Design and Implementation of a Management Toolset for TINA-based PCS Components

Diploma Thesis / Diplomarbeit

Andreas Guther

Matrikelnummer 114481

Technische Universität Berlin
Fachbereich Informatik
Institut für Kommunikations- und Softwaretechnik
Fachgebiet Offene Kommunikationssysteme (OKS)
Franklinstraße 28-29
10587 Berlin

Aufgabensteller und Gutachter:
Prof. Dr. Radu Popescu-Zeletin und Dr. Thomas Magedanz

Der Autor erklärt an Eides statt, daß er diese Arbeit ohne unerlaubte fremde Hilfe und unter ausschließlicher Verwendung der genannten Quellen und Mittel angefertigt hat.

Berlin, den

Andreas Guther

Design and Implementation of a Management Toolset for TINA-based PCS Components

Diploma Thesis

Andreas Guther

Vorbemerkung

Die vorliegende Ausgabe stimmt mit meiner Diplomarbeit, wie ich diese am 2. Mai 1997 eingereicht habe, in Inhalt und Aufbau überein. Wo mir nachträglich Rechtschreibefehler und ähnliches aufgefallen sind bzw. ich auf diese hingewiesen wurde, habe ich eine Korrektur vorgenommen, ohne dies explizit anzuzeigen.

Contents at a Glance

Zusammenfassung

Die Telekommunikation ist einer der zur Zeit am meisten expandierenden Bereiche der internationalen Wirtschaft. Bereiche neuerer Technik wie Audio und Video wachsen mit herkömmlichen wie Fernsehen und Kommunikation über das Telefon immer stärker zusammen und können generell unter dem Schlagwort Multimedia eingeordnet werden. Mit dieser Verschmelzung einhergehend wächst die Anforderung an telekommunikationsunterstützende Software immens. Der Forderung nach kürzeren Entwicklungs- und Einführungszyklen neuer Telekommunikationsdienste stehen veraltete Softwaresysteme gegenüber, die diesen Ansprüchen selten wirklich gerecht werden können.

Seit Anfang der 90er Jahre sind Bestrebungen im Gange, Lösungen zu entwikkeln, die es ermöglichen, Telekommunikationsdienste jeglicher Art schneller und unkompliziert einsetzen zu können. Als Beispiele seien hier Intelligente Netze (*Intelligent Network*, kurz IN), Telekommunikations Management Netze (*Telecommunication Management Network*, kurz TMN) oder auch *Advanced Intelligent Network* (AIN) genannt.

Seit 1992 besteht das *Telecommunications Information Networking Architecture Consortium* (TINA-C), welches ein Zusammenschluß der wichtigsten Netzwerk Betreiber und Computer Hersteller ist und bis Ende 1997 projektiert wurde. Erklärtes Ziel des TINA-C ist es, basierend sowohl auf bestehenden Lösungen wie z.B. IN und AIN als auch auf neuen Konzepten eine Telekommunikations-Architektur zu entwerfen, die für beliebige Netze wie PSTN, ISDN, B-ISDN etc. benutzt werden kann. Dies gilt gleichermaßen für Telekommunikationsanwendungen als auch für das Management dieser Anwendungen und deren Ressourcen.

Zur begleitenden Auswertung und Evaluierung der TINA-C Ergebnisse wurde 1996 das TINA-Evaluierungsprojekt mit den Bereichen **Evaluierung von DPEs**, **Definition einer TINA-konformen Vorgehensweise** und dem **TINA-C Auxiliary Project Personal Communications Support in TINA (PCS in TINA)** ins Leben gerufen. Das Projekt wird im Zusammenschluß der Technischen Universität Berlin und der GMD Fokus unter der Schirmherrschaft der Deutschen Telekom AG durchgeführt und endet Mitte 1997. Das *TINA-C Auxiliary Projekt PCS in TINA* führt Konzepte zur Unterstützung der Mobilität und der verbesserten Erreichbarkeit von Benutzern moderner Telekommunikationsdienste in die TINA Architektur ein. Eigens dazu wurden bestimmte Komponenten der TINA Architektur zum Teil erweitert als auch neue Komponenten eingeführt. Im Projektgesamtzusammenhang wurde ebenfalls an der GMD die *TANGRAM DPE* Plattform entwickelt, die eine Umsetzung der TINA Architektur auf der Ebene der verteilten Anwendungen unter Verwendung der noch jungen Technologie der *Common Object Request Broker Architecture* (CORBA) ist. In diese Plattform wurden die *PCS in TINA* Komponenten erfolgreich integriert.

Aufgabe der vorliegenden Arbeit ist es, einen Teil der *PCS in TINA* Komponenten anhand benutzerfreundlichen Anwendungen zu verwalten. Dies besonders unter Berücksichtigung von Erweiterbarkeit und Plattformunabhängigkeit. Bei der Realisierung dieser Ziele kommt zum einen die Benutzung der neuen, plattformunabhängige Entwicklungssprache *Java* zum Tragen, das Anwenden von CORBA Funktionalitäten als auch der Einsatz der noch sehr jungen Software Entwicklungs-Handbücher, der *Design Patterns*.

Abstract

Telecommunications is one of the fastest expanding areas in today's international economy. The newest hightech developments in audio and video are converging more and more with the traditional television and communication systems connected by the telephone, to form a new arena for advancements in these areas termed "multimedia". As a by-product of this melting of medias, the demand for telecommunications supportive software is growing rapidly. Outdated software systems are seldom able to meet the need for the ever shorter development and introduction cycles of new telecommunications services

Since the early 90's, great effort has been put into developing ways of speeding up and simplifying telecommunication services of every kind. Some of the resulting service networks are for example: the Intelligent Network with the abbreviation IN, the Telecommunication Management Network or TMN, and the Advanced Intelligent Network or AIN just to name a few.

In 1992 a development project called the Telecommunications Information Networking Architecture Consortium or TINA-C was brought to life and has since successfully joined the most important network carriers with the computer industry. The project is projected to continue running through 1997. The declared goal of TINA-C is, using solutions already provided by existing networks like IN and AIN as a base, as well as developing new concepts, to design a telecommunications architecture that could be used for every kind of network, for example the PSTN, ISDN, and B-ISDN nets. This goes equally for telecommunications applications and the management of these applications and of their resources.

A companion project to the TINA-C project, the **TINA-C Evaluation project** was formed in 1996. It's aims are to support and evaluate TINA-C's results in the areas of: **Evaluation of DPEs**, **Definition of a TINA-conform Procedure** and the **TINA-C Auxiliary Project Personal Communications Support in TINA (PCS in TINA)**. The project, which is in partnership with the Technical University of Berlin and the GMD Fokus under the umbrella of the German Telekom AG, will continue until the middle of 1997. The *TINA-C Auxiliary Project PCS in TINA* developed concepts to support mobility and to improve accessibility for users of modern telecommunications services which are with in the TINA architecture. To the same purpose, a certain number of completely new components as well as the partial extension of existing ones, have been introduced. For the overall continuity of the project, the TANGRAM DPE platform was developed—also by the GMD. The TANGRAM DPE platform shifted the TINA architecture to the level of distributed applications using the still recent Technology of the Common Object Request Broker Architecture (CORBA). With in this platform, the PCS in TINA components were successfully integrated.

The task I am setting out to accomplish with my program, and with the toolset it is based on, is to assure that a part of the PCS in TINA components can be administered with user friendly applications and to give special regard to extensibility and platform independence. To realize this goal, one is led to the use of Java, the new platform independent programing language, to the application of CORBA functions and to the quite recent software development handbooks—Design Patterns.

X

Acknowledgments

This diploma thesis was made possible by the Department of Open Communication Systems at the Technical University and the German National Research Center for Information Technology GMD FOKUS. It is an independent work under the guidance of Professor Dr. Dr. h.c. Popescu-Zeletin.

First, I would like to thank my wife Hally, who guided me in finding the right english sentences and expression and becoming more clear and consistent in expressing my work.

I would like to express my thanks to those people who offered me the possibility to do research in the topics of this diploma thesis and / or guided me during all phases of this work: Professor Dr. Dr. h.c. Radu Popescu-Zeletin, Dr. Thomas Magedanz, Dipl.-Inform. Tim Eckardt and specially Dipl. Inform. Ute Scholz.

I would also like to thank all my colleagues at GMD FOKUS for their various kinds of support. Especially the members of the PCS in TINA project, the members of the departments OKS of the Technical University of Berlin, and the members of the department ICE of GMD FOKUS: Dipl.-Inform. Stefan Arbanowski, Dirk Ahrens, Metin Çetinkaya, Andreas Dannert, Thomas Gringel, Dipl.-Inform. Lars Hagen, Dipl.-Inform. Ali Hafezi, Dipl.-Inform. Stephan Hübener, Dipl.-Inform. Patrick Kielhöfer, Dipl.-Inform. Sven Krause, Thomas Masjosthusmann, Dipl.-Ing. Tom Pfeifer, Dipl.-Inform. Martin Vetter, Dipl.-Inform. Sven van der Meer, Dipl.-Inform. Hao Wang, Dipl.-Inform. Sven Winter and all those I forgot to mention.

Andreas Guther

Berlin, April 1997

Table of Contents

Part 2 **Requirements and Design**

Part 3 **Implementation**

Part 4 **Views–The Graphical User Interface**

Part 5 **Conclusion**

Part 6 **Appendix**

List of Figures

List of Tables

1 Introduction

The *TINA-C Auxiliary Project Personal Communications Support in TINA* (*PCS in TINA*) enhanced the TINA Access Session in terms of the 'reachability' of end-users for incoming calls. *PCS in TINA* aims to incorporate concepts of Personal Communications Support (PCS) into the TINA Service Architecture in order to provide generic personal communications-related capabilities in a uniform way to an open set of TINA-based telecommunication services[1]. Although terminal mobility is not part of the *PCS in TINA* project, the concept of PCS as defined within *PCS in TINA* addresses aspects of personal mobility and can be understood as an extension of the concepts of personal mobility as defined by Universal Personal Telecommunications (UPT). The *PCS in TINA* concepts allow the end-users to register at end-user systems (terminals) as well as at locations which were not foreseen in the original TINA Service Architecture. Furthermore, *PCS in TINA* introduces mechanisms for advanced invitation handling to define handling policies according to which invitations sent to the invitee's User Agent will be automatically processed.

1.1 Motivation

To provide the mechanisms described above, it was necessary for *PCS in TINA* to introduce new components into the TINA architecture, as well as to enhance existing ones. Some of these components, i.e. computational objects, are being created automatically by the system whenever they are needed, others have to be managed by system operators or even end-users during the lifetime of a system.

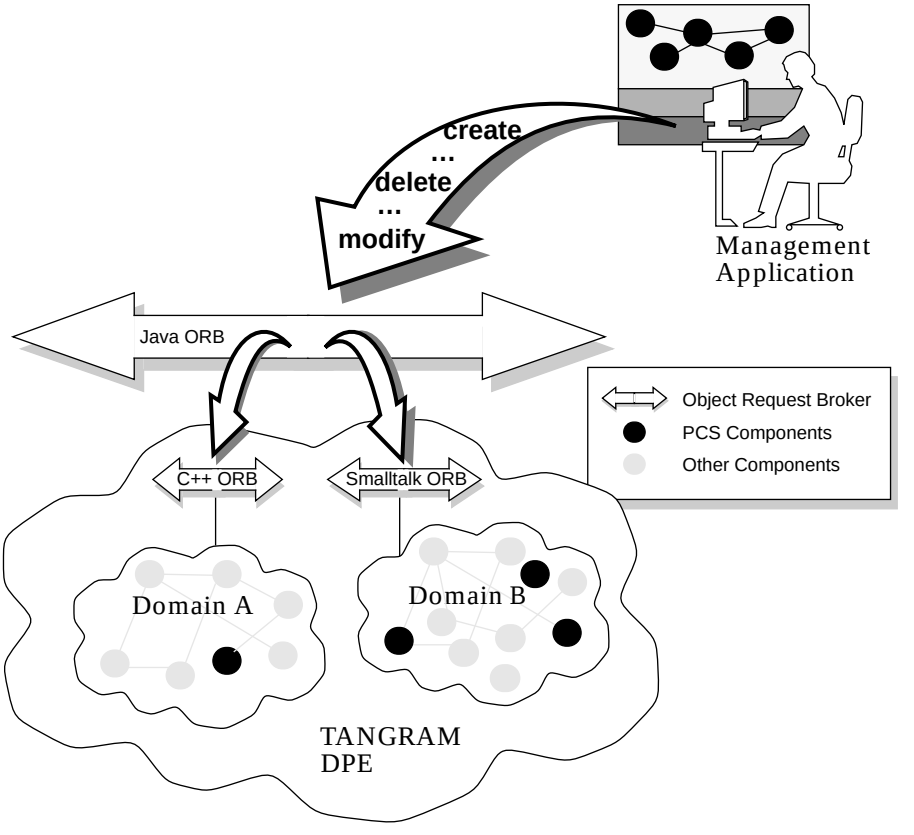
For the management of these computational objects, I have designed and implemented a generic management toolset especially for this thesis project. This toolset can be seen as a craftsman's tool box. It consists of different, 'plugable' components serving the needs of a programmer, to build a management application to administer some of the *PCS in TINA* components. The management applications which have resulted from this project, are a compound of those components. The design was influenced by different points of view concerning the technical constraints of the underlying platform and how such a management toolset should be used. The following briefly sketches those contrasting points of view.

Both system operators and end-users need support so that they can manage the components in a transparent manner. That means, the user should be, as far as possible, unaware of and unhampered by the complexity of the underlying system.

Thus, management applications with graphical user interfaces should provide easy management of the component which is uncoupled with the technical aspects of the underlying implementation. Such management applications should, ideally, be available on the most common computer architectures, i.e. independent from an underlying operation system.

1. [Eckardt+96a], p. 13

Speaking from a technical standpoint, access to the computational objects is rather complicated. Access to the different PCS-nodes is granted through different Object Request Broker implementations. Depending on the purpose of the action, some computational objects have to be accessed in more than one way.



The TANGRAM DPE consists of different domains accessible through different ORB implementations. The management applications are implemented in Java and use a Java ORB. The management applications were constructed using a three layer architecture.

Figure 1-1. Distributed Access to PCS Components

It is desirable to have an application programming interface that hides the complexity of distributed access as much as possible, and provides a simplified but still powerful access to the different interfaces of the computational objects.

Another point of central importance for the motivational impetus behind my design, was the state of the underlying distributed processing environment; the TANGRAM DPE. The TANGRAM DPE is a DPE implementation which mirrors the actual state of the TINA-C concepts—and these concepts are still in flux. During the design phase of this thesis project in the fall of 1996, the TANGRAM DPE was based on the TINA Service Architecture 2.0 or SA 94 from 1995[2]. It was obvious then, that the Service Architecture would go through major changes before the end of the year. In december 1996, TINA-C delivered the Service Architecture

2. [Berndt+95]

4.0[3] or SA 96 which was followed one month later by the Service Architecture 4.1[4]. At the time of this writing, the TANGRAM project works on a migration of their DPE implementation from SA94 to SA96.

For these reasons, it was important to design the access to a platform that is, so to speak, 'under construction' in such a way that changes in the platform would cause as little change as possible in the implementation of the management toolset.

When researching this project, I came across a book which, by introducing me to the design pattern movement, revolutionized my way of thinking about the design of object oriented software. The book is titled, Design Patterns: Elements of Reusable Object-Oriented Software'[5]. The impact of this book and related publications on my design can be seen in the resulting implementation—it is more efficient, easier to maintain and, I would go as far as saying, more elegant than it would have been without the knowledge that the pattern catalogues have to offer. While getting familiar with design patterns, I also discovered their impact on the design of Common Object Request Broker Architecture as well as on the design of the programming language Java, which was used for the implementation of this management toolset.

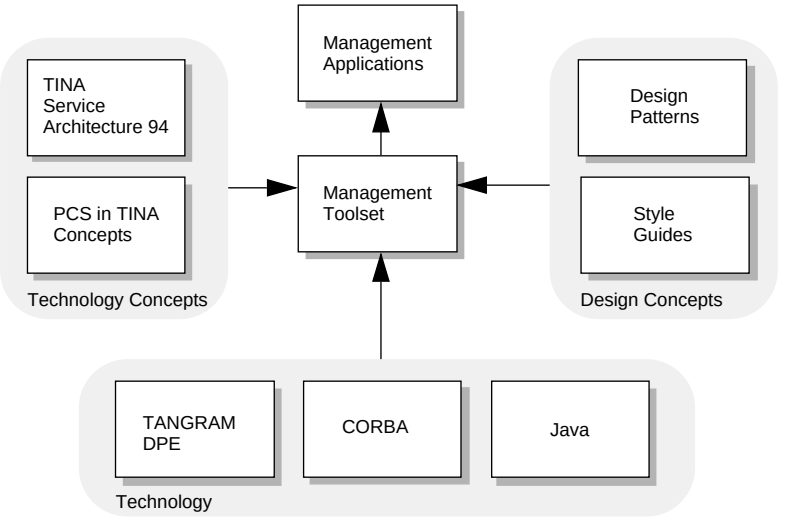


Figure 1-2. Impacts on this Work

1.2 Thesis Project Scope

This thesis project aims at designing reusable, platform independent generic components which serve to build management applications for TINA-based PCS components in any environment. The integration of those components into and the design and implementation of management applications with graphical user interfaces is part of this project.

3. [Abarca+97]
4. [Farley+97]
5. [Gamma+94]

I have designed a three layer architecture in order to implement reusable and easy to extend applications which are independent from any underlying system.

The exciting new movement in design patterns has had a major impact on this project. The programming language I have used is Java, the new 'Internet' programming language. I have demonstrated platform independence by implementing an application programmer interface that can access a CORBA based distributed environment platform.

1.3 Guide to Readers

This book consists of six parts. The first part introduces the basic concepts, principles, and rules which I followed in designing and implementing my management toolset for TINA-based components. The second part describes the design of the toolset. Part three explains the implementation of the toolset. Part four introduces the graphical user interface which allows the user to use the toolset in an uncomplicated way. Part five draws some conclusions and suggests some ideas for further extensions of the toolset. Finally, there is an appendix including a bibliography, a glossary and an index.

The chapters are grouped into parts. A short description of the contents of each chapter is given in the following sections.

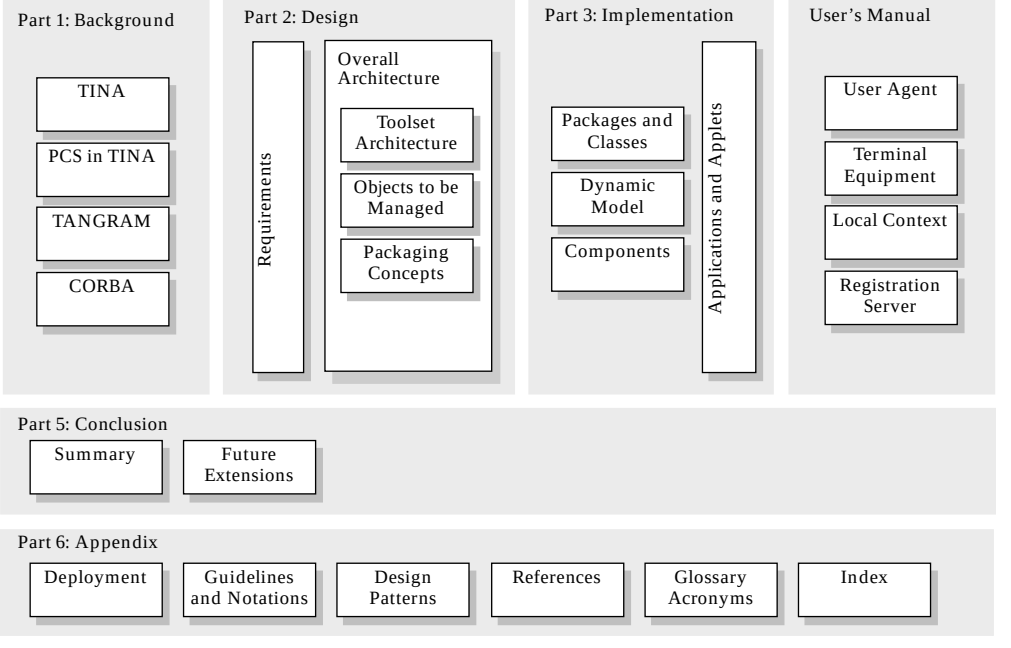


Figure 1-3. Structure of this Work

Part One (Basic Concepts, Principles and Rules) describes the background related to this thesis project. *Chapter 2: Telecommunications Information Networking Architecture* gives an introduction to the Telecommunication Architecture designed by the TINA-Consortium as far as is related to this project. *Chapter 3: Personal Communication Support* explains the personal communication support

concepts introduced into the TINA architecture by the TINA-C Auxiliary Project Personal Communication Support in TINA. *Chapter 4: The TANGRAM DPE in Relationship to the PCS* introduces the concepts of a TINA compliant DPE platform which was used to evaluate the TINA architecture. *Chapter 5: Common Object Request Broker Architecture* gives a brief introduction into the CORBA technology which is necessary for understanding this work.

Part Two (Requirements and Design) delineates the design of the management toolset. *Chapter 6: Requirement Specification* sketches what impacted my design decisions. *Chapter 7: Management Toolset Architecture* outlines the architecture that I applied to all the management toolset implementations. *Chapter 8: Objects to be Managed* presents the computational objects to be managed within the range of this project. *Chapter 9: Package Concepts and Design* describes the organization of the modules in terms of packages. *Chapter 11: Abstract Classes, Interfaces and Exceptions* specifies the abstract classes which determine the overall behavior for all of my toolset components.

Part Three (Implementation) details the classes which are part of the toolset. *Chapter 10: Package Usage* Here I have listed the classes of each package and their mapping for my design decisions on the informational viewpoint. *Chapter 11: Abstract Classes, Interfaces and Exceptions* lists and depicts all classes that I used for the implementation of all managed objects. Chapters 13 to 16 specify the classes used for the managed objects. *Chapter 17: Utilities* describes utility classes which I designed and implemented especially for this project. *Chapter 18: Graphical User Interfaces* introduces the packages and classes which I used to implement the graphical user interfaces. *Chapter 19: Java's Applications and Applets* characterizes the different aspects which I considered when designing and implementing stand alone applications and applets—which can be used with WWW browsers.

Part Four (Views–The Graphical User Interface) can be seen as a manual, which describes the usage of the graphical user interface within the management toolset. It consists of *Chapter 20: User Data Management*, *Chapter 21: Terminal Equipment Management*, *Chapter 22: Location and Location Context Management* and *Chapter 23: Registration Management*.

Part Five (Conclusion) The conclusion gives a summary of this thesis project, and discusses proposals for further extensions.

Part Six (Appendix) consists of miscellaneous topics: *Chapter 26: Deployment* gives useful hints on how to install and use the toolset applications, focusing on the end-user side. On the other hand, *Chapter 27: Programmers Guide*, focuses on the installation and usage of the management toolset from the programmer's point of view. In *Chapter 28: Style Guide*, the guidelines I used for structuring and documenting, for naming conventions and recommendations, and for implementation are given. *Chapter 29: Notations* explains very briefly the used notation. *Chapter 30: Catalog of Applied Design Patterns* introduces the design patterns used in this project. *Chapter 31: A Cookbook for Portable Clients—A Pattern System* compiles the design decisions introduced in this thesis project into a 'programmer's cookbook' for implementing applications for distributed processing environments.

1.4 Map Through this Book

Figure 1-4 presents the map that accompanies this book. Each column represents one part of the book, each box represents a chapter. At the beginning of each chapter, the according column is displayed to indicate the location of the actual chapter. The box representing the actual chapter is set in black typeface, the other boxes, representing the remaining chapters, are set in grey typeface.

