool support for the design of decentralized distributed systems. Tools such as DIVE are indispensable for larger systems. We will need more powerful tools for design, development, testing and debugging.

Continuous extension could be combined with current research into traceability, so that developers can link user requests from requirements elicitation directly to the behavior of the running system. A starting point for this would be to integrate DIVE with the requirements analysis tool RAT [143].

Agile processes such as Scrum would be particularly interesting to test with the technique of design at run time, as new ideas can then be dynamically integrated into the development project schedule.

Combining architecture and process An enticing idea is to more tightly integrate the adaptive service dependency architecture and design at run time. Just as a service can dynamically develop a new need for another service, a user can spontaneously develop a desire for a new application feature. Combining these two concepts would allow the user’s new wish to be immediately modeled as a need in the middleware, however as one with currently unknown type, predicate, and penalty. If the system cannot find a currently running service to satisfy the need, it could help users and developers by suggesting an appropriate extension. A starting point for implementing this idea is the concept of non-communicating needs and abilities.

Domain analysis The analysis of UAR challenges in Section 2.4 may have disregarded important aspects of the application domain, such as its "spatialness," the embodiment of functionality in the physical world. If, in implementing real-world UAR system, such new challenges occur, they will have to be investigated.

Experimental validation Finally, there are several areas for experimentation with the existing concepts, in order to validate and improve them.

First, the middleware's distributed algorithms could be simulated in large-scale networks. Appropriate simulation tools would make improving the middleware's scalability easier.

Second, the concepts of design at run time have so far been tested only in a vary limited scope, in the building of prototype systems. Given appropriate tool suites, both jam sessions and continuous development should be tested in large-scale development projects, over a long time span.

Finally, to test both the value of the concepts of this dissertation and that of ubiquitous augmented reality itself, real systems for real applications have to be built and deployed in the field, and evaluated by end users. Only if those systems can overcome the technological difficulties, the development challenges, and the problems of social acceptance, will ubiquitous augmented reality become reality.

Appendix

Middleware and services included in the current version of DWARF.

Many concepts of the adaptive service dependency architecture have been implemented in the framework DWARF, especially in the DWARF middleware. This appendix shows how the middleware is designed and implemented, and what remains to be done for a full implementation of the adaptive service dependency architecture. It also gives an overview of the services currently included in DWARF.

A.1 The DWARF Middleware

This section describes the implementation of the middleware in the current version of DWARF, DWARF 2004. It is not a detailed reference to the implementation; rather, it shows an overview for current and future developers working with DWARF.

A.1.1 Design and Decomposition

The DWARF middleware consists of several parts. The service manager, helper libraries, and CORBA ORBs are visible to the DWARF services, whereas several internal components are hidden from them. The middleware itself can also be extended by further services.

Service manager The service manager is the main component of the DWARF middleware. One service manager runs on each computer of a system using DWARF. It is responsible for managing the services in each of the phases described in Chapter 6. Together, the service managers coordinate the services of the distributed system.

Use of CORBA Services communicate with the service manager, and with each other, using CORBA. This allows services to be implemented in a variety of languages and on different operating systems, and makes network communication transparent.

DWARF uses several different implementations of CORBA: *OmniORB* [98] for services written in C++, and for the service manager; *OmniORBpy* [98] for services written in Python, and for scripting; and *OpenORB* [99] for services written in Java. Each of these is available on all platforms (Linux, Mac OS, Windows) that DWARF runs on.

The two main part of each CORBA implementation are the ORB itself, which is a library that is linked to a service executable, and an IDL compiler, which generates stubs and skeletons from interface definitions so that the ORB can transparently allow remote method calls.

Helper libraries For each implementation language, DWARF includes a small middleware helper library. This includes commonly-used functions that let services initialize the ORB, establish communication to the local service manager, abstract operating system functions, set up logging and debugging facilities, etc. The simple design of the helper libraries are important, as this makes it easier for new developers to learn DWARF.

Default phase handlers Within the service manager, there are components responsible for default strategies in particular phases, as described above. For example, in the selection phase, the default behavior is to prefer partners that are already running, and are on the local machine. Services can override this behavior.

Internal components of service manager Other internal components of the service manager include locators, responsible for the service discovery of the location phase; connector factories, which allocate resources for communication in the connection phase; and the XML service describer, which creates service descriptions from XML files.

These components are all used by the *service manager core*, which consists of a collection of *active service descriptions*, one for each service managed by the service manager.

Notification service For communication via events, DWARF uses *OmniNotify* [97], a CORBA notification service based on OmniORB. This runs as a separate process, is managed by the service manager, and communicates with services via CORBA. The *push connector factory* communicates with the notification service on behalf of the service manager.

SLP daemon and library For service location, the middleware uses *OpenSLP* [100], an open-source implementation of the Service Location Protocol. OpenSLP consists of a daemon running on each machine, and a library that is linked into the service manager. The SLP locator communicates with the OpenSLP library on behalf of the service manager.

Extensions: management services Beyond the service manager's functionality, the middleware allows the addition of external management services with application-specific or domain-specific functionality. An example is the Ubitrack middleware agent [11, 90], which contains special middleware optimizations for tracking services.

Development history A first prototype of the middleware of DWARF 2001 is described in [77] and [109]. Since then, the middleware has been significantly reworked and extended, but much of the basic design has remained the same.

A.1.2 Interfaces

The CORBA interfaces of services and the service manager are shown in table A.1. A service has callback-style interfaces to be informed of changes of the system structure, and the middleware

Phase	implemented by middleware / called by service	implemented by service / called by middleware
Description	DescribeServices: create, delete, enumerate service descriptions ServiceDescription: query and modify a service description NeedDescription: query and modify a need description AbilityDescription: query and modify an ability description ConnectorDescription: query and modify a connector description Attributes: query and modify attributes of a service, need, connector or ability	AttributesChanged: notification that a service's attributes have changed
Loading	RegisterServices: register a running implementation of a service with its description AsdLoading: register an implementation of a need or ability with its description	SvcLoad: load implementation of a given service into memory Service: return service's status
Startup		SvcStartup: start or stop running, allocate resources
Selection	AsdSelection: choose partners for a service's need	SvcSelection: notify a service of new potential partners for its needs
Session	AsdSession: manage and terminate sessions	SvcSession: notify a service of new sessions to other services; allow creation of session-handling objects in service
Connection		SvcProtObjrefImporter: set reference to remote CORBA object for communication SvcProtPushSupplier: set up communication for sending push-style CORBA notification service events SvcProtShmem: set parameters for local shared memory communication
Communication	StructuredPushConsumer: forward CORBA notification service events	StructuredPushConsumer: receive CORBA notification service events
Instantiation	AsdInstantiation: instantiate a template service in a given context	SvcInstantiation: choose which contexts to instantiate a template service in
Adaption	ServiceDescription: see *Description* phase	AttributesChanged: see *Description* phase

Table A.1: DWARF middleware interfaces by phases. Top box, the phases of the *single service* high-level use case; middle box, of the *two services* use case; bottom box, of the *service in context* use case.

has interfaces allowing a service to control its communication with other services. Different interfaces are used in different phases; not all phases currently have defined interfaces. An example interface definition is shown in Figure A.1 on the next page.

A.1.3 Delegation of Interfaces

Services do not necessarily have to implement all of the interfaces themselves; if they do not, the service manager provides a default behavior. In fact, there is a fairly complex chain of delegation, allowing service developers great flexibility in where to implement which interfaces.

