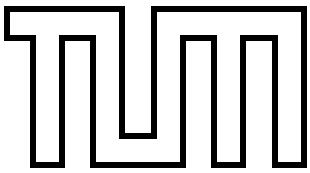
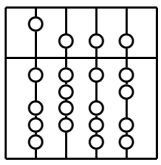


Technische Universität
München
Fakultät für Informatik



Systementwicklungsprojekt

Personalized Ubiquitous Computing with Handhelds in an Ad-Hoc Service Environment

Franz Strasser

Aufgabensteller: Univ-Prof. Bernd Brügge, Ph.D.

Betreuer: Dipl.-Inf. Asa MacWilliams

Abgabedatum: 15. April 2003

Erklärung

Ich versichere, dass ich diese Ausarbeitung des Systementwicklungsprojektes selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. April 2003 Franz Strasser

Abstract

Current AR application are mostly static configurated systems which only hardly or not at all be altered by the user at runtime to fit his/her specific needs. However, the DWARF framework enables the system to provide default configuration values which can be merged or overwritten by personal preferences of the users. This work introduces concepts which are realized in the ARCHIE system by using DWARF.

Overview

1 Introduction 1
Introduction, Goals of my work, DWARF

2 The ARCHIE system 7
Common chapter of the ARCHIE project: The description of the ARCHIE system

3 The application of handhelds in AR scenarios 17
Using handhelds in AR applications overs new possibilities

4 Ad-Hoc configuration and personalisation 21
Handhelds can be used to store user-specific configuration information. They can be used to configure AR applications

5 Personalized selection of services 27
Application example: Using handhelds to choose pre-configured DWARF services at runtime

6 Internals of the `Selector` service 33
Programming internals of the selection core service

7 A security model for DWARF 37
Personalisation needs a security model for DWARF: first ideas

8 The ARCHIE demo 41
Usage of the described components in the ARCHIE demo

9 Future work 45
Concepts which can be discussed

Contents

1 Introduction **1**
- 1.1 Handhelds and their applications . . . 1
- 1.2 Security in Ad-hoc systems . . . 1
- 1.3 Goals of this work . . . 2
- 1.4 The DWARF framework . . . 3
- 1.5 Service Descriptions and Attributes . . . 3
- 1.6 DWARF applications and service chains . . . 5

2 The ARCHIE system **7**
- 2.1 Problem Statement . . . 7
- 2.2 Scenarios . . . 8
- 2.3 Requirements . . . 13
  - 2.3.1 Functional Requirements . . . 13
  - 2.3.2 Nonfunctional Requirements . . . 14
- 2.4 System Design . . . 15

3 The application of handhelds in AR scenarios **17**
- 3.1 Usage of handhelds as AR devices . . . 17
- 3.2 Scenarios from SHEEP . . . 18
- 3.3 Scenarios from ARCHIE . . . 18
  - 3.3.1 Application selection . . . 18
  - 3.3.2 TouchGlove profiles . . . 20
  - 3.3.3 Summary . . . 20

4 Ad-Hoc configuration and personalisation **21**
- 4.1 The approach for personalisation . . . 21
  - 4.1.1 Personalized configuration . . . 22
  - 4.1.2 Personalized selection of services . . . 22
- 4.2 Distributed Configuration of DWARF applications . . . 23
- 4.3 The `Configuration` service . . . 23
- 4.4 The `Preference` service . . . 24
- 4.5 Modeling Context with Attributes . . . 24
- 4.6 Example from ARCHIE: TouchGlove Profiles . . . 25

5 Personalized selection of services **27**
- 5.1 Selection of service chains . . . 27
- 5.2 Using a `Selector` service . . . 28
  - 5.2.1 The Delegated Service . . . 28
  - 5.2.2 Modifying the partner services . . . 30

6 Internals of the `Selector` service **33**
6.1 Desing of the `Selector` components . . . 33
6.1.1 The system design . . . 33
6.1.2 The object design . . . 34
6.2 The `Collector` class . . . 34
6.3 The Service Description . . . 36

7 A security model for DWARF **37**
7.1 Distributed Authentication . . . 37
7.1.1 The Kerberos Protocol . . . 37
7.1.2 Authentication Protocol for DWARF . . . 37
7.2 Roles and Certificates . . . 38
7.3 Data Structures for the Communication . . . 39
7.4 Conclusion . . . 40

8 The ARCHIE demo **41**
8.1 The demo set up . . . 41
8.2 The Application `Selector` . . . 41
8.3 The overall system . . . 41

9 Future work **45**
9.1 An improved `MenuDisplay` component . . . 45
9.2 Realisation of the `Preference` service . . . 45
9.3 Implementation and evaluation of the security model . . . 46
9.4 User profile for `TouchGlove Interpreter` services . . . 46

A The `MenuDisplay` service **47**
A.1 `MenuDisplay` IDL data types and interfaces . . . 47
A.2 The `ListMenuDisplay` service . . . 48

Bibliography **51**

List of Figures

1.1 Example XML description . . . 4
1.2 Tracker service chain . . . 5

2.1 The ARCHIE selection menu displayed on the iPaq . . . 9
2.2 HMD calibration with a pointing device . . . 10
2.3 Modeling and Form Finding . . . 11
2.4 Hardware setup for the location awareness . . . 12
2.5 Presentation of a planned building to an audience . . . 12
2.6 Live visualization of user performance in usability study . . . 13
2.7 ARCHIE architecture . . . 16

3.1 A classification of application fields of handhelds for AR . . . 17
3.2 The SHEEP demo . . . 19
3.3 The *TouchGlove* subsystem . . . 20

4.1 The `UserProxy` service . . . 21
4.2 Configuration of DWARF applications . . . 22
4.3 The `Configuration` service . . . 23
4.4 Configuration of a `TouchGlove Interpreter` service . . . 25

5.1 Ambiguities with two service chains . . . 27
5.2 The `ArchieUIC` service . . . 31
5.3 The `UserProxy` service . . . 31

6.1 The `Selector` service: bridge between middleware and user . . . 33
6.2 The system design of the `Selector` service . . . 34
6.3 The `Selector` XML Service Description . . . 36

7.1 The `UserInfo` structure . . . 39
7.2 The `Certificate` structure . . . 39

8.1 The application `Selector` for ARCHIE . . . 42
8.2 Screenshot of DIVE showing the whole ARCHIE application . . . 43

9.1 Design concept for a `Wizard` service . . . 46

A.1 The core data structure of the MenuDisplay services . . . 47
A.2 XML description of the `ListMenuDisplay` service . . . 48
A.3 Example screenshot of the GUI part of the `ListMenuDisplay` service . . . 49

1 Introduction

Introduction, Goals of my work, DWARF

1.1 Handhelds and their applications

Since new technologies enable new generations of wearable computers, so called *handhelds* or *PDA*s, it is now possible to find new application sectors. One of them is Augmented Reality, because using handhelds for new input devices in AR applications and as well as small output devices gives more flexibility: you can use the device itself as a tangible object while getting information on the display.

Another field is Ubiquitous Computing. Handhelds will be the "swiss army knife" for the next generation: you can enter an intelligent building and use distributed services provided by the network infrastructure there. This is realest by ad-hoc networking, which means, that the handheld can connect to the service infrastructure without knowing it in advance. The information is delivered by the system itself and not by the user.

A difficult issue in ad-hoc systems is personalization, which means enabling individual, configurable services to the user and ensuring that only authorized users can connect to their own services. The main problem is the difficulty to establish and control security mechanism, because the infrastructure changes dynamically when new previously unknown users connect and many different heterogeneous components must communicate with each other.

So the problem is: How can a distributed system offer individual services to the users and maintain simultaneously a certain level of security?

1.2 Security in Ad-hoc systems

Establishing security mechanism in ad-hoc networks is not trivial because of the decentralized character of such systems: every client can connect to the system anytime and anywhere. However even in such decentralized environments, there must exist one authority which is responsible for granting rights to the services. By registering with this authority, the identification of the user is checked and a certificate for each accessible application is created. Users, who are not allowed to use the system, do not own a valid certification and can be rejected before even the application is started and configured. After that point, intruders in running systems can be detected since public key mechanisms identify the user.

By introducing an access control model, it is also possible to distinguish between the different users and to save personal preferences. These preferences are needed to reconfigure the system at runtime. To avoid contacting this central component again when an access

to an application is desired, the authority generates *certificates* which are then stored on the user's PDA.

1.3 Goals of this work

The goal of this work is to enhance DWARF, a framework for building AR applications, to allow running of personalized services on wearable computers. User-specific configuration cannot be integrated into running applications developed with the current framework. This reduces flexibility when developing more advanced applications. For example in the ARCHIE project, the required authentication ensures that the user can only choose between the applications which are available to him (i.e. the main ARCHIE application and HMD calibration) and can load personal preferences to configure the application (i.e. personal HMD calibration values).

The goal of my work is to use small wearable computers with Augmented Reality applications. There are two main focuses of the work:

Setting up the infrastructure The first step is to set up the infrastructure for using the Compaq iPaq as a fully supported device in a DWARF application. This requires the DWARF Service Manager and a suitable CORBA implementation for small StrongARM devices. Since native compilation on an iPaq is not desirable, a cross-compiling environment has to be set up to build the ORB libraries as well as DWARF for StrongARM processors on Intel Linux machines. Then the ORB libraries and the Service Manager had to be ported to the iPaq. The first demo setup which used this handheld as I/O device was the SHEEP demo [13]. Details of this step can be found in a separate SEP documentation [14].

Running personalized services In the second goal is to find a new model for user dependencies. In current applications time and location can already be implemented as part of the application's context. The new personalization model enables the developer to design personalized context-aware applications. By using the new handheld technologies it is also possible to hold the user context information at the place where it belongs to: the user. User-specific services can run on the actual handheld and connect to the system at runtime. An example application which uses these new methods will be the ARCHIE system.

A security model for DWARF The original idea behind the above mentioned focus is a long term goal: a security model for DWARF. The problem statement for each project can define several user roles for the application with different rights. To configure the application for use within a certain role, the user has to identify himself to the system and provide personal preferences. The identification can be checked via a passport service running on his handheld as well as with traditional login mechanism like user name and password entered in a terminal in the room. An application selection service and a personal configuration service are now started to enable access to the system.

1.4 The DWARF framework

The Distributed Wearable Augmented Reality Framework (DWARF) [1], which was developed at the TU München, is focusing on combining the aspects of AR and Ubiquitous Computing. The framework consists of the distributed DWARF Service Manager which dynamically locates and connects the different services, e.g. tracker or view services. The idea behind DWARF was that using distributed components on several machines and connecting them ad-hoc will provide more flexibility when developing new AR applications. It is possible to build a new AR system using already existing services. For example, a tracker service or a viewer component can be reused by more applications; in some cases even at the same time.

The connections between the components is handled by *Needs* and *Abilities*. If a service offers certain data types or methods for others then it has an ability which describes the used interface. When another service has a respective need for it, the two services are connected. Every service delivers its own *Service Description*. It contains all information which is necessary for using this service in the system. This information contains for example:

- the name of the service, which is used to identify it,
- several service flags (e.g. startOnDemand),
- service or context *attributes* (see chapter 1.5 on page 3 for detailed information on attributes),
- all *Needs* the service has including the kind of data interface is needed and
- all *Abilities* it offers including the interface description (like the need).