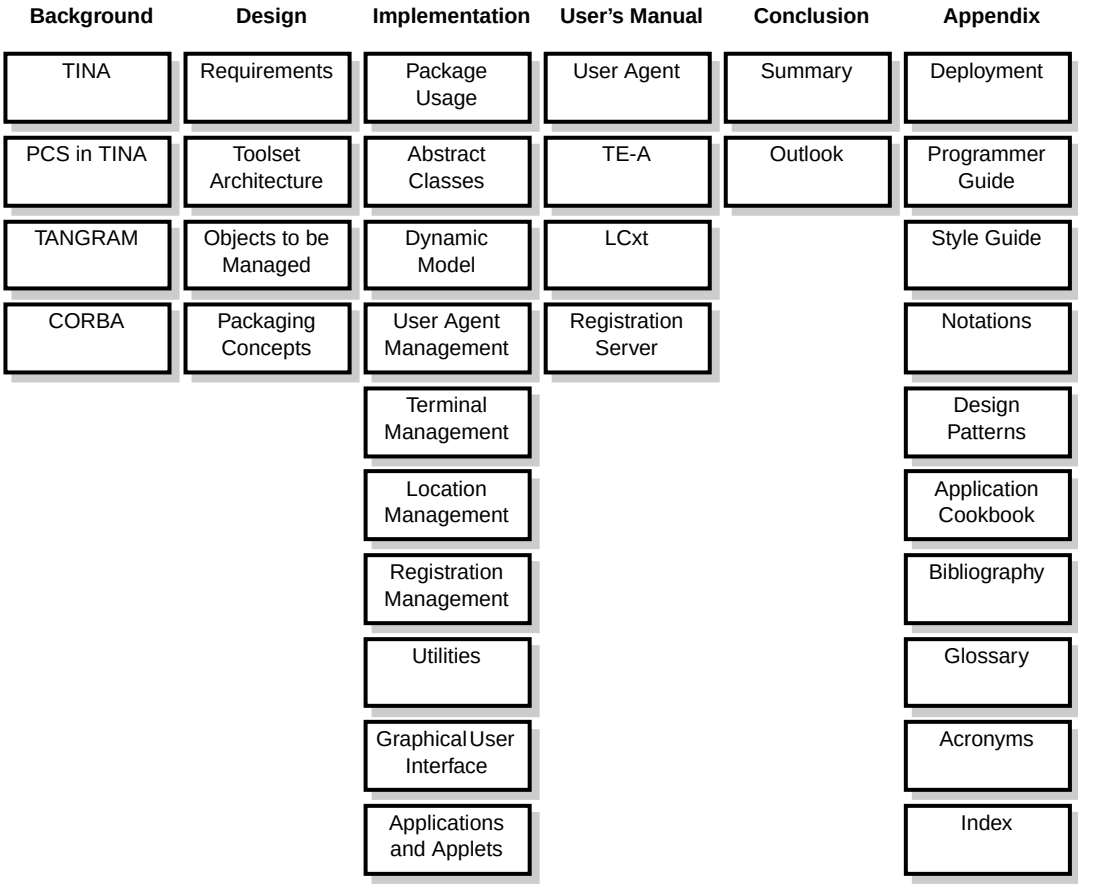
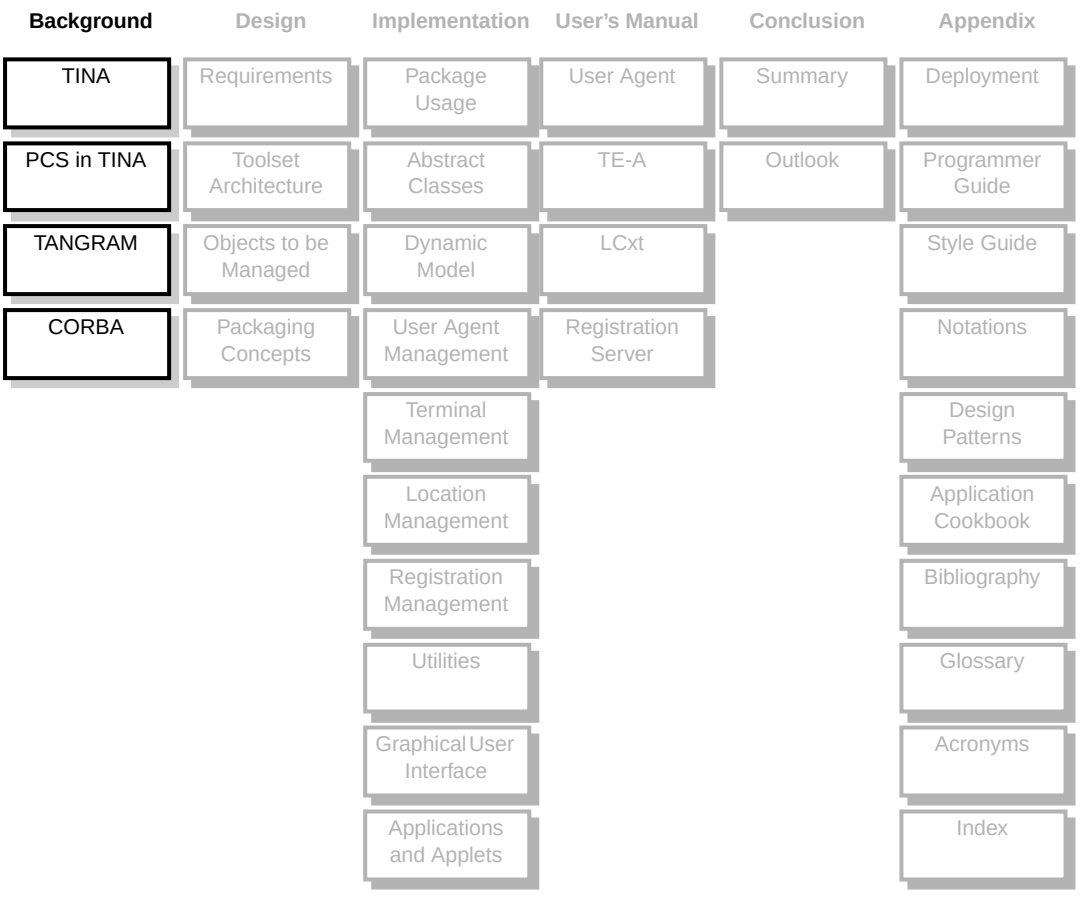


Figure 1-4. Map Through this Book

Part 1

Basic Concepts, Principles and Rules

This Chapter describes the concepts, principles and rules which guided this thesis project. It begins with background for the Telecommunications Information Networking Architecture which was introduced by the TINA Consortium. This is followed by an introduction to the concepts of Personal Communication Support as developed by the TINC-C Auxiliary Project Personal Communication Support in TINA. The TANGRAM DPE, a distributed processing environment, developed for evaluation of the TINA-C Service Architecture, which is used in the context of the *PCS in TINA* project, is presented in the following chapter. Part one ends with an overview of the applied concepts of the Common Object Request Broker Architecture technology.


Background
Design
Implementation
User's Manual
Conclusion
Appendix
TINA
PCS in TINA
TANGRAM
CORBA
Requirements
Toolset Architecture
Objects to be Managed
Packaging Concepts
Package Usage
Abstract Classes
Dynamic Model
User Agent Management
Terminal Management
Location Management
Registration Management
Utilities
Graphical User Interface
Applications and Applets
User Agent
TE-A
LCxt
Registration Server
Summary
Outlook
Deployment
Programmer Guide
Style Guide
Notations
Design Patterns
Application Cookbook
Bibliography
Glossary
Acronyms
Index

2 Telecommunications Information Networking Architecture

One major goal of the TINA-C initiative is the application of well-known concepts within the computer science field to both future telecommunication services and to traditional telecommunication services. Both types of applications are to the purposes of supporting heterogeneous environments and hiding the complexity of distributed computing. These concepts include *software engineering, object-oriented methodologies, distributed computing,* and *network management*.

The first set of TINA-C documents, also called TINA Baseline Documents, were finalized in 1993, a second set became available in the spring of 1995, followed by several Stream Deliverable Documents which are also referred to as Technical Proposals or TINA-C Reports.

This chapter gives an overview of those concepts and principles of TINA-C that are used in the scope of this project and the *PCS in TINA* project it is based on. It starts with an introduction into the TINA layered architecture, followed by a description of TINA session concepts.

2.1 TINA Layered Architecture

TINA defines a telecommunications supportive software architecture which is divided into four basic layers:

- Telecommunication Applications Layer,
- Distributed Processing Environment (DPE),
- Native Computing and Communications Environment (NCCE),
- Hardware Resource Layer.

The layering supports the need for an abstract view of resources and guarantees their independence from the network infrastructure[1]. A short description of the layers is given in the following sections.

2.1.1 TINA Applications Layer

The TINA *Application Layer* is the topmost layer. It contains a set of interacting objects to build telecommunication services.

2.1.2 Distributed Processing Environment Layer

The *Distributed Processing Environment Layer* or the DPE layer lies under the TINA *Application Laye*r. It provides a technology independent view of computing resources, which minimizes the technology-dependent aspects in application

1. [Handegård+96]

software. The DPE allows for easier designing of applications, software re-use and software portability. All of the above described capabilities help to hide the complexity of a distributed environment and provide support for object location and remote interaction.

The DPE is the layer where the computational objects are located.

2.1.3 Native Computing and Communications Environment

The Native Computing and Communication Environment (NCCE) is a software layer, which lies under the DPE. It contains software-like operation systems, communication and other supporting software.

2.1.4 Hardware Resource Layer

The hardware resource layer consists of hardware resources such as processors, memory, and communication devices.

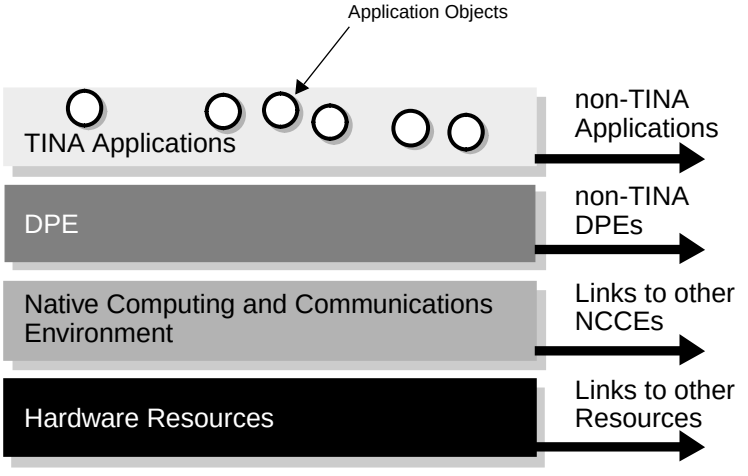


Figure 2-1. Basic Structure of Telecommunications Software in a TINA Environment

2.2 TINA Session Concept

A TINA session represents the purpose of a service which is achieved by performing a collection of activities during a temporal period. The following types of sessions, which have been defined by TINA-C from the informational viewpoint, will be explained below.

- Access Session
- Service Session
- Communication Session

2.2.1 Access Session

The Access Session is the entrance to TINA telecommunication services for users. It supports a user while accessing, requesting and retrieving any available service. The Access Session supports services which are independent and exclusively dedicated to the user. It provides mechanisms to control the life-cycle of Service Sessions, i.e. Service Session creation, suspension, resumption, and deletion. In addition, the Access Session provides the personalization of services by adapting the appearance of a service to the individual preferences of its user. The Access Session aids various mobility aspects such as: personal mobility, terminal mobility, and session mobility. The preceding capabilities made the Access Session of special interest to the *PCS in TINA project*. In fact it is solely the Access Session which the *PCS in TINA* project enhances.

2.2.2 Service Session

A Service Session represents the actual usage of a service. It provides an environment to support the execution of a service for a user or a group of users. Service sessions may consist of:

- *User (Service) Sessions* and
- a *Provider (Service) Session*.

User (Service) Session

A User (Service) Session represents the local view of a user as a participant of a Service Session. It represents service customization and maintains user specific resources. A User Session is created when a user joins a Service Session and is deleted when he leaves it. User Service Sessions reflect the settings and constraints given by the user or his communication end-point, e.g. terminal limitations.

Provider Service Session

A Provider Service Session expresses the global view on the service usage from a provider's perspective maintaining resources used by all participants. Both user and Provider Service Sessions can make use of several Communication Sessions.

2.2.3 Communication Session

A Communication Session provides an abstract view of connection related resources and supports the activities needed to establish the communication between users. A Communication Session is service independent.

While Communication Sessions and Access Sessions are service independent, Service Sessions representing service execution instances are service dependent although parts of the Service Session can be modeled in a service independent way. As depicted in Figure 2-2, an Access Session is able to maintain multiple Service Sessions, and in turn each Service Session may use multiple Communication Sessions.

2.3 Separation Aspects

One purpose of this session concept is to achieve a separation of concerns, another is to promote the distribution of processing. The separation of Access Sessions and Service Sessions allows both the access methods and the technology used by different users to vary. It also supports session mobility by, for example, allowing the accessing users to change location while a service is in progress. The separation of Service Sessions and Communication Sessions supports the division of the service activity from the set of connections currently associated.

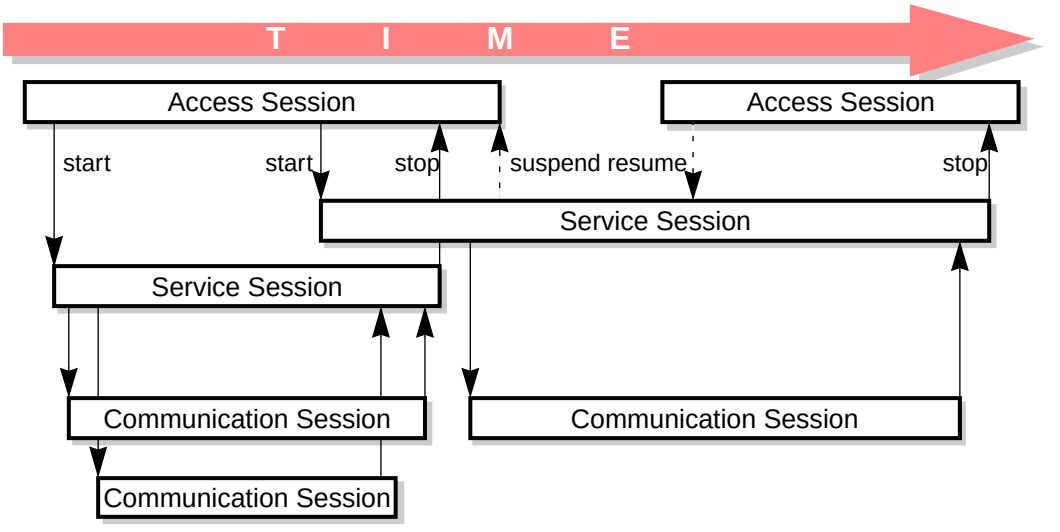


Figure 2-2. Support of Multiple Communication Sessions in TINA

2.4 Processing Environment for Distributed Objects

It must be possible for large software systems to interwork. This is a difficult task, because one is then faced with many applications, developed by different designers, distributed over long distances at various locations, which run on different hardware systems and networks. One solution for the handling of such large systems is the usage of object-oriented technology. A distributed object-oriented application is a set of interacting objects which are located at different hosts and communicate via networks. In order to have a uniform approach, some mechanisms to overcome the problems of heterogeneity, interoperability and transparency are needed. Some examples are: a standardized communications protocol, an interface description language, procedures for setting up and closing connections, distribution mechanisms, and exception handling are therefore needed.

The goal of the Object Management Group (OMG) is to develop such mechanisms. The OMG was founded to develop an open architecture for distributed applications in heterogeneous environments using object-oriented software tech-

nology. The Object Management Architecture (OMA)[2] specifies a reference model for distributed applications called the Common Request Broker Architecture (CORBA). It includes the Object Request Broker[3] , the CORBAservices[4] and the CORBAfacilities[5] . These items will be explained in Chapter 5.

2.5 Summary

The current implementation of the PCS-enhanced Access Session is based on the TINA Service Architecture 2.0 which is also called Service Architecture 94 or SA 94.

In order to define more precisely the computing structure TINA deals with, it distinguishes between several layers also called the basic structure of telecommunications software in a TINA environment. The layers for telecommunications applications and for Distributed Processing Environment are those relevant to this project.

TINA distinguishes in its session concepts between different session types. Among those sessions, the Access Session is the only one in which the *PCS in TINA* concepts can be applied.

TINA Sessions are realized by computational objects which provide usage and management interfaces; management interfaces being the one pertinent to the management toolset in this project.

2. [OMG:Mgmt95]
3. [Orfali+96]
4. [OMG:Services95]
5. [OMG:Facilities95]

3 Personal Communication Support

The main targets of the *PCS in TINA* project are, the definition of a PCS-enhanced Access Session by identification, specification and implementation of the new PCS-related computational objects (COs) and the enhancement of already defined COs related to the 'basic' TINA Access Session according to the PCS concept.

This chapter gives an overview of the PCS concepts and thus briefly clarifies which enhancements of the 'basic' Access Session have been necessary to realize the PCS concept. The Information Model of a PCS-extended Access Session is depicted at the end of the chapter.

3.1 Overview

The targeted functionalities that the *PCS in TINA* project aims to realize are: personal mobility support, the personalized control of reachability and user profile management as described in the following sections.

For a better understanding of the various mobility aspects defined by TINA, and supported by the TINA Service Architecture, I have given brief descriptive explanations of the concepts, followed by a look at how the PCS concepts were realized.

3.1.1 Personal Mobility Support

TINA defines the following types of mobility[1]:

- *Personal/Service/User Mobility* enables a user to utilize a service ubiquitously—independent of both the user's physical location and specific equipment. Personal Mobility enables the invited user to be directly addressed using the invited party's user ID instead of addressing a terminal or other equipment assumed to be with, or near to the user.
- *Terminal Mobility* enables a terminal to be identified (by a unique terminal identifier) and used independently of the point of attachment to the network and its current location. The capabilities for locating, identification and validation of terminals must be provided by the network.
- *Session Mobility* enables a Service Session to virtually 'follow' a participant, independently of the user location, the terminal or of the access point to the network. This implies that the session can be suspended and later on resumed at a different terminal independent of changes in the equipment used to support it.

The TINA Service Architecture supports various mobility aspects in the following manner:

1. [Berndt+95]

- The support of *personal mobility* is realized within TINA by the application of a unique user ID for the purpose of addressing an invited party. A proper terminal or CPE, will be selected among those devices registered with by the user. The TINA Service Architecture defines a *Usage Context* computational object (UCxt) designed to contain registration information, that is, terminal IDs and Network Access Points which are related to the user.
- The support of *terminal mobility* is handled by the separation between Terminal Equipment Agent (TE-A) and Network Access Point (NAP).
- The support of *session mobility* is modelled as a part of the TINA session concept including the strong separation between global and party (user) related resources.

TINA considers the relationship between users, terminals and services to be highly dynamic. The provision of services to an end user depends on the type of terminal being used, the network access point being accessed and the set of services being subscribed. To support mobility, it is necessary to assign unique identifiers to user and terminal agents, Network Access Points and Service Session Manager COs.

The TINA overall session concept distinguishes between service and user dependent session management, and maintenance. In order to support the provision of session mobility, a TINA-compliant system keeps the user relevant part of a session (i.e., the User Session) in the *User Session Management* CO (USM) after a session suspension. For a session resumption, the user session must be adaptable to a different end user system.

The *PCS in TINA* Auxiliary Project concentrates on personal mobility support based on user registration at terminals (considered to be a basic capability of the TINA Service Architecture) as well as on advanced support for personal mobility based on user registration at locations.

Registration at Terminals

Here the term *registration* has two different but related meanings: (1.) it denotes the association between a user and a terminal and (2.) it denotes the process of establishing that association. The association between a user (or more precisely, a user ID) and a terminal (i.e., Terminal ID / Network Access Point) has to be maintained by the system.

It should be noted that the TINA Service Architecture 2.0 was vague about whether registration at terminals would be applied during the Access Session when a Customer Premises Equipment (CPE) was to be selected for the invited user or whether it would be applied by the Service Session. In contrast to the definition of the TINA Service Architecture 4.0, where the user registration is considered to be part of the Service Session, *PCS in TINA* states, that user registration should be part of the Access Session. This is based on the fact that the process of identification and authentication will be performed very often. But, in the case of a user *only* wanting to register without using a service, no resources are required and therefore accounting is not absolutely necessary.

In the case of an invitation, the invited user will be alerted at a CPE / terminal selected with the help of the terminal registration information. In order to support user or *personal mobility*, the Service Architecture defines the Usage Context informational and computational object (UCxt), designed to contain registration information such as terminals and network access points related to the user. This object needed enhancements in order to interact with the newly introduced objects (informational and computational).

The TINA Terminal Equipment Agent also needed enlargements which allow for more flexible reactions to communication requests in the context of PCS.

Registration at Locations

Currently, TINA supports only the registration at terminals—TINA does not consider registration at locations or scheduled registration. Registration at terminals limits the set of usable terminals (e.g. CPE) to those the user manually registers at. There is no implicit, transparent registration. This issue has been addressed by the *PCS in TINA* project. The PCS-enhanced Access Session supports user registration at locations. Through user registration at locations, the system associates users with specific well known locations (e.g., rooms or zones), thereby trying to minimize the required user cooperation in order to keep the registration data up to date. With personal mobility support being based on the *PCS in TINA* concepts of registration at locations, the system is more flexible in selecting terminals suited for the requested service types. The association of users with specific terminals to be used is postponed to the moment of the arrival of invitations (i.e., incoming invitations), according to the availability of service specific terminals at the registered location. This association (i.e., selection) can also be based on user preferences.

The realization of registration at locations requires a representation of communication capabilities which are available at locations. Therefore, the informational objects *Location* and *Location capabilities* have been newly introduced into the TINA architecture, and were then realized by the computational object *Local Context*, which contains the context of a user. Each Local Context has the references to the TE-As at a specific location in the User Domain in order to allow the PCS-enhanced CO *Usage Context* to find terminals related to this specific location and therefore select an appropriate CPE.

To reflect location registration, the informational object *User Location Registration* has been also newly introduced, and then realized by the computational object *User Location*. This object holds the current location information of one user. It allows the (PCS-enhanced) UCxt to find terminals related to locations and thereby to select an appropriate CPE. This is done by querying the LCxt to this location.

An additional PCS-enhancement for registration is scheduled registration. For scheduled registration, the object *Registration Schedule*[2] (IO and CO) was introduced. It allows a user to indicate at what time he will be reachable (terminal or location). Note: the Registration Schedule allows CPE selection among registered (available) CPE which the user is registered at (terminal or location). So the Registration Schedule is considered to be the completion of the Invitation Handling Policy (realized by the CO Invitation Handling Logic, see below), which is personalized but independent of available terminals.

3.1.2 Control of Reachability

As a consequence of personal mobility and the resulting enhanced reachability, the users must be protected from an omnipotent communication environment. A personalized, automatic invitation handling (by a kind of electronic ‘secretary’) is required to control and, if necessary, to restrict reachability in order to preserve the user’s privacy. The PCS-enhanced Access Session has been designed to provide service-control related capabilities for an automatic, personalized control of

2. This object’s name was changed and was before Personal Schedule (PS).

the invited user's reachability. It is based on a user-defined policy which is described in terms of rules on how invitation requests sent to a particular user (agent,) should be handled (negative / positive communication filtering).

Using these service-control functionalities, a set of common service features can be modeled and realized in the TINA Service Architecture, such as

- Unconditional Invitation Forwarding,
- Invitation Forwarding on (user) Busy / Don't answer,
- Time Dependent Invitation Handling, and
- Originating Invitation Screening.

The *PCS in TINA* project has introduced the Invitation Handling Policy as an information object, which was realized by the CO Invitation Handling Logic. The Invitation Handling Policy represents personalized reachability control which is related to the PD_AccessSession[3] as a computational object supporting the User Agent. The TINA User Agent was enhanced in order to interact with the above mentioned newly introduced components.

3.1.3 User Profile Management

The third area of abstract target functionality is related to user profile management. In this area, the *PCS in TINA* project realizes capabilities allowing the user to manually register at a terminal or location, to view and edit the Registration Schedule, and to view and edit the Invitation Handling Logic. All of these three areas of management functionality will be briefly described in the following sections.

In the TINA Service Architecture 2.0, the User Applications (UAPs) communicate exclusively with the GSEP to access the PCS-enhanced Access Session. In the current realization stage, there are also direct relationships between UAPs and the Access Session. This may be reworked during the process of adapting to TINA Service Architecture 4.0.

GSEP

The *Generic Session End Point* (GSEP) computational object is a service independent computational object that models the capabilities needed to control Access Sessions as well as a minimal set of capabilities to control Service Sessions. The GSEP can be seen as the main client of the User Agent. It conveys every Access Session related action initiated by the user from the User Application to the User Agent. Due to its knowledge of provider domain objects, the GSEP may be seen as an agent of service providers in the user domain. On the other hand, the GSEP is the contact point for accessing components or devices in the user domain. Thus, it is capable of delivering invitations by starting or notifying user applications (UAP) in order to alert a user. According to the now available TINA Service Architecture 4.0[4], the GSEP functionality, will be supplemented by a new computational object called the Provider Agent (PA), which encapsulates the Access Control capabilities of the GSEP[5].

3. Provider Domain Access Session
4. [Farley+97]
5. which is also referred to as GSEP95 in current TINA documents

User Registration

User registration is the most important prerequisite for the provision of personal mobility. Users have to make their current location or access terminal known to the system in order to indicate to the system which terminal to select and use for alerting and communicating in the case of incoming invitation requests. The *PCS in TINA* Auxiliary Project provides a dedicated management functionality allowing the user to register at locations as well as at terminals via a specific application called *Registration Management Application*.

Invitation Handling Logic Management

The management of the user's reachability in an environment of maximized reachability is an important aspect of Personal Communications Support (PCS). The project *PCS in TINA* provides dedicated management functionality allowing the user to manage his own invitation handling. That is, the user will be enabled to specify a personal policy that prescribes how to handle invitations. The user must be enabled to define a policy which is kept in a personal rule base that allows the system to automatically control who can reach the user, in what form to reach the user (e.g. mail or interactive voice), and under what conditions. On the other hand, the management of invitation handling also comprises functionality allowing the user to store or re-direct incoming invitations if he is temporarily not reachable or not able to handle an incoming invitation.

Registration Schedule Management

Registration Schedule Management comprises functionality allowing the user to maintain and manage a network-stored schedule that serves for example as a fallback source of registration information. The schedule would indicate at what location or terminal a user plans to be reachable. In the case that no other registration information (resulting from a manual registration at a terminal or location) is available for a particular invited user, the registration schedule of that user will be approached to provide a probable (i.e., planned) location or terminal address where the user (presumably) can be reached.