Extension interface pattern The design follows the *extension interface* design pattern [120]: the service manager performs a check at run time to determine which CORBA interfaces a service supports.

When the service manager invokes a method of an interface, e.g. foundPartners in the SvcSelection interface (Figure A.1 on the facing page), it tries, in order of decreasing priority:

- a *session handler object* allocated by the service in response to the newSession method (this only applies to the interfaces of the connection and communication phases);
- a *need object* that the service has registered using the AsdLoading interface's registerNeed method (or analogously an *ability object* registered with registerAbility);
- the main object of the service itself, as registered with the RegisterServices interface's registerService method;
- an object of another service (e.g. the Selector service, for delegated selection);
- an external object that has been registered using the RegisterServices interface's registerServiceCallbacks method on the service description;
- an external object that has been registered using the RegisterServices interface's registerServiceCallbacks method on the service description of the template service, if this service has been instantiated from a template;
- and finally, the default phase handler in the service manager, e.g. in the DefaultSvcSelection class (see below).

This design allows a service implementation to use default functionality where appropriate, and add specific implementations to needs, abilities or services when they become necessary.

In order to implement several CORBA interfaces at the same time, services may implement predefined aggregated interfaces. For example, the interface BasicService_PushConsumer is derived from the Service, SvcStartup and SvcProtPushConsumer interfaces.

A.1.4 Default Phase Handlers

Within the service manager, there are default implementation of strategies for three phases: the loading, selection and the *instantiation* phases.

The *default service loader*, implemented as the class DefaultSvcLoad in the service manager, loads services by starting them as new processes. On Unix-based systems, the output of each

```
// Information on a potential communication partner for a need
struct Partner {
    string protocol;    // communication protocol at this end of the connection.
    string location;    // unique string identifying the communication partner
    boolean selected;   // true if this partner has been selected with selectPartner()
    boolean up;         // true if a session to this partner has been set up
    Attributes attrs;   // attributes of the partner, cached locally
};
typedef sequence<Partner> PartnerSeq;

// Interface to notify a service about set of potential communication partners
interface SvcSelection {
    // Callback to notify a service about set of communication partners
    // In this method, choose one or more communication partners
    // for this need, and call AsdSelection::selectPartner.
    void foundPartners(
        in ActiveServiceDescription serviceDesc,  // service description of this service
        in string needabilityname,                // name of the need
        in PartnerSeq partners);                  // set of possible communication partners
};

// Choose partners to communicate with
interface AsdSelection {
    // Select a communication partner for a need.
    // Call this method after receiving a list of potential communication partners
    // in SvcSelection::foundPartners.
    oneway void selectPartner(
        in string needabilityname,   // name of the need
        in string protocol,          // communication protocol to be used
        in string partner);          // string identifying the partner

    // Replace a communication partner for a need.
    oneway void replacePartner(in string needabilityname, in string protocol,
        in string oldpartner, in string newpartner);
};
```

Figure A.1: Example interface definition within the DWARF middleware.
Annotated excerpt from the file Service.idl, showing the interfaces of the selection phase in CORBA Interface Definition Language.

process is piped to a file, which is displayed in an X Windows terminal. This allows developers to visually distinguish and to debug the individual services. The service is started with the additional command line parameters -DserviceName=..., so that the name of a service, specified in its XML service description, does not have to be hard-coded into the executable.

The startCommand tag of a service description specifies the name of the executable to start, as well as additional command line parameters for the service. If the executable is a .jar file, default service loader starts a java interpreter to load it.

The *default service selection* strategy is implemented in the DefaultSvcSelection class. It selects as many partners as possible, limited only by the maxInstances tag in the need description. When there are several alternative partners to chose from, the default strategy is to prefer partners that are already running, then ones that are running under the same user account, and then ones that are managed by the same service manager. Partners that are not startable are not selected.

The *default service instantiation* strategy, in the DefaultSvcInstantiation class, is to perform all possible instantiations of a template. For each set of bound context attributes, it instantiates the template once. This can lead to partially instantiated services, which are then still templates.

When a partially instantiated template is, itself, instantiated, the service manager computes the combined context of the partially instantiated service and the additionally bound context, and checks that the resulting context has not been instantiated yet. Then, it instantiates the original template service with the new context.

Also, the default service instantiation strategy evaluates the clonepredicate attribute and only instantiates services in contexts for which this attributes evaluates to true. This lets developers limit the number of instantiated services. However, equivalence classes and discriminators are currently not supported.

A.1.5 Service Description Format

One of the main methods that services can use to interact with the middleware is by manipulating their own service descriptions. Similarly, developers and administrators must write and maintain service descriptions of installed services.

Service descriptions are persistently stored in a simple XML format, defined in Figure A.2 on the next page. The DescribeServices, ServiceDescription, NeedDescription and related interfaces of the description phase have get... and set... methods that correspond to the elements of this XML format. For example, the NeedDescription interface has getPredicate and setPredicate methods.

The XML files of each service description reside in a directory whose name can be passed to the service manager upon startup. This makes organization of different configurations easy. For example, if the directory localsheep contains all service descriptions necessary for a demonstration of the SHEEP system on a single machine, the command run-servicemgr localsheep will start the service manager and cause it to read those service descriptions.

The service manager can also re-export services' descriptions that have been programmati-

```
<!ELEMENT service (attribute*,(need|ability)*)>       // service description...
    <!ATTLIST service
        name CDATA #REQUIRED                          // unique service name
        isTemplate          (true|false) "false"      // is this a template?
        startOnDemand       (true|false) "false"      // start service on demand
        startAutomatically  (true|false) "false"      // always start service
        stopOnNoUse         (true|false) "false"      // stop unused service
        startCommand        CDATA        ""           // how to start service
    >
    <!ELEMENT attribute EMPTY>                        // attribute definition...
        <!ATTLIST attribute
            name            CDATA        #REQUIRED    // unique attribute name
            value           CDATA        #REQUIRED    // attribute value
        >
    <!ELEMENT need (attribute*,connector*)>           // need description...
    <!ATTLIST need
        name                CDATA        #REQUIRED    // unique need name
        type                CDATA        #REQUIRED    // type of need, matches ability type
        predicate           CDATA        ""           // boolean predicate over attributes
        delegated           CDATA        ""           // need to delegate to, e.g. selection
        minInstances        CDATA        "1"          // minumum partner abilities
        maxInstances        CDATA        "1"          // maximum partner abilities
    >
        <!ELEMENT connector (attribute*)>             // connector description...
        <!ATTLIST connector
            protocol        CDATA        #REQUIRED    // communication protocol
        >
    <!ELEMENT ability (attribute*,connector*)>        // ability description...
    <!ATTLIST ability
        isTemplate          (true|false) "false"      // is this a template?
        name                CDATA        #REQUIRED    // unique ability name
        type                CDATA        #REQUIRED    // type of ability, matches need type
        >
```

Figure A.2: Document type definition for service descriptions in DWARF.
Annotated service.dtd file. The ServiceDescription interface allows programmatic access to the same elements of a service description.