These Service Descriptions are stored in XML files which are evaluated by the *Service Manager*. This component is located on every node of the system and dynamically finds all other managers in the net. Every Service Manager knows its locally runnable services and locates and connects matching services by evaluating every available description. Therefore it parses the local XML files or contacts remote Service Managers to exchange their information.

1.5 Service Descriptions and Attributes

The base mechanism to get configuration and context information into the running DWARF system, is through Service Descriptions. As already mentioned before, Service Descriptions are used by the Service Manager to specify the interfaces and relations of a service to others. This section is only an introduction. For detailed information, it is recommended to read [9].

Needs and Abilities The relationships between services are modeled by needs and abilities. If a services requires data to perform a certain task, it must have a *need* for this data. If a service provides data, it has a correspondig *ability*. The specification of a DWARF need or ability contains three major parts:

- A name for the identification of the need or ability

```
<service name="TouchGloveInterpreterService" >
   <attribute name="timeLimit" value="200"/>
   <attribute name="buttonTimeLimit" value="500"/>

   <ability name="sendAnalog" type="InputDataAnalogLimited">
      <attribute name="configurationKey" value="*"/>
      <attribute name="numOfInstances" value="0"/>
      <connector protocol="PushSupplier"/>
   </ability>

   <ability name="sendBool" type="InputDataBool">
      <attribute name="configurationKey" value="*"/>
      <attribute name="numOfInstances" value="0"/>
      <connector protocol="PushSupplier"/>
   </ability>


   <need name="receiveRawData" type="TGloveDataRaw">
      <connector protocol="PushConsumer"/>
   </need>

   <need name="getConfiguration" type="Configuration"
                     predicate="application=TouchGloveProfile">
      <attribute name="application" value="TouchGloveProfile"/>
      <attribute name="configurationKey" value="*"/>
      <connector protocol="ObjrefImporter"/>
   </need>

</service>
```

Figure 1.1: Example XML Service Description of the `TouchGlove Interpreter` service

- A type which describes what information is provided or needed
- The used communication protocol for exchanging this information

Attributes Needs and abilities can have *attributes* which are informal name/value pairs which can contain additional information about the current need/ability. They can be used to restrict possible communication partners by specifying a *predicate* in a need. This limits all possible connections to those who have the given attribute name set to the given value. Besides attributes for every need and ability, the Service Description can include *service attributes*. They are valid for the whole service i.e. for every need and ability. This type of attributes is later used for the *context attributes* (see 4.5).

Template Services and Wildcard Attributes In SHEEP [13] and ARCHIE this system was enhanced by several new key mechanism which make personalization possible. Attributes

in Service Descriptions need not be filled with a certain value. They can contain an asterisk (*) which stands for a wildcard. These kind of attributes are allowed in abilities and services (see 4.5). If an ability contains such attributes, it is called *Template Ability*; if a service contains such attribute it becomes a *Template Service*. This work will focus on the use of template services, because currently template abilities are not used in ARCHIE.

A template service can be connected to all services which provide a defined value for the wildcard attribute. But a template is not a runnable version of a service, because all attributes have to be set. The DWARF Service Manager parses these description and instantiates for every possible connection a new service, a *clone*. These clones are then connected to the services with the defined attributes. This new technique was first introduced in [8]. An example description is the `Selector` service (figure 6.3 on page 36).

1.6 DWARF applications and service chains

Applications are developed by using separated DWARF services with adapted Service Descriptions. These interdependent services are forming *chains* which are providing a certain sub-functionality of the whole system. For example, the tracking sub-system can consist of more services[1]:

1. The `Tracker` service computes the position of markers in the tracked area. It delivers raw data containing 6D position matrix and a marker ID.
2. This information is used by a `Calibration` service to map the marker on tangible user interfaces (TUI) or other objects which are relevant for the system.
3. A `Collision Detection` uses calibrated input data to determine collisions between real and virtual/real objects and sends events for every collision.

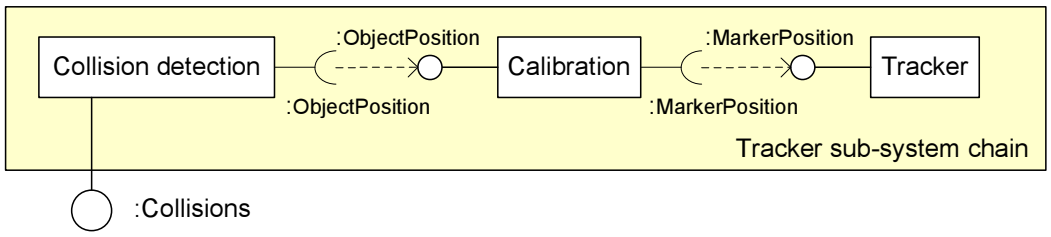


Figure 1.2: Tracker service chain

By grouping of services into chains the abilities or needs of the ends of the chain become the interface services for the outside. This model simplifies the development process and improve the re-usability of whole services.

[1] You can also find a more detailed example in [11].

2 The ARCHIE system

Common chapter of the ARCHIE project: The description of the ARCHIE system

This chapter introduces the ARCHIE project[1]. The name is an acronym for

Augmented **R**eality **C**ollaborative **H**ome **I**mprovement **E**nvironment.

ARCHIE is an interdisciplinary project between the Universität Stuttgart represented by Manja Kurzak, a student graduating in architecture [7] and a team consisting of seven members from the Technische Universität München. Manja Kurzak provided information about design processes and the planning of the construction of e.g. public or private buildings. Her information is summarized in the following problem statement section.

This project provided a starting point for requirements elicitation on the different single projects of all team members. It was intended to use ARCHIE in a prototypical implementation to give a proof of the requested components and their underlying concepts. To build an application with full functionality for architectural requirements was never a project goal.

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

2.1 Problem Statement

A *Problem statement* is a brief description of the problem the resulting system should address [2].

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

[1] This chapter was part of the group diploma theses of the ARCHIE members (see `http://wwwbruegge.in.tum.de/projects/lehrstuhl/twiki/bin/view/DWARF/ProjectArchie` for more details). It was shortened to the relevant details which are necessary for this SEP documentation.

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

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

2.2 Scenarios

A *Scenario* is a concrete, focused, informal description of a single feature of a system. It is seen from the viewpoint of a single user [2]. This section describes the Augmented Reality relevant scenarios of the ARCHIE system. The following list does not describe a full architectural system, because the ARCHIE project is only intended to be a baseline for the development of reconfigurable DWARF services.

Scenario: **Selecting Current Task**

Actor instances: `Alice:User`

Flow of Events:
1. Alice enters her laboratory which contains the necessary environment for the ARCHIE application such as a *ARTtrack 1* tracking system. Furthermore she is wearing a backpack with several objects mounted on it. There is for example a laptop that provides the viewing service for her HMD. She also holds an iPaq in her hand.
2. As her iPaq attains the range of the wireless ARCHIE-LAN, its Service Manager connects to the one running in the laboratory, and they ex-

change their service information to each other. Now the `Selector` service knows the available applications, and the menu pictured in figure (2.1) is displayed on the iPaq.

3. By selecting either entry and confirming, Alice can start the desired task.

Figure 2.1: The ARCHIE selection menu displayed on the iPaq

Scenario: **Calibrating the Devices**

Actor instances: `Bridget:User`

Flow of Events:

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

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

Figure 2.2: HMD calibration with a pointing device

Scenario: **Modeling and Form Finding**

Actor instances: `Charlotte, Alice:User`

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

Scenario: **Mobility - Location Awareness**

Actor instances: `Dick:User`

Flow of Events:
1. Dick starts the location-awareness subsystem based on the ARToolkit [5], running on his laptop attached to his backback. In addition to the former setup an iBot camera is mounted on his shoulder to deliver video images.
2. The system is configured with the current room information. The information consists of the current room and the outgoing transitions (doors) to other rooms. A pie-menu giving information about the current services of the room appears on the laptop.
3. Dick exits the room while detecting an ARToolkit marker attached to the

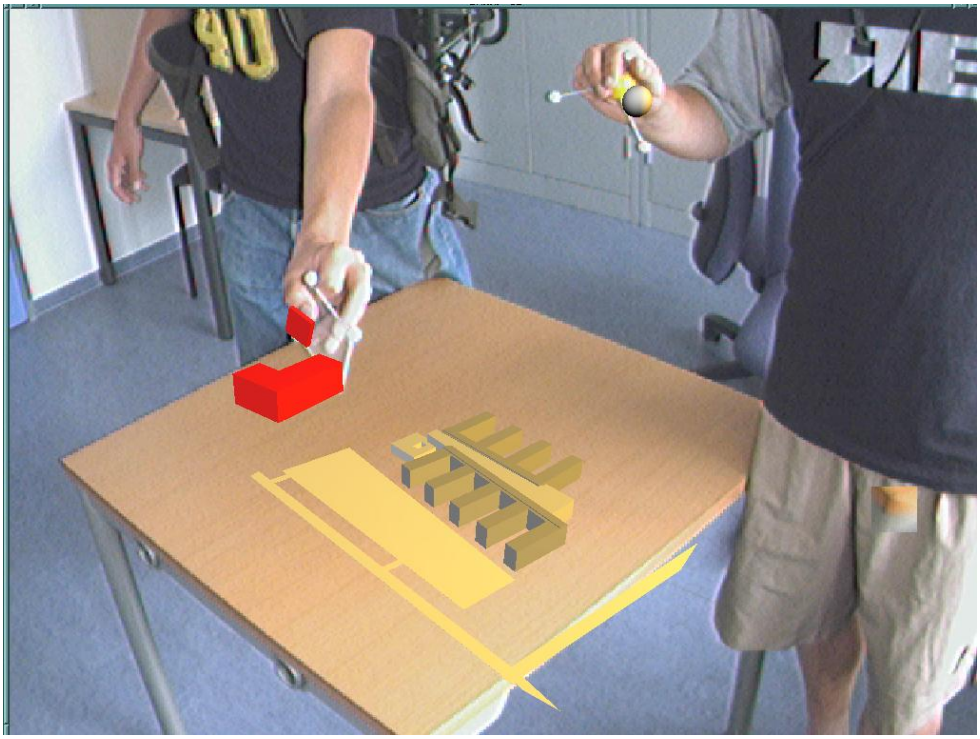


Figure 2.3: Modeling and Form Finding

door with his iBot camera. The system changes its configuartion according to the new state (new room). The old information is dropped and Dick has a new set of services availabe now which can be used in the current environment dynamically. The pie-menu is updated.

4. Dick is able to exit and enter rooms and is enabled to use the corresponding services and room environment.

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

Scenario: **Presentation**

Actor instances: `Alice:User`

Flow of Events:

1. Alice starts the system, but instead of the previous used HMD, now a video beamer view is started, providing scenes as seen from a tangible camera object.
2. Alice takes this camera and moves it around the virtual model of the planned building.
3. The public can get a spatial understanding of the proposed building which is displayed on the beamer screen. The model shown in figure 2.5 is rendered in anaglyphic red-cyan 3D. For a realistic 3D view the visitors need to wear the corresponding red-cyan glasses.