3.2 Access Session Information Model

The information and semantics needed in the Access Session encompass flexible access for users to the TINA system, independent of specific terminals.

The information objects of the Access Session can be divided into objects settled in the user domain and objects settled in the provider domain (and some objects in-between). This means that the responsibility for information contained by these objects is divided.

The requirements for accessing TINA services by a user are represented by an *Access Session* providing functions to the *User* it belongs to. The Access Session is decomposed into zero or more *User Domain Access Sessions* (UD_AccessSession) which deals with local information and one or more *Provider Domain Access Sessions* (PD_AccessSession) which is mainly responsible for user admission. The object *Access Application* represents the semantics necessary in the user domain for performing a customized Access Session. The link attributes *Identification* and *User Identification* model the information necessary to authenticate the user of a function. The user needs Identification information for a local logging and User Identification information for accessing the TINA Network.

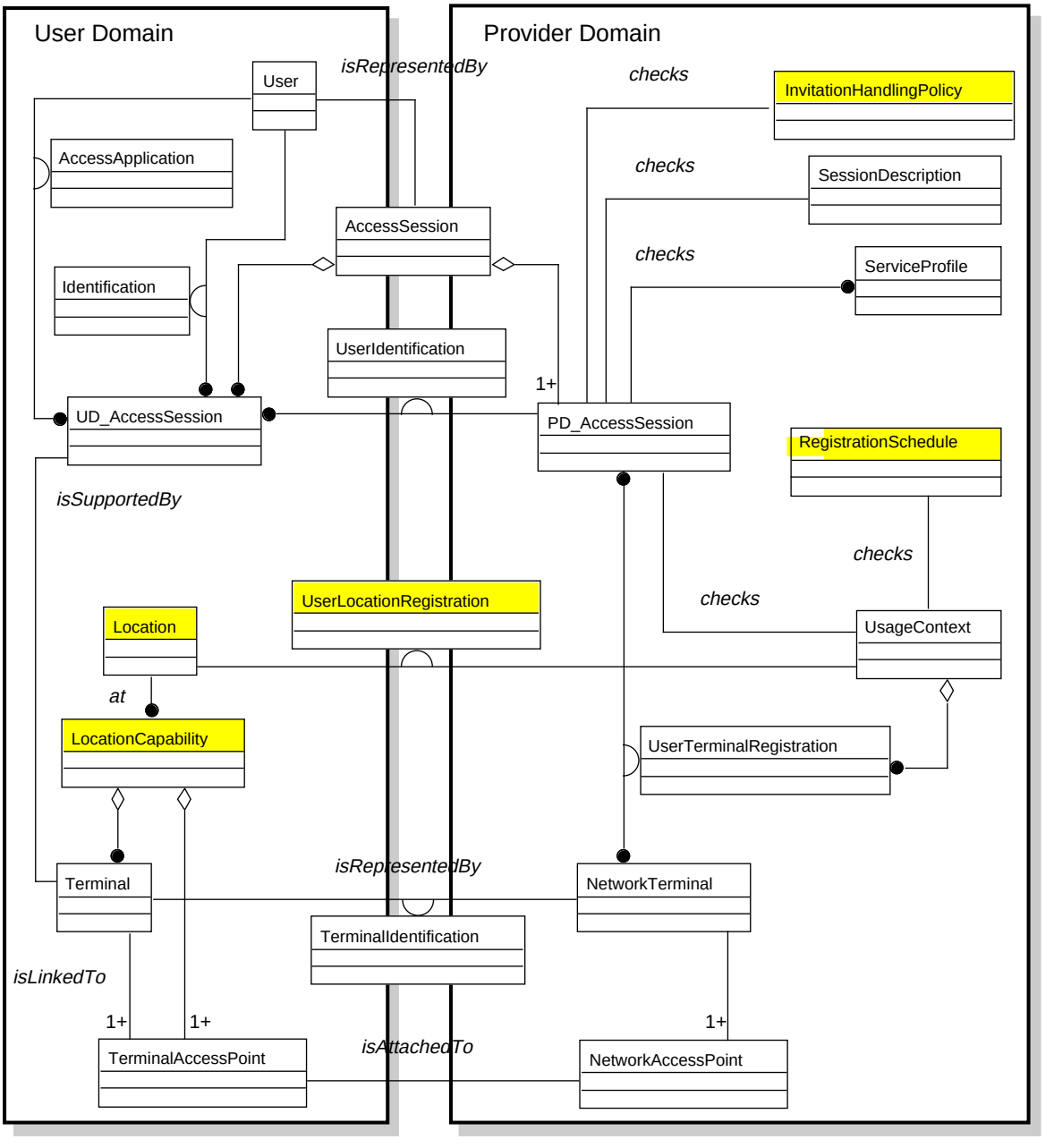


Figure 3-1. PCS–Enhanced Access Session Information Model

4 The TANGRAM DPE in Relationship to the PCS

This section presents the concepts, services and interfaces of the TANGRAM Distributed Processing Environment used within the scope of *PCS in TINA*.

4.1 Introduction

TINA-C defines TINA compliant distributed processing environments in detail in the TINA Baseline *TINA Distributed Processing Environment*[1]. In order to realize it's concepts, the *PCS in TINA* project chose the TANGRAM DPE which is a TINA-C compliant platform.

The TANGRAM DPE provides some development and runtime support for distributed telecommunication services. It supports the basic DPE object services, and is implemented using the commercially available CORBA 2.0 technology of IONA Orbix[2] and HP Distributed Smalltalk[3,4]. The underlying transport network (NCCE[5]) is represented by an Ethernet using TCP / IP or ATM. A part of this thesis project was the design and implementation of an API, which allows computation objects with in the TANGRAM DPE to be accessed.

For a better understanding—in particular of the API which I have implemented—this chapter will give a more detailed, engineering viewpoint oriented description of the concepts of the TANGRAM DPE than has been given in the proceeding chapters.

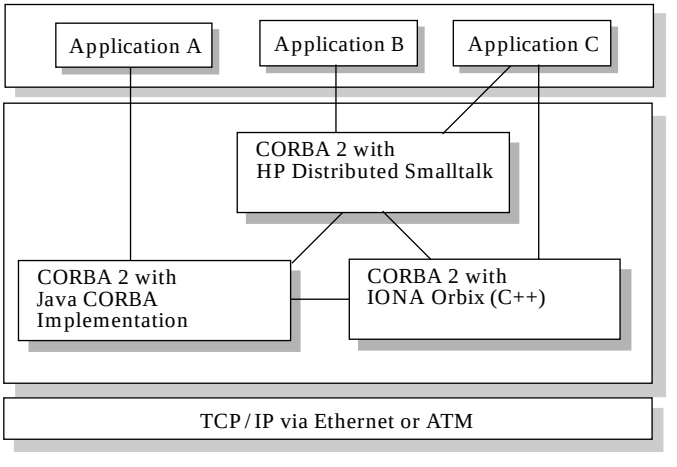


Figure 4-1. The TANGRAM DPE

1. [Leydekkers+95]
2. [IONA94]
3. [HP95]
4. [HP95a]
5. See "Native Computing and Communications Environment" on page 10.

4.2 General Concepts

Distribution Transparency

The description of an abstract infrastructure, which enables the execution of TINA applications, is called the TINA Distributed Processing Environment (DPE). The Distributed Processing Environment supports the execution of distributed telecommunication applications. It is a platform on which distributed applications, like a multimedia communication service, can operate. The DPE provides distribution transparency and is an infrastructure which consists of interconnected DPE nodes.

The DPE node is under control of a DPE kernel which is an abstraction of the NCCE. A DPE node consists of a number of capsules which can also consist of a set of object groups called clusters. The communication mechanism needed between different clusters uses the concept of channels. The DPE offers a number of functionalities from the engineering viewpoint which are needed to support the distributed applications[6,7,8]. The DPE provides distribution transparency, allows object interactions and supports the object life-cycle. This is accomplished by a variety of services. The Object Management Group (OMG) defines a DPE in the Object Management Architecture (OMA) called the Common Object Request Broker (CORBA)[9]. The TANGRAM DPE uses the CORBA technology.

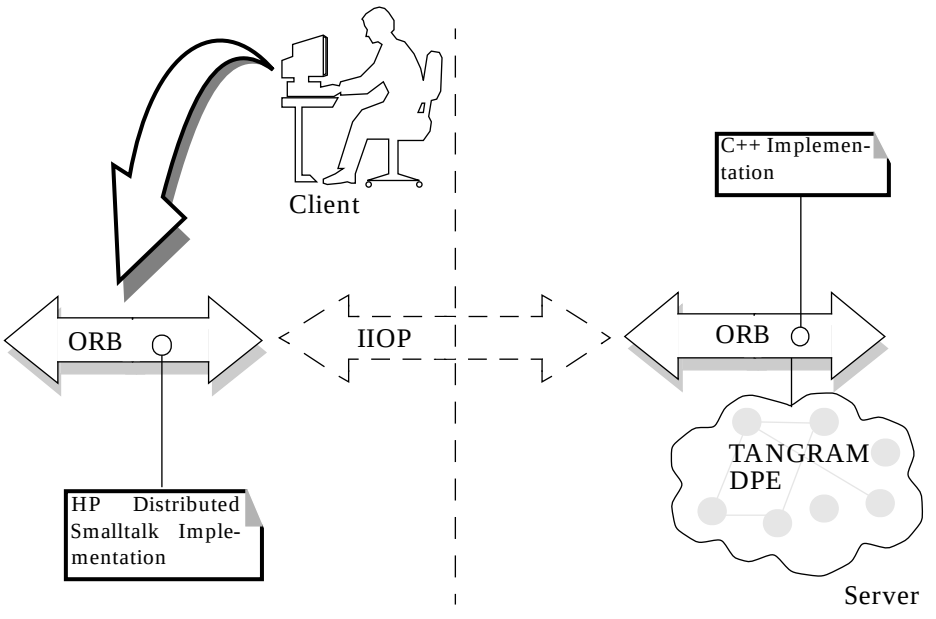


Figure 4-2. Usage of Different ORB Domains in TANGRAM

6. [Graubmann+94]
7. [Leydekkers+95]
8. [Leydekkers+95a]
9. [OMG:ORB95]

Interfaces

Every object in TINA can have more than one interface. In contrast, a CORBA object has only one single interface. To reconcile this conflict, TINA objects must be mapped to a number of CORBA objects. That means, that each interface is represented by a CORBA object. Furthermore, there is one CORBA object which represents the core TINA object[10].

Core TINA Objects

CORBA defines one interface per object, i.e. the object is the interface. In TINA one Computational Object can have multiple interfaces. To make this possible, one TINA object must be mapped to a set of CORBA objects. One Basic Engineering Object (BEO) consists of a number of interface objects and one core object which realizes the semantics defined by the CO specification.

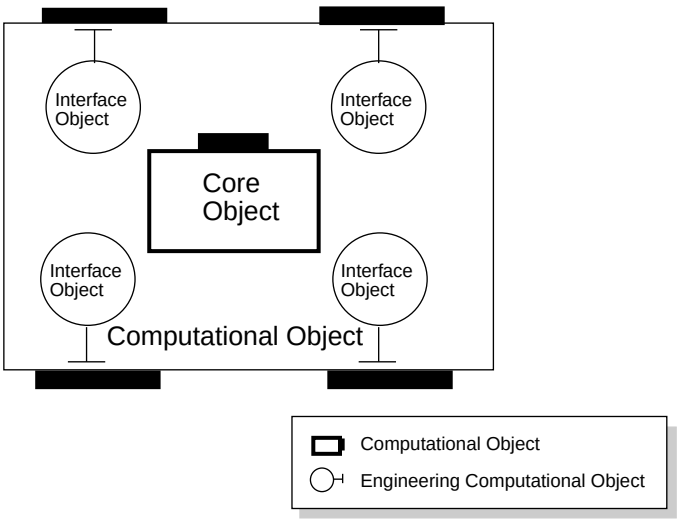


Figure 4-3. Mapping of TINA Computational Objects to CORBA Objects

The TANGRAM DPE supports the basic DPE object services, described in Chapter 5 and is implemented using the commercially available CORBA 2.0 technology of IONA Orbix[11] and HP Distributed Smalltalk[12]. The different applications use one or more of these CORBA-implementations. The underlying transport network (NCCE) is represented by an Ethernet using TCP/IP or ATM. Figure 4-1 shows the TANGRAM platform as a DPE in a TINA compliant environment.

10. [Eckert96]
11. [IONA95]
12. [HP95a]

4.3 TANGRAM Services

The CORBA Services used in the TANGRAM project are

- Naming Service,
- Life Cycle Service and Relationship Service.

Repositories

These services are supported by two kinds of repositories: the I*nterface Repository* and the *Implementation Repository*. The Interface Repository provides persistent storage of interface definitions specified in IDL. The Implementation Repository maintains the information that allow the system to locate and activate implementations of objects[13].

Naming Service

The Naming Service is used to obtain the interface references of a specific CO from within the system. It is implemented using HP-DST 5.0 and has an interface specified in IDL, so it can be used by different ORBs, such as for example Orbix. Since the version 2.0.1 of IONA's Orbix used in this project does not offer a Naming Service, the CORBA-conform Naming Service of HP-DST is used to provide a common Naming Service in the TANGRAM DPE[14].

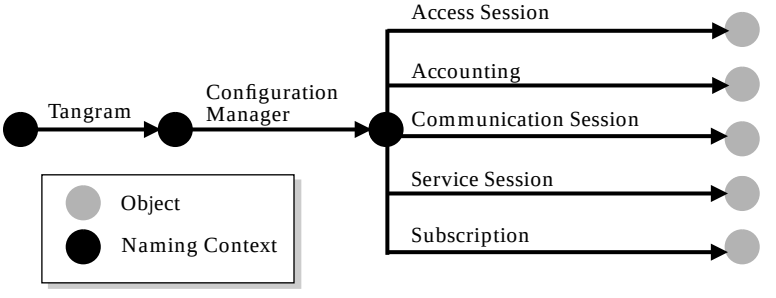


Figure 4-4. The TANGRAM Naming Graph

The root context of the Naming Service is TANGRAM, which contains the context *ConfigurationManager* containing the Configuration Managers of the different sessions. From the different Configuration Managers the corresponding session objects can be accessed. As there may be several objects of one kind (e.g. various service sessions for one user or various User Agents in one Access Session), each object is also identifiable by a unique name. Figure 4-4 shows the TANGRAM Naming Graph with the corresponding sessions objects[15].

IOR

As the Naming Service usually resides in the Provider Domain, it represents the entry point for accessing the components of this domain. In order to allow the utilization of these components, the IOR of the Naming Service has to be stored in a well known file, so that it may be transferred to another domain (e.g. to the User Domain via ftp). Each component that will be added to a domain (such as, for example, the Registration Server) has to register one or more of its interfaces at the Naming Service in order to be accessible for other components.

13. [OMG:ORB95]
14. [Schoo+96]
15. [Schoo+96]

Life Cycle Service

The Life Cycle Service together with the Naming Service support location transparency, which provides a logical view of naming, independent of the actual physical location of the object[16].

The TANGRAM DPE provides objects which serve for the life-cycle of Computational Objects. The *Configuration Manager* (CM) is responsible for a group of COs which can be assigned together. The *Life Cycle Manager* (LCM) manages the life-cycle of the instances of one specific Computational Object type. These managers and their interfaces will be explained in the next section. Figure 4-5 shows the CM, LCM and the associated CO-types.

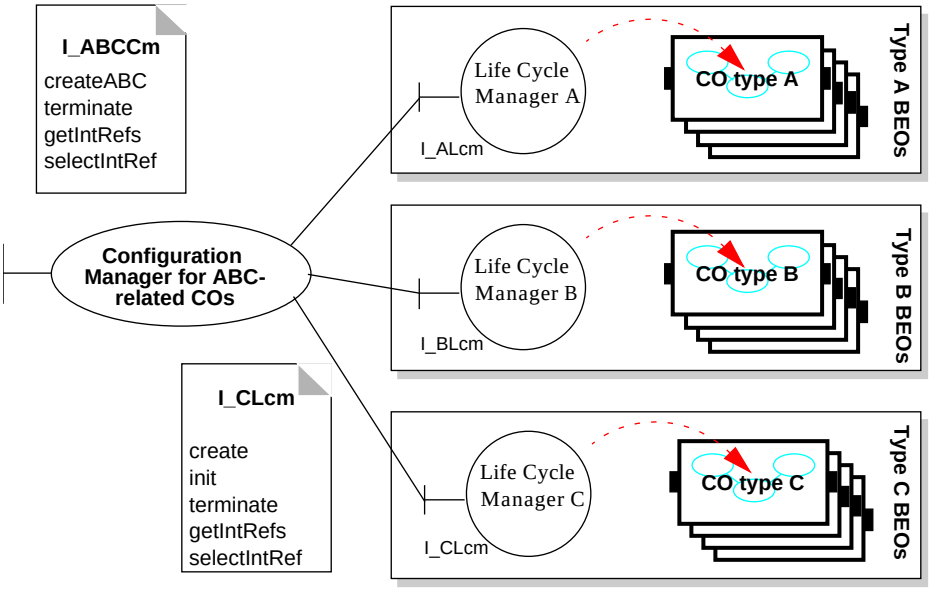


Figure 4-5. The TANGRAM Configuration and Lifecycle Managers

The Relationship Service of the TANGRAM DPE is implemented using HP-DST and is not compliant to the CORBA standard[17]. Since it offers an IDL-interface, it may also be used by other ORBs. Note that this service is only used by one type of *PCS in TINA* COs, the Invitation Handling Logic.

4.4 TANGRAM Engineering Concepts

This section describes the interfaces of the objects responsible for configuration management, life-cycle management and the control of COs. These interfaces may be considered as template classes, which are inherited by the CO-classes. The implementation of the inherited methods is then class-specific. This mechanism ensures that the various COs can be accessed in a uniform way.

16. [Schoo+96]
17. [HP95a]

The definition of the above described types, structures and methods is found in the file i_TnManagers.odl, which is included by the CO-classes of *PCS in TINA*. For a complete ODL-listing refer to Appendix A in the *PCS in TINA Report 2*[18].

4.4.1 Common Data Types

The following data types are used by the TANGRAM DPE.

T_CompObjectType

The enumeration type T_CompObjectType enumerates all the computational objects available. These are, for example, TnUserApplication defining a User Application or TnPcsLocalContext defining the PCS Local Context (LCxt).

T_CoIntRefList

The sequence T_CoIntRefList represents a list of CO interface references (T_CoIntRef), each reference consisting of a pair: the interface type (T_InterfaceType, a CORBA Repository ID represented as a string) and the reference to this interface (T_IntRef, represented as a CORBA object). This list is returned when the interfaces of a CO are requested.

T_IntRefList

The sequence T_IntRefList represents a list of interface references of a collection of COs. Each element (T_CoIntDescr) consists of the type of the CO (T_CompObjectType) and the associated list of references (T_CoIntRefList). This list is returned when the create-operation of a Configuration Manager is called (see below).

Exceptions

Additionally, an exception is defined, which may be raised by every method described in the following section. The exception E_TnNotYetImplementedError informs the calling party, that the invoked method has not yet been implemented.

4.4.2 Configuration Manager

The Configuration Manager (CM) is responsible for creating, terminating and requesting the interfaces of the Computational Objects belonging to the Configuration Managers. It registers itself at the Naming Service. Objects that want to use the services offered through the interfaces have to retrieve the specific CM interface reference through the Naming Service.

AccessSession Configuration Manager

Environment Configuration Manager

In the Access Session two different types of Configuration Managers are implemented. One CM is responsible for COs which are associated with a specific user. This CM is called AccessSession Configuration Manager (AssCm). The other CM deals with the COs which represent environment objects. This CM is called the Environment Configuration Manager (EnvCm). Every CM knows its own Life Cycle Managers, which are explained below. The definition of the interface of the CM template Manager (I_ConfigurationManager) is also given below.

18. [Arbanowski+96a]

Interface Definitions

The CM-interface defines **four methods** which have to be implemented by the inheriting class. One method is class-specific (i.e. its parameters are specific to each CO-class) the other three are general 'virtual' methods.

The **class-specific method** is the create method, which is called whenever an object requests the creation of an instance of the class(es) for which the CM is responsible. It requires at least two parameters plus some class-specific parameters. The required parameters are:

- the instance name of the type T_InstanceName, which is the TANGRAM external representation of an entity, as input parameter
- the ID for the created entity of the type T_InstanceId, which is the TANGRAM internal representation of an entity, as inout parameter

As a result, the method returns the list of all interface references(T_IntRefList), or raises an exception (either factory or naming).

The **three general methods** are:

- terminate: stops execution of the CO-instance with the given instance ID and removes it or raises a naming exception if it does not exist.
- getIntRefs: returns the list of interface references (T_CoIntRefList) of the requested CO type and instance ID or raises a naming exception if it does not exist.
- selectIntRef: returns the interface reference (T_IntRef) to the requests CO-type and interface-type with the given instance ID or raises a naming error.

The interface declares its own exceptions for two categories of faults: those concerning the creation and deletion of COs (factory exceptions - E_CmFactoryError) and those concerning the naming / retrieval of COs (naming exceptions - E_CmNamingError).

Factory Exception Codes

The factory exception codes (T_CmFactoryExceptionCodes) are:

- CmNamingConflict, (some) COs with the same name are already existent
- CmTooManyCOs, there are insufficient (system) resources
- CmNoSuchCO, the requested CO (i.e. the type) is not existent
- CmNoSuchInstance, the requested CO-instance (identified through an ID) is not existent
- CmNoSuchInterface, the requested interface for a CO is not existent
- CmInvalidInitParams, the given initialization parameters for the creation of a new CO-instance are not valid.

The **naming exception codes** (T_CmNamingExceptionCodes) raised are:

- CmUnknownCO, the requested CO is unknown
- CmUnknownInstance, the requested CO-instance (identified through an ID) is unknown
- CmUnknownInterfaceType, the requested interface type for a CO is not known

4.4.3 Life Cycle Manager

The interface I_LifeCycleManager is defined by the TANGRAM DPE and represents a template for factory functionalities needed to control and enable the creation, deletion and initialization of objects[19],[20]. It is intended to standardize the interfaces of the COs used on the TANGRAM platform in order to facilitate their interactions. For each CO-class, there is a Life Cycle Manger (LCM), which is derived from this template interface. That means, each LCM is responsible (i.e. provides the factory functionalities) for exactly one CO-type. Like the CM-interface, the LCM-interface defines its own exceptions and data types.

A **read-only attribute** of the type T_CompObjectType represents the type of CO for which the LCM is the factory.

Interface Definitions

The interface defines **five methods**, where one is class-specific (i.e. the parameters are specific for this class).

The **class-specific method** is the initialization (init) method, which is called after the creation of a CO-instance. It requires at least two parameters plus some class-specific parameters. The required parameters are:

- the instance ID (type T_InstanceId) of the CO which will be initialized
- The list of required interfaces for this CO of the type T_IntRefList

The four general methods are:

- create: creates an instance of the CO-class with the given name (T_InstanceId) and returns the list of interface references (T_CoIntRefList) if the creation was successful, otherwise it raises a factory exception. The init-method may be called immediately after the creation.
- terminate: stops execution of the CO-instance with the given instance ID and removes it or raises a factory exception if it is not existent.
- getIntRefs: returns the list of interface references (T_CoIntRefList) of the requested CO instance or raises a naming exception if it is not existent.
- selectIntRef: returns the interface reference (T_IntRef) to the requested interface-type with the given instance ID or raises a naming error.

Except for the create-method, all above explained methods are mapped to the methods of the same name which are declared in the I_CoControl interface described in the next section.

Factory Exception Codes

The interface declares its own exceptions for two categories of faults: those concerning the creation and deletion of COs (factory exceptions - E_LcmFactoryError) and those concerning the naming / retrieval of COs (naming exceptions - E_LcmNamingError).

The factory exception codes (T_LcmFactoryExceptionCodes) are:

- LcmTooManyCOs, there are insufficient (system) resources
- LcmNoSuchInstance, the requested CO-instance (identified through an ID) is not existent
- LcmAlreadyExistingInstance, a CO-instance with the given ID already exists

19. [Eckert96]
20. [Schoo+96]

- LcmNoSuchInterface, the requested interface for a CO is not existent
- LcmConfigurationFailure, the configuration of the CO failed
- LcmInvalidInitParams, the given initialization parameters for the creation of a new CO-instance are not valid.

Naming Exception Codes

The naming exception codes (T_LcmNamingExceptionCodes) raised are:

- LcmUnknownInstance, the requested CO-instance (identified through an ID) is unknown
- LcmUnknownInterfaceType, the requested interface type for a CO is not known

4.4.4 Computational Object Control Interface

As explained above, a CO may have more than one interface. Each of these interfaces will then be represented (and also instantiated) as an independent object internally, whereas externally this fact is transparent. All interface(-objects) forming an object with multiple interfaces are controlled by the so called core-object. The core-object is responsible for the creation of each of the interface objects upon request and also for their destruction. For these purposes, it offers the I_CoControl interface which is used by the LCM-interface. In fact, most of the methods defined in the LCM-interface are mapped to the methods defined in this interface[21]. Figure 4-6 depicts the creation of an object instance.