Class	Responsibility	External interfaces
`ServiceManager_i`	Main facade class of the service manager	`DescribeServices`, `RegisterServices`
`ActiveServiceDescription_i`	Class responsible for a single service in the service manager	`ServiceDescription`, all `Asd...` interfaces
`ActiveNeedDescription_i`	Class responsible for a single need of one service in the service manager	`NeedDescription`
`ActiveAbilityDescription_i`	Class responsible for a single ability of one service in the service manager	`AbilityDescription`
`ConnectorDescription_i`	Base class for need and ability connector descriptions	`ConnectorDescription`
`NeedConnectorDescription_i`	Handles one connector description of a need	`ConnectorDescription`
`AbilityConnectorDescription_i`	Handles one connector description of an ability	`ConnectorDescription`
`Session`	Represents one end of a connection between an ability and a need	

Table A.2: Core classes in the DWARF service manager

cally modified, in XML format; this allows developers to persistently store changes they have made to a running system.

A.1.6 Implementation of the Service Manager

Figure A.3 on the facing page shows a class diagram of the current implementation of the DWARF service manger. The classes can be roughly divided into three categories:

- The main classes of the service manager core: a facade class and active entity classes. The entity classes represent services, needs, abilities and connectors (for managing single services), as well as sessions (for managing pairs of services). These classes are shown in Table A.2.
- Helper classes for the service manager core, performing various additional functions, such as storing and retrieving attributes (Table A.3 on page 162).
- Extension classes, shown in Table A.4 on page 162: these are the default phase handlers, the connector factories, the locators, and the XML service describer.

Service manager core: active objects, state machines, callbacks The service manager core consists of classes representing services, needs, abilities, and other objects from the meta-model of the adaptive service dependency architecture shown in Figure 5.1 on page 66. These classes follow the *active object* design pattern [120], in that they perform actions on behalf of the services they represent in their own thread of control.

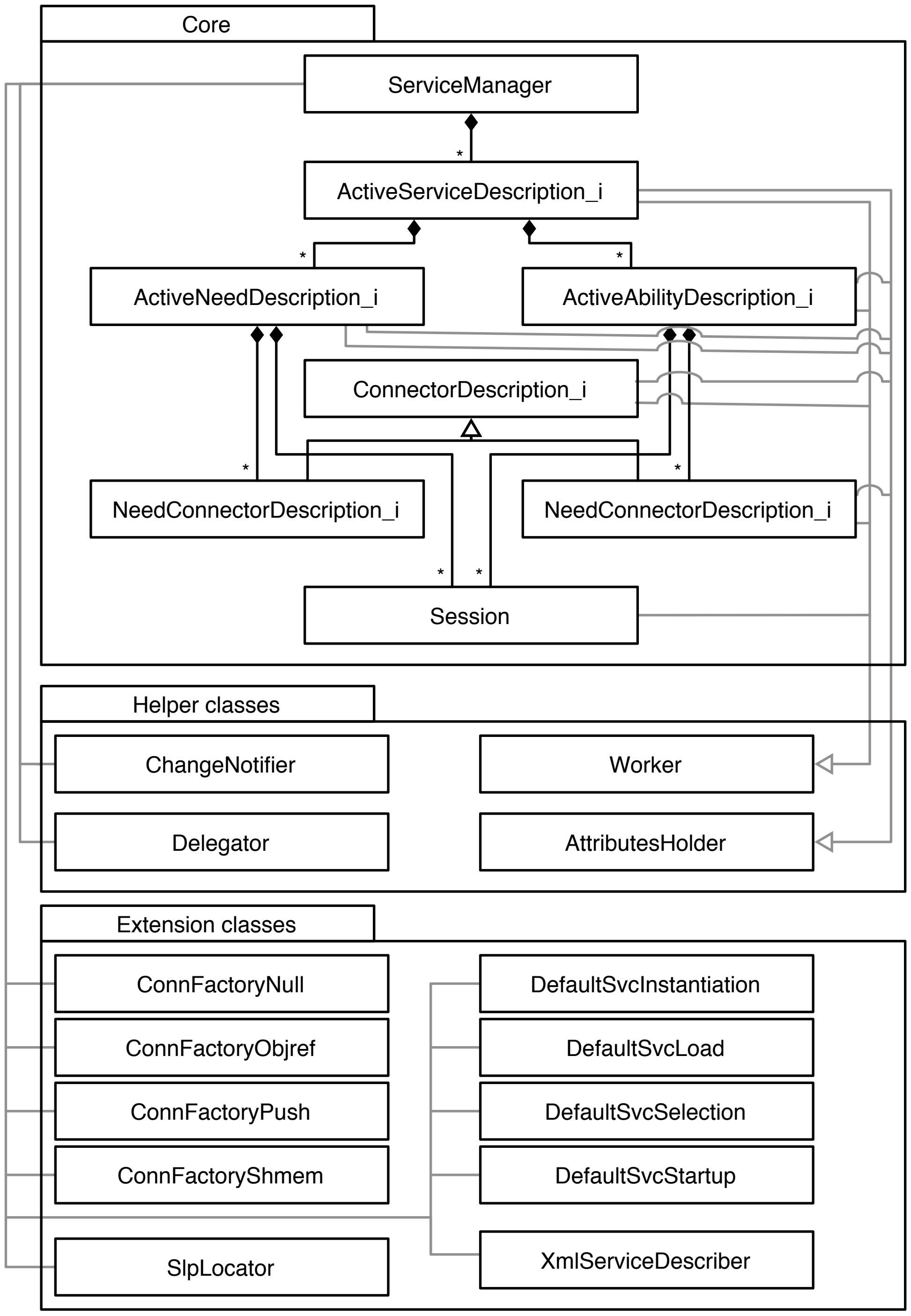


Figure A.3: Class diagram of the DWARF service manager implementation.
Interfaces omitted for simplicity. The core classes, in the topmost package, are modeled after the adaptive service dependency architecture meta-model.

Class	Responsibility	External interfaces
`Worker`	A helper class for threading, cleanup, and reentrant outcalls. Each ActiveService-Description_i is at the root of a tree of worker objects	
`ReentrantMutex`	A helper class for reentrant locks.	
`AttributesHolder`	A mixin helper class for holding attributes; implements The methods of the DWARF::Attributes interface for other classes.	`Attributes`
`ChangeNotifier`	Sends notifications when a service changes	
`ContextSet`	Helper class for holding a set of attribute/value contexts.	
`Delegator`	Importing delegator for the SvcSelection interface, allows delegated selection.	`SvcProtObjrefImporter`

Table A.3: Helper classes in the DWARF service manager

Class	Responsibility	External interfaces
`DefaultSvcInstantiation`	Default handler for the SvcInstantiation interface.	`SvcInstantiation`
`DefaultSvcLoad`	Default SvcLoad handler	`SvcLoad`
`DefaultSvcSelection`	Default SvcSelection handler	`SvcSelection`
`DefaultSvcStartup`	Default SvcStartup handler	`SvcStartup`
`ConnFactoryNull`	Connector factory for ”null” connector; does nothing.	
`ConnFactoryObjref`	Connector factory for CORBA object references.	
`ConnFactoryPush`	Connector factory for CORBA notification service events.	
`ConnFactoryShmem`	Connector factory for shared memory.	
`SlpLocator`	Responsible for all SLP communication in the service manager	
`XmlServiceDescriber`	XML parser for service descriptions.	

Table A.4: Extension classes in the DWARF service manager

Each ActiveServiceDescription_i object runs in its own thread of control (with an optimization of a thread pool for services that are not loaded). When a method of the service manager is invoked by a service or by another service manager, it is forwarded to the appropriate active service description, and there triggers an event waking the service description's worker thread.