Figure 2.4: Hardware setup for the location awareness

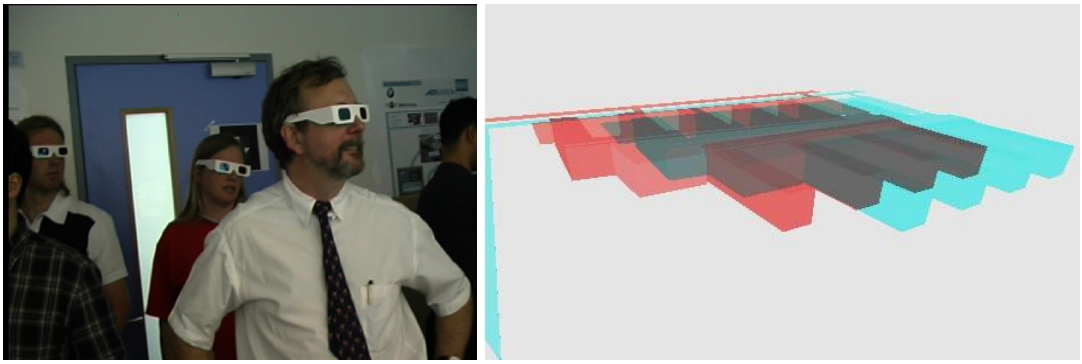


Figure 2.5: Presentation of a planned building to an audience

Scenario: **User Interaction Evaluation**

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

Flow of Events:

1. Gabriel starts the ARCHIE application and configures it for an usability evaluation task.
2. He sets up the logging system and initializes it with the study and task name Felicia will be performing for an unambiguous log file.
3. After briefing Felicia appropriately, she is asked to perform a number of tasks which are being monitored.
4. The logging system is automatically taking task completion times while incrementing the task counter too, so Gabriel can fully concentrate on Felicias reactions to the system.
5. While Felicia is performing her tasks, Gabriel observes a number of real-time, updating charts which visualize her performance by e.g. applying standard statistical functions.

6. During the course of the study, Gabriel is always fully aware of what Felicia is seeing in her augmented display by looking at a special screen which duplicates Felicias HMD view.
7. Felicia is debriefed after she has completed posttest questionnaires handed out by Gabriel.

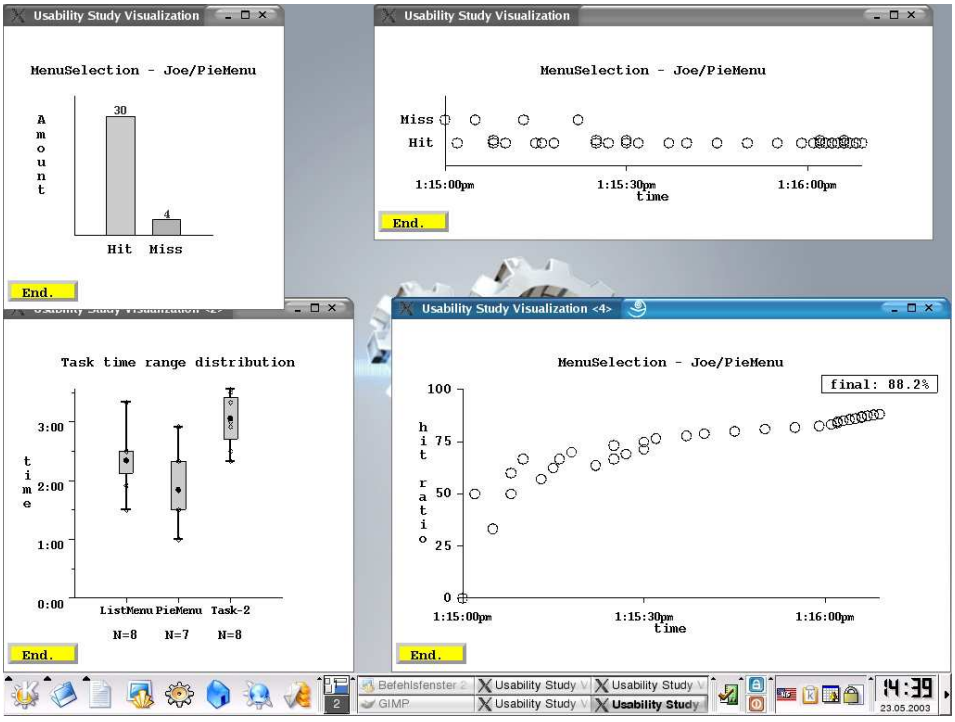


Figure 2.6: Live visualization of user performance in usability study

2.3 Requirements

This section describes the requirements our team elicited during the startup phase of the ARCHIE project. The methodology described in [2] was used to refine them step by step.

2.3.1 Functional Requirements

Functional Requirements describe interactions between a system and its environment [2]. They are independent from the implementation, but can get realized almost directly in code. This section declares the *Functional Requirements* for the ARCHIE system.

Modeling process

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

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

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

Collaborative work

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

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

2.3.2 Nonfunctional Requirements

In contrast to the *functional requirements*, the *nonfunctional requirements* describe the user-visible aspects of a system [2]. These may not directly relate to the system's functional behavior. *Nonfunctional requirements* can be seen as overall requirements a system must fulfill. They influence multiple locations allover the later realization. In decomposition they can be seen as *functional requirements*, but this view would be hard to understand for the client during requirements elicitation. This section reveals the *nonfunctional requirements* of the ARCHIE project that lead to the design goals of general DWARF framework components.

Augmentation

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

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

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

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

Ubiquity

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

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

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

The development for framework components also yield requirements.

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

Dynamic Application Configuration Framework components may rely on context information. These services must be configurable via interfaces as described in [8].

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

2.4 System Design

An overview over the architecture of ARCHIE is shown in figure (2.7).

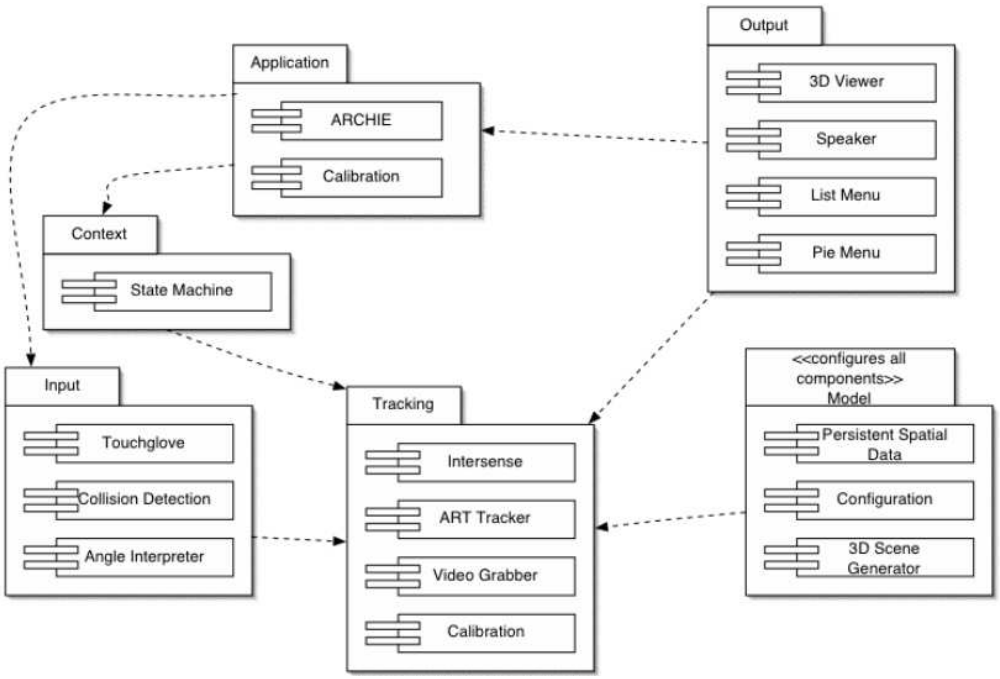


Figure 2.7: ARCHIE architecture

3 The application of handhelds in AR scenarios

Using handhelds in AR applications overs new possibilities

This chapter gives an overview over the possibilities of using wearable computers, so called *handhelds*, in Augmented Reality applications. The focus in this chapter will be the coverage of the demo systems SHEEP and ARCHIE.

3.1 Usage of handhelds as AR devices

Handhelds are very flexible devices, since their main purpose is to provide a variety of tasks. The classical field for an wearable computer is providing information for the user when he needs it. So the interaction between the user and the device is the primary focus. Communication between the device itself and its environment is now possible since new wireless technologies for networks are available. This offers new opportunities of interactions between handhelds and the environment, especially in the field of Augmented Reality applications. A possible classification of the usage of handhelds can be found in figure 3.1.

- **Input device**
  One application field is using the handheld as an *classical input device* for the system. You can use the touchpad of the screen for sending mouse clicks, i.e. button click events [16], or string events through the gesture recognition.

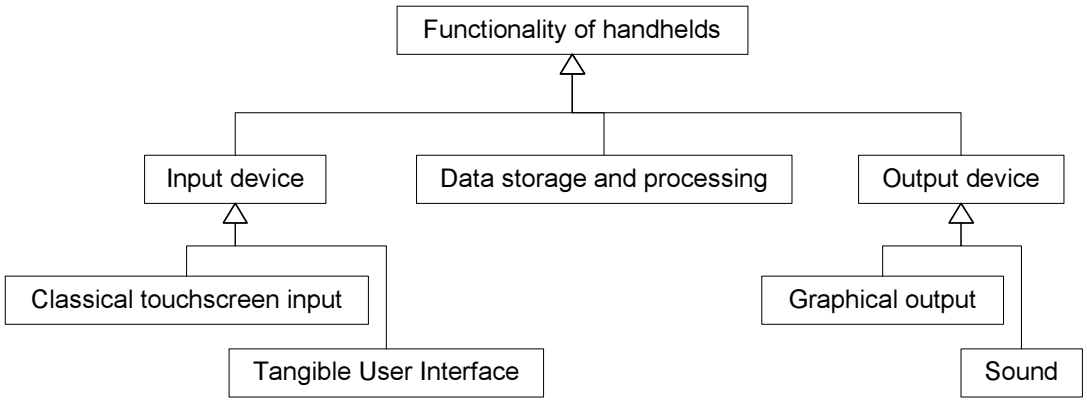


Figure 3.1: A classification of application fields of handhelds for AR

Especially since the device is used within an Augmented Reality applications, we can use tracking technologies to estimate the location of the wearable. In this way it is possible to calculate collision data with other real or virtual objects. The device itself becomes a *tangible input device*.

- **Output device**
  You can also use the output capabilities of the handhelds. The most basic possibility is the screen itself. By using 2D or 3D techniques, the user can have an overview of the status information or can see virtual 3D objects (*graphical output*).

  Additionally, *speech output* can be preferable in some situations if the hardware supports audio devices.

- **Data storage and processing**
  This is the main advantage of a wearable computer: to save and process information before it is distributed or needed in the system. Since every device can be associated with one user[1], personal data can easily be stored on the device.

By combining several of these techniques (shown in figure 3.1), new application sectors can be found. In the SHEEP and ARCHIE demos, a Compaq iPaq is used since it is possible to run Linux on this architecture.