I_CoControl interface

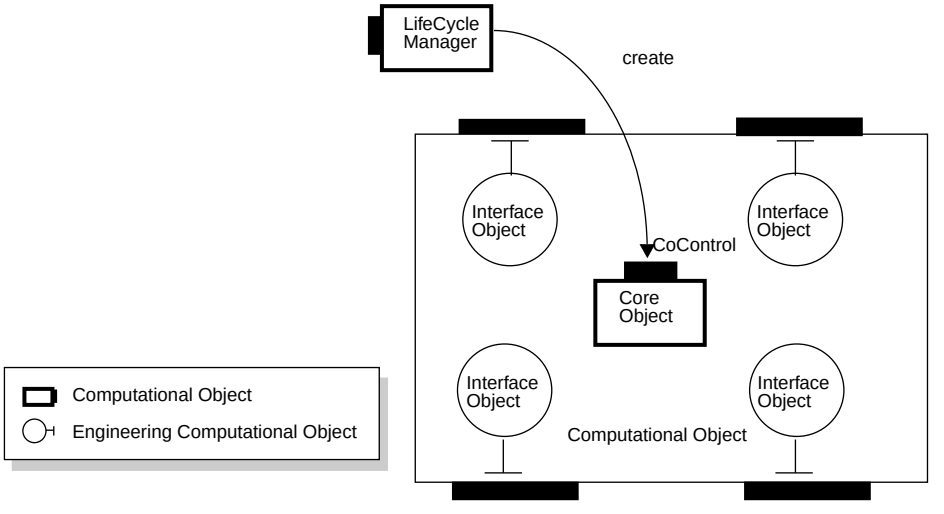


Figure 4-6. Creation of an Object Instance

A read-only attribute of the type T_CompObjectType represents the type of CO which is controlled.

21. [Schoo+96]

Interface Definitions

The interface defines five methods, where one is class-specific (i.e. the parameters are specific for this class). The class-specific method is the init method, which is called after the creation of a CO-instance.

The four general methods are:

- create: creates an instance of the CO-class with the given CO-type (T_CompObjectType) and returns the list of interface descriptions (T_CoIntDescr) if the creation was successful, otherwise it raises a factory exception. The init-method may be called immediately after the creation. Note: The specification of the CO-type to be created is redundant in this context, as the request for creation came from a LCM, and each LCM is only responsible for the management of one type of COs.
- terminate: stops execution of the CO-instance and removes the interface object as well as the core object (i.e. itself).
- getIntRefs: returns the list of interface references (T_CoIntRefList) of the CO instance or raises a naming exception if not existent.
- selectIntRef: returns the interface reference (T_IntRef) to the given interface-type of the CO or raises a naming error.

Exceptions

Like the CM- and the LCM-interface, the COControl-interface defines its own exceptions for factory (E_CoFactoryError) and naming (E_CoNamingError) errors.

The factory exception codes (T_CoFactoryExceptionCodes) are:

- CoTooManyCOs, there are insufficient (system) resources
- CoNoSuchInterface, the requested interface is not available for this CO-class
- CoConfigurationFailure, an error occurred during the configuration of the CO
- CoInvalidInitParams, the passed parameters for the initialization of the CO are invalid

The only **naming exception code**(T_CoNamingExceptionCodes) is:

- CoUnknownInterfaceType, the type of interface is unknown for this CO-class

4.4.5 Access Session Configuration Manager

The Access Session Configuration Manager (AssCm) looks for the following user specific COs:

- UA,
- UCxt,
- ULoc,
- PPrf,
- PCL,
- PS,
- Authentication and
- Session Description.

All these objects are highly user related and represent the user and his preferences.

Each user of the system has one unique identifier. Figure 4-7 shows the AssCm with its Life Cycle Managers and the objects which are handled by the LCMs.

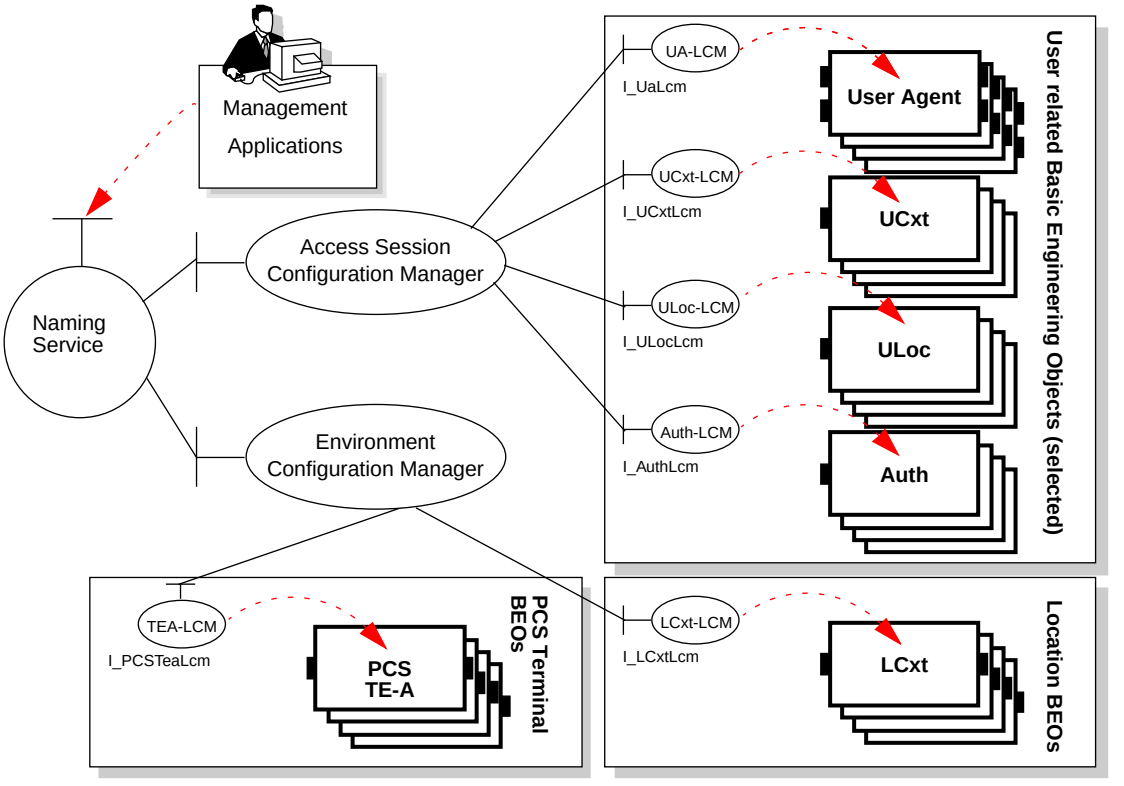


Figure 4-7. Engineering Viewpoint on Management

5 Common Object Request Broker Architecture

This chapter gives an introduction to the Common Object Request Broker Architecture (CORBA) as far as is necessary for the understanding of this project. The CORBA is used in the realization of the TANGRAM DPE to support distributed processing.

5.1 Introduction

An OMG compliant Object Request Broker (ORB) serves as a bus for the transport of requests from a client to a server in distributed processing environments. The ORB relays the invocation from the client to the object implementation on the server side and then relays the result back to the client. The ORB is responsible for locating the object implementation and for transporting the data from the client to the server independent of location, implementation and host. It lets objects send requests transparently to other locally or remotely located objects. The ORB is able to locate the specific object and the interface needed to operate the request. The answer to the request retraces its steps in reverse through the ORB to the client. The request is made via a dynamic invocation interface or a stub. The interface definition language (IDL), describes the interface between the client and the server. The object adapter is the interface between the ORB core and the object implementation[1].

5.2 CORBA Components

A Common Object Request Broker Architecture (CORBA) consists of the following components:

- Object Request Broker
- Application Objects
- Common Facilities
- Common Object Services

5.2.1 Application Objects

Application Objects are components specific to end-user applications which represent an enterprise model. An application is typically built from a number of cooperating business components that together, serve a specific purpose. These application objects are built on top of services provided by the ORB, the Common Facilities and the Common Object Services.

1. More details can be found in [OMG:ORB95].

5.2.2 Common Facilities

Common Facilities are a collection of components that provide services for direct use to applications objects. OMG defines two categories of Common Facilities:

- Horizontal Common Facilities[2]
- Vertical Market Facilities

The horizontal set of Common Facilities includes functions shared by many or most systems, regardless of the type of content the application has.

The vertical set of Common Facilities (Vertical Market Facilities) represents standards for interoperability in particular specialty areas, e.g. Computer Integrated Manufacturing (CIM). Each speciality area represents the needs of an important computing market. There is no limitation on the amount of Vertical Market Facilities.

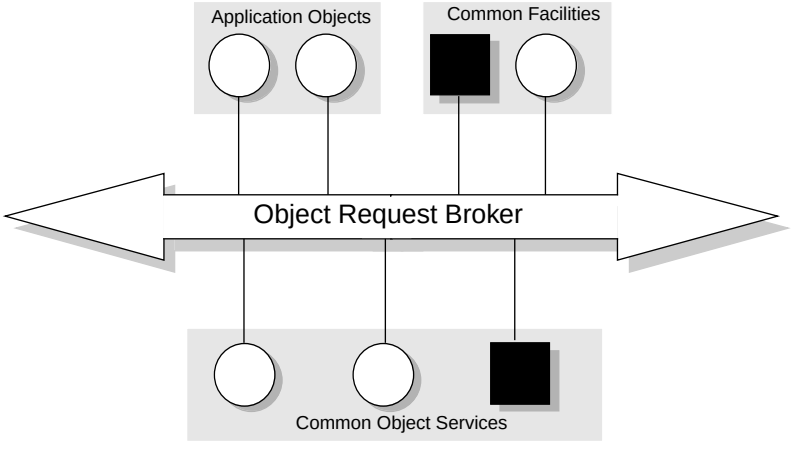


Figure 5-1. Common Object Request Broker Components

5.2.3 Common Object Services

CORBA Object Services are collections of system-level services. CORBA Object Services are described in more detail in the following section.

5.3 Object Services

CORBA Object Services extend and complement the functionality of the Object Request Broker. CORBA object services are used to create a component, to name it and to introduce it into the environment[3].

2. OMG defines four Horizontal Common Facilities: User Interface, Information Management, Systems Management and Task Management
3. [Orfali+96]

The following services are defined as standards by OMG[4]. Services in bold typeface are used in the context of the *PCS in TINA* project and are described below.

- Naming Service
- Event Service
- Life Cycle Service
- Persistent Object Service
- Transaction Service
- Concurrency Control Service
- Relationship Service
- Externalization Service

Note, that the underlying platform of the present implementation only makes use of the following CORBAservices, which are explained in the next sections:

- Naming Service,
- Relationship Service,
- Lifecycle Service.

5.3.1 CORBA Naming Service

The Naming Service defines conventions for binding a name to an object relative to a naming context. Names are humanly recognizable values that identify an object. Names can be passed to other objects. Objects can address the naming service and resolve the name in a given context. The naming service allows the creation of naming hierarchies. Clients can navigate through different naming context trees in search of a specific object. Name contexts from different domains can be used together to create federated naming services for objects. A CORBA naming hierarchy does not require a 'universal' root[5],[6] .

5.3.2 CORBA Relationship Service

Objects of the real world do not exist in isolation. They build relationships that come and go. Therefore a system, modelling the real world should provide a way to create relationships between objects that could be dynamic, static or created ad hoc. The CORBA Relationship Service allows the creation of relations between objects that do not know of each other and therefore do not have to keep track of existing relationships between themselves and other objects. The Relationship Service provides a way to create relationships between immutable objects[7].

The CORBA Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. OMG defines two kinds of objects:

- Relationships and
- Roles.

---

4. [OMG:Services95]
5. [Orfali+96]
6. [Schoo+96]
7. [Orfali+96]

A role represents a CORBA object in a relationship. A compound object is a structure of objects which are linked together and appear to be a single object. It may consist of several CORBA objects. Any life-style request applied to the compound object affects the entire cluster of CORBA objects. Clients are unaware of the implementation of objects they interact with and link to. They may apply a 'delete' request to a compound object to delete an entire cluster of objects including their relationships.

5.3.3 CORBA Life Cycle Service

OMG defines the Life Cycle Service as follows:

> The CORBA Life Cycle Service defines services and conventions for creating, deleting, copying and moving objects. Because CORBA-based environments support distributed objects, the Life Cycle Service defines conventions that allow clients to perform life cycle operations on objects in different locations[8].

In CORBA 2.0 the Life Cycle Service was expanded to handle associations between groups of related objects, which include containment and reference relationships. The Relationship Service is used to define these relationships. The Life Cycle Service provides interfaces that are derived from the Relationship Service[9].

5.4 The Inter-ORB Communication Architecture

The implementation of the TANGRAM DPE, which is used in the *PCS in TINA* project, makes usage of different ORBs. Therefore it is necessary to have a closer look to the concepts which define the communication between different ORB implementations of different vendors.

For communication between different ORB implementations, OMG defines special protocols. These protocols are the Internet Inter-ORB Protocol (IIOP) and the Environment-Specific Inter-ORB Protocol (ESIOP). The later is not used in this implementation and therefore is not described. To pass object references between different ORB implementations, OMG defined the Interoperable Object Reference or IOR.

Since the version 2.0 of CORBA, the interoperability between ORBs of different vendors is defined by specifying a mandatory Internet Inter-ORB Protocol. The IIOP is basically TCP/IP with some CORBA-defined message exchanges that serve as a common backbone protocol. To be considered CORBA compliant, an ORB must either implement IIOP natively or provide a 'half-bridge'[10] to it.[11]

The following sections will give a short description of this terms.

8. [OMG:Services95]
9. [Orfali+96]
10. half-bridge because IIOP is the 'standard' CORBA ORB.
11. [Orfali+96]

5.4.1 GIOP

For the communication between ORBs, a specially built protocol is set forth; the General Inter-ORB Protocol (GIOP). It specifies a set of message formats and common data representations for the Inter-ORB communication. GIOP is designed to work directly over any connection-oriented transport protocol and defines seven message formats that cover all the ORB request and reply semantics. Therefore, no format negotiations are needed.[12]

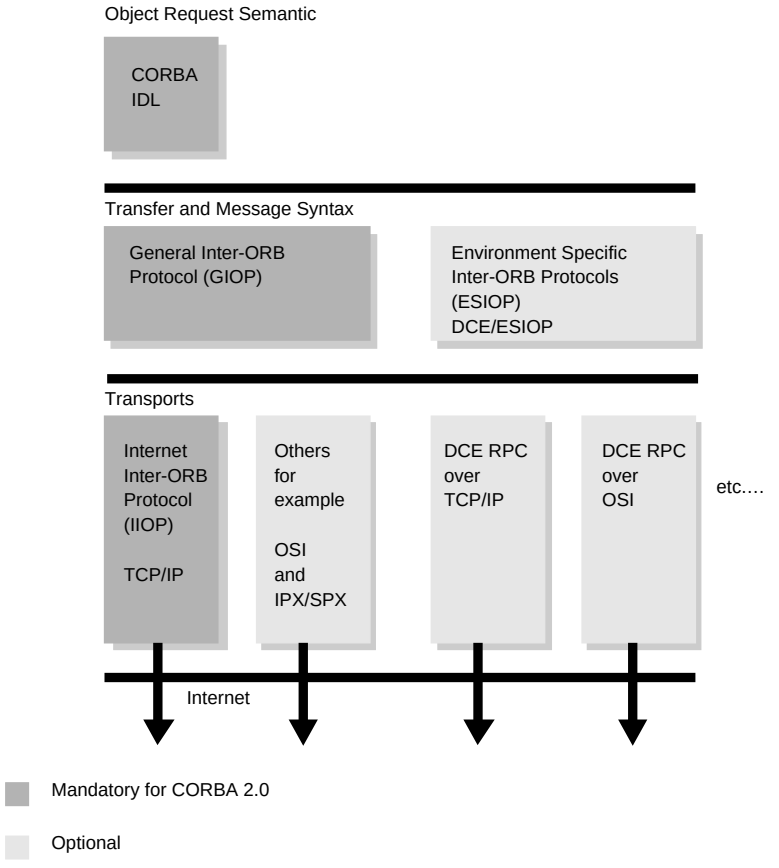


Figure 5-2. CORBA Inter-ORB

5.4.2 IIOP

The Internet Inter-ORB Protocol (IIOP) specifies how GIOP messages are exchanged over a TCP/IP network. The IIOP makes it possible to use the Internet itself as a backbone ORB through which other ORBs can bridge. To be CORBA 2.0 compliant, an ORB must support GIOP over TCP/IP or connect to it via a half-bridge.

12. [Orfali+96]

5.4.3 IOR

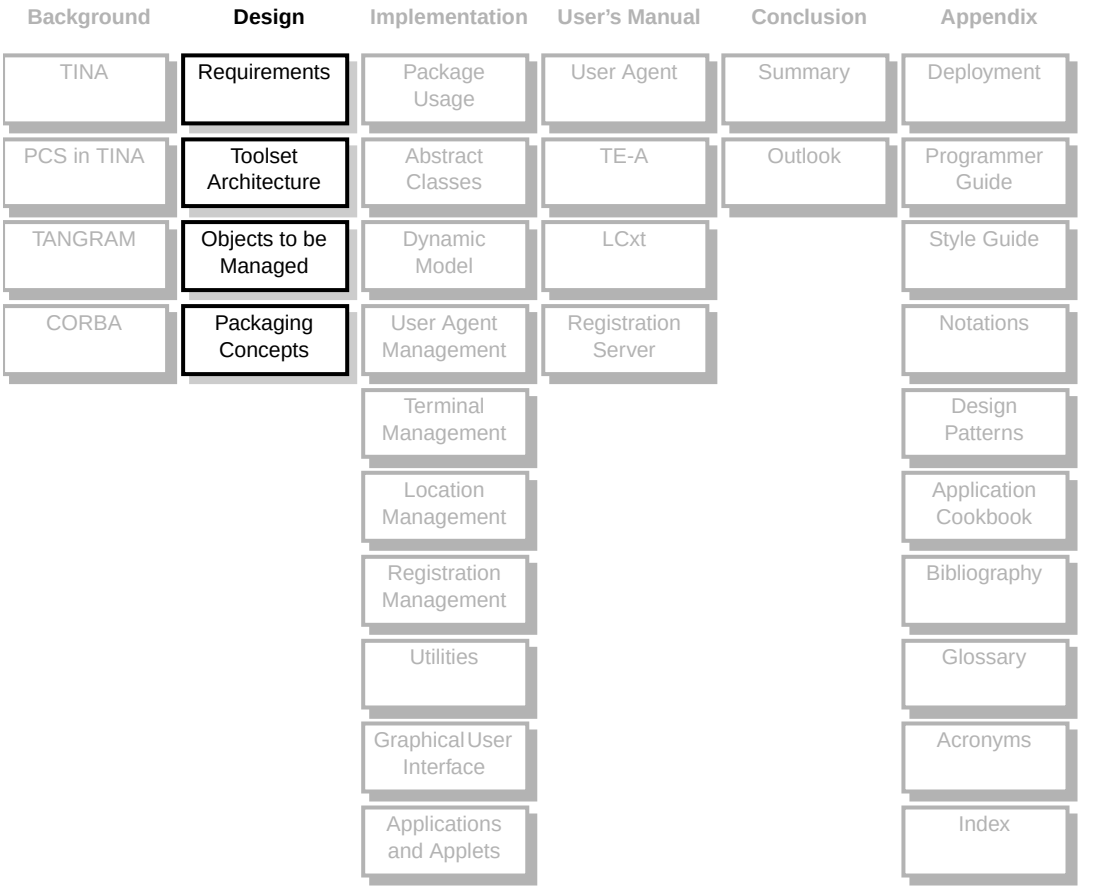
GIOP also defines a format for Interoperable Object References (IORs). To pass an object reference between ORBs, an IOR must be created. An IOR describes how an object is to be contacted using a special ORB mechanism. An IOR includes self-describing data that identifies the ORB domain to which a reference is associated and the protocols it supports.[13]

13. [Orfali+96]

Part 2

Requirements and Design

This portion of the book presents the design of the management toolset. It starts with the requirement specifications for this project. Next, the overall architecture of the toolset is introduced followed by a chapter about the design of the managed objects. This part closes with a description of the package subdivision of the architecture and the delineation of the design of the overall objects.


Background
Design
Implementation
User's Manual
Conclusion
Appendix
TINA
PCS in TINA
TANGRAM
CORBA
Requirements
Toolset Architecture
Objects to be Managed
Packaging Concepts
Package Usage
Abstract Classes
Dynamic Model
User Agent Management
Terminal Management
Location Management
Registration Management
Utilities
Graphical User Interface
Applications and Applets
User Agent
TE-A
LCxt
Registration Server
Summary
Outlook
Deployment
Programmer Guide
Style Guide
Notations
Design Patterns
Application Cookbook
Bibliography
Glossary
Acronyms
Index

6 Requirement Specification

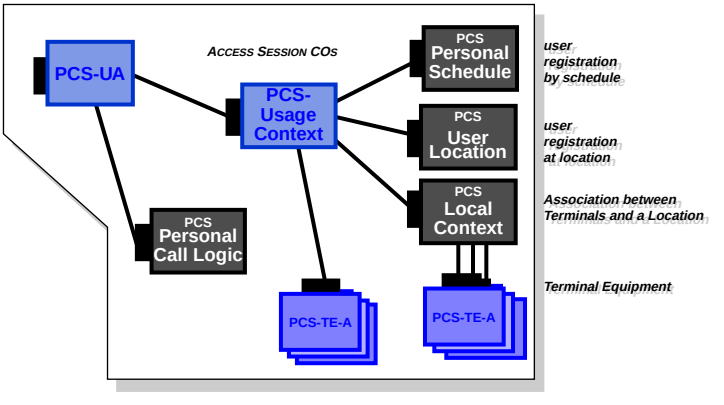
6.1 Introduction

The *TINA-C Auxiliary Project PCS in TINA* (*PCS in TINA*) introduces personal mobility supporting aspects into the TINA-C Service Architecture. The objectives of the *PCS in TINA* project are to enhance and extend the TINA Access Session by concentrating on the management of reachability. Thus, by providing the user with capabilities for a personalized invitation management, the user is able to manage his or her own reachability in a uniform and efficient way, abstracting as much as possible from technicalities.

To achieve this goal, *PCS in TINA* had to introduce new computational objects into the TINA-C Service Architecture (SA 94)[1] and to enhance existing ones.

The following computational objects have been enhanced or newly introduced by *PCS in TINA* into the TINA-C Architecture (cf. Figure 6-1):

- PCS User Agent (PCS-UA)
- PCS Usage Context
- PCS Terminal Agent (PCS-TE-A)
- PCS Personal Schedule
- PCS User Location
- PCS Local Context.



Dark shaded boxes represent computational objects which have been introduced into the Service Architecture 94.

Light shaded boxes are enhanced computational objects

Figure 6-1. PCS Enhancements to the TINA Access Session

1. The *PCS in TINA* project is based on the TINA-C Service Architecture 94. Since beginning this thesis project the fall of 1996, TINA-C has delivered a revised Service Architecture, called SA 96.

These computational objects have to be managed in compliance with the TINA standards, i.e. they must be accessed at their provided management interfaces.

6.2 Objective of this Work

The objective of this work is to design and implement a management toolset which is to be used by the management applications for the following PCS computational objects:

- PCS User Agent
- PCS Terminal Agent
- PCS Local Context.

An other central objective is the design and implementation of a management application for the **Registration Server Management**.

Technical Requirements

The following technical requirements had to be fulfilled:

- The computational objects are located in a TINA Service Architecture 94 compliant distributed processing environment (DPE).
- The required DPE is the TANGRAM DPE.
- The TANGRAM DPE uses the Common Object Request Broker Architecture (CORBA) technology. Therefore, the management toolset has to communicate with computational objects via their CORBA Interface Definition Language (IDL) interfaces using an Object Request Broker (ORB).
- Different ORB domains are supported by the TANGRAM DPE. Therefore, CORBA IIOP concepts must be considered.
- Design and implementation must be object-oriented.
- The management toolset should be available on different operating systems, therefore;
- The programming language is Java.
- To access the ORB, the Visigenic Visibroker for Java has to be used.
- Graphical user interfaces must be provided as end-user front ends.
- Different kinds of end-user types (beginner, expert, secretary, administrator, etc.) need different access controls which should be customizable.
- Reusability and maintainability should be considered.
- The possibility to access different platforms (e.g. ORACLE Database) should be provided.

To manage the computational objects means, to use the provided facilities of that platform and its embedded objects. The management toolset has to offer the functionality to create, modify and delete these computational objects. Those functionalities must be available from within management applications.