The state of a service is modeled using state machines. There is a state machine for each service, need, ability, connector and session. The states reflect the states of the various phases—for example, the Session class has a state WaitForSelect, representing a possible parter for a need partner that has completed the location phase, but not the selection phase.

It is worth pointing out that the Session object represents only one "half" of a connection between one service's need and another service's ability. Thus, for each matching pair of needs and abilities, one Session object is allocated within each of the two service descriptions. The Session objects communicate with one another to establish communication between a pair of services.

Incoming method calls change the state of one of these state machines. In response, the worker thread traverses all state machines of a service (service, needs, abilities, sessions, etc.) and performs whatever actions are appropriate. This includes invoking callback methods of the service itself, allocating connectors from a connector factory, registering abilities with the locator, loading or unloading the service, etc.

Reentrancy The service manager does not call methods reentrantly; this means that when a service invokes a method of the service manager, the service manager does not perform callbacks to the service from the same (distributed) thread of control. This helps prevent deadlocks in services that are not reentrant.

However, the service manager can itself be called back reentrantly: when the service manager invokes a service's method such as selectPartners, the service may perform calls back to the service manager. To avoid deadlocks, an active service description's worker thread releases the lock on its internal variables when performing outcalls (i.e. invoking callback methods on the service), and tries to reacquire it after the call has completed. If internal data structures have changed due to callbacks from the service, the worker thread then aborts its traversal of the state machines and starts over.

This design prevents deadlocks between the service manager and its services, and ensures thread-safe access to the service manager's internal data.

Exception Handling When an exception occurs in communication between two services, that session is terminated, without affecting other sessions. When an exception occurs in communication between the service manager and a service, the active service description of that service terminates, killing the service. If the service description has been persistently stored as an XML file, it will be re-loaded afterwards.

This means that implementation bugs in a single DWARF service, which cause the service to crash, are generally not critical: the service manager detects that the service has failed and reloads it. As long as the communication between the failed service and other services is stateless, the system can recover. An example is the Viewer service used in ARCHIE and SHEEP; for a long time, it contained a bug causing (infrequent) crashes. However, the service manager could detect this and restart the Viewer service, so the system as a whole continued running.

Locators The two locators available in the DWARF service manager, the SimpleLocator and the SLPLocator, implement a common Locator interface. Thus, locator implementations can be exchanged transparently.

By default, the DWARF service manager uses the SLP locator, which registers abilities with the SLP daemon running on the local host, and periodically sends multicast queries to the network for each need.

However, with the extra environment variable SERVICEMANAGER_PARAMS='-DnoSLP', the service manager will instead use the simple locator, which only compares needs and abilities on the local host, without network communication.

Connector Factories Similarly, the connector factories in DWARF all implement a common ConnectorFactory interface, allowing the allocation and deallocation of communication resources. Thus, implementations of connector factories can be exchanged transparently.

However, each connector factory has to communicate with the "matching" connector factories on other hosts; thus, there are protocol-specific interfaces that connector factories use to communicate with each other.

A.1.7 Future work

There are several areas of important future implementation work on the DWARF middleware.

Unimplemented concepts Many of the architectural and middleware concepts have not yet been implemented in DWARF. To improve the framework and to test the concepts, Several features should be implemented:

- Attribute change connectors, which receive events and change attributes of service descriptions. This allows developers to easily take advantage of the context-adaptive features of the middleware.
- Interfaces allowing services to override the middleware's default behavior in the phases where this is not possible yet, e.g. the location phase.
- Penalties and distances automating a need's selection between multiple alternatives.
- A full binding rule specification syntax, as shown in Figure 6.13 on page 111.
- Equivalence classes and discriminators, limiting the number of instantiations of service templates.
- A general rule specification syntax for the binding and induction phases.
- Connector factories for additional communication protocols, such as video streaming.
- In the connection phase, a callback to notify services that the communication link to the partner has been established.

Mobility and roaming Both OmniORB and OpenSLP currently cannot deal with dynamically changing IP addresses; this limits the applicability in mobile scenarios. Patches to both would be feasible, as both are open source software.

Performance improvements Several areas would benefit from performance enhancements:

- Better service location technology, e.g. an extension to SLP that uses announcements and subscriptions in addition to polling.
- Colocation of services within a single process space to avoid excessive interprocess communication, e.g. with language specific loader services.
- More scalable locator, allowing context-aware locator federations.
- Integration of external data flow frameworks to improve communication efficiency, e.g. by adding support for multicast communication protocols.

Packaging and distribution Several features would be helpful in making DWARF more accessible to developers:

- Additional classes in the helper libraries, e.g. for shared memory communication.
- Installers and binary distributions for various common platforms.
- A *launcher service*, allowing easy startup of DWARF applications.

Implementation alternatives It would be worthwhile to consider extending the DWARF middleware so that services can be written using the CORBA component model. The DWARF service manager could then act as a partial CORBA component model implementation, or extend an existing implementation. This would make service development easier, and allow access to existing modeling tools.

A.2 Services Included in DWARF 2004

The current version of DWARF includes implementations of many services. Some are generically applicable, whereas some are specialized to certain hardware or certain types of applications. An overview is presented in tables A.5 to A.11; the most important services have been fully documented elsewhere, e.g. in student developers' project reports [88]. The services are organized according to the subsystems of Section 2.3, with the addition of a *development subsystem*, which consists of services that are used by developers to test, configure and improve the system.

The implementation status and the stability of these services vary; some are quite stable and complete, whereas others are still in an "alpha" stage. This is to be expected of a developing research platform. Similarly, several services overlap in functionality (e.g. different services for collision detection)—this shows how different implementations were tested in different applications. In future versions of DWARF, these services should be consolidated into a complete UAR development platform.

Service	Functionality
ART Tracker	Adapter for ART's DTrack tracking system
Intersense Tracker	Adapter for Intersense trackers
Xsens Tracker	Adapter for the Xsens MT9-B tracker
Video Grabber	Video capture for FireWire cameras
V4L Grabber	Video capture for USB webcams on Linux
AR Toolkit Marker Detection	Wrapper for the AR Toolkit's marker detection
AR Toolkit Pose Reconstruction	Wrapper for the AR Toolkit's pose reconstruction
Object Calibration	Interactive calibration of tracked objects
Calibration	Interactive HMD calibration
Static Calibration	Provide statically calibrated measurements
Multi Tracker	Transition between two trackers with overlapping range
Inference	Inference of pose data from several tracker inputs
Ubitrack Middleware Agent	Find and combine different tracking services

Table A.5: DWARF services in the tracking subsystem

Service	Functionality
Configuration	Retrieve configuration data from a database
Marker Configuration	Provide marker descriptions to ARToolkit tracker
State Machine	Distributed model of transitions between rooms

Table A.6: DWARF services in the context subsystem

Service	Functionality
Model	Consistent manipulation and management of 3D model
Model Server	Database adapter for persistent storage of 3D models

Table A.7: DWARF services in the world model subsystem

Service	Functionality
Viewer	Rendering of 3D scenes for HMDs and other screens, using Coin, an OpenInventor implementation
Virtual Camera	Simple Open-GL-based rendering of static scenes
VRML Display	Rendering of 3D scenes, using FreeWRL (used on palmtop)
3D Audio	Adapter for spatial audio output hardware
Sound Service	Plays sound files for audio feedback
Layout Manager	Dynamic layout of labels for 3D scene annotations
Menu Display	Display simple 2D menu to select services, using QT
Fairy Tail	Shows trail of motion behind tracked object
Speech Out	Speech synthesis from english text
GPS Position	Adapter for GPS devices