3.2 Scenarios from SHEEP

In the SHEEP demo the iPaq is used as tangible input device with graphical output. You can scoop sheeps with the handheld which can then be kept in the iPaq and released somewhere else on the table. So the wearable was primarily used as an interaction device between the user and the system. I will not go into detail here because the SHEEP demo is covered by other papers, especially [13], [3] and [14]. You can find detailed information there.

3.3 Scenarios from ARCHIE

In ARCHIE the iPaq is not used as a tangible user interface but again its classical usage as wearable computer is preferred to control the system. Moreover, the iPaq stores additional data of its user, so it becomes a "wearable database".

3.3.1 Application selection

The iPaq realizes the first scenario *"Select the current task"* (see 2.2 on page 8). The user enters the room where the ARCHIE application is ready to run. A window pops up on his/her handheld with all possible applications which are offered. By selecting one, i.e. clicking on the entry and confirm by pushing the `Select` button, the application starts up and is ready to run. The handheld was used as a classical input device here.

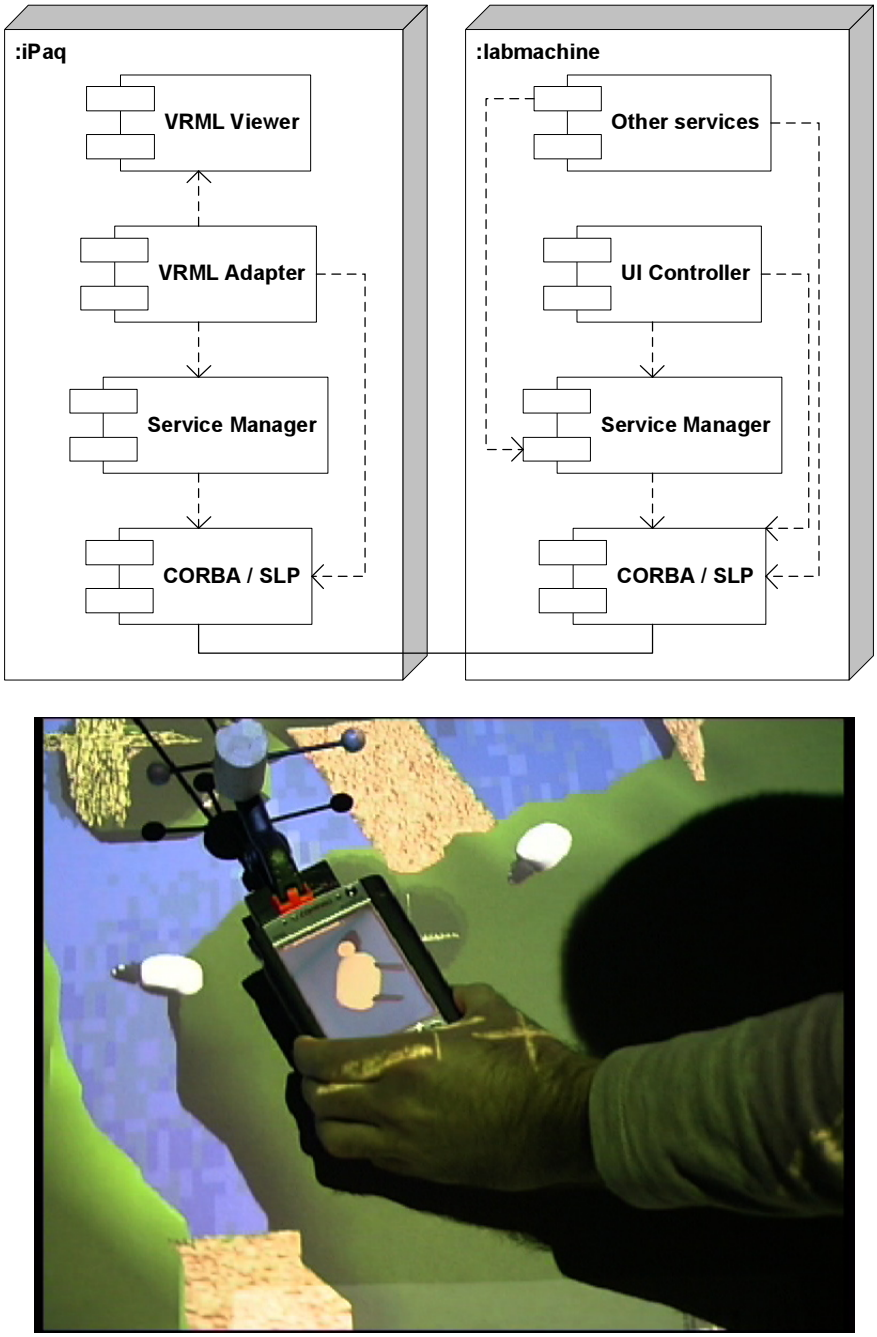


Figure 3.2: The SHEEP demo

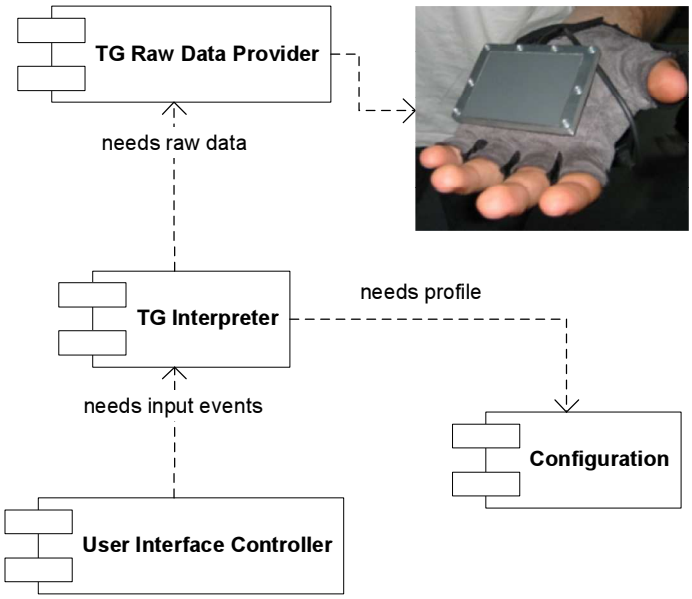


Figure 3.3: The *TouchGlove* subsystem

3.3.2 TouchGlove profiles

Johannes Wöhler built a new input device for ARCHIE called the *TouchGlove* [16] which can be used to provide generic DWARF input data. The touchpad itself can be divided into several regions which are either sliders or buttons. This information is stored as *profiles* (figure 3.3).

The design of the subsystem allows the user to choose his/her own personal profile in addition to the system profiles. User-specific information, like the thickness of the fingers, can be integrated into the systems profile to improve the layout of the sliders and button on the touchpad. This information can easily be stored on the user's PDA and is available on demand. The data storage and processing ability is the primary focus in this case. [2]

3.3.3 Summary

The two above mentioned scenarios in ARCHIE use new methods of user-driven interaction with DWARF. In the first scenario the user implicitly enters information by choosing menu items whereas in the second personal profiles are loaded automatically into the system if they are available. The ARCHIE team realized the scenarios in the demo setup (see chapter 8 on page 41). How these new techniques are implemented is covered in the next chapter.

[1] Normally, a handheld operates in single-user mode as a PDA. So we can assume that a user has control over the device and it is acting for him/her.

[2] This scenario is not mentioned in the ARCHIE scenarios (section 2.2) since it is not part of the ARCHIE problem statement. It is a sub-scenario which is used independently from ARCHIE in any DWARF application.

4 Ad-Hoc configuration and personalisation

Handhelds can be used to store user-specific configuration information. They can be used to configure AR applications

---

This chapter introduces the concepts how personalized configuration is realized in DWARF. By using attributes to distinguish between services which are running especially for specific users, DWARF applications can become user-aware. Handhelds are the ideal place where user-specific services can run.

4.1 The approach for personalisation

DWARF applications were not aware of different users since now. In the SHEEP demo, different types of users (roles) were introduced: the god, the wizard and the iPaq user. No real information about the actual user is known nor it is even needed.

The new approach tries to integrate the user into the system. As discussed in the chapter before, wearable computer can be used to interact with AR systems. Services represent the units of DWARF applications. We can view users of DWARF applications as such units. They can be represented as "services" which have needs for the desired application. This new service can be called `UserProxy` service. It accepts all applications (`application=*`) and runs for user Alice (`user=Alice`).

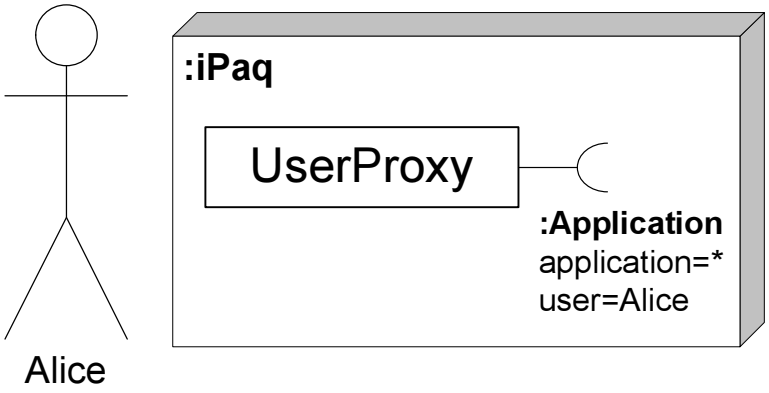


Figure 4.1: The `UserProxy` service

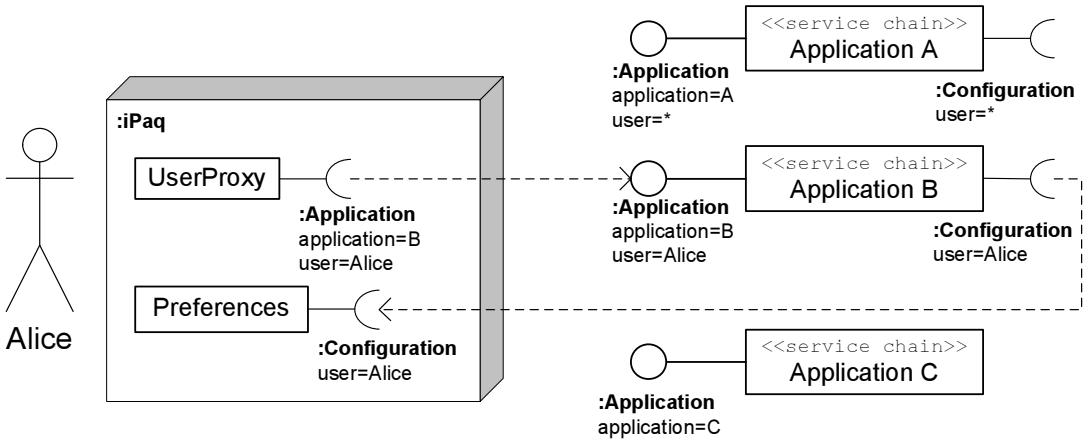


Figure 4.2: Approach: Configuration of DWARF applications by introducing a `Configuration` need

4.1.1 Personalized configuration

Since every user has its own personal handheld, you can use the device to store information on it.[1] By merging this personal configuration information with the system configuration, the user can influence the application's default behavior: The system is pre-configured with reasonable settings which can be replaced or modified with the ones from the user. This is realized by adding a `Configuration` need to configurable service. In diagram 4.2 the application chains consists of all services which form the application and which can be set up by the user. `Application A` and `B` are configurable applications for all users whereas `C` is not. The appropriate attributes are set according to the application name. User-specific configuration data is called *preferences* and they are stored on the user's handheld.