Target Group

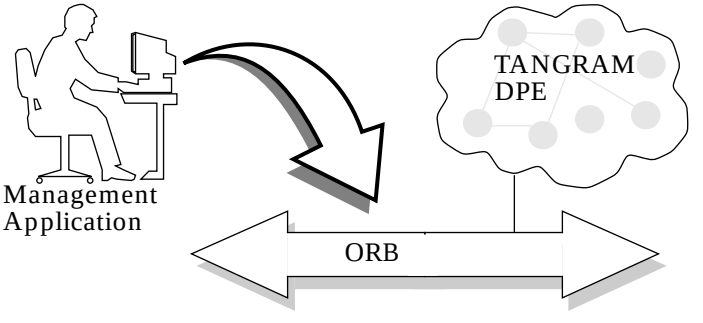
The target group for these management applications, are experienced system administrators, as well as end-users who simply want to configure computational objects within their system. Each component has to be managed independently—that means that each component is managed with its own independent application.

Therefore, one aspect of this project is to provide applications with graphical user interfaces which faciliate customer access to the management services provided by the PCS components. The guiding intention during the design process was, that the graphical user interfaces (GUIs) should provide state of the art quality and userfriendly access to the underlying management functions. On the one hand, these GUIs must hide the complexity of the system providing an intuitive management of the components, and on the other, they must give an advanced user full control over the managed components.

Platform Independence

Another design criteria was the aspect of independence from an underlying platform. Although the management toolset has to operate on the TANGRAM DPE, it is desirable to supply other platforms as well, whenever needed.

The design was largely guided by the need for applications which are independent from an underlying existing platform. In other words, it is important to be able to add functionality to the application with ease and be able to access different platforms, e.g Database Management Systems, without having to rewrite the existing application.

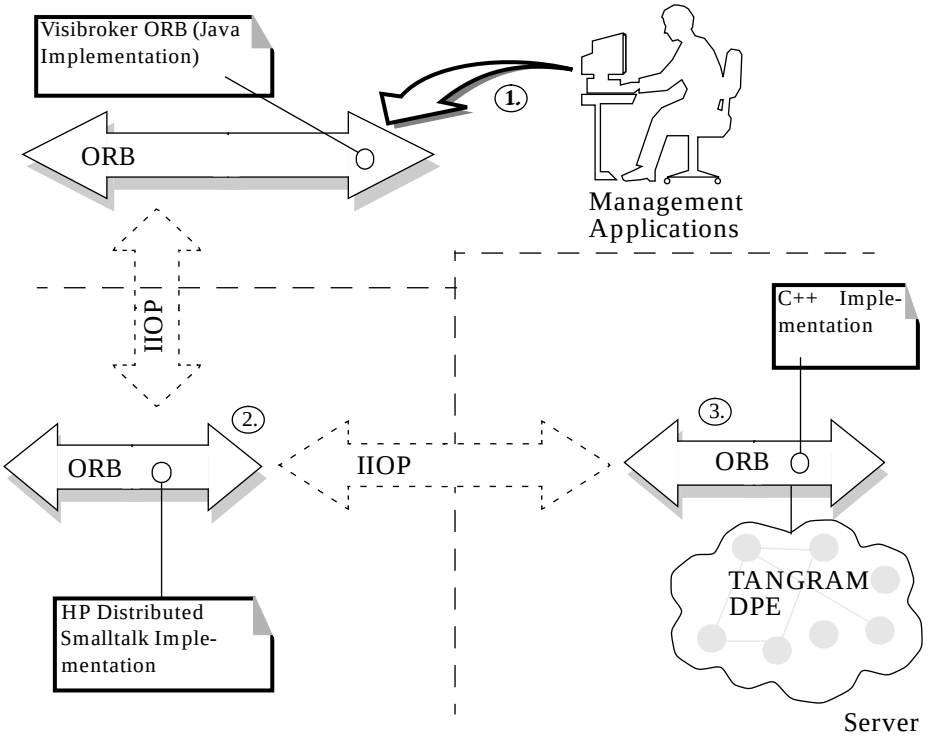


Access to the TANGRAM DPE is accomplished from ‘outside’ using the provided management interfaces. The management facilities are part of the application.

Extensibility

Access to the managed objects had to be oriented on the existing TANGRAM DPE, which is compliant with the TINA Service architecture 2.0 from March 1995. In December 1996, during the design phase of this project, TINA-C delivered a new version of their service architecture; the Service Architecture 4.0, also named SA 96. This version was followed one month later by the version 4.1 which included minor changes. It was obvious then, that the TANGRAM platform had to be changed sooner or later to fit the constraints of the new architecture. Therefore, this design was guided by the intension to create an application architecture which allows a migration to the new platform with as few changes as possible in the existing implementation.

Requirements concerning naming and programming conventions that I compiled for the implementation can be found in Chapter 28 ’Style Guide’ on page 135.



To get access to the computational objects to be managed, three different ORB Implementations were used: (1) The management applications contact the Visigenics Java ORB. (2) A HP Distributed Smalltalk implementation of an ORB is then contacted which contacts a C++ Implementation of an ORB to get access to the computational objects.

Figure 6-2. Bridging With Three Different ORB Implementations

6.3 Summary

The goal of this project is to design and implement a management toolset which administers PCS components in a TINA compliant distributed environment. The required DPE platform is the TINA compliant TANGRAM DPE which uses the recent CORBA technology. Besides a reusable and extendable design, the toolset should provide a graphical user interface to support end-users with different experiences. The programming language is the platform independent recent ‘Internet’ programming language, Java.

In order to get lean applications which could be used in Java enabled browsers, I decided not to build one monolithic application but instead to design a set of independent applications; small enough to be loaded in an acceptably short time.

Although based on the above mentioned requirements, the design of the management toolset was influenced by the design pattern languages that have emerged in recent years. Chapter 30 presents a catalog of the applied design patterns used in this thesis project.

7 Management Toolset Architecture

My main objective during the design phase of this project was to develop an architecture which can be applied to every single implementation of the management applications of the management toolset. This motivating idea was driven by the need for an application which is independent from both its underlaying platform, as well as from the service which the platform provides. It should also be easy to maintain, to extend and be as reusable as possible. This chapter explains my design decisions for the overall architecture of the management toolset.

7.1 Layering Concepts

Each management toolset follows the layering concepts that help to structure the applications into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Each management toolset is structured into three levels:

- An Application Layer. The application layer contains those parts of the management toolset described in the Model-View-Controller design pattern.
- A Service Access Manager Layer. The Service Access Manager Layer hides the access functionality of a specific platform from the Application Layer and offers a consistent interface for it. This allows an application to exchange a Service Access Manager for a specific platform with a Service Access Manager of a different platform at anytime.
- A Service Access Layer. The Service Access Layer contains the application programmer interface of a specific platform. There is no restriction on how such an API should be realized.

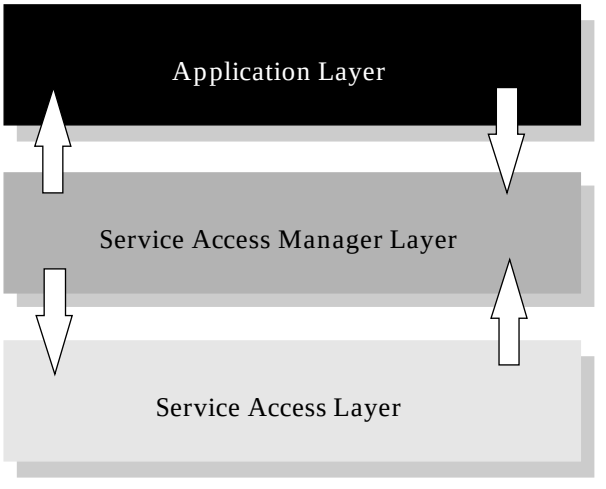


Figure 7-1. Toolset Layering Concepts

The following sections explain each layer in more detail.

7.1.1 Layer for Application

The application layer is the topmost layer. It is responsible for representing the informational aspects of the managed object. The application layer is designed following the Model-View-Controller design pattern.

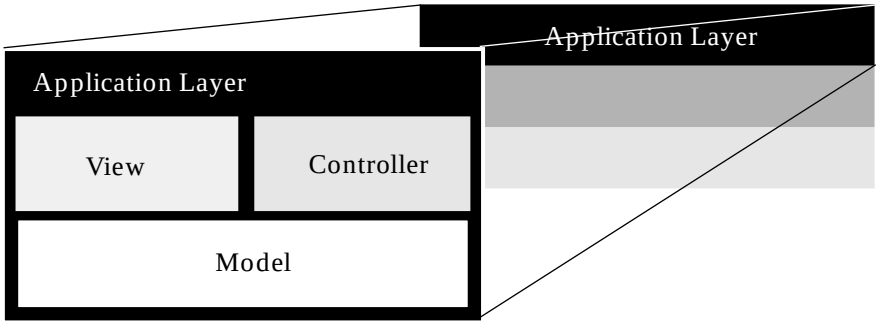


Figure 7-2. Toolset Layer for Application

The design guidelines for the application are explained in Section 7.3 on page 51.

7.1.2 Layer for Service Access Management

The Service Access Management Layer consists of the Service Access Managers, which are responsible for managing the access to their dedicated platforms. A Service Access Manager provides a unified interface to the application layer and is therefore exchangeable. The Application Layer does not specify what kind of Service Access Manager it deals with as long as the a Service Access Manager supports the expected interface. The Service Access Manager Layer also hides the complexity of an application programmer interface. This fulfills the constrains of the Facade pattern.

Facade

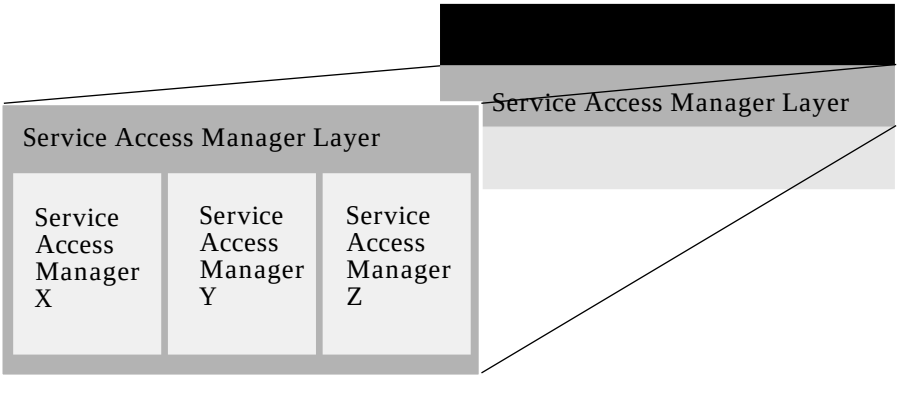


Figure 7-3. Toolset Layer for Service Access Manager

The service management layer is responsible for providing an unified interface to the application layer. The service management layer converts data retrieved from the service access layer into the format used by the application. The service management layer also hides the service access layer from the application layer.

This design provides the possibility to change platforms during runtime. If more than one Service Access Manager is available, the management applications are capable of managing different platforms simultaneously. This is made possible by introducing an abstract manager which defines the standardized interfaces for a Service Access Manager. The application only deals with the standardized interface of an AbstractManager which will be replaced by a concrete manager during runtime.

Another responsibility of the Service Access Manager is to convert data structures, which are retrieved from the Service Access Layer, to the data format structure expected from the Application Layer. This is an other aspect of achieving independence from an underlying platform.

7.1.3 Layer for Service Access

The Service Access Layer houses the application programmer interfaces (API)[1]. An API might be a package of modules which access specific platform or just one single monolithic unit. There are no restrictions on how the API is implemented and on what interfaces the API offers. All these aspects are wrapped by the Service Access Manager.

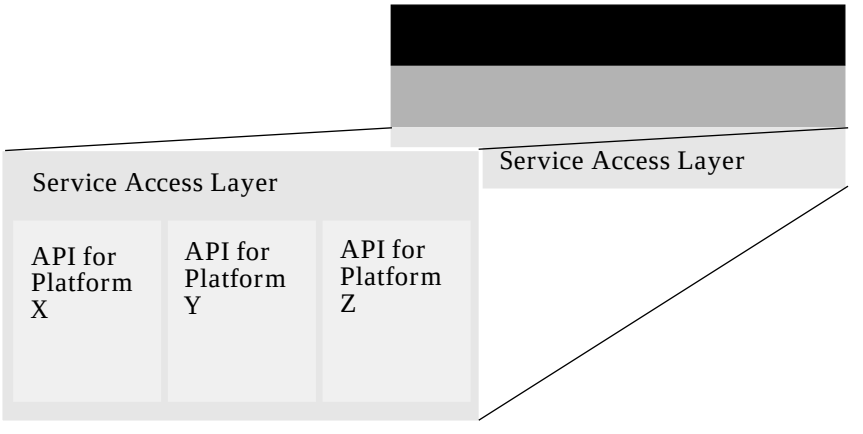


Figure 7-4. Toolset Layer for Service Access Manager

For example, one API could provide the access to a CORBA Object Request Broker while an other could provide the access to a Database Management System. Different API implementations could be exchanged, for example, because one offers better access to a system and the other, better performance. This exchange would only affect the Service Access Manager while the Application Layer would remain untouched.

1. A pattern description for API can be found in Mowbray's 'CORBA Design Patterns' [Mowbray+97] Library(144).

7.2 Design of the Overall Objects

This chapter describes the design of the general architecture for special classes which are implemented in every tool package.

7.2.1 Factories

The Factory classes are based on the design pattern Factory Method[2].

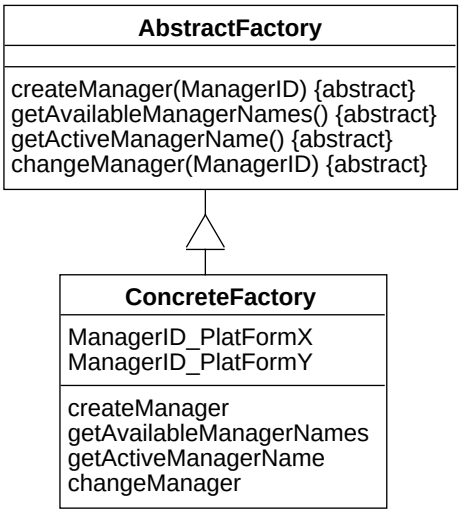


Figure 7-5. Abstract and Concrete Factories

The Factory classes responsibility is to create Service Access Manager (SAM) instances. They provide interfaces for creating SAMs of all supported platforms during runtime. Factories provide methods for querying a listing of all available SAMs. For the moment, only the TANGRAM platform is supported. An API for the PCS ORACLE database is in the implementation phase. The Access Manager to that platform will be implemented as soon as an API for that platform is available.

7.2.2 Models

A model is the representation of a TINA-based PCS component. The model is comprised of all the functionalities needed to administer a managed object.

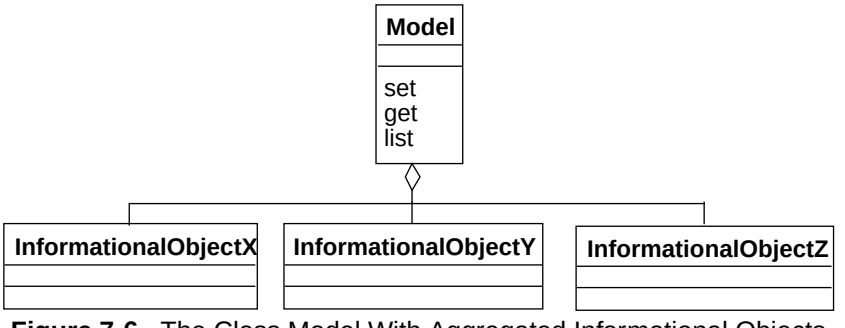


Figure 7-6. The Class Model With Aggregated Informational Objects

2. [Gamma+94], Factory Method (107)

Due of the structure of such a managed object, a model aggregates a set of classes. An extreme example is the Terminal Equipment Agent component, which consists of six informational objects. Each object is modeled in a class and each class could use a set of helping classes. The model keeps those parts together and provides an operational interface to operate on that objects in an consistent way. The model fulfills the requirements of the Facade pattern[3].

7.2.3 Views

Views are responsible for the representation of the presented data. A view might represent a group of data fields as well as a single field or a list. Each view has a controller which controls access to the model. A View is normally permitted to read data from the Model without using a controller.

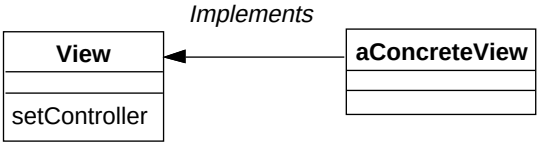


Figure 7-7. The Class View

7.2.4 Controllers

Each View communicates with the Model through a Controller. A Controller can be configured with more or less rights. If desired, it is possible to exchange a controller during runtime. This could allow, for example, a user to change from a beginner mode, which permits modification of data, to a more advanced mode with full access rights.

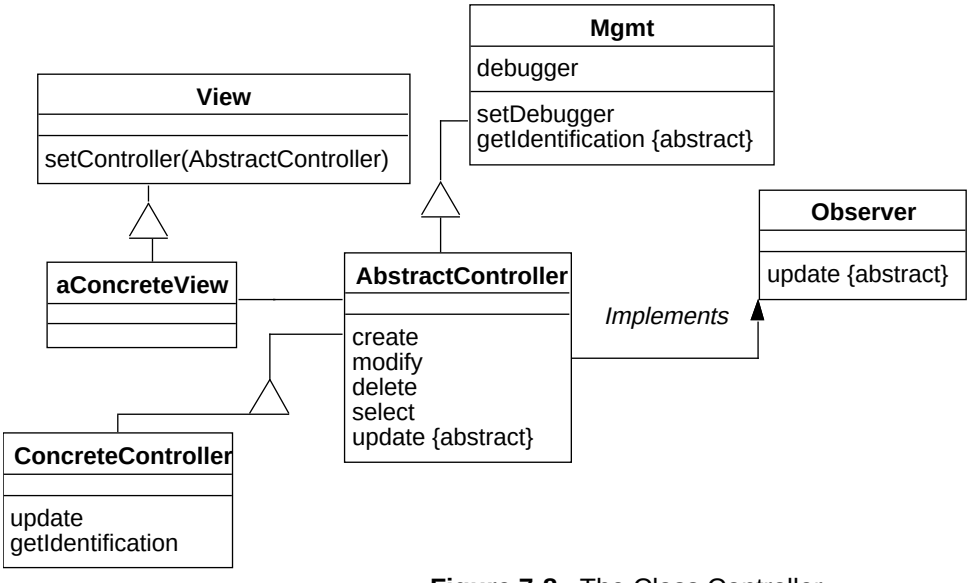


Figure 7-8. The Class Controller

3. [Gamma+94], Facade (185)

7.2.5 Service Access Manager

A Service Access Manager manages the access to a specific platform, e.g. TANGRAM or ORACLE database. It uses the API and converts data formats from a specific platform to the data format for the model. It separates the model from the API.

An AbstractManager defines the common signature of Manager methods. The specificAbstractManager, e.g. TEAAbstractManager describes more specifically the parameters needed by a specific Object Manager. The concreteManager, e.g. TEAManagerTANGRAM, implements the access methods for a specific platform using the provided application programmer interface (API).

An API does not need to support a predefined interface or special signatures. It is up to the concreteManager to wrap the specific methods and to marshall all the parameters needed by the API. This allows an easy exchange of APIs by just implementing a new concreteManager. Higher layers are not affected.

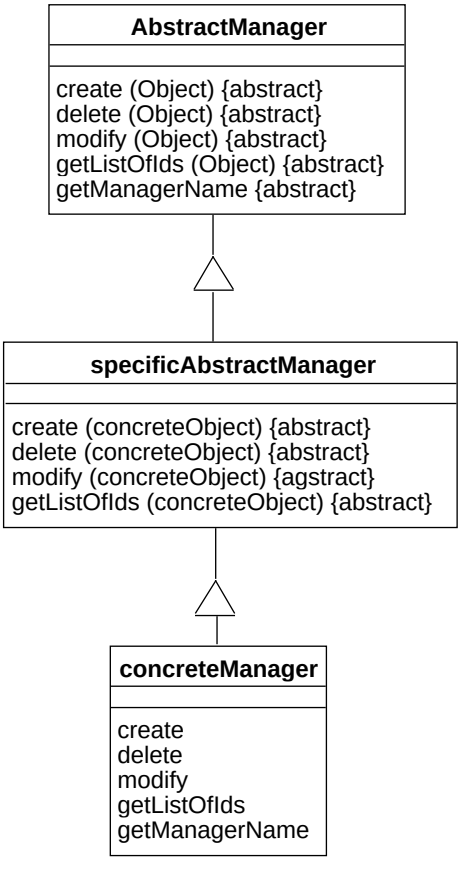


Figure 7-9. The Class Service Access Manager

7.3 Design of the Applications

Model-View-Controller

Applications are located in the Application Layer. The design of an application follows the Model-View-Controller pattern.

Command Processor

All applications support Undo and Redo commands, therefore the Command-Processor pattern is applicable.

Factory Method

The model grants access to the system. To do so, it needs a Service Access Manager. The application is responsible for choosing and creating a Service Access Manager and passing its references to the model. Therefore, the application uses a Service Access Manager Factory to create an appropriate Service Access Manager. This is also an application of the Adapter Pattern[4].

Adapter

Mediator

Where needed, the communication between different views is managed by a mediator— following the Mediator pattern.

Singleton

There is only one model for a specific type of managed object available. In order to avoid having more than one instance of a Model at runtime, I used the Singleton pattern. Nevertheless, an application can operate using different models of different managed object types. For example, the Location Manager uses the User Model to retrieve a list of all available users, and at the same time it can use the Terminal Equipment Agent Model to retrieve a list of all available terminals. In other words: there is no limitation on the number of different model-represented managed objects an application can use; but at runtime there will always be only one instance of a specific model.

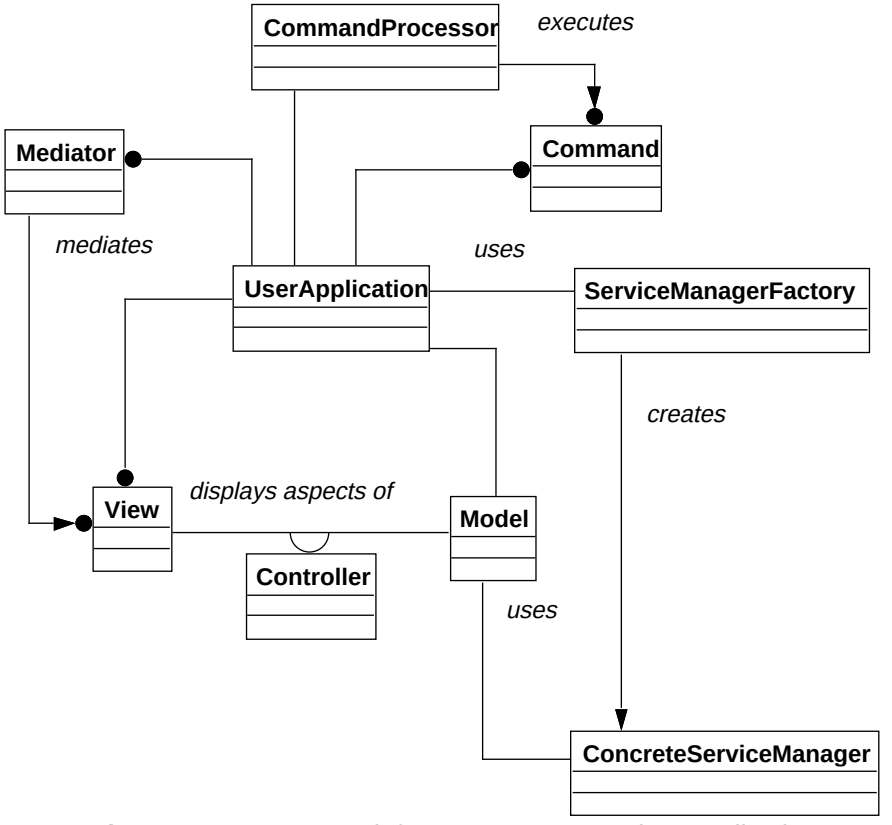
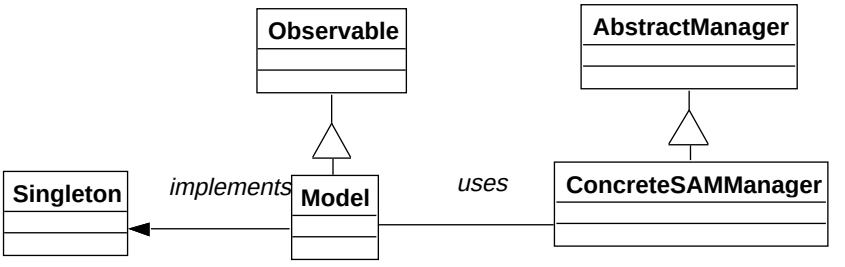


Figure 7-10. Framework for Management Toolset Applications

4. [Gamma+94]. Adapter(139)

7.3.1 Administering the Data

Operation on managed objects are only possible using the model which reflects the actual state of the managed object. The model uses the Service Access Manager to manipulate objects belonging to the system. The model has knowledge about the representation of the data only; it doesn't know how and where those objects are stored or how to retrieve them. Therefore, it is unaware of which Service Access Manager it uses because all of them support the standardized interface of the AbstractManager.



7.3.2 Displaying the Data

A view is used to display a specific aspect of the model. There is no limitation on the amount of views an application might have. Each view has its own controller which regulates a view's access to the model.