Table A.8: DWARF services in the presentation subsystem

Service	Functionality
UI Controller	User Interface Controller; manage multimodal interaction using Petri nets
Speech Recognition Adapter	Adapter for external speech recognition system
Touchpad Glove	Manage input from hand-mounted touchpad
Janus Adapter	Adapter for the Janus eye tracking system
Powermate Driver	Adapter for Griffin Technology's Powermate input device
Pattern Collision Detection	Detect collisions between tracked and virtual objects, for gesture input
Collision Detection	Detect collisions for gesture input (older version)
Discretizer	Generate discrete input events from continuous data streams
PoseData Angle Interpreter	Interpret gesture input for selection of menu items
Parsephone	Map input data to formula-defined function, e.g. for gesture input
Interpolator	Map input data to function interpolated between given points, e.g. for tuning gesture input
Selector	Allows a user to chose between alternate services

Table A.9: DWARF services in the interaction subsystem

Service	Functionality
Sheep	Simulation of a sheep within a herd (SHEEP project)
BreakOut	Simple AR version of classic “breakout” game
Filter	Several custom services for transforming streams of continuous data for user interaction (CAR project)
Street Map	Provide navigational information for the blind based on GPS position and street map (NAVI project)

Table A.10: DWARF services in the application subsystem. Only a few example services are shown here, since most applications include several custom services.

Service	Functionality
DIVE	Visualization, monitoring and configuration of distributed services
DISTARB	Invoke arbitrary CORBA methods
CARmelion	Configuration of arbitrary services by sending events
Manual Tracker	Simulation of tracking data
Error Visualizer	Visualization of tracker errors
Pose Tool	Simulate filtering operations on tracking data
Data Logger	Logs data for usability tests
PoseData Logger	Log streams of tracking data
PoseData Player	Replay streams of tracking data
PoseData Show Lag	Show network delay of events
Visualisation Python	Visualization of 3D marker deployment
Viewer Controller	Interactive scripting tool to control the Viewer service
Time Service	Simple service returning the time of day, used in tutorial
Test String Sender	Historically, the first DWARF service; send debugging strings to test the middleware

Table A.11: DWARF services in the development subsystem. These services are used to monitor, debug, and extend a running system.

Bibliography

[1] Michael Adams, James Coplien, Robert Gamoke, Robert Hanmer, Fred Keeve, and Keith Nicodemus. Fault-tolerant telecommunication system patterns. In *Pattern languages of program design 2*, pages 549–562. Addison-Wesley Longman Publishing Co., Inc., 1996. 104

[2] Advanced Realtime Tracking GmbH. Home page. `http://www.ar-tracking.de/`. 56

[3] Ronald T. Azuma. A survey of augmented reality. *Presence*, 6(4):355–385, August 1997. 1, 7

[4] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman, and Deborra Zukowski. Challenges: an application model for pervasive computing. *Proceedings of 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000)*, Aug. 2000. 33

[5] Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams, Thomas Reicher, Stefan Riss, Christian Sandor, and Martin Wagner. Design of a component-based augmented reality framework. In *Proceedings of the International Symposium on Augmented Reality (ISAR)*, October 2001. 8, 46, 133, 136

[6] Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams, Thomas Reicher, Christian Sandor, and Martin Wagner. An architecture concept for ubiqutous computing aware wearable computers. In *International Workshop on Smart Appliances and Wearable Computing (IWSAWC)*, July 2002. 54, 58, 68

[7] Martin Bauer, Otmar Hilliges, Asa MacWilliams, Christian Sandor, Martin Wagner, Joe Newman, Gerhard Reitmayr, Tamer Fahmy, Gudrun Klinker, Thomas Pintaric, and Dieter Schmalstieg. Integrating Studierstube and DWARF. In *International Workshop on Software Technology for Augmented Reality Systems (STARS)*, October 2003. 92, 134

[8] R. Bauernschmitt, E.U. Schirmbeck, M. Groher, P. Keitler, M. Bauer, H. Najafi, G. Klinker, and R. Lange. Navigierte platzierung endovaskulaerer aortenstents. *Zeitschrift für Kardiologie*, S3:116, 2004. 138

[9] Kent Beck. *eXtreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. 45, 121, 122

[10] Blaine Bell, Steven Feiner, and Tobias Hollerer. View management for virtual and augmented reality. In *UIST*, pages 101–110, 2001. 16

[11] Dagmar Beyer. Construction of decentralized data flow graphs in ubiquitous tracking environments. Diplomarbeit, Technische Universität München, Institut für Informatik, 2004. 60, 118, 154

[12] R. Bjornson, N. Carriero, D. Gelernter, T. Mattson, D. Kaminsky, and A. Sherman. Experience with linda. Technical Report YALEU/DCS/TR-866, Yale University, Department of Computer Science, Yale University, New Haven, Connecticut, U.S., 1991. 33

[13] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001. 148

[14] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns and Java*. Prentice Hall, 2003. 38, 49, 70, 150

[15] Bernd Brügge, Asa MacWilliams, and Thomas Reicher. Software architectures for augmented reality systems – report to the ARVIKA consortium. Technical Report TUM-I0410, Technische Universität München, July 2004. 12, 25, 26

[16] B. Brumitt, J. Krumm, B. Meyers, and S. Shafer. Ubiquitous computing and the role of geometry. *IEEE Personal Communications*, pages 41–43, October 2000. 60

[17] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England, 1996. 32, 49, 67, 100

[18] Andreas Butz, Clifford Beshers, and Steven Feiner. Of vampire mirrors and privacy lamps: Privacy management in multi-user augmented environments [technote]. In *Proceedings of UIST'98*, pages 171–172. ACM SIGGRAPH, 1998. 27

[19] Andreas Butz, Tobias Höllerer, Steven Feiner, Blair MacIntyre, and Clifford Beshers. Enveloping users and computers in a collaborative 3d augmented reality. In *Proceedings of the International Workshop on Augmented Reality IWAR '99*, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264, 1999. IEEE Computer Society Press. 27

[20] Renato Cerqueira, Carlos Cassino, and Roberto Ierusalimschy. Dynamic component gluing across different componentware systems. In *Proceedings of the International Symposium on Distributed Objects and Applications*, page 362. IEEE Computer Society, 1999. 32

[21] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. Addison-Wesley, 2002. 49, 72, 81

[22] Michael Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the computational needs of intelligent environments: The

metaglue system. In Paddy Nixon, Gerard Lacey, and Simon Dobson, editors, *1st International Workshop on Managing Interactions in Smart Environments (MANSE'99)*, pages 201–212, Dublin, Ireland, December 1999. Springer-Verlag. 34

[23] Microsoft Corporation. COM: Component object model technologies. `http://www.microsoft.com/Com/default.mspx`. 98

[24] Microsoft Corporation. Microsoft .Net information. `http://www.microsoft.com/net/`. 98

[25] D. Curtis, D. Mizell, P. Gruenbaum, and A. Janin. Several devils in the details: Making an AR application work in the airplane factory. In Reinhold Behringer, Gudrun Klinker, and David W. Mizell, editors, *Augmented Reality - Placing Artificial Objects in Real Scenes*. A K Peters, Ltd., 1999. 11

[26] N. Davies, S. Wade, A. Friday, and G. Blair. Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications. In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, pages 291–302, Toronto, Canada, May 1997. 32