`Application B` is already connected to the `Preference` service from Alice. Why actually this application has connected is discussed in the next section 4.1.2. When this connection has been established, the application chain propagtes the `user` attribute from the `Configuration` need through all services to the `Application` ability.

4.1.2 Personalized selection of services

If the configuration information from the user is not sufficient for forming the user-aware application, ambiguities may exist in the chains which cannot be solved by the middleware automatically via above described mechanism. The source of the ambiguities is the fact that several services can offer similar functionalities. The user must decide himself what service is the best fit for his personal needs. The possible choices are presented to the user who selects one of them interactively. The example in figure 4.2 shows different applications which have all the same ability type `Application`. The `UserProxy` does not know with which application is desired by its user. After the user entered his decision, the `UserProxy` connects to the selected application whereby the application gets configured and the `UserProxy`'s application attribute is set to the name.

[1] The wearable computer acts as an *intelligent cookie*.

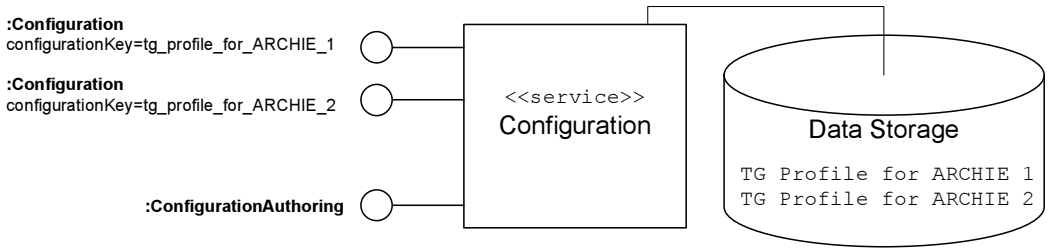


Figure 4.3: The `Configuration` service

4.2 Distributed Configuration of DWARF applications

Configuration is a set of data which is be used to bring additional information into the system, e.g. user context information. Since the set is stored in DWARF services the data can be distributed over the system. Every service can potentially get parts of its configuration from different sources:

The application developer A standard method when writing high-quality applications is to use as generic as possible program and to feed it with additional external information which increases the re-usability of the software: the configuration.

The UI designer When using the same application on different I/O devices, it is necessary to change the user interfaces accordingly. This information can also be stored as external data and be evalutated when needed.

The user Last but not least, the third information source is the user. He should be able to change the application according to his current needs by providing his own set of configuration data.

To provide the possibility of a distributed configuration, there must be two semantically distinguishable databases available for the data: one for the system domain and one for each user. In DWARF these two data-storage components are realized by services which provide a interface for back-end databases.

4.3 The `Configuration` service

Configuration data is provided by the system itself when starting an application. Services, which need additional information before they are able to be used, connect to a `Configuration` service. This service deals as interface between the application services and a database which stores all important information for the system. The `Configuration` parses all database entries and creates for everyone a new ability with a *configuration key*. When a service needs configuration data, it connects to the ability with the appropriate configuration key and gets the data via a pre-defined interface. This service was implemented for the ARCHIE project by Marcus Tönnis. In his diploma thesis [15] you can find a detailed description about the interfaces of this service.

4.4 The `Preference` service

In addition to configuration, services can also need *preferences* which come directly from the user. In contrast to the `Configuration` service which is a central component of DWARF, every user has its own preferences store: the `Preference` service. It can run on the user's handheld or as a central service for all users. It behaves very similar to the `Configuration` service because it parses the backend database and generates the abilities dynamically. The only difference is the service description and the type of the abilities since the services have to distinguish whether they are getting configuration or preferences.

4.5 Modeling Context with Attributes

Context is the information about the environment and the situation of the overall system. Personalization is realized by using context information. This section gives an short introduction on how DWARF can be made context-aware.

By propagating service attributes through the DWARF application (see 1.6 on page 5), the services get aware of context changes throughout the system [11]. Therefore *context attributes* are introduced which are service attribute with a specific semantical interpretation. Context is modeled and exchanged by using the normal propagation mechanism of these attributes. A proposal for valid context attributes are:

`application`: the name of the current running application.
This attribute is used in ARCHIE to distinguish between HMD calibration scenario and the real modeling application ARCHIE.

`location`: the current location of the service.
Every service runs at a special location, e.g. the room number of the computer on which the service is running. This information can be interpreted as location context. In ARCHIE, the mobile scenario was based on this mechanism.

`user`: the actual user identification for which the service is running.
This attribute is part of the user context. It contains the name of the person who is actually using the application. It may also contain session information (e.g. timestamps, session cookies etc.), but currently it only contains the user name.

`role`: the role of the current user.
A role is a classification for different users. When a user identifies himself to the system, he is assigned the roles for which tasks he is permissioned to use. In ARCHIE , the role `Architect` was used to identify subjects which are allowed to use the modeling part.

This four attributes are sufficient to model the context information in current DWARF applications. However, it is not granted that this is a complete list and it will be possibly necessary to add new attributes or refine the existing ones.

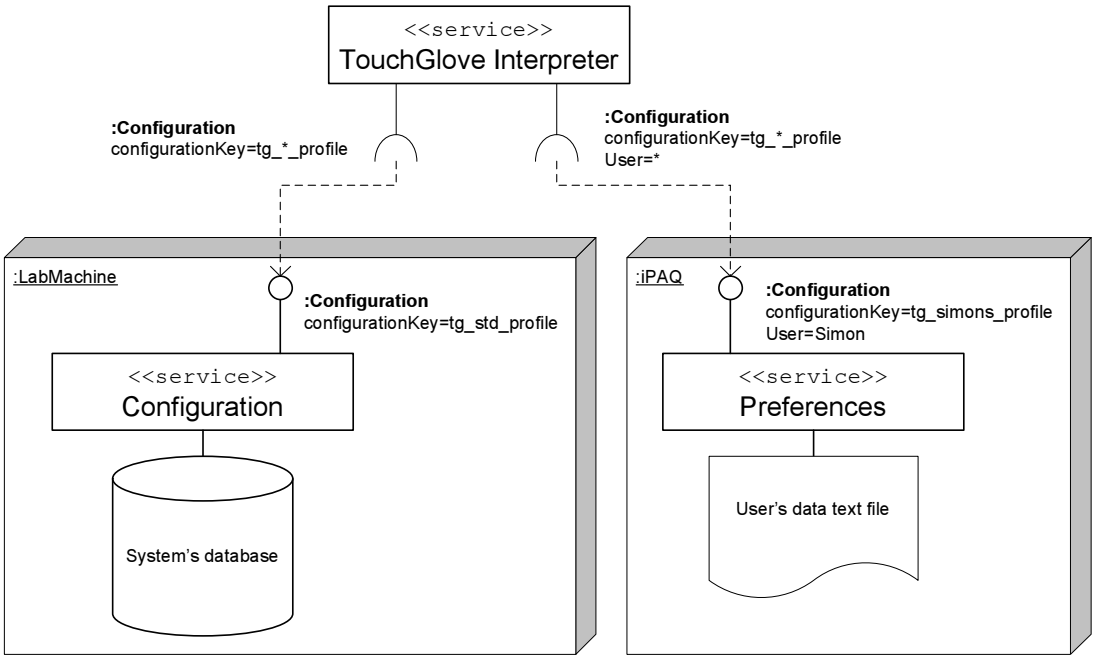


Figure 4.4: There are two different sources which provide a suitable profile for the `TouchGlove Interpreter` service.

4.6 Example from ARCHIE: TouchGlove Profiles

In ARCHIE the above mentioned mechanism are used to form a flexible AR application which can be dynamically configured at runtime. As mentioned in 3.3.2 on page 20 the TouchGlove needs profiles to provide either button clicks or slider values. These profiles depend mainly on the application, i.e. what user events should be generated by this input device. So the standard profile has to be stored for every application in a `Configuration` service. By setting up a need with type `Configuration` the interpreter service gets the desired profile from the database by evaluating the configuration key.

However, these standard profiles are not suitable for every user, e.g. the width of the slider is too small compared to the thickness of the finger or a button cannot be reached at all. These information can be stored in the user's `Preference` service. The `TouchGlove Interpreter` service has a second need for configuration, but with an additional `user` attribute set with an asterisk. This need connects to the `Preference` service since it has a valid and matching `user` attribute (see figure 4.4).

Through this mechanism the `TouchGlove Interpreter` service gets both profile. How the two are merged or one is replaced by the other for a new layout depends on the design and implementation of the interpreter service and the application. Currently only the system's default profile is evaluated since the `Preference` service is not implemented yet.

5 Personalized selection of services

Application example: Using handhelds to choose pre-configured DWARF services at runtime

---

After introducing the two concepts how personalization can be achieved this chapter goes into detail about how ambiguities of not fully-configured personal services can be solved: the personalized selection of services or service chains.

5.1 Selection of service chains

As discussed in the introduction (section 1.6) DWARF applications are generally formed via service chains. Service chains can be configured to suit the desired functionality (chapter 4). But sometimes it is easier that the application developer provide certain pre-configured alternatives instead of implementing a whole configurable chain.

One example would be that the system has more input or output devices which can be used for a specific task. Instead of providing a service chain which gets dynamically configured depending which device the user chooses, it is more convenient for the developer to write services for each device. These new chains are equal in their interfaces (needs and abilities) for their potential partner services. This situation leads to *ambiguities* in the system which cannot be solved at runtime automatically. So the user has somehow to tell the system which device (and with that which service chain) he/she wants to use. Another ex-

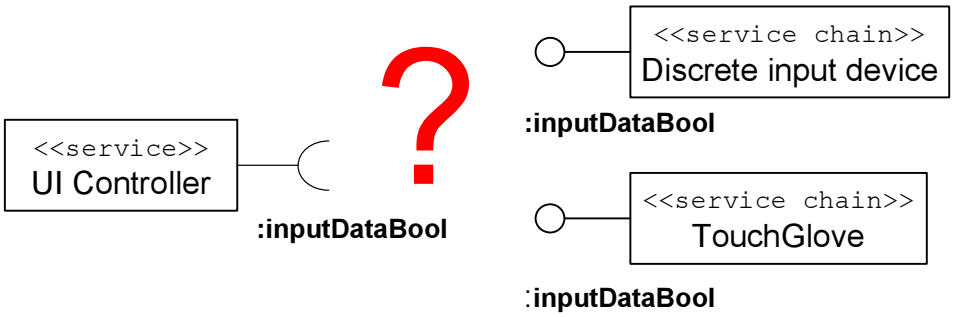


Figure 5.1: Ambiguities with two service chains

ample would be the situation shown in figure 4.2 on page 22: there are several applications available to the user but the system does not know which one it should start and possibly configure.

These two examples illustrate the needs for an additional component which interacts with the user on one side and with the system, especially with the DWARF Service Manager on

the other. This component is the `Selector` service. It handles the ambiguities by presenting the possible choices to user via a selection menu. After the user selected one entry it starts the service (and when set up correctly the entire service chain).