7.3.3 Controlling the Access to the Data

The controller used in this design differs from the classical controller of the Model-View-Controller pattern. The classical responsibility of the controller is to control the mouse and keyboard input of a user. Usually, controllers are already implemented in modern graphical application toolkits. Java, which is chosen as target language, provides two methods to respond to user input: action() and handleEvent(). Modern descriptions of the MVC are more likely to emphasis the Publisher-Subscriber or the Observer Design Patterns.

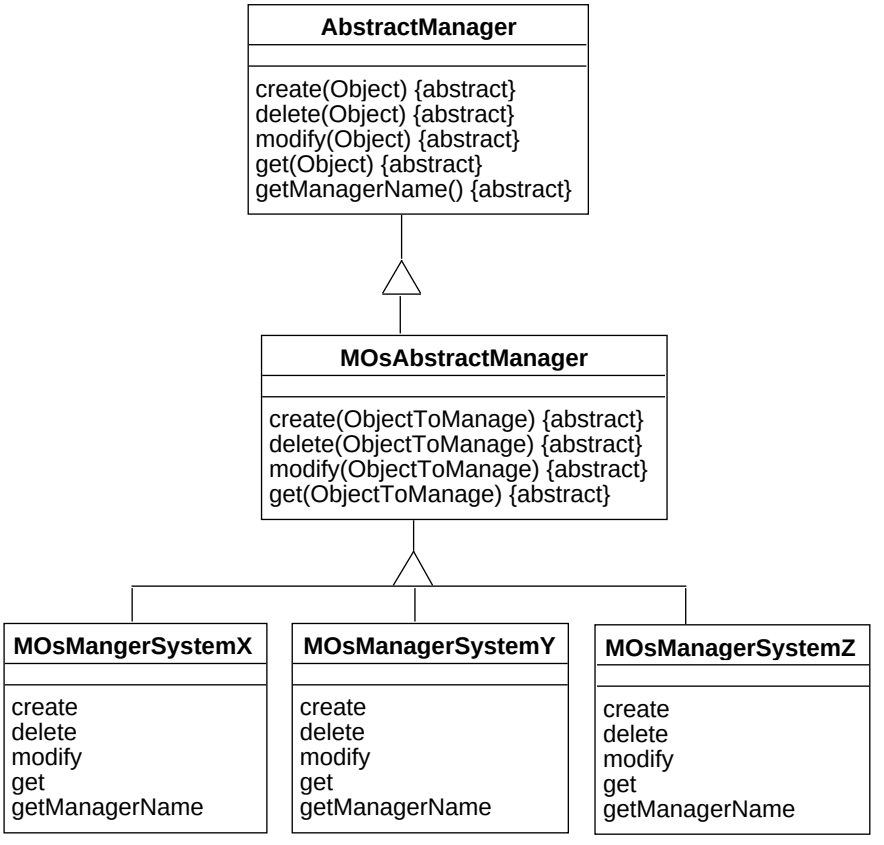
Controlling Access Rights

In this thesis project, the controller serves as an interface to the Service Access Manager layer. Depending on the implementation of a controller, the user has more or less access rights. For example, one controller may read and modify data only while an other may add the capability to create new data objects.

Using controllers allows one, for example, to change during runtime from novice mode to expert mode or vice versa.

7.4 Design of the Service Access Manager

Every concrete Service Access Manager is derived form the AbstractManager class which defines standardized interfaces. AbstractManager class defines the overall behavior of the Service Access Management classes.



Abstract methods declared in AbstractManager are redefined in MOsAbstract-Manager by naming the concrete needed parameter types.

Figure 7-11. From Abstract Manager to Concrete Manager

7.5 Design of the Inter Layer Communication

Each layer communicates in a specific way with its neighbor layers, and only with them. The following sections explains what kind of communication between layers is established and how it is accomplished.

7.5.1 Exception Handling

Exceptions provide a clean way to check for errors without cluttering code. Exceptions also provide a mechanism to signal errors directly rather than using flags or side effects such as fields that must be checked. Exceptions make the error

conditions that a method can signal, an explicit part of the method's contract. The list of exceptions can be seen by the programmer, checked by the compiler, and preserved by extended classes that override the method[5].

Buschmann et al. give in the Layers pattern description an explicit hint as to how exceptions should be dealt with:

> Error handling can be rather expensive for layered architectures with respect to processing time and, notably, programming effort. An error can either be handled in the layer where it occurred or be passed to the next higher layer, in the latter case, the lower layer must transform the error into an error description meaningful to the higher layer. As a rule of thumb, try to handle errors at the lowest layer possible. This preserves higher layers from being swamped with many different errors and voluminous error-handling code. As a minimum, try to condense similar error types into more general errors. If you do not do this, higher layers can be confronted with error messages that apply to lower-level abstractions that the higher layer does not understand. And who hasn't seen totally cryptic error messages being popped up top the highest layer of all—the user?[6]

Therefore, exceptions are dealt with in the following way in each management toolset:

- Similar errors are condensed, which results in the fact that;
- Each layer has its own exception.
- Exceptions are treated where they occur or;
- Exceptions provide a meaningful message which can be presented to the user of the application.

Mapping of Exception Classes

The following mapping of exception classes to layers is applied as shown in Table 7-1.

Table 7-1. Mapping of Exceptions

Layer	Exception Type	Remark
Service Access Layer	ApiException	
Service Access Manager Layer	ManagemetException	
Application Layer	ModelException	Thrown by the Models
	ControllerException	Thrown by the Controllers
	ViewException	Thrown by the Views
	CommandException	Thrown by Commands

The Exception flow is in only one direction, from the lower to the upper layer.

5. [Arnold+96], p 133
6. [Buschmann+96], p 43

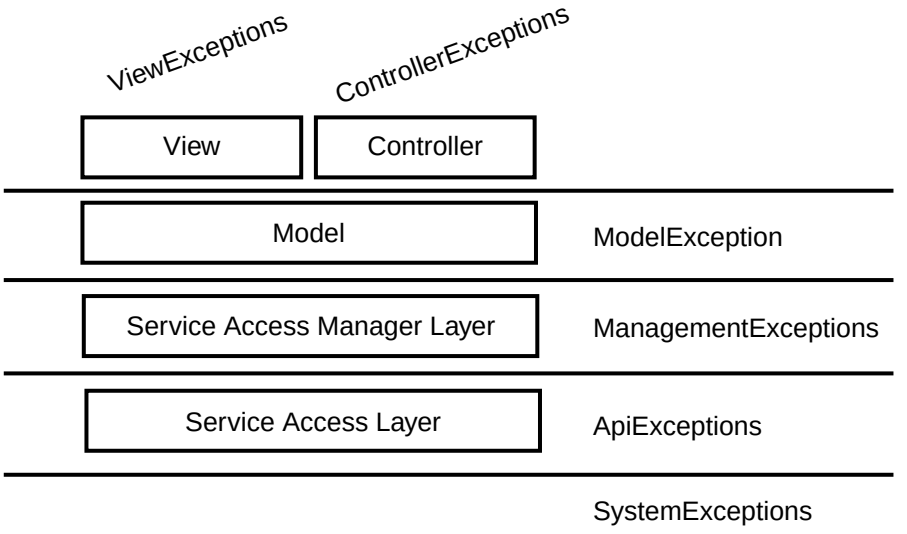


Figure 7-12. Exceptions Thrown by Layers

The inheritance hierachy of Exceptions is shown in Figure 7-13. Since Java is the target language, all exceptions are subclasses of the Java class Exception. AbstractMngmtException enhances the Exception class with a method getHint and an enhanced constructor, which can be used to pass a hint additionally to the standard message provided by Java.

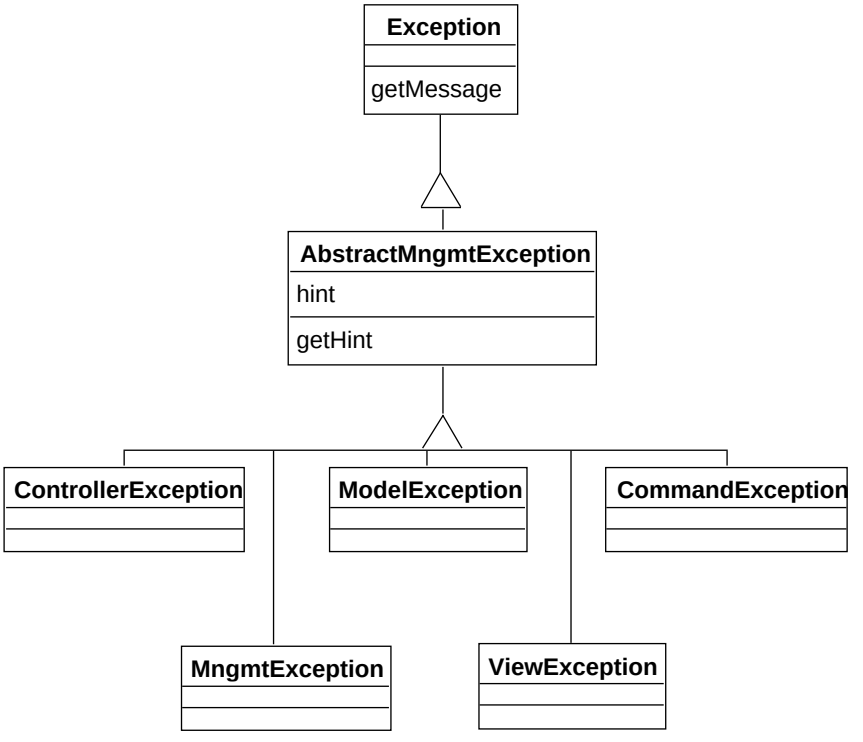


Figure 7-13. Exception Inheritance Hierachy

7.5.2 Application Layer to Service Access Manager Layer

The communication between Application Layer and Service Access Manager is managed by the Model of the Application Layer. The Model aggregates a specific Service Access Manager which it uses to get data from and set data in the system. The Model represents the business model and therefore knows about the relations between the classes which comprises the model. If the end-user request for data from the system or wants to create or delete objects, the view sends this request to the controller which decides, if the view is allowed to execute this special command. If so, the command sends the request to the model. The Model hands over the needed data to the Service Access Manager. The Service Access Manager converts the data retrieved form the Model into the special data format needed by the Service Access Layer and invokes the according command from that layer.

7.5.3 Service Access Manager Layer to Service Access Layer

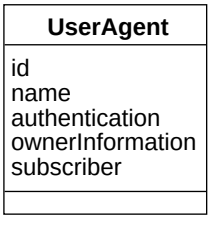
The Service Access Manager wraps the application programmer interface calls of the Service Access Layer. It converts data from the Service Access Layer into the data format expected by the Application Layer and vice versa.

8 Objects to be Managed

This chapter describes the informational structure of the managed objects, namely the User Agent, the Local Context, the Terminal Equipment Agent and the Registration Server.

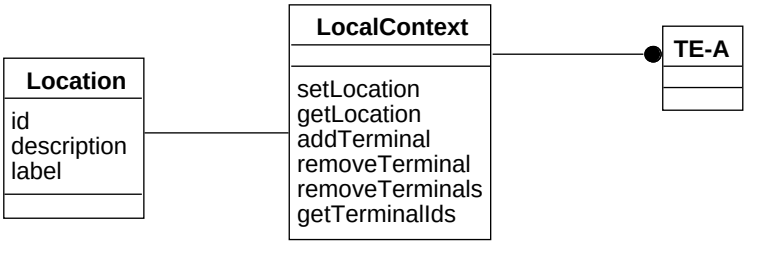
8.1 User Agent

The User Agent represents a user in the provider domain and contains user specific information. The visible parts of this object are depicted in the OMT diagram below.



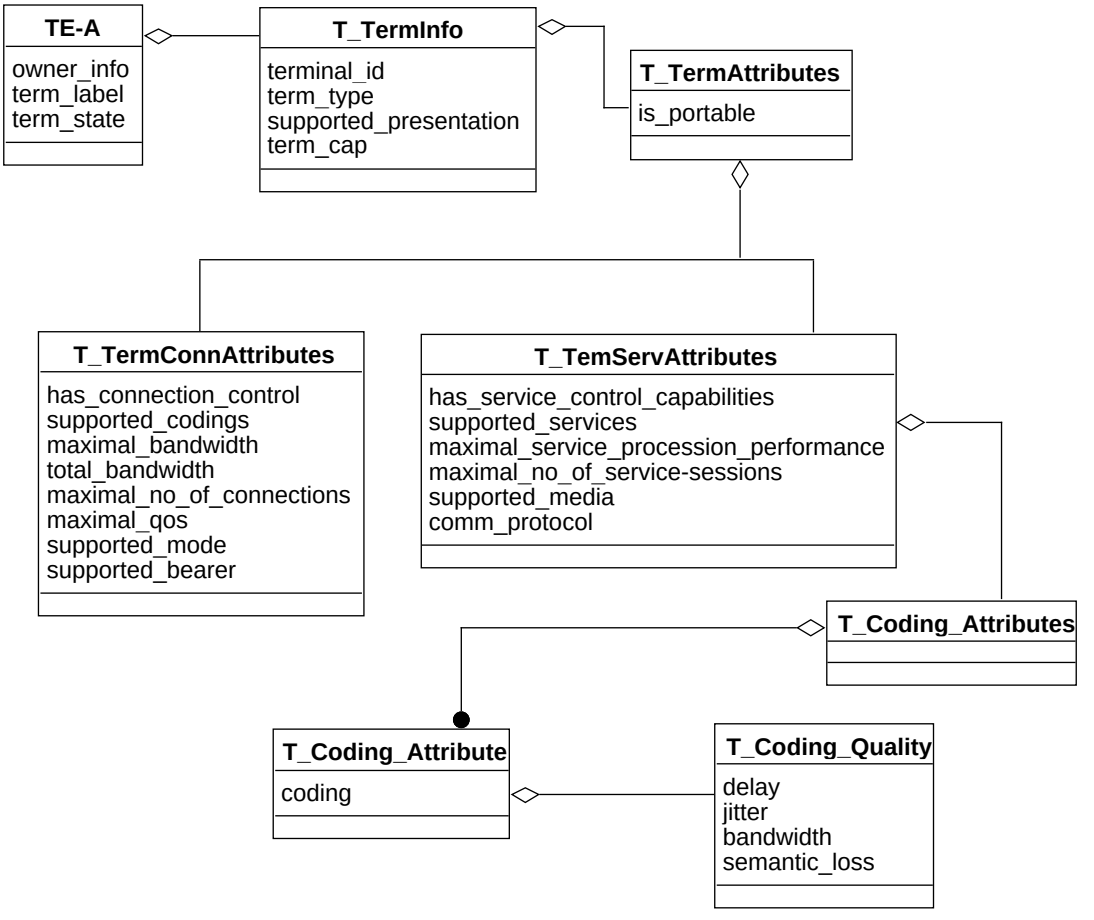
8.2 Local Context

The Local Context describes the terminal equipment at a specific location. The Local Context consists of an object representing the location and a list of terminals.



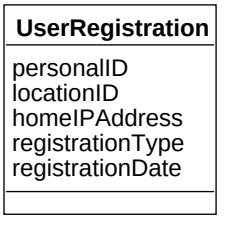
8.3 Terminal Equipment Agent

The Terminal Equipment Agent (TE-A) is the most complicated object to be managed in the scope of this work. The TE-A IDL definition is comprised of several nested structures which had to be mapped to classes.



8.4 Registration Server

The Registration Server allows the user to manage registrations at locations as well as to register a user manually at a location. The informational object User-Registration contains the relevant attributes of a user registration.



9 Package Concepts and Design

This chapter gives an introduction to the package concept. I will describe the meaning and purpose of the package concept, and how it is applied to this project. The chapter ends with a listing and explanation of all the packages in the management toolset.

9.1 What Are Packages for?

In his article about packaging a system[1], James Rumbaugh gives an introduction into the packaging concept:

> The most important high-level decision about the development of a system is how the model is divided into parts. Any complex system must be constructed in parts, so that teams of developers can work productively in parallel. This requires that the work units be identified and isolated to some extent, so that different work teams do not interfere with each other. Furthermore, it is difficult or impossible to understand a large monolithic system. Engineering practice in all disciplines has shown that systems can best be understood as collections of loosely-coupled subsystems that are connected using well-specified interfaces. […]
>
> A package has no inherent semantic meaning. It is just a subdivisions of the model. It is *desirable* that packages follow natural semantic boundaries, but this is just a design goal, not the inherent meaning of the concept.
>
> What is a package then? It is a management unit for models, a unit for organizing, controlling, and managing the model. It is first and foremost a meaningful work unit, something that represents a group of closely related modelling elements that must be managed together. It is an access control, unit for controlling updates to the model. It is a configuration control unit for saving and versioning parts of the model. […]
>
> A package is also a name space for naming modelling elements, such as classes. This may seem odd at first, because I said, that packages have no semantics. This is correct; name spaces are fundamentally software engineering constructs for team development and not semantic entities. Why do we need name spaces at all? Because teams working in parallel can accidentally choose the same name for different things. If there is a flat name space for everything, then all developers must synchronize their choices of names. While this might have some advantages, it introduces an central bottleneck that is incompatible with parallel development. If we avoid the central bottleneck, then developers can conflict on the use of names. The solutions is a recursively-nested tree of name spaces with unique

1. [Rumbaugh96a]

> names within a given name space and path names to identify the name space within the tree. any modelling element can be uniquely identified by its path name and local name within its name space. This is an old solution long used in programming languages. A package must be a name space, because it is a unit of independent work and therefore must isolate itself from naming conflicts.

9.2 Guidelines for Naming Packages

In their book “The Java Programming Language” [Arnold+96], the authors give a description of how package names should be chosen. This package naming convention is becoming more and more common in commercial software packages written in, and for Java.

> A package name should be unique for classes and interfaces in the package, so choosing a name that’s both meaningful and unique is an important aspect of package design. But with programmers all around the globe developing Java language packages, there is no way to find out who is using what package names. Choosing unique package names is therefore a problem. If you are certain a package will be used only inside your organization, you choose a name using an internal arbiter to ensure no two projects pick clashing names.
>
> But in the world at large this is insufficient. Java package identifiers are simple names. A good way to ensure unique package names is to use an Internet domain name.[2]

The authors suggest the use of a reversed domain name and to spell the highest-level domain name in capital letters, e.g. DE.gmd.fokus.tools. They suggest choosing the capital letters to “ […] prevent conflicts with package names chosen by those not following the convention, who are unlikely to use all upper-case, but might name a package the same as one of the many high level domain names.”

However, the use of the highest-level names seems to be more common. For example, Visigenic uses the name, com.visigenic.vbroker…, OMG in the Visigenic software package uses, org.omg.CORBA…, Roguewave uses com.roguewave… just to name a few.

Naming Convention

The packages with in this project start with de.gmd.fokus.ice.pcs. Our domain name is fokus.gmd.de. The work unit name is ICE (Intelligent Communication Environments). The management toolset supports Personal Communication Support (PCS) components, therefore, the umbrella name is de.gmd.fokus.ice.pcs.mngmt…

2. [Arnold+96], pp 186

9.3 Packages of the Management Toolset

The management toolset package is the topmost package of the management toolset. It contains a set of nested packages supplying the different needs of the toolset. The package name for the management toolset is de.gmd.fokus.ice.pcs.mngt. Besides the nested packages, the management package contains abstract classes which define a standardized behavior, exception classes used by other packages, and interface definitions.

Grey Shaded Package Symbols

Figure 9-1 shows the packages which are nested in the management package (mngmt). Grey shaded packages contain packages themselves, which will be displayed separately.

The package for the graphical user interface contains classes for the purpose of displaying data. Spoken in the language of the design pattern Model-View-Controller, those classes represent the view. The package for the applications contains the classes with which to start a management applications or applet. The package for the application programmer interfaces contains packages for accessing different platforms.

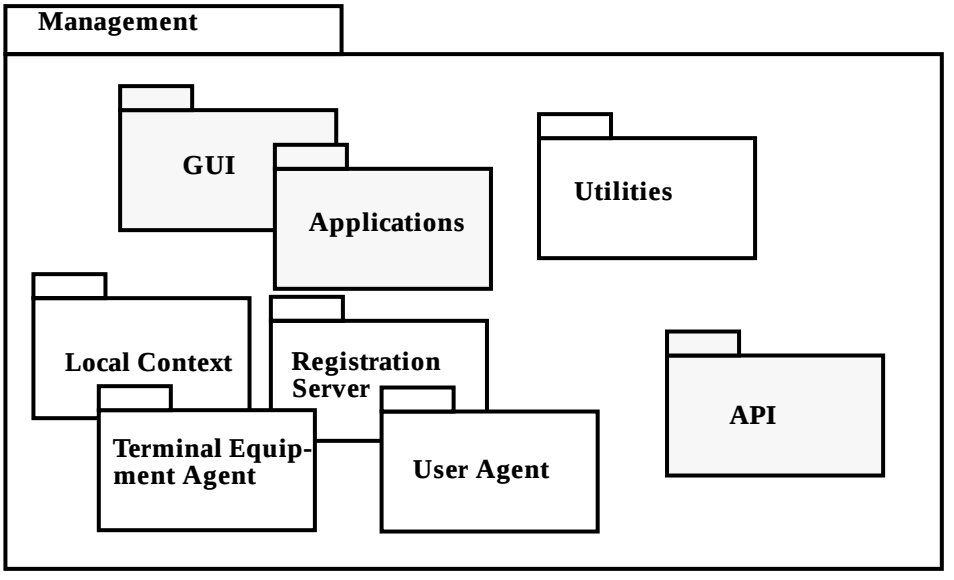
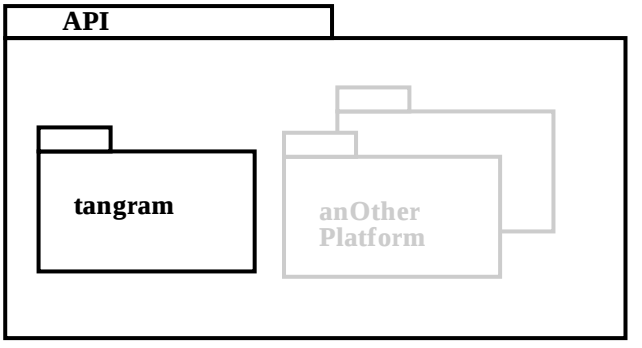


Figure 9-1. The Management Toolset Package

9.4 Application Programming Interface Related Packages

The application programming interface package contains a set of packages which provide access to different platforms. Additional packages providing access to other platforms should be placed inside of the API package. Figure 9-2 depicts the application programming interface package and its sub-packages. Packages containing other packages are shaded grey.



The grey shaded packages stands for possible additional packages which could be implemented at a later time.

Figure 9-2. The Application Programming Interface Package

9.5 Package Tangram

The tangram packages contain classes to access the TANGRAM platform.

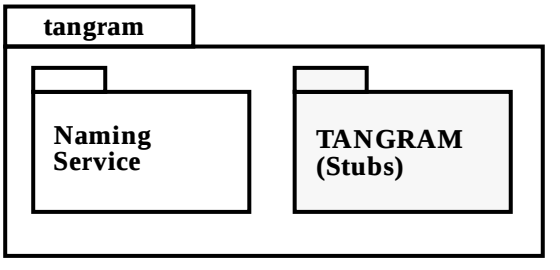


Figure 9-3. The Tangram Package

9.6 Graphical User Interface Related Packages

The graphical user interface package contains a set of packages which provide the different views used inside of applications for the display of management relevant data.

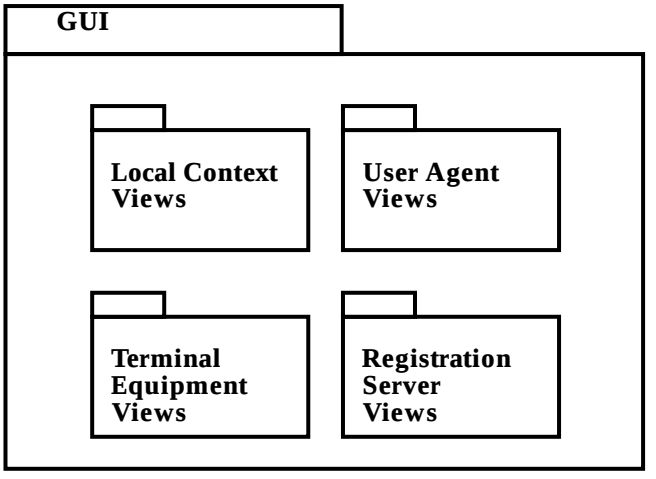


Figure 9-4. The Graphical User Interface Package

The responsibility of each package is to hold the data that is to be displayed. Each package includes a set of classes which define specific groups of data to be displayed. These groups are held together by groupboxes as a ‘visual container’. Each groupbox builds a component which can be reused to build new applications.

9.7 Packages for Accessing the TANGRAM Platform

Stubs

The packages containing the stubs for access to the TANGRAM platform are placed inside the API package de / gmd / fokus / ice / pcs / mngmt / api. This is done for reasons of convenience only, and is not due to any restrictions on the TANGRAM project.

The complete package is created automatically by the idl2java compiler.

Package TANGRAM

The package TANGRAM contains all modules concerning the TANGRAM platform. The PCS in TINA supporting components are also nested inside this package.