[27] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction (HCI) Journal*, 16(2-4):97–166, 2001. 15, 31

[28] Nikolas Dörfler, Peter Hallama, Korbinian Schwinger, and Maximilian Schwinger. Development of a dynamic network of filters that is configurable with tools. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2004. 63, 129, 130

[29] Florian Echtler, Hesam Najafi, and Gudrun Klinker. FixIt. In *Demonstration Session at the International Symposium on Augmented and Mixed Reality (ISMAR 2002)*, Darmstadt, Germany, 2002. 137

[30] Florian Echtler, Fabian Sturm, Kay Kindermann, Gudrun Klinker, Joachim Stilla, Joern Trilk, and Hesam Najafi. The intelligent welding gun: Augmented reality for experimental vehicle construction. In S.K Ong and A.Y.C Nee, editors, *Virtual and Augmented Reality Applications in Manufacturing, Chapter 17*. Springer Verlag, 2003. 4, 137

[31] W. Keith Edwards, Victoria Bellotti, Anind K. Dey, and Mark W. Newman. The challenges of user-centered design and evaluation for infrastructure. In *Proceedings of the conference on Human factors in computing systems*, pages 297–304. ACM Press, 2003. 149

[32] Christoph Endres. Towards a Software Architecture for Device Management in Instrumented Environments. In *Adjunct Proceedings of UbiComp–The Fifth International Conference on Ubiquitous Computing*, pages 245–246, Seattle, Washington USA, October 2003. Doctoral Colloquium. 32

[33] Christoph Endres. *A Software Architecture for Device Management in Instrumented Spaces*. PhD thesis, Saarland University, Germany, 2005. To appear. 32

[34] Christoph Endres, Andreas Butz, and Asa MacWilliams. A survey of software infrastructures and frameworks for ubiquitous computing. *Mobile Information Systems Journal*, 1(1), January–March 2005. 25, 147

[35] Alois Ferscha and Friedemann Mattern, editors. *A Context-Aware Communication Platform for Smart Objects*, number 3001 in LNCS, Linz/Vienna, Austria, April 2004. Springer-Verlag. 103

[36] Marco Feuerstein. Erstellen einer DWARF-basierten Darstellungskomponente für 3D Szenen auf dem iPAQ. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2003. 58

[37] Wolfgang Friedrich. ARVIKA - Augmented Reality for Development, Production and Service. In *Proceedings of the International Symposium on Mixed and Augmented Reality*, Darmstadt, Germany, Oct. 2002. 3

[38] Wolfgang Friedrich, editor. *ARVIKA – Augmented Reality für Entwicklung, Produktion und Service*. Publicis, Erlangen, 2004. 9, 11, 14, 15, 18, 26

[39] Maribeth Gandy, Steven Dow, and Blair MacIntyre. Prototyping Applications with Tangible User Interfaces in DART, The Designers Augmented Reality Toolkit. In *Positional Paper at Toolkit Support for Interaction in the Physical World Workshop at the IEEE Pervasive Computing*, April 2004. 27

[40] 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. 10

[41] Markus Geipel. Run-time development and configuration of dynamic service networks. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2004. 63, 125, 127, 128, 129

[42] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985. 33

[43] Gnutella home page. `http://www.gnutella.com`. 104

[44] Phil Gray, Joy Goodman, and James Macleod. A lightweight experiment management system for handheld computers. In *CADUI 2004*, Madeira, Portugal, January 2004. 8

[45] William G. Griswold, Robert Boyer, Steven W. Brown, and Tan Minh Truong. A component architecture for an extensible, highly integrated context-aware computing infrastructure. In *Proceecings of the International Conference on Software Engineering*, pages 363–372, Portland, Oregon, 2003. IEEE Computer Society. 10, 29

[46] Object Management Group. UML 2.0 infrastructure final adopted specifcation. http://www.omg.org/cgi-bin/doc?ptc/2003-09-15. 65

[47] Object Management Group. UML 2.0 superstructure final adopted specification. http://www.omg.org/cgi-bin/doc?ptc/2003-08-02. 65

[48] E. Guttman, C. Perkings, J. Veizades, and M. Day. Service location protocol, version 2, Jun. 1999. 101

[49] Günther Harrasser. Design eines für Blinde und Sehbehinderte geeigneten Navigationssystems mit taktiler Ausgabe. Diplomarbeit, Technische Universität München, Institut für Informatik, 2003. 138

[50] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *Mobile Computing and Networking*, pages 59–68, 1999. 14, 15, 34

[51] Thomas Heider and Thomas Kirste. Supporting goal-based interaction with dynamic intelligent environments. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 596–600, Lyon, France, July 2002. IOS Press. 31

[52] Michael Hellenschmidt and Thomas Kirste. SodaPop: A Software are Infrastructure Supporting Self-Organization in Intelligent Environments. In *Proceedings of the 2nd IEEE International Conference on Industrial Informatics, INDIN'04*, Berlin, Germany, 2004. 31

[53] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA, 1999. 44

[54] Otmar Hilliges. Design of a 3d view component for dwarf. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2003. 58, 96, 138

[55] Otmar Hilliges. Interaction management for ubiquitous augmented reality user interfaces. Diplomarbeit, Technische Universität München, Institut für Informatik, 2003. 63, 129, 139

[56] 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 & Graphics*, 23(6):779–785, 1999. 8

[57] Roberto Ierusalimschy and Luiz Henrique de Figuereido andWaldemar Celes. Lua: An Extensible extension language. In *Software: Practice and Experience*, 1996. 32

[58] Nikos Ioannidis. Archeoguide. http://archeoguide.intranet.gr/project.htm, 2002. 12

[59] Tony Jebara, Bernt Schiele, Nuria Oliver, and Alex Pentland. Dypers: Dynamic personal enhanced reality system. Technical Report Vision and Modeling No. 463, M.I.T. Media Laboratory, May 1998. 9

[60] Xiaodang Jiang, Nicholas Y. Chen, Jason I. Hong, Kevin Wang, Leila Takayama, and James A. Landay. Siren: Context-aware computing for firefighting. In *Proceedings of the Second International Conference on Pervasive Computing*, Vienna, Austria, April 2004. 11, 15, 18

[61] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, 1987. 125

[62] Hirokazu Kato and Mark Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Proceedings of the 2nd International Workshop on Augmented Reality (IWAR 99), San Francisco, USA*, 1999. 25

[63] Hirokazu Kato, Mark Billinghurst, and Ivan Poupyrev. *ARToolKit version 2.33 Manual*, 2000. Available for download at `http://www.hitl.washington.edu/research/shared_space/download/`. 3

[64] Sungil Kim, Ungyeon Yang, Namgyu Kim, and Gerard Jounghyun Kim. COVRA-CAD: A CORBA based Virtual Reality Architecture for CAD. In *Proceedings of International Conference on Virtual Systems and Multimedia*, 1998. 41

[65] Gudrun Klinker. Introduction to augmented reality. Lecture 1, `http://wwwbruegge.in.tum.de/Lehrstuhl/AugmentedRealityIntroductionWiSe2003`, 10 2003. 13

[66] Gudrun Klinker, Allen Dutoit, Martin Bauer, Johannes Bayer, Vinko Novak, and Dietmar Matzke. Fata morgana – a presentation system for product design. In *International Symposium on Aumgented and Mixed Reality ISMAR 2002*, 2002. 4