5.2 Using a `Selector` service

Building an application which should use a `Selector` is exactly the same procedure like programming one without it. The interfaces between the services remain the same. The communication is completely independent from the use of a `Selector`. It collects all similar abilities and generates a list of selectable services. This list is sent to a display component for selecting one item out of a menu list: the `MenuDisplay` service (appendix A). The only places where changes are relevant are the Service Descriptions. The following descriptions have to be modified to set up the `Selector` properly:

- The partner services with the similar abilities which lead to the ambiguity and
- the service containing the need.

This section gives a step-by-step tutorial for adding manual selection support for two application components[1]: a User Interface Controller for the ARCHIE modeling application and one for the HMD calibration. Since the system does not know which application should be started the selection must be delegated to the user. We use the `UserProxy` service which is connected to the UICs via a *application* need to start all services in the chain. The relevant parts of the XML descriptions can be found in the figures 5.3 and 5.2 at the end of this chapter.

5.2.1 The Delegated Service

The main modifications have to be made in the service description of the service which contains the need for the ambiguous abilities.

Delegated Needs

The need for which there are more satisfiable abilities becomes the *delegated need*. The application developer must find the XML description where this need is and change it. There, the main modifications must be made by adding three new attributes. The following need description shows an example:

```
<need name="anyApplication" type="Application">
   <attribute name="selectionAttribute" value="application"/>
   <attribute name="selectionTitle" value="Please select the
                                           application"/>
   <attribute name="selectionHelp" value="applicationHelp"/>
   <connector protocol="Null"/>
</need>
```

The new attributes have the following meanings:

[1] taken from the ARCHIE demo, see chapter 8.

selectionAttribute It stores the name of the attribute which should be evaluated to distinguish the different services. In our example this would be the attribute application. Every partner service must have an application ability with exactly this attribute set to the name of the application. Only the name must be the same, the value can be arbitrary. This value will be displayed in the MenuDisplay service.

selectionTitle The value is only evaluated by the Selector for the MenuDisplay service. Every menu should have a title so that the users know what exactly they should choose from the menu. This is only for informational purpose and does not change the selection process. In our example we inform the users that they should choose an application.

selectionHelp This attribute stores the name of the attribute which contains additional information which can be displayed in the MenuDisplay when the user highlights a certain menu entry. In our example the attribute is called applicationHelp. Like for the selectionAttribute the value of the partner service is evaluated. However, all currently implemented MenuDisplay services do not support additional help strings. So the Selector simply ignores them.[2]

From the perspective of the DWARF framework this mechanism is nothing more than to delegate the service selection phase of the Service Manager to the user (see [9] for description of all different phases).

The Selector system need

The next step is introducing a need for the Selector. This is a special *system need*.

```
<need name="selector" type="Selector" delegated="anyApplication"
                  predicate="(selectionNeedName=anyApplication)">
   <connector protocol="ObjrefImporter"/>
</need>
```

As you can see this need has a very special structure (derived from the design of the Selector service and its Service Description). Only when all of these points are given the system need for a Selector service is set up correctly.

1. The type must be Selector.
2. A new keyword was introduced to indicate the delegated need for the Service Manager: delegated. It stores the name of delegated need.
3. The predicate must contain an attribute named selectionNeedName which value is also the name of the delegated need.
4. The connector protocol **must** be an ObjrefImporter. This is due to the fact that the Selector itself is an ObjrefExporter which is imported by the DWARF Service Manager.

[2]The name selectionHelp is not intuitive. It should be renamed correctly to selectionHelpAttribute.

The `RemoteSelection` ability

This step is needed due to a implementation weakness in the current DWARF Service Manager. By design it should be possible to pass attribute values from needs to abilities in the corresponding template service to instantiate them. But this mechanism is not implemented yet: it is only possible to propagate attributes from abilities to needs between distinct services. So we have to set up a "dummy" ability only for this purpose. The `selectionNeedName` attribute of this ability must be set to the name of the delegated need.

```
<ability name="presentApplication" type="remoteSelection">
   <attribute name="selectionNeedName" value="anyApplication"/>
   <connector protocol="Null"/>
</ability>
```

When this weakness is fixed, this step is completely unnecessary because the attribute can then be propagated by the `Selector` system need.

5.2.2 Modifying the partner services

The Service Description of the partner service must be modified only at some locations. As described above, some attributes contain names of attributes which have to be set in the partner services. So new attributes must be added to the original XML description so that the `Selector` can read them.

```
<attribute name="application" value="ARCHIE"/>
<attribute name="applicationHelp"
           value="This UIC is needed for ARCHIE"/>
```

Other changes do not have to be made here, because the Service Description of the service already contains the right ability. In our example:

```
<ability name="application" type="Application">
   <connector protocol="Null"/>
</ability>
```

```
<service name="UICArchitectNet" startOnDemand="true"
         stopOnNoUse="true" startCommand="UIC.jar ArchitectNet"
         isTemplate="false">

   <attribute name="application" value="ARCHIE"/>
   <attribute name="role" value="Architect"/>

   <ability name="application" type="Application">
      <connector protocol="Null"/>
   </ability>

   ... other needs / abilities

</service>
```

Figure 5.2: The ArchieUIC service

```
<service name="UserProxy" startOnDemand="false"
         stopOnNoUse="false" startCommand="UserSelection"
         isTemplate="false">

   <need name="application" type="Application">
      <attribute name="selectionAttribute" value="application"/>
      <attribute name="selectionTitle"
                 value="Select the application"/>
      <attribute name="selectionHelp" value="applicationHelp"/>
      <connector protocol="Null"/>
   </need>

  <!-- delegate selection to the selectors -->
  <need name="selector" type="Selector" delegated="application"
        predicate="(selectionNeedName=application)">
    <connector protocol="ObjrefImporter"/>
  </need>

  <!-- this starts up the selectors -->
  <ability name="presentApplication" type="RemoteSelection">
    <attribute name="selectionNeedName" value="application"/>
    <connector protocol="Null"/>
  </ability>

</service>
```

Figure 5.3: The UserProxy service

6 Internals of the `Selector` service

Programming internals of the selection core service

This chapter describes the internals of the implementation of the `Selector` service. It gives an overview of the system design and the interfaces between the user and the DWARF Service Manager.

6.1 Desing of the `Selector` components

6.1.1 The system design

The `Selector` is the bridge between the middleware and the user[1]. So the underlying design pattern is the bridge pattern (see figure 6.1). It is realized as a DWARF template service.

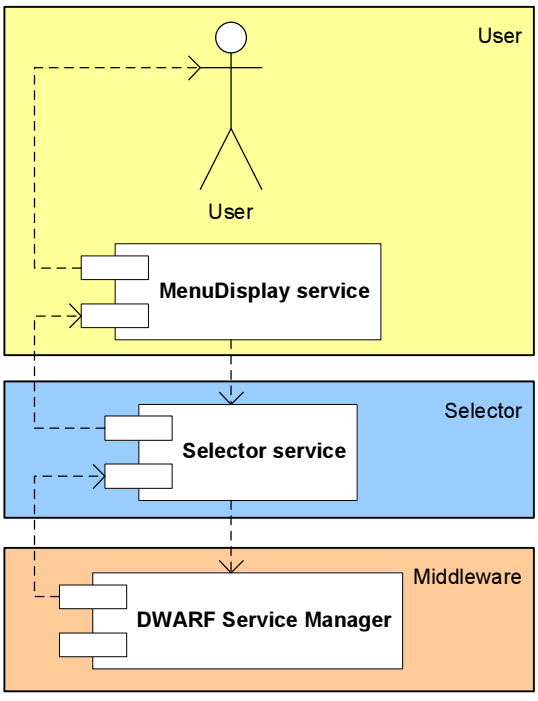


Figure 6.1: The `Selector` service: bridge between middleware and user

[1] in conjunction with the `MenuDisplay` service which is described in appendix A on page 47

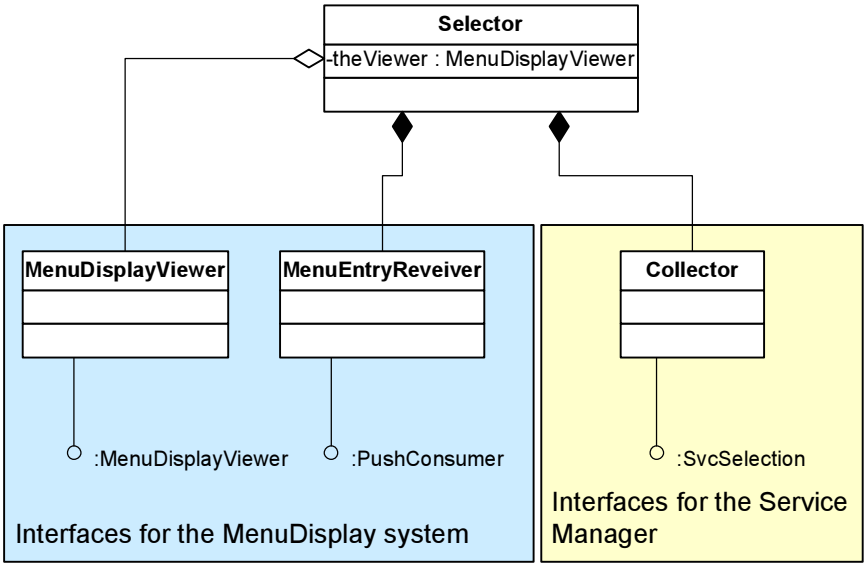


Figure 6.2: The system design of the Selector service

6.1.2 The object design

As you can see in figure 6.2 the Selector consists of three major classes:

Selector This is the main class. It implements the service interface as well as the ObjrefImporter interface which imports the MenuDisplayViewer component of a MenuDisplay service.

MenuEntryReceiver This class receives *selection* events from the MenuDisplayViewer and transfroms them into items which can be compared with the service list in the Collector class.

Collector This class is the interface to the DWARF Service Manager. It implements a certain middleware IDL interface: the DWARF *ServiceSelection* interface. This object is imported by the Service Manager and handles from now on the selection phase[2] of the delegated need in the original service. It gets a list of all possible partners, transforms them into menu entries for the MenuDisplay and evaluates the results of the returned item of the MenuEntryReceiver.

6.2 The Collector class

The Collector is the interface to the DWARF Service Manager. It implements the DWARF *ServiceSelection* interface and is imported by the Service Manager to handle the selection for the original service. It converts the service list given by the Service Manager to a list of valid menu entries which can then delivered to a MenuDisplay service. A summary of all methods is shown in table 6.1.

[2] See [9] for all connection phases.

Name	**`foundPartners`**
Return value	void
Parameters	service:ActiveServiceDescription needabilityname:String partners:PartnerSequence
Description	Is called by the Service Manager to provice the own Service Description and a list (sequence) of possible partners.

Name	**`setPartner`**
Return value	void
Parameters	partner:String
Description	Is called by the `Selector` with the menu entry which should be selected. The `Collector` searches for the associated partner string and calls `selectPartner` in the Service Manager.

Name	**`constructMenuList`**
Return value	MenuList
Parameters	none
Description	Constructs a menu entry list from the partner sequence for the `MenuDisplay` service.

Name	**`getTitle`**
Return value	String
Parameters	none
Description	Returns the title string for the `MenuDisplay` service.