Figure 9-5 depicts the TANGRAM package and its subpackages of the first level. Packages inside the TANGRAM package which also contain packages are shaded grey.

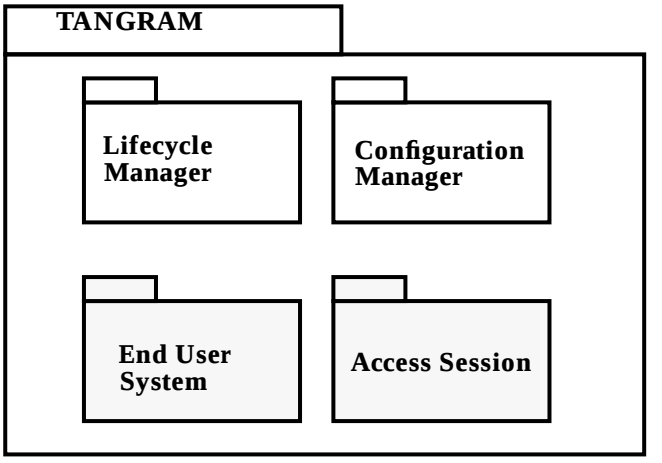


Figure 9-5. The TANGRAM Package

Package Life Cycle Manager

The package Life Cycle Manager contains all classes for lifecycle management of the objects inside the TANGRAM platform. The Life Cycle Manager represents a template for factory functionalities needed to control and enable the creation, deletion and initialization of objects.[3]

Package Configuration Manager

The package Configuration Manager's task is to manage a group of objects which are associated in a specific manner. Its responsibilities are to request, create, and terminate the interfaces of the Computational Objects belonging to that Configuration Manager[4]. These capabilities make possible the achievement of ODP location transparency.

Package End User System

The package *End User System* contains classes which enable communication with the TINA GSEP.

Package Access Session

The package *Access Session* contains classes which support the TINA Access Session.

9.8 Package for Accessing the Naming Service

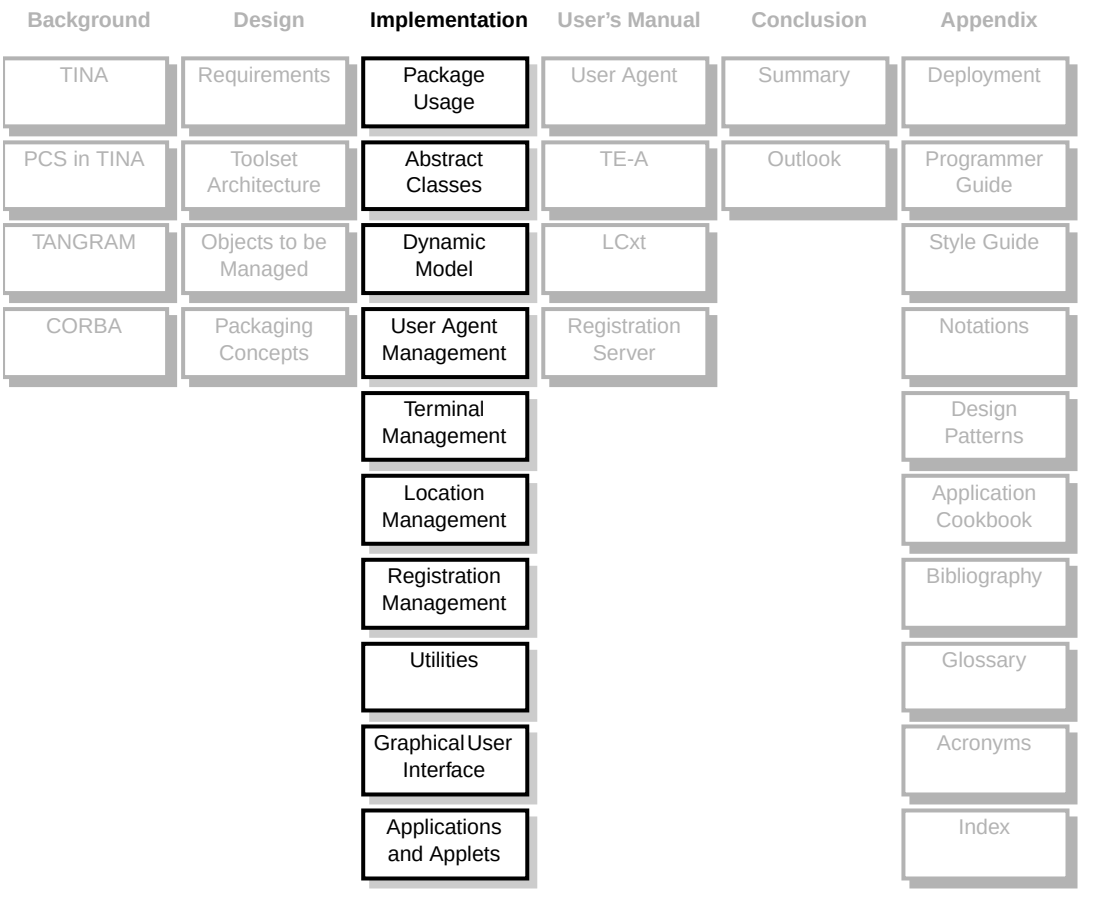
The package *Naming Service* provides the classes which support the naming service functionalities for the accessing of TANGRAM Modules and Interfaces

3. see [Eckardt+96a]
4. loc.cit.

Part 3

Implementation

This part describes the implementation of the management toolset. It starts with descriptions of the organizational aspects of the implementation, and the package and their mapping to the file structure. This is followed by explanations of the abstract classes and interfaces, followed by a chapter which introduces the dynamic model and explains how the computational objects interact. The remaining chapters list and describe all classes—thereby supporting comprehension of the implementation and giving guidance for finding needed components.


Background
Design
Implementation
User's Manual
Conclusion
Appendix
TINA
Requirements
Package Usage
User Agent
Summary
Deployment
PCS in TINA
Toolset Architecture
Abstract Classes
TE-A
Outlook
Programmer Guide
TANGRAM
Objects to be Managed
Dynamic Model
LCxt
Style Guide
CORBA
Packaging Concepts
User Agent Management
Registration Server
Notations
Terminal Management
Design Patterns
Location Management
Application Cookbook
Registration Management
Bibliography
Utilities
Glossary
Graphical User Interface
Acronyms
Applications and Applets
Index

10 Package Usage

Packages serve for better dividing of a model into parts. They are units for managing, organizing, and controlling a model. Packages are access control and support the concept of unique naming spaces. This chapter gives an overview over the mapping of the designed packages into the naming spaces of this work.

10.1 Package Tree for the Toolset

The management toolset is organized in a tree structure which is shown in Figure 10-1.

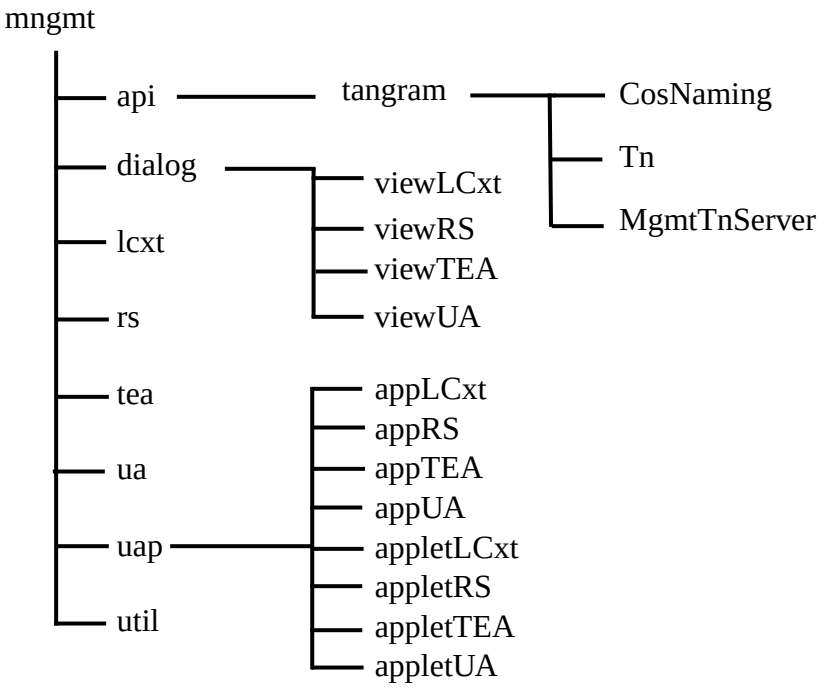


Figure 10-1. The Package Tree of the Management Toolset

10.2 Package ‘Management’

The management package’s name is mngmt. This package is the container package for the whole management toolset.

Table 10-1. Packages of the Management Package

Package Name	Description
de.gmd.fokus.ice.pcs.mngmt	Basic classes. Container Package for all other packages.
de.gmd.fokus.ice.pcs.mngmt.util	Utilities, used in almost all classes
de.gmd.fokus.ice.pcs.mngmt.uap	Applications and applets. Contains other packages.
de.gmd.fokus.ice.pcs.mngmt.ua	User Agent
de.gmd.fokus.ice.pcs.msngmt.tea	Terminal Equipment Agent
de.gmd.fokus.ice.pcs.mngmt.lcxt	Local Context
de.gmd.fokus.ice.pcs.mngmt.rs	Registration Server
de.gmd.fokus.ice.pcs.mngmt.api	Application Programming Interface. Contains other packages.
de.gmd.fokus.ice.pcs.mngmt.dialog	Views, Graphical User Interface dependent packages. Contains other packages

10.3 Package ‘Dialogs’

The Dialogs package includes all classes and packages related to graphical user interfaces. Each graphical user interface for a special managed object resides in its own packages.

Table 10-2. Graphical User Interface Dependent Packages

Package Name[a]	Mapping	Description
….dialog	GUI	Common views, used in several applications
….dialog.viewLCxt	Local Context Views	Views, only used in application for the management of the Local Context
….dialog.viewRS	Registration Server Views	Views, only used in application for the management of Registration Server
….dialog.viewTEA	Terminal Equipment Views	Views, only used in application for the management of the Terminal Equipment Agent.
….dialog.viewUA	Registration Server Views	Views, only used in application for the management of the User Agent

a. “…” stands for de.gmd.fokus.ice.pcs.mngm

10.4 Package ‘Tangram’

The tangram package’s name is Tn. It includes all classes and packages created by the idl2java compiler. The idl2java compiler transforms modules found in the IDL file into packages. IDL structures are converted into classes as well as enumeration types.

Table 10-3. Packages Created from the Tn IDL[a]

Package name[b]	Mapping	Description
….api.Tn	TANGRAM	TANGRAM root
….api.Tn._I_ConfigurationManager	Configuration Manager	Configuration Manager
….api.Tn._I_LifeCycleManager	Lifecycle Manager	Life Cycle Manager
….api.Tn._I_CoControl		Classes to support CO control
….api.Tn.Eus	End User System	End User System, i.e. Generic Session Endpoint (GSEP)
….api.Tn.Eus._I_GsepUApp		Generic Session Endpoint User Application
….api.Tn.Ass	Access Session	Access Session
….api.Tn.Ass.Pcs		PCS enhanced components
….api.Tn.Ass._I_UCxt		Interfaces for the Usage Context
….api.Tn.Ass._I_PcsRSUserLocation		Classes to support the Interface for the User Location components
….api.Tn.Ass._I_PcsRSUsage		Classes to support the Registration Server usage interface
….api.Tn.Ass._I_PcsRSMgmt		Classes to support the Registration Server management interface
….api.Tn.Ass._I_SessionDescription		Classes to support the interface to the Service Description.
….api.Tn.Ass._I_UaSession		Classes to support the User Agent Session
….api.Tn.Ass._I_UaInvitation		Classes to support User Agent Invitation

a. Grey shaded packages are not used in the management toolset.
b. “…” stands for de.gmd.fokus.ice.pcs.mngmt

10.5 Package ‘Naming Context’

The Naming Context package’s name is CosNaming. It includes several packages that are only of internal importance.

11 Abstract Classes, Interfaces and Exceptions

This chapter describes classes, interfaces and exceptions defined in the package mngmt. These classes define a common mode of behavior for the remaining classes of the management toolset.

11.1 Classes

In Table 11-1 the abstract classes of the package mngmt are listed. Each class is an abstract class and describes common signatures for derived classes.

Table 11-1. Abstract Classes Defined in Package mngmt

Class Name	Mapping	Description
AbstractFactory	Factory Method	Describes the signatures for all factory classes of the management toolset.
AbstractManager	Management Layer	Describes the signatures for all service access manager classes of the management toolset.
AbstractController	Model-View-Controller	Describes common signatures of all controller classes.
Command	Command Processor	Describes common signatures of all command classes.

11.2 Interfaces

In Table 11-2, interfaces of the package mngmt are listed. So far, there is only one interface definition.

Table 11-2. Interfaces Defined in Package mngmt

Interface Name	Mapping	Description
View	Model-View-Controller	Defines common signatures for all Views.

11.3 Exceptions

Exceptions declared in the package *mngmt* are thrown by different layers and therefore used by all related classes. For more information about Exception handling in the management toolset see Section 7.5 on page 53.

The behavior of all Exception classes is the same. Each class is derived from the AbstractMngmtException class, which is derived from the Java Exception class. All Exception classes consist of an additional method getHint() which allows the deliverance of more detailed information.

Table 11-3. Exceptions Defined in Package mngmt

Class Name	Thrown by Layer	Description
AbstractMngmtException	Never thrown.	Abstract Exception class which defines an extension of the Java Exception class.
MngmtException	Service Access Manager Layer	Exceptions thrown by the Service Access Manager layer.
CommandException	Application Layer	Exceptions thrown by command classes
ControllerException	Application Layer	Exceptions thrown by controller classes.
ViewException	Application Layer	Exceptions thrown by view classes.
ModelException	Application Layer	Exceptions thrown by model classes.

12 Dynamic Model

This chapter describes the dynamic aspects of the management toolset which are applied to all implementations of the management applications.

12.1 Application Layer

The application layer contains the application relevant computational objects. This section gives an overview of interaction between computational objects with in the application layer.

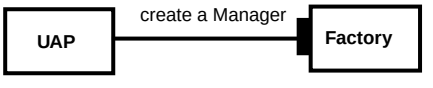
Initialization Phase

The initialization of a management application is normally completed in a few steps:

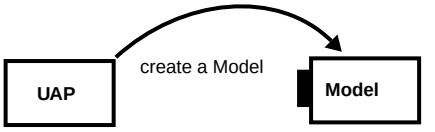
1. Create a factory.
2. Query factory for a specific Service Access Manager.
3. Create the model.
4. Create the views.
5. Create the command processor.
6. Create the controllers.
7. Pass the controller to a view component.
8. Create mediators, if needed.
9. Create visual components (menu, statusbar, buttons).

Steps one to seven are typical for the initialization phase of all management (user) applications. For convenience, the management application computational object is abbreviated in the following graphics as UAP. Note: In the actual implementation of the management toolset only the TE-A management application uses a mediator computational object to administer its views.

The first action for an application is to create a Service Access Manager Factory which can then be queried for the desired Service Access Manager. The model needs the Service Access Manager in order to manage the computational objects.

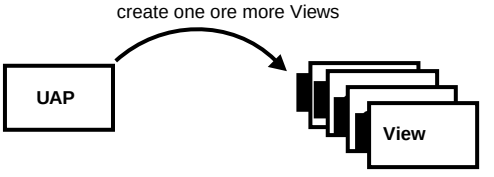


The createManager method of a factory returns a specific Service Access Manager.



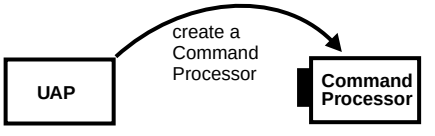
During creation of a model, the UAP passes the Service Access Manager to the model.

After the model is created, the next step is to create the views needed. A view needs to know the model it has to communicate with. Therefore, a model is passed as a parameter to a view during creation.



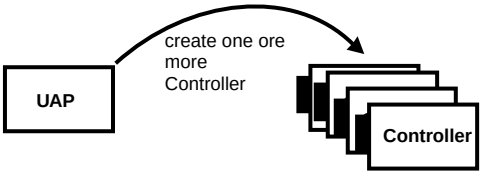
A user application creates as many view objects as are needed.

With the goal of creating a controller, the next step is to create a CommandProcessor object. The CommandProcessor object will be passed to it during creation of a controller.



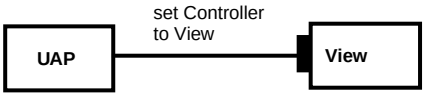
During creation of a model, the UAP passes the Service Access Manager to the model.

After the model, the views and the command processor are created, the UAP creates controller objects. Each controller objects gets it's associated view and the command processor as parameters during its creation.



A user application creates as many controller objects as are needed to pass to the views.

After the needed controllers are created, they must be set in the associated view.



If needed, the UAP might create a mediator object which is responsible for managing a statusbar object to display short messages at the bottom of the application window, a menu object, and all necessary buttons and graphic objects.

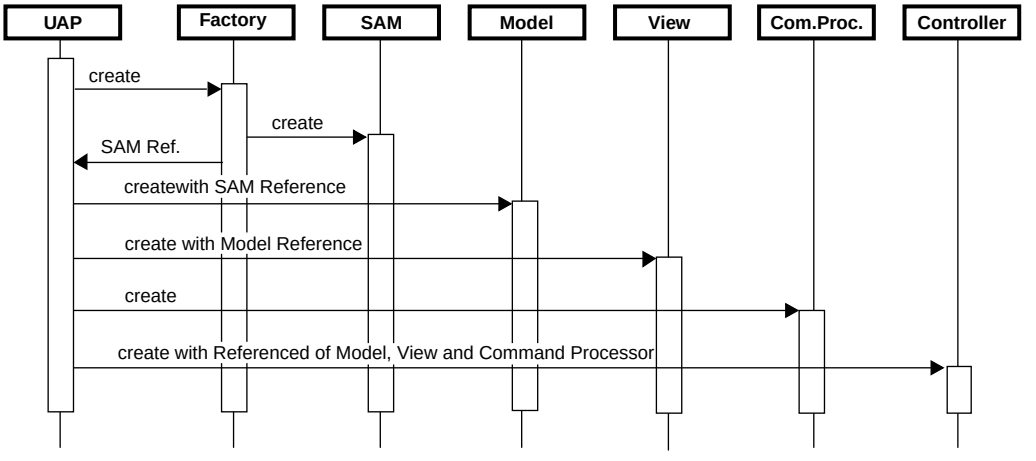


Figure 12-1. Initialization Phase of a Management Application

12.2 Platform Access

To access a specific platform, the model uses the Service Access Manager (SAM) for the modification of data. For the model, and therefore also for the application layer, the access to a platform is transparent. In fact, the model never knows which platform it is actually dealing with.

The following example exemplifies the activities between the different computational objects while they are modifying data. First, a user queries to save modified data. The action of modification implies the act of saving the data in the system to retrieve a persistent state of the data.

The controller receives the query for saving. In case the controller is allowed to permit modification, it builds a modification command and passes to it the changed data. It then passes the interface of the modification command to the command processor. The command processor executes the modification command and stores a reference to the command for further undo actions. The command CO contacts the interface of the model and invokes the modification command of the model. The model sends the request with the changed data to the Service Access Manager (SAM). The SAM converts the data into the requested

format of the Service Access Layer. After converting the data, the SAM invokes all necessary command of the API which now will send the request for modification to the platform system.

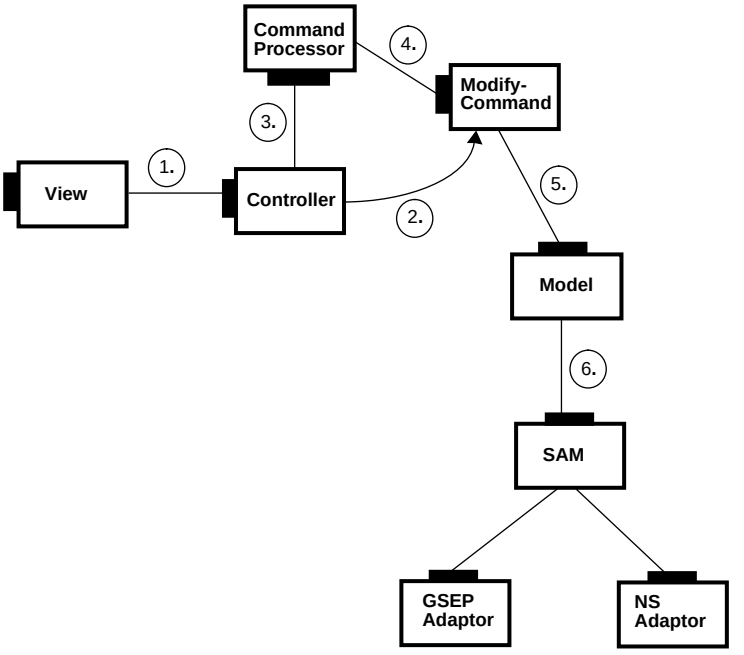


Figure 12-2. Modification of Data

1 The View asks the Controller to save modified data.

2 The controller creates a ModifyCommand.

3 The controller passes the ModifyCommand to the CommandProcessor.

4 The CommandProcessor invokes the ModifyCommand to modify (save) the data.

5 The ModifyCommand contacts the Model interface to invoke the Model's modify command.

6 The Model contacts the Service Access Manager interface for modification.

13 User Agent Management

This chapter lists all classes used to build the management tools for User Agent management. The classes can be found in the package de.gmd.fokus.ice.pcs.mngmt.ua. For more detailed information please refer to the online documentation[1].

The following description of a User Agent computational object (Figure 13-1) is taken from the file UA.ODL which is part of the implementation of the TANGRAM platform.

```
UA.ODL

behavior
"The User Agent (UA) computational object rep-
resents a user in the service provider domain
of a TINA system.

It supports the user by accessing TINA ser-
vices.

The UA is involved, when a user wants to start
a service session, join in an existing service
session, register at terminals, maintain the
users preferences on service execution and
when a user is invited to a service session.

The UA is supported by some User Agent Sup-
porting objects, such as Usage Context, Per-
sonal Profile, Authentication and Session
Description.

The UA controls the life-cycle of service ses-
sions.";
```

Figure 13-1. ODL Extract of UA.ODL

Table 13-1. Classes in Package 'ua'

Name	Mapping to Design Pattern	Description
AbstractUserManagement		Redefines Methods of Abstract Manager concerning the special needs of the User Agent computational object.

1. See "Source Code Documentation" on page 134.

Table 13-1. Classes in Package ‘ua’

Name	Mapping to Design Pattern	Description
CommandCopy	Command Processor	Command object for copy operations.
CommandCreate	Command Processor	Command object for create operations.
CommandDelete	Command Processor	Command object for delete operations.
CommandModify	Command Processor	Command object for modification operations.
EnumUserNames		Returns an Enumeration Object of all users found in the system. Note: an enumeration object is always a snapshot.
ORACLE_UserManager	Layers / Facade	Manager for ORACLE platform access.
TANGRAM_UserManager	Layers / Facade	Manager for TANGRAM platform access.
Ua		Superclass for all classes of this package
User		Represents a single user’s data.
UserController	Model-View-**Controller**	Controller for end-user interactions on the user data.
UserDataController	Model-View-**Controller**	Extends UserController.
UserException		Intern exception class, thrown by class User.
UserList		List to store objects of type User.
UserListController	Model-View-**Controller**	Controller for managing interactions of end-users on a list of user objects in a list of type UserList.
UserManagerFactory	Factory Method	Factory for creating a specific Service Access Manager computational object.
UserModel	**Model**-View-Controller / Observable	Functional core of the application.
UserModelMessage	Message	Message object sent by the model to its observers while invoking the update method.
UserView	Model-View-Controller	Interface for defining signatures which a view for user data has to provide

14 Terminal Management

The management of terminals in a TINA-C system is the management of Terminal Equipment Agents. TINA-C describes a Terminal Equipment Agent in the Service Architecture 2.0 as follows:

> A Terminal Equipment Agent (TE-A) is defined to model a user system as a computational object within the provider domain. It maintains minimum information of resource configuration (e.g., access points, UAPs, stream interfaces, and GSEPs) of a user system. A TE-A is one of the key computational objects for mobility, since it keeps tracks of association between a terminal and access points in the provider domain. Details of how the TE-A can cooperate with computational objects for connection management (e.g. Communication Session Manager) to support terminal mobility are for further study.[1]

An other description of the TE-A can be found in the behavior section of the TANGRAM ODL file TEA.ODL and is cited in Figure 14-1.

```
TEA.ODL

behavior
“The Terminal Equipment Agent (TE-A) object
belongs to the set of Infrastructure Capabil-
ity Describing Objects.

It represents terminal equipment in a TINA
system.

Various characteristics of an individual ter-
minal are stored in a TE-A.

Basically, a subscriber or a user requests
creation and deletion of a TE-A.

Since an actual terminal resides in the user
domain, the object which resides in the pro-
vider domain doesn’t execute terminal func-
tions per se. Instead, a TE-A can provide
sufficient
information from a specific terminal to exe-
cute a service.”;
```

Figure 14-1. ODL Extract of TEA.ODL

The package *tea* contains the classes which model the nested structure of the PCS enhanced Terminal Equipment Agent (TE-A). Since Java, as an object oriented language, does not provide structures, every structure of the TE-A struc-

1. [Berndt+95], p 6-7.

ture had to be mapped to classes. The package also contains classes supporting the used Model-View-Controller pattern as well as various other application supporting classes which represent and mange the TE-A.