[67] Gudrun Klinker, Allen Dutoit, Martin Bauer, Johannes Bayer, Vinko Novak, and Dietmar Matzke. Fata morgana – a presentation system for product design. In *Proceedings of ISMAR 2002*, Darmstadt, Germany, 2002. 136

[68] Gudrun Klinker, Hesam Najafi, Tobias Sielhorst, Fabian Sturm, Florian Echtler, Mustafa Isik, Wolfgang Wein, and Christian Truebswetter. Fixit: An approach towards assisting workers in diagnosing machine malfunctions. In *Proc. of the International Workshop exploring the Design and Engineering of Mixed Reality Systems - MIXER 2004, Funchal, Madeira, CEUR Workshop Proceedings*, 2004. 137

[69] Gudrun Klinker, Thomas Reicher, and Bernd Bruegge. Distributed user tracking concepts for augmented reality applications. In *Proceedings of the IEEE and ACM International Symposium on Augmented Reality – ISAR 2000*, Munich, Germany, October 2000. 46

[70] Donald Kossmann and Frank Leymann. Web services. *Informatik Spektrum*, 2004. 43

[71] Christian Kulas. Usability engineering for ubiquitous computing. Diplomarbeit, Technische Universität München, Institut für Informatik, 2003. 58, 123, 132, 138

[72] Christian Kulas, Christian Sandor, and Gudrun Klinker. Towards a development methodology for augmented reality user interfaces. In *Proc. of the International Workshop exploring the Design and Engineering of Mixed Reality Systems - MIXER 2004, Funchal, Madeira, CEUR Workshop Proceedings*, 2004. 60, 123, 126, 131, 138

[73] R. L. Lagendijk. Ubiquitous communications (ubicom) - updated technical annex 2000. Technical report, Ubiquitous Communications Program TU-Delft, Jan. 2000. 36

[74] James A. Landay and Gaetano Borriello. Design patterns for ubiquitous computing. *Computer*, 36(8):93–95, 2003. 37

[75] Felix Löw. Context-aware service selection based on the artoolkit. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2003. 58

[76] Blair MacIntyre and Steven Feiner. A distributed 3d graphics library. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 361–370. ACM Press, 1998. 27

[77] Asa MacWilliams. DWARF – Using ad-hoc services for mobile augmented reality systems. Diplomarbeit, Technische Universität München, Institut für Informatik, February 2001. 46, 133, 154

[78] Asa MacWilliams, Thomas Reicher, and Bernd Brügge. Decentralized coordination of distributed interdependent services. In *IEEE Distributed Systems Online – Middleware '03 Work in Progress Papers*, Rio de Janeiro, Brazil, June 2003. 84, 105

[79] Asa MacWilliams, Thomas Reicher, Gudrun Klinker, and Bernd Bruegge. Design patterns for augmented reality systems. In *Proceedings of the International Workshop exploring the Design and Engineering of Mixed Reality Systems (MIXER), Funchal, Madeira, CEUR Workshop Proceedings*, January 2004. 37, 92, 147

[80] Asa MacWilliams, Christian Sandor, Martin Wagner, Martin Bauer, Gudrun Klinker, and Bernd Brügge. Herding sheep: Live system development for distributed augmented reality. In *Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR)*, October 2003. 3, 10, 16, 18, 55, 56, 57, 58, 130

[81] Marco Lohse and Michael Repplinger and Philipp Slusallek. Dynamic Distributed Multimedia: Seamless Sharing and Reconfiguration of Multimedia Flow Graphs. In *Proceedings of the 2nd International Conference on Mobile and Ubiquitous Multimedia (MUM 2003)*, pages 89–95. ACM Press, 2003. 36

[82] Marco Lohse and Michael Repplinger and Philipp Slusallek. Session Sharing as Middleware Service for Distributed Multimedia Applications. In *Interactive Multimedia on*

Next Generation Networks, Proceedings of First International Workshop on Multimedia Interactive Protocols and Systems, MIPS 2003, volume 2899 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2003. 36

[83] Marco Lohse and Philipp Slusallek. Middleware Support for Seamless Multimedia Home Entertainment for Mobile Users and Heterogeneous Environments. In *Proceedings of The 7th IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA)*, pages 217–222. ACTA Press, 2003. 36

[84] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. 38, 44

[85] Mutlicast DNS home page. `http://www.multicastdns.org`. 101

[86] Technische Universität München. AR Patterns home page. `http://wwwbruegge.in.tum.de/ARPatterns/`. 92

[87] Technische Universität München. DWARF home page. `http://wwwbruegge.in.tum.de/DWARF/`. 45

[88] Technische Universität München. DWARF teaching overview. `http://wwwbruegge.in.tum.de/DWARF/TeachingOverview`. 165

[89] Hani Naguib, George Coulouris, and Scott Mitchell. Middleware support for context-aware multimedia applications. In *DAIS*, 2001. 35

[90] J. Newman, M. Wagner, M. Bauer, A. MacWilliams, T. Pintaric, D. Beyer, D. Pustka, F. Strasser, D. Schmalstieg, and G. Klinker. Ubiquitous tracking for augmented reality. In *Proc. of International Symposium on Mixed and Augmented Reality*, Arlington, VA, USA, Nov. 2004. 60, 61, 62, 118, 147, 154

[91] Joseph Newman, David Ingram, and Andy Hopper. Augmented reality in a wide area sentient environment. In *Proceedings of the International Symposium on Augmented Reality (ISAR)*, October 2001. 4, 8, 10, 34

[92] Joseph Newman, Martin Wagner, Thomas Pintaric, Asa MacWilliams, Martin Bauer, Gudrun Klinker, and Dieter Schmalstieg. Fundamentals of ubiquitous tracking for augmented reality. Technical Report TR-188-2-2003-34, Vienna University of Technology, December 2003. 60

[93] Daniela Nicklas, Matthias Großmann, Thomas Schwarz, Steffen Volz, and Bernhard Mitschang. A model-based, open architecture for mobile, spatially aware applications. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 117–135. Springer-Verlag, 2001. 33

[94] Laurence Nigay, Philippe Renevier, Jullien Bouchet, and Laurence Pasqualetti. Generic interaction techniques for mobile collaborative mixed reality systems. In *MIXER*, 2004. 10

[95] Vinko Novak. Attentive user interfaces for DWARF. Diplomarbeit, Technische Universität München, Institut für Informatik, 2004. 63

[96] Object Management Group home page. `http://www.omg.org`, 2004. 44

[97] OmniNotify home page. `http://omninotify.sourceforge.net`. 154

[98] OmniORB home page. `http://omniorb.sourceforge.net`. 153

[99] OpenORB home page. `http://openorb.sourceforge.net`. 153

[100] OpenSLP home page. `http://www.openslp.org`. 101, 154

[101] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA*. Wiley, New York, NY, 1997. 44

[102] Charles Owen, Arthur Tang, and Fan Xiao. ImageTclAR: A blended script and compiled code development system for augmented reality. In *Proceedings of the International Workshop on Software Technology for Augmented Reality Systems (STARS)*, October 2003. 27

[103] Juha Pärssomem, Teemu Koponen, and Pasi Eronen. Pattern lanugage for service discovery. In *Ninth European Conference on Pattern Languages of Programs (EuroPLoP)*, July 2004. 102, 103, 104

[104] W. Pasman and F. W. Jansen. Distributed low-latency rendering for mobile ar. In *Proceedings of the International Symposium on Augmented Reality (ISAR)*, October 2001. 36, 41