Table 6.1: Important methods of the `Collector` class

6.3 The Service Description

The XML description can be found in figure 6.3.

```
<service name="Selector" isTemplate="true" startOnDemand="true"
                                           stopOnNoUse="true">

   <ability name="selector" type="Selector">
      <attribute name="selectionNeedName" value="*"/>
      <connector protocol="ObjrefExporter"/>
   </ability>

   <need name="remoteSelection" type="RemoteSelection">
      <attribute name="selectionNeedName" value="*"/>
      <connector protocol="Null"/>
   </ability>

   <ability name="provideMenuList" type="MenuList">
      <attribute name="selectionNeedName" value="*"/>
      <connector protocol="ObjrefImporter"/>
   </ability>

   <need name="selection" type="MenuSelection">
      <attribute name="selectionNeedName" value="*"/>
      <connector protocol="PushConsumer"/>
   </ability>

</service>
```

Figure 6.3: The `Selector` XML Service Description

7 A security model for DWARF

Personalisation needs a security model for DWARF: first ideas

This chapter introduces a security model for DWARF. The goal was to enable reliable ad-hoc networking as well as to ensure flexibility when using DWARF for developing new AR applications which relies on the identity of the user.

Security was the original topic of this work. However by searching for solution of this issue, the problem of the personalized configuration and selection has to be solved first. So this chapter is only a proposal and should be evaluated.

7.1 Distributed Authentication

7.1.1 The Kerberos Protocol

For DWARF we need a distributed model of authentication. There are many already existing mechanisms available such as *Kerberos*.[1] In short words, Kerberos uses tickets from a trusted *Ticket Granting Service* (TGS) to verify the identity of the *principle* (a user or program which wants to use a Kerberos service). A good introduction of Kerberos can be found in [4].

To establish a secure and authenticated connection between a user *Alice* and a service *Print*, following simplified steps are executed:

1. Alice logs into the system providing her identity (e.g. with username and password). The request is sent to the Kerberos system which generates a session key and ticket for communication with the TGS.
2. Alice requests a session with the Print service at the TGS. Therefore, the TGS creates a ticket for the Print service.
3. Alice contacts the Print service and delivers the ticket. If the ticket is accepted by the Print service, Alice can now use Print.

This section is very brief and only describes the protocol from a coarse view. But the main point is, that tickets are issued by a trusted third party which are evaluated by the single services. This model can be adapted to DWARF.

7.1.2 Authentication Protocol for DWARF

Since DWARF also consists of distributed services one can modify the Kerberos protocol so that is can be integrated into DWARF. Every user has its own `UserProxy` service which can

[1] See the homepage of the MIT project (`http://web.mit.edu/kerberos/www/`) or RFC 1510 [12]

also handle a *login ability*. These ability provides *certificates* for every application which the user wants to use. These certificates are issued by a *certificate authority* in a administrative domain (e.g. a service which is under control of the administrators). If users want to use an application, they have to prove their identity by several ways:

1. By manually entering a username/password pair in a terminal in the room (location) where the application runs. A `Password` service is running on that terminal which evaluates the entries with the database of the certificate authority. This method must be used if the user has no wearable computer, and therefore no `UserProxy` service.

2. By automatically connecting to the `UserProxy` service and fetching the certificates for the available applications. If a valid one is received, the application can be used.

The differences to the Kerberos model are that the communication channels are not encrypted. It is purely used for authentication and access control. So an intruder has the possibility to modify the communication because neither encryption nor integrity is considered in the above approach. The authentication is base on public key encryption whereas Kerberos relies on symmetric key encryption. The advantage of the first mentioned mechanism is that the administrative services must only store public keys and the unique identity of the owner. Private keys stay at the users.

```
- describe the new model
```

7.2 Roles and Certificates

As already discussed, we introduced a context attribute "*role*" in the DWARF Service Descriptions. Every application defines new roles for the user. Only if a user is assigned the right role, he/she can use the application. The role assignment is also done by the administrators.

For every role in every application the administrators issue new certificates for the user which can be stored on the personal wearable computer. For every application/role pair extracted from the certificate, the `UserProxy` service generates an application need with the attributes `application` and `role` set to the stored values. The selection of the actual desired application is then handled via a `Selector` service.

For example, Alice has two certificates for the applications ARCHIE and SHEEP. The administrators assigned her the roles "God" and "Architect". The `UserProxy` service generates now two needs:

```
<need name="archie" type="application"
      predicate="(&amp;(appliction=ARCHIE)(role=Architect))">

   <attribute name="application" value="ARCHIE"/>
   <attribute name="role" value="Architect"/>
   <connector protocol="Null"/>

</need>
```

```
<need name="sheep" type="application"
      predicate="(&amp;(appliction=SHEEP)(role=God))">

   <attribute name="application" value="SHEEP"/>
   <attribute name="role" value="God"/>
   <connector protocol="Null"/>

</need>
```

Only configured applications which have the appropriate attributes set can now connect to the `UserProxy` service and are available in the `Selector`.

7.3 Data Structures for the Communication

This section gives a brief overview of possibly suitable data structure which can be used as IDL structures for communication. Figure 7.1 shows a possible structure for identifying users. It contains a unique username and the public key.

```
struct Userinfo
   name: String;
   myPublicKey: PublicKey;
};
```

Figure 7.1: The `UserInfo` structure

A certificate which is issued by the certificate authority contains the following fields:

authority This is an identifier for the authority which issued this certificate

application The name of the application

roles A sequence of roles for which the user is registered within the application

userinfo The id of the user

generated Time when this certificate was created

valid Time when this certificate expires

```
struct Certificate {
   authority: String;
   application: String;
   roles: Sequence of Strings;
   userinfo: Userinfo;
   generated: Timestamp;
   valid: Timestamp;
};
```

Figure 7.2: The `Certificate` structure

7.4 Conclusion

The model and protocol which as introduced here are not evaluated nor implemented in any scenario in ARCHIE or in SHEEP. The ideas are based on concepts which should be applicable for DWARF, however no security or performance analysis has been made. Since my focus switched to the personalization, a whole integration picture in DWARF is missing. This has to be done in future works.

8 The ARCHIE demo

Usage of the described components in the ARCHIE demo

This chapter gives on overview how the `Selector` and the `MenuDisplay` services are used in ARCHIE and what hardware was used to realize the demo.

8.1 The demo set up

As shown in the ARCHIE chapter (chapter 2 on page 7) the system consists of several independant scenarios:

1. Calibration of the devices
2. Modeling of a building
3. Navigation through the building for maintenance
4. Usability studies and evaluation of the system

The main focus of this chapter is on the steps 1 and 2, because they are using the `Selector` for choosing the application. Both the calibration and the modeling scenarios use a *User Interface Controller* (UIC) for handling user actions and controlling the views in the HMD. In brief, the UIC has needs for *user input* which are generated e.g. by the `TouchGlove-Interpreter` service. By combining user input events using a petry net the UIC generates *user action* events and changes the model in the viewer of the HMD accordingly. In case of the clibration UIC, the user action events are used to initiate the different calibration steps, in case of the modeling UIC they are used e.g. to save the current model.

8.2 The Application `Selector`

By adding an *application ability* to the UICs the `Selector` can present the different scenarios to the user. How to do this is described in chapter 5. The `Selector` presents the different applications to the user (see figure 8.1) in a `ListMenuDisplay` service. This instance of the service is called *Application Selector* since it is used to let the user choose the desired application scenario.

8.3 The overall system

The whole ARCHIE system with all services is shown in the DIVE[1] screenshot in figure 8.2.

[1] Distributed Interactive Visualisation Environment

Figure 8.1: The application `Selector` for ARCHIE

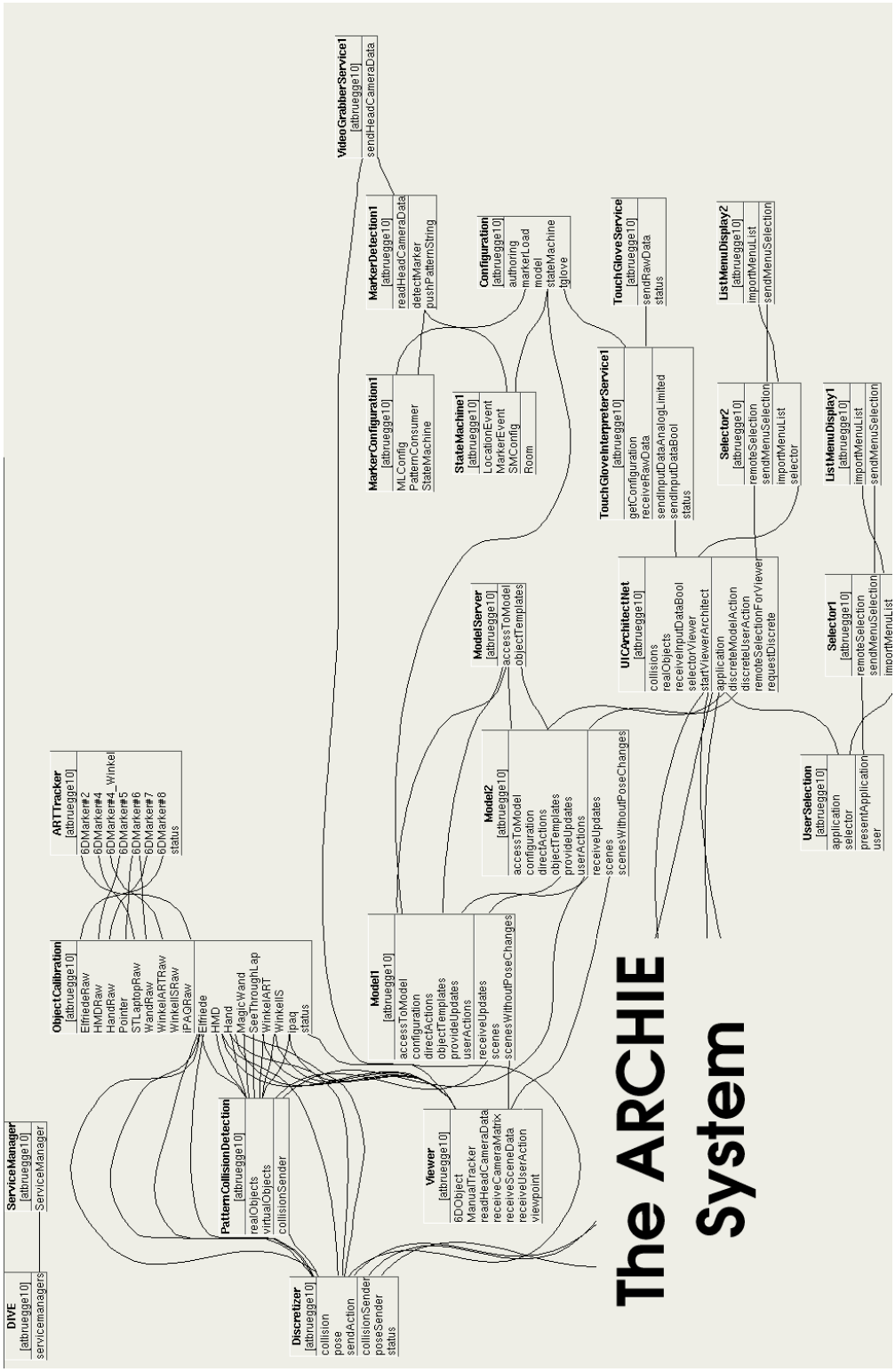


Figure 8.2: Screenshot of DIVE showing the whole ARCHIE application

9 Future work

Concepts which can be discussed

9.1 An improved `MenuDisplay` component

In modern GUI environments and frameworks, there are widgets which group several menus together and let the user navigate through them. This concept can be used for improving the selection of service chains in future DWARF applications.

In the current implementation, the user chooses the application on his PDA using a `ListMenuDisplay`. This menu pops up when the `UserProxy` service was started locally on his PDA. So the display component is also started locally as desired. After that however, all other `ListMenuDisplay` services start up on the machine where the ad-hoc chain cannot be formed. So the menus are generated at different places throughout the network which can be useful when standing straight to the hardware.

But sometimes, it would be preferable (especially when starting applications and the user knows exactly what he wants to do) when all decisions from the user are made on his PDA once. This means that instead of starting new `MenuDisplay` on every machine, all `Selector` services connect to the single `Wizard` service running on the PDA. The `Wizard` gets all possible menu entry through the generic `MenuDisplay` interface defined in the IDL (see appendix A on page 47 for the IDL source code).

Figure 9.1 on 46 shows the design concept of such a `MenuDisplay` component. For every `Selector` which connects to the `Wizard`, a new menu must be generated within the display component. The user can navigate through all menus and make his decisions. After that, he commits everything and the system forms all service chains with this information.

9.2 Realisation of the `Preference` service

For using user specific preferences, we need a light-weighted implementation of the `Configuration` interface for handheld devices. The `Configuration` service is not a suitable candidate since it needs a fully functional database as backend. A `Preferences` service for example can use a text file for storing its information. For example, one can use the SQLite library[1].

[1] `http://www.hwaci.com/sw/sqlite/`

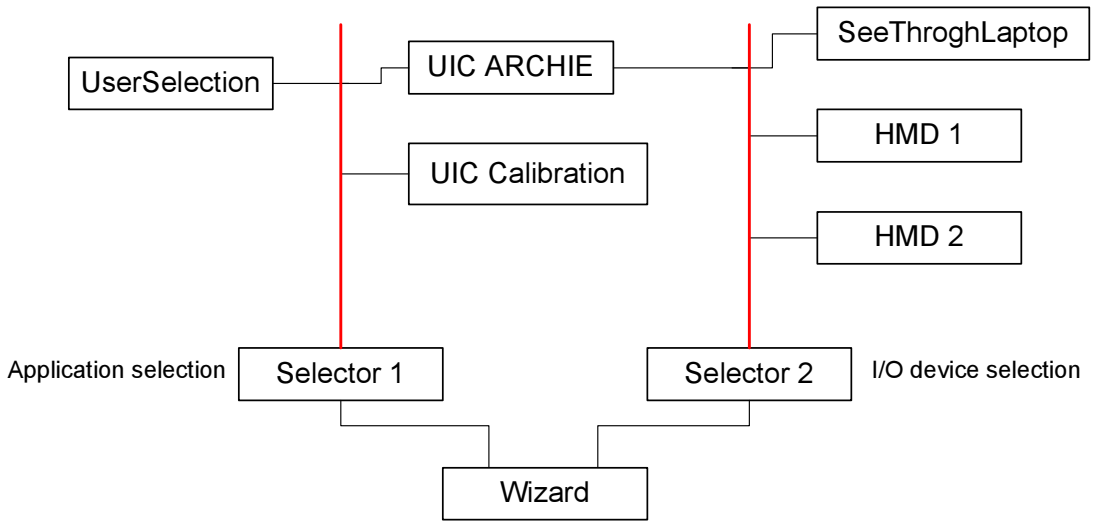


Figure 9.1: A concept for the design of a central `MenuDisplay` component which handles all menus coming from the `Selector` at once. The user can navigate through the menus and commit them when everthing is configured by him.

9.3 Implementation and evaluation of the security model

Since the security model described in chapter 7 is currently only a concept, it has to be evaluated considering today's security needs. This will cover for example:

- Use security engineering to find weaknesses and intrusion possibilties considering current DWARF implementation
- Refine the data structures and interfaces for the exchange of certificates
- Implemnent the security model and integrate it into DWARF

Since the model is only a proposal, it can and should be replaced when a working solution is found.

9.4 User profile for `TouchGlove Interpreter` services

As proposed in chapter 4 the TouchGlove should be able to be configured by combining system default settings with user specific preferences. However, this is not implemented in the current `TouchGlove Interpreter` service. One possiblity is to use the `user` attribute and two different needs for configuration to merge the different sets of settings (see section 4.6 on page 25).

A The `MenuDisplay` service

This chapter introduces the MenuDisplay component. It was developed by Chris Kulas [6] and me to provide a new 2D interaction component based on the widget system Qt [10]. It is used to select one entry of a list.

A.1 `MenuDisplay` IDL data types and interfaces

The data structures and interfaces of the MenuDisplay are specifed in the IDL file at

`src/idl/DWARF/MenuDisplay.idl`.

in the CVS source tree.

The `MenuEntry` structure

The core data structure is shown in figure A.1. Every entry in the list contains of the text which is displayed in the list. The field `entry` contains that text. The `state` field contains the current status of the entry. We defined the following semantic:

- **0:** The entry is disabled. Depending on the used widget the appearence of the entry is different: in ordinary lists the entry is not displayed at all whereas in other it may be displayed as gray entry which cannot be selected.
- **1:** This is entry is enabled. It appears as a normal entry in the list.
- **10:** After an entry is highlighted the state remains at 1. This entry gets selected when the highlight is confirmed, e.g. by pressing an `Ok`-button. Then the state changes to 10.

The last field `help` may contain additional information which is displayed when an entry is highlighted. Currently this is not implemented by any menu display component.

Data structure for one menu entry	
entry	The displayed entry text
state	The state of the entry 0: entry is disabled 1: entry is enabled 10: entry is selected
help	An optional help string which can be displayed in a supported viewer

```
struct MenuEntry {
  string entry;
  short state;
  string help;
};
```

Figure A.1: The core data structure of the MenuDisplay services

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

   <attribute name="type" value="ListMenu"/>
   <attribute name="typeHelp" value="This is a ListMenuDisplay"/>

   <ability name="sendMenuSelection" type="MenuSelection">
      <connector protocol="PushSupplier"/>
   </ability>

   <need name="importMenuList" type="MenuList">
      <connector protocol="ObjrefExporter"/>
   </need>

</service>
```

Figure A.2: XML description of the `ListMenuDisplay` service

The interfaces

If a service wants to provide a menu display component, this compoent must implement the `MenuDisplayViewer` interface. It consists of two methods:

- `void listHasChanged (in MenuList list)`
  This method receives a list of `MenuEntry` entries and displays them in the implemented GUI.

- `void setMenuTitle (in string title)`
  A menu display component can show a title which gives additional information about all entries. For example, a suitable title can be: *"Select the desired file"* from a list of filenames.

A.2 The `ListMenuDisplay` service

The menu display components are realized by DWARF services. Currently there are two different GUI versions of menu display services: The `PieMenuDisplay` and the `ListMenuDisplay` services. This work will focus on the `ListMenuDisplay`, but the concepts described here are also implemented in the `PieMenuDisplay`. Figure A.2 shows the XML description of the `ListMenuDisplay`.

Every DWARF service which is interested in a menu display component must have a *need* of type `MenuSelection`. It uses the `PushSupplier` connector to retrieve the highlighted entries. By default all highlighted entries are sent over the channel, i.e. whenever the user clicks on an entry, this entry is sent. When the user confirms this by pressing the `Ok`-button, the entry changes to state 10 and is sent again. The receiving services has only to check for the state to determine the selected entry.

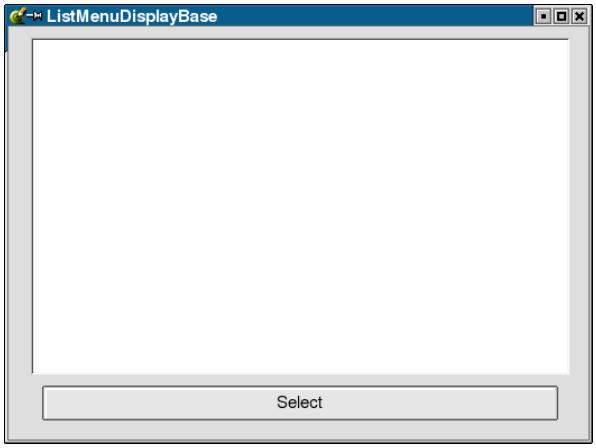


Figure A.3: Example screenshot of the GUI part of the `ListMenuDisplay` service

To be able to inform the menu display about changes in the menu list, the service must have an ability of type `MenuList`. The menu display component is is imported by this ability. After the import, the above mentioned interface methods are available. The service can update the menu display's list and change the title.

Bibliography

[1] 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 IEEE and ACM International Symposium on Augmented Reality (ISAR 2001)*, 2001. 1.4

[2] Bernd Brügge and Allen H. Dutoit. *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*. Prentice Hall, Upper Saddle River, NJ, 2000. 2.1, 2.2, 2.3, 2.3.1, 2.3.2

[3] Marco Feuerstein. Erstellen einer DWARF-basierten Darstellungskomponente für 3D Szenen auf dem iPAQ. SEP, Technische Universität München, 2002. 3.2

[4] Jeffrey I. Schiller Jennifer G. Steiner, Clifford Neuman. Kerberos: An authentication service for open network systems. *Technical Report*. 7.1.1

[5] 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/`. 1

[6] Christian Kulas. Usability engineering for ubiquitous computing. Master's thesis, Technische Universität München, 2003. A

[7] Manja Kurzak. Architecture and urban planning. Master's thesis, Universität Stuttgart, 2003. 2

[8] A. MacWilliams and T.Reicher. Decentralized coordination of distributed interdependent services. *Technical Report*, 2003. 1.5, 2.3.2

[9] Asa MacWilliams. Using ad-hoc services for mobile augmented reality systems. Master's thesis, Technische Universität München, 2001. 1.5, 5.2.1, 2

[10] Qt homepage qt trolltech. `http://www.trolltech.com/products/qt`. A

[11] Thomas Reicher, Asa MacWilliams, and Franz Strasser. A decentralized architectural style and infrastructure for mobile augmented reality. *DOA*, 2003. 1, 4.5

[12] Request for comments. `http://www.rfc.net`. 1

[13] Christian Sandor, Asa MacWilliams, Martin Wagner, Martin Bauer, and Gudrun Klinker. SHEEP: The shared environment entertainment pasture. In *IEEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2002*, 2002. 1.3, 1.5, 3.2

[14] Franz Strasser. Evaluation of CORBA implementations for small wearable computers. SEP, Technische Universität München, 2003. 1.3, 3.2

[15] Marcus Tönnis. Data management for AR applications. Master's thesis, Technische Universität München, 2003. 4.3

[16] Johannes Wöhler. Driver development for touchglove input device for dwarf based applications. Systementwicklungsprojekt, Technische Universität München, 2003. 3.1, 3.3.2