[105] Brenton Phillips. Metaglue: A programming language for multi-agent systems. M.eng thesis, MIT, 1999. 34

[106] W. Piekarski and B. H. Thomas. An object-oriented software architecture for 3d mixed reality applications. In *Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR)*, October 2003. 28

[107] Daniel Pustka. Visualizing distributed systems of dynamically cooperating services. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2003. 63, 125, 127

[108] Daniel Pustka. Handling error in ubiquitous tracking setups. Diplomarbeit, Technische Universität München, Institut für Informatik, 2004. 60, 114

[109] Thomas Reicher. *A Framework for Dynamically Adaptable Augmented Reality Systems*. PhD thesis, Fakultät für Informatik, Technische Universität München, 2004. 13, 46, 49, 51, 68, 90, 133, 136, 147, 154

[110] Thomas Reicher, Asa MacWilliams, and Bernd Bruegge. Towards a system of patterns for augmented reality systems. In *Proceedings of the International Workshop on Software Technology for Augmented Reality Systems (STARS)*, October 2003. 37, 92, 147

[111] Gerhard Reithmayr and Dieter Schmalstieg. OpenTracker – an open software architecture for reconfigurable tracking based on XML. Technical report, TU Wien, 2000. 14, 28

[112] Gerhard Reitmayr and Dieter Schmalstieg. Data management strategies for mobile augmented reality. In *Proceedings of the International Workshop on Software Technology for Augmented Reality Systems (STARS)*, October 2003. 16

[113] Florian Reitmeir. Benutzerorientierter Datenfilter für Umgebungsinformationen als Erweiterung eines Navigationssystems für sehbehinderte und blinde Mitmenschen. Diplomarbeit, Technische Universität München, Institut für Informatik, 2003. 138

[114] Lothar Richter. Design and practical guideline to a CORBA-based communication framework. Studienarbeit, Technische Universität München, Institut für Informatik, 2002. 99

[115] Manuel Roman, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, pages 74–83, Oct-Dec 2002. 32

[116] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI*, pages 434–441, 1999. 4

[117] Christian Sandor and Gudrun Klinker. A rapid prototyping software infrastructure for user interfaces in ubiquitous augmented reality. In *Journal for Personal and Ubiquitous Computing*. Springer Verlag, 2004. To appear. 17, 131

[118] Christian Sandor, Asa MacWilliams, Martin Wagner, Martin Bauer, and Gudrun Klinker. Sheep: The shared environment entertainment pasture. In *Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR)*, October 2002. 55

[119] D. Schmalstieg, A. Fuhrmann, G. Hesina, Zs. Szalavari, L. Miguel Encarnação, M. Gervautz, and W. Purgathofer. The Studierstube Augmented Reality Project. *Presence*, 11(No.1), 2002. 3, 17, 18, 28

[120] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture - Pattern for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, Ltd., 2000. 156, 160

[121] R. Schönfelder, G. Wolf, M. Reeŝing, R. Krüger, and B. Brüderlin. A pragmatic approach to a VR/AR component integration framework for rapid system setup. In *Proceedings of the Paderborn Workshop "Augmented und Virtual Reality in der Produktentstehung"*, Paderborn, 2002. 29

[122] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002. 45, 121

[123] João Pedro Sousa and David Garlan. AURA: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In Bosch, Gentleman, Hofmeister, and Kuusela, editors, *Software Architecture: Design, Development, and Maintainance (Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture)*, pages 19–43. Kluwer Academic Publishers, August 2002. 30

[124] João Pedro Sousa and David Garlan. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. Technical Report CMU-CS-03-183, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213-3890, August 2003. 30

[125] Franz Strasser. Evaluation of CORBA implementations for small wearable computers. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2002. 141

[126] Franz Strasser. Personalized ubiquitous computing with handhelds in an ad-hoc service environment. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2003. 58, 105, 138

[127] Franz Strasser. Bootstrapping of sensor networks in ubiquitous tracking environments. Diplomarbeit, Technische Universität München, Institut für Informatik, 2004. 60

[128] Fabian Sturm. Issues in view management for 3d objects on multiple displays. Diplomarbeit, Technische Universität München, Institut für Informatik, 2004. 63

[129] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990. 125

[130] Peter Tandler. Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices. In *Proceedings of UbiComp 2001: Ubiquitous Computing*, number 2201 in LNCS, pages 96–115. Springer Verlag, Heidelberg, 2001. 31

[131] Peter Tandler. The BEACH application model and software framework for synchronous collaboration in ubiquitous computing environments. *Journal of Systems and Software*, 69(3):267–296, 2004. 31

[132] Peter Tandler. *Synchronous Collaboration in Ubiquitous Computing Environments*. PhD thesis, Technische Universität Darmstadt, Fachbereich Informatik, to be published in 2004. 31

[133] Bruce Thomas, Ben Close, John Donoghue, John Squires, Phillip De Bondi, Michael Morris, and Wayne Piekarski. ARQuake: An outdoor/indoor augmented reality first person application. In *ISWC*, pages 139–146, 2000. 10

[134] Marcus Tönnis. Data management for augmented reality applications. Diplomarbeit, Technische Universität München, Institut für Informatik, 2003. 58, 59, 130, 131

[135] Jörg Traub, Marco Feuerstein, Martin Bauer, Eva U. Schirmbeck, Hesam Najafi, Robert Bauernschmitt, and Gudrun Klinker. Augmented reality for port placement and navigation in robotically assisted minimally invasive cardiovascular surgery. In *8th Annual Conference of the International Society for Computer Aided Surgery (ISCAS)*, 2004. 138

[136] Shinji Uchiyama, Kazuki Takemoto, Kiyohide Satoh, Hiroyuki Yamamoto, and Hideyuki 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, Oct. 2002. 3

[137] Martin Wagner. *Tracking with Multiple Sensors*. PhD thesis, Fakultät für Informatik, Technische Universität München, 2005. To appear. 60, 149

[138] Martin Wagner, Sven Hennauer, and Gudrun Klinker. Easing the transition between multiple trackers. In *Proc. of IEEE and ACM International Symposium on Mixed and Augmented Reality*, Arlington, VA, USA, November 2004. 146

[139] Martin Wagner, Asa MacWilliams, Martin Bauer, Gudrun Klinker, Joseph Newman, Thomas Pintaric, and Dieter Schmalstieg. Fundamentals of ubiquitous tracking. In *Second International Conference on Pervasive Computing, Hot Spots section*, Vienna, Austria, April 2004. 60

[140] Jim Waldo. Jini architecture overview. Sun Microsystems, `http://java.sun.com/products/jini/whitepapers/architectureoverview.pdf`, February 2001. 101

[141] Mark Weiser. Hot topics: Ubiquitous computing. *IEEE Computer*, Oct. 1993. Three topics: wireless communication, mobility, window systems. 1, 7

[142] Johannes Wöhler. Driver development for touchglove input device for dwarf based applications. Systementwicklungsprojekt, Technische Universität München, Institut für Informatik, 2003. 58

[143] Timo Wolf and Allen Dutoit. A rationale-based analysis tool. In *13th International Conference on Intelligent & Adaptive Systems and Software Engineering (IASSE'04)*, 2004. 131, 150

[144] Bernhard Zaun. Calibration of virtual cameras for augmented reality. Diplomarbeit, Technische Universität München, Institut für Informatik, 2003. 58

Index