An overview of the classes of the package tea and their mapping to the TANGRAM TE-A is given in the following Table 14-1. Also, for a better understanding of the nested structure of the Terminal Equipment Agent, the mapping to the corresponding TANGRAM (Tn) Type is given.

More detailed information on each class can be found in the online documentation[2].

Table 14-1. Classes in Package ‘tea ‘

Classes	Mapping to TANGRAM Data Type	Mapping to Design Patterns	Description
CodingAttribute	T_Coding_Attribute		Models a Coding Attribute which contains a list of Coding Qualities.
CodingAttributes	T_Coding		A list of Coding Qualities.
CodingAttributesController		Model-View-**Controller**	Controller for Modification of Coding Qualities.
CodingQuality	T_CodingQuality		Class for representing the attributes of one Coding Quality entity.
EnumCommunicationProtocols	T_Comm_protocol		Enumeration for predefined protocol types.
EnumPresentationSupport	T_PresentationSupport		Enumeration
EnumSupportedBearer	T_Bearer		Enumeration
EnumSupportedCodings	string		Enumeration
EnumSupportedMedia	T_Media		Enumeration
EnumSupportedMode	T_Mode		Enumeration
EnumSupportedServices	T_ServiceIdList		Enumeration
EnumTerminalNames			Enumeration for listing available terminals.
EnumTerminalType	T_TermType		Enumeration for listing available terminal types.
ServiceIdList	T_ServiceIdList		List of available Services
TEAAbstractManager			Abstract Class which defines signatures of Service Access Managers for Terminals.
TEAManagerFactory		Factory Method	Factory for creating a special Service Access Manager
TEAManagerTANGRAM			Service Access Manager for TANGRAM platform.
TeA			Superclass for all tea objects of this package.

2. See “Source Code Documentation” on page 134.

Table 14-1. Classes in Package ‘tea (Continued)‘

Classes	Mapping to TANGRAM Data Type	Mapping to Design Patterns	Description
TeaModel		**Model**-View-Controller	
TeaModelMessage		Message	Message class for passing while invoking update of observers
TeaProducer		Shopper	Converter
TeaProducerContainer		Shopper	Converter
TeapConstants		Shopper	Converter
TermAttributes	T_TermAttributes		
TermConnAttributes	T_TermConnAttributes		
TermInfo	T_TermInfo		
TermServAttributes	T_TermServAttributes		
TermState	T_TermState		
Terminal	T_Terminal		Computational Object TE-A
TerminalController		Model-View-**Controller**	Controller for modification of a terminal object. Used in views.

15 Location Management

This chapter lists the classes needed to build the management tool for location management. For more detailed information, please refer to the provided online documentation. All classes listed here can be found in the package de.gmd.fokus.ice.pcs.mngmt.lcxt.

More detailed information on each class can be found in the online documentation[1].

Table 15-1. Classes in Package 'lcxt'

Class	Mapping to Design Patterns	Description
AbstractLCxtManager		Redefines Methods of Abstract Manager concerning the special needs of the Local Context computational object.
CommandCopy	Command Processor	Command object for copy operations.
CommandCreate	Command Processor	Command object for create operations.
CommandDelete	Command Processor	Command object for delete operations.
CommandModify	Command Processor	Command object for modification operations.
LCxt		Represents a location with its associated terminals.
LCxtController	Model-View-**Controller**	Controller to control access to Local Context data.
LCxtDataController	Model-View-**Controller**	Controller to control access to Local Context data. Extends LCxtController.
LCxtListController	Model-View-**Controller**	Controller for list containing locations.
LCxtModel	**Model**-View-Controller	Functional core of the application.
LCxtModelMessage	Message	Message object sent by the model to its observers while invoking the update method.
LCxtSamFactory	Factory Method	Factory for creating a specific Service Access Manager computational object.
LCxtSamTANGRAM		Service Access Manager to the TANGRAM platform.

1. See "Source Code Documentation" on page 134.

Table 15-1. Classes in Package ‘lcxt’ (Continued)

Class	Mapping to Design Patterns	Description
LCxtView	Model-View-Controller	Interface for defining signatures which a view on Local Context Data has to provide.
ListOfTerminals		
Location		Representation of a location.

16 Registration Management

The following table presents the classes used to build the management tool for the management of registrations. This tool allows registrations to be managed. It can also be used for manual registrations. The classes can be found in the package de.gmd.fokus.ice.pcs.mngmt.rs.

More detailed information on each class can be found in the online documentation[1].

Table 16-1. Classes in package ‘rs’

Class	Mapping to Design Patterns	Description
AbstractRegServerManager		Redefines Methods of Abstract Manager concerning the special needs of the Registration Server Manager computational object.
RegServerModel	Model-View-Controller	Functional core of the application.
RegServerModelMessage	Message	Message object sent by the model to its observers while invoking the update method.
RegServerSamFactory	Factory Method	Factory for creating a specific Service Access Manager computational object.
RegServerSamTANGRAM	Layers / Facade	Service Access Manager to the TANGRAM platform.
UserRegistration		Representation of a user registration.

1. See “Source Code Documentation” on page 134.

17 Utilities

This chapter lists the classes which are located in the package ‘util’. The utility classes are helper classes which are not specialized to serve only the management toolset. This package provides for example classes to debug any kind of software written in Java.

In addition to debugging classes the package provides classes for programming by contract. Programming by contract means, that a routine assumes certain preconditions and guarantees certain preconditions and postconditions[1]. During implementation, those methods help to check for those conditions and to detect programming flaws.

More detailed information on each class can be found in the online documentation[2].

17.1 Classes

Table 17-1 lists all classes of the package util. For more detailed information please refer to the online documentation.

Table 17-1. Classes in Package ‘util’

Name	Description
Debugger	Abstract class. Provides signatures for concrete debugging classes like StandardDebugger or ToFileDebugger.
Ensure	Provides a set of static methods for programming by contract.
Environment	Provides a set of static methods concerning environment information.
Pictures	Provides a set of static methods concerning the retrieval of pictures.
Require	Provides a set of static methods for programming by contract.
SilentDebugger	This debugger is used when no debugging output at all is desired.
StandardDebugger	Prints all debugging messages to the standard output.
ToFileDebugger	Prints all debugging messages into a given file.

17.2 Interfaces

So far there is only one interface definition in the util package.

1. [Meyer94]
2. See “Source Code Documentation” on page 134.

Table 17-2. Interfaces in Package ‘util’

Interfaces	Description
Debuggable	Provides a signature to set the debugger.

18 Graphical User Interfaces

This chapter presents the implemented classes of the different views. Beginning with a table of commonly used classes for displaying data, there follows sections listing and explaining the classes used to implement the User Agent, the Local Context, the Terminal Equipment Agent and the Registration Server Management tool.

More detailed information on each class can be found in the online documentation[1].

18.1 Shared Views and Dialogs

The classes listed in Table 18-1 are classes that are used by different components of the management toolset.

Table 18-1. Classes in Package ‘dialog’

Class Names	Description
Box	Utility for representing the dimension of a rectangle.
DlgAbout	Dialog window for presenting the mission statement of an application (cf. Figure 20-4 on page 102).
DlgListOfLocations	Dialog window for presenting a list of locations. Allows for selection and management of elements in the list.
DlgListOfUser	Dialog window which presents a list of users. Same functionality as DlgListOfLocations.
DlgListTerminalLabels	Dialog containing list of terminals
DlgLoggingOptions	Dialog for setting device for logging output (cf. Figure 20-3 on page 102).
DlgTellUser	Dialog to inform user about problems, errors etc.
GrpBoxAvailableUser	Groupbox to display a list of users
GrpBoxListOfLocations	Groupbox to display a list of locations
GrpBoxListOfTerminals	Groupbox to display a list of terminals
GrpBoxTerminalInfos	Common Information about a terminal, i.e. TE-A and TermInfo fields
MgmtApplet	Base class for applets
MgmtDialog	Base class for dialogs
MgmtFrame	Base class for frames
MgmtLFLGroupBox	Base class for GroupBoxes
MgmtPanel	Base class for panels

1. See “Source Code Documentation” on page 134.

Table 18-1. Classes in Package ‘dialog’

Class Names	Description
PanelUserList	Panel to hold groupbox GrpBoxAvailableUser

18.2 User Agent

The User Agent management application is a very simple one. The application provides a listbox to select a user and a groupbox to display and modify user data. The listbox to select a user is used in different applications and therefore can be found in the dialog package.

Table 18-2. Classes in Package ‘viewUA’

Class Name	Description
GrpBoxUserInfo	Container for holding all user related information. Can be used as plug-in-group for user data into other applications.

18.3 Local Context

The Local Context management application is arranged using a tabbed notebook. Two sheets are provided, one for location information and one for associating a location with terminals.

Table 18-3. Classes in Package ‘viewLCxt’

Class Name	Description
GrpBoxLCxt	Container to display all information about a location.
GrpBoxTEAsOfLCxt	Container to display list of all terminals of a given location.
PanelLCxt	Container to display a list of available terminals to add to a chosen Local Context.
PanelLocations	Container to display a list of Locations and their attributes.

18.4 Terminal Equipment Agent

The Terminal Equipment Agent management application presents a set of views and is the most extensive application concerning the used components.

Table 18-4. Classes in Package ‘viewTEA’

Class Name	Description
DlgCodingQuality	Dialog to set the coding quality.

Table 18-4. Classes in Package ‘viewTEA’

Class Name	Description
DlgSupportedCodings	Dialog to set and choose the supported codings.
GrpBoxCodingName	Groupbox to display a coding name.
GrpBoxCodingQuality	Groupbox to display coding quality parameter.
GrpBoxConnectionControl	Groupbox to display connection control parameter.
GrpBoxListOfCodingNames	Groupbox to display a list with coding names.
GrpBoxServiceControl	Groupbox to display service control parameter.
GrpBoxTerminalControl	Groupbox with checkboxes to set the attributes has_connection_control_capabilities and has_service_control_capabilities
GrpBoxTerminalName	Groupbox to display the name of the actual selected terminal
GrpBoxTerminalState	Groupbox with four Checkboxes. Displays the actual state of the terminal. i.e. busy, idle, up or down.
TerminalMediator	Mediator class for centralized component management.

18.5 Registration Server

The Registration Server management application consists of some dialog windows and three different views.

Table 18-5. Classes in Package ‘viewRS’

Class Name	Description
DlgRemoveInfo	Dialog to display hints before deleting registrations.
DlgWarning	Dialog for warnings.
GrpBoxRegisteredUsers	Groupbox to display all registered users.
GrpBoxRegistrationDeletion	Groupbox to set parameter for registration deletion.
GrpBoxWarning	Groupbox used by class DlgWarning

19 Java's Applications and Applets

This chapter discusses the differences between Java's applications and applets and what to consider while designing both of them. I will give an overview of my experiences building a management application as an applet and finally I will describe the steps needed to build CORBA-based applets.

19.1 The Usage of Applications and Applets

In Java, two kinds of executable binaries exist: applications and applets. Applications are an equivalent to stand-alone, 'old-fashioned' software. Applets are software that run inside a Java-enabled browser like Netscape's Navigator, Microsoft's Internet Explorer or Sun's HotJava.

The differences between Java applications and Java applets—at first sight—are slight; while applications are executed by using the Java interpreter *java*, applets can be run outside Java-enabled browsers using the Java interpreter *appletviewer*. Java's applications and applets also need different methods: where Java's applications need the *main* method, which serves as a starting point for the interpreter, the applets need the *init* method as starting point. An applet is a Java class which extends java.applet.Applet. A class which extends java.applet.Applet and also has a main() method is both an application and an applet.

Using different methods as starting points enables one to define different initialization methods. It is possible with in the same class to define one *main* method and one *init* method. Thus, permitting the distribution of only one 'executable' for both kinds of software. This may seem easy, but is deceptive; for the writing of an applet or an application there is more that must be taken into consideration.

First, an application has more visual elements to communicate with a user then an applet. Applets can not have menus and they also should not provide buttons to exit. You do not exit an applet like an application. You quit an applet by choosing a new page or by closing the browser.

Secondly, it should be considered that an applet will be loaded over the internet which can take far more time than just loading a local application from your host. Thus, an applet should be made as lean as possible. As few graphical elements should be used as possible, such as background pictures, to prevent extended loading time. In general, an applet should be kept as simple as possible.

Thirdly an applet can be divided into smaller parts which can be loaded when ever necessary. It is even possible to put links on the Web page which let the user retrieve those parts at will.

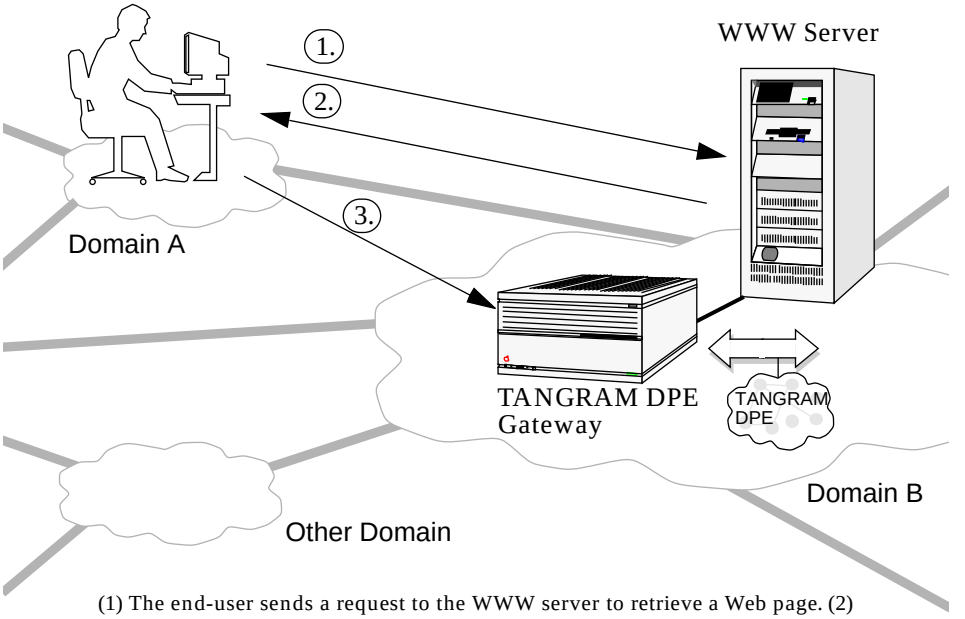
All of those reasons—and more, as are detailed in the following section—led me to the decision to distinguish between applications and applets.

19.2 The Management Toolset Used With Applications and Applets

All graphical user interfaces of the management toolset are available as applications. The toolset is built in such a way that all components check if they are running inside a Java application or a Java applet. Therefore, there is no need to make any changes in the basic components if they are to be used in applets.

The executables to start a management application or applet for each manageable computational object is located in its own package. Both kinds of executables are located inside the package de.gmd.fokus.ice.pcs.mngmt.uap.

After finishing the implementation of the management applications, the integration of the toolset in applets was started. The first intension was: add the init method, eliminate the menu and run it in a browser. But these steps alone were not adequate for using an applet together with CORBA. Several problems occurred that must first be solved before it is possible to implement further extensions of the management toolset as applets.



(1) The end-user sends a request to the WWW server to retrieve a Web page. (2) The WWW server sends the Web page to the user's host. The Web page includes a management applet which will also be loaded on the user's host and executed there. (3) The management applet, now on the user's host, tries to contact the TANGRAM DPE Gateway. This is not permitted in a Netscape browser!

Figure 19-1. Management Applet Loaded With a Netscape Browser via the Internet

19.3 Problems While Using Applets

19.3.1 Connecting Different Hosts

Contrary to the way applications function, applets can be loaded over the internet from any point. Applets that are loaded over the internet are untrusted. Before you executing an applet for the first time, one is not sure if the applet will do any malicious action like deleting files. For that reason, applets run in a very restrictive environment witch permits actions such as[1]:

- Read files on the local system.
- Write files to the local system.
- Delete files on the local system.
- Rename files on the local system.
- Create a network connection to any computer other than the one which the applet was loaded.
- Listen for or accept network connections on any port of the local system.

and many more.

The most interesting points in terms of management applications are the two last ones. In the distributed environment of the *PCS in TINA* project, a local host is used as a Web Server[2]. The TANGRAM DPE Smalltalk Image, which serves as a Gateway to the TANGRAM DPE, is available on a different host[3]. Therefore, to retrieve any computational object, a browser has first to connect the Web server in order to load the Web page with the management applet. The management applet checks the IOR[4] which contains information about how and where to retrieve objects. In this case, the objects were located on an other host which had to be connected by the applet. This is not permitted in the Netscape browser and causes a security exceptions which prohibits the applet to continue (cf. Figure 19-1).

To solve this problem, a server has to be used, which has to run on the same host the Web server is running on.

19.3.2 Loading CORBA Functionalities

An other problem that occurs when using an applet is in the performance while loading an applet. It takes too long to load all needed CORBA functionalities onto the user's host. My suggestion for solving the performance problem is, to implement the Service Access Layer completely as a server which will be contacted by the Service Access Manager Layer. In that case it would be no need to load any CORBA functionalities onto the user's host. The applet would be shrunk down from a 'fat client' to a 'thin client'.

1. [Flanagan96], pp. 197-198
2. As I am writing this, the Web server's name is hughes.
3. Until April, the servers name was bell.
4. See "The Inter-ORB Communication Architecture" on page 36.

19.4 Possible Solutions

19.4.1 Usage of A Gateway-Server

To prevent an applet from contacting an other host than the one it is loaded from, a server has to run on the WWW server which serves as a gateway to other hosts.

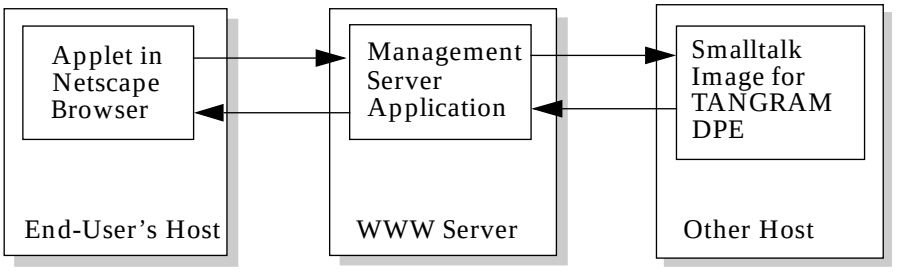


Figure 19-2. Using an Additional Server as Gateway to Other Hosts

With Visigenics Visibroker for Java, it is very easy to implement a server if you carry out the following steps. First certain methods needed for the server have to be identified. Next the IDL definition for the server has to be written and finally, after running the idl2java compiler, the skeletons have to be filled with 'flesh' to implement the server.

19.4.2 Usage of 'Thin Clients'

Instead of loading a 'fat client' with full CORBA functionality, the management applet should only consist of the application layer and the Service Access Manager Layer. The Service Access Layer, which consists of the API functionality must be locate on the WWW server host and act there as a gateway.

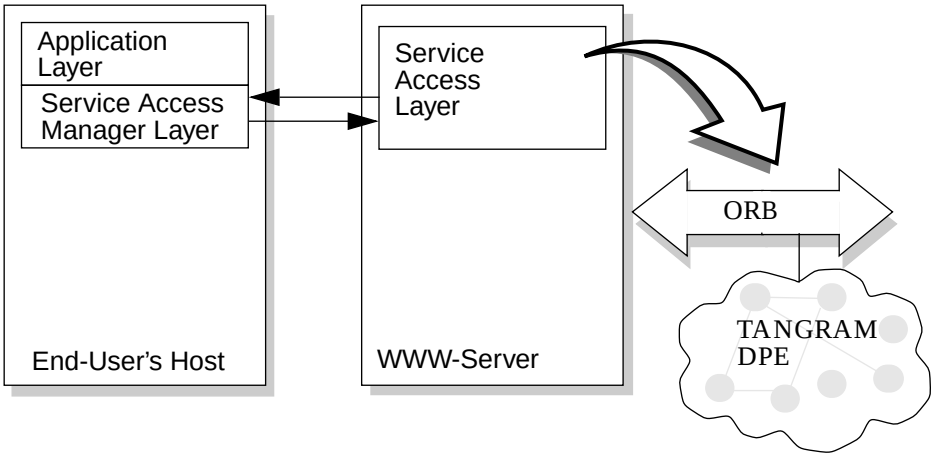


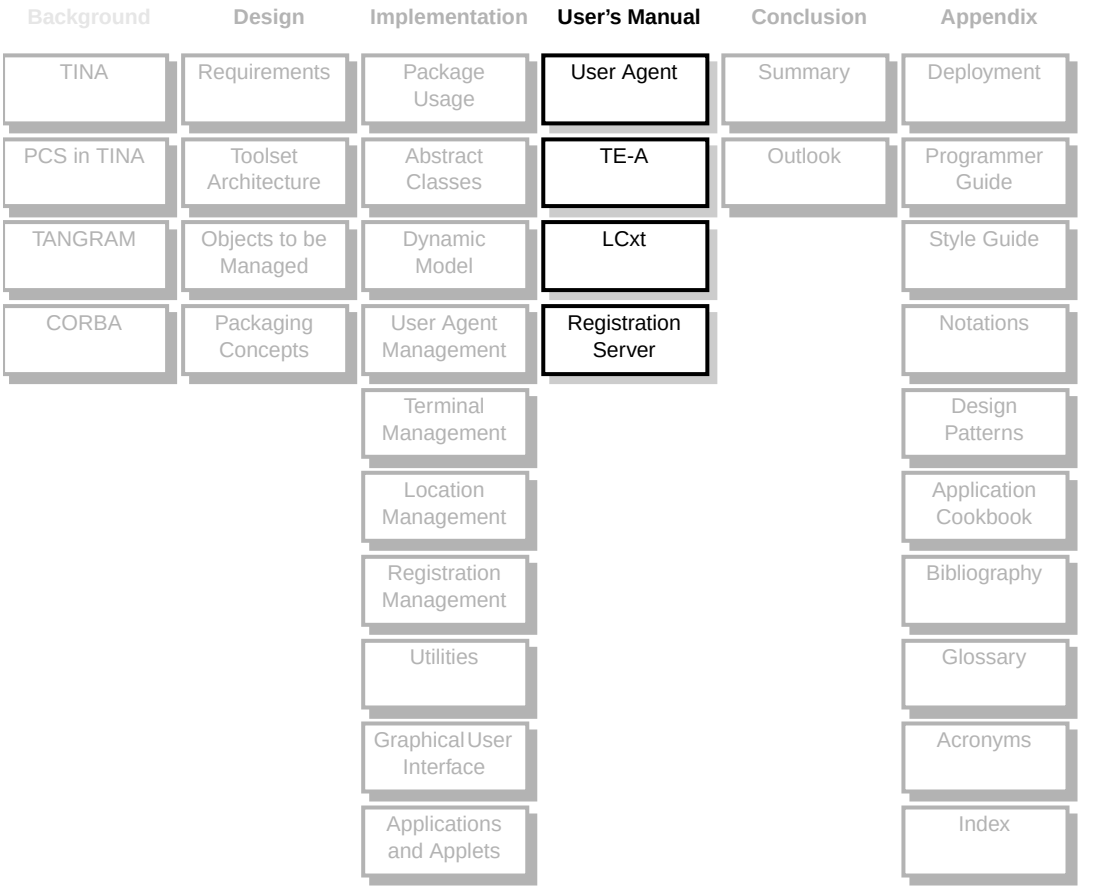
Figure 19-3. Management Applets as 'Thin Clients'

Part 4

Views–The Graphical User Interface

This part of the book introduces the graphical user interface of the management toolset as presented to an end-user. The usage for each management application will be explained step by step.

This part can be seen as a user's manual.


Background
Design
Implementation
User's Manual
Conclusion
Appendix
TINA
PCS in TINA
TANGRAM
CORBA
Requirements
Toolset Architecture
Objects to be Managed
Packaging Concepts
Package Usage
Abstract Classes
Dynamic Model
User Agent Management
Terminal Management
Location Management
Registration Management
Utilities
Graphical User Interface
Applications and Applets
User Agent
TE-A
LCxt
Registration Server
Summary
Outlook
Deployment
Programmer Guide
Style Guide
Notations
Design Patterns
Application Cookbook
Bibliography
Glossary
Acronyms
Index

20 User Data Management

This chapter describes the usage of the management toolset application for managing User Agents.

20.1 The PCS User Agent

The PCS User Agent is a s-independent component representing a user in the service provider domain. It acts on behalf of the user, and may be seen as a simple intelligent agent-like component. The management application for user data configuration enables the user to create, delete and modify a user agent.

The User Data Management application enables the user to create user data computational objects (PCS User Agents) as well as to modify and delete data, which is the visible part of the User Agent.

20.2 Usage

The User Data Management application is located in the package **de.gmd.fokus.ice.pcs.mngmt.uap.appUA** and is named UserConfigurationTool. A script file to start the UCT is also provided[1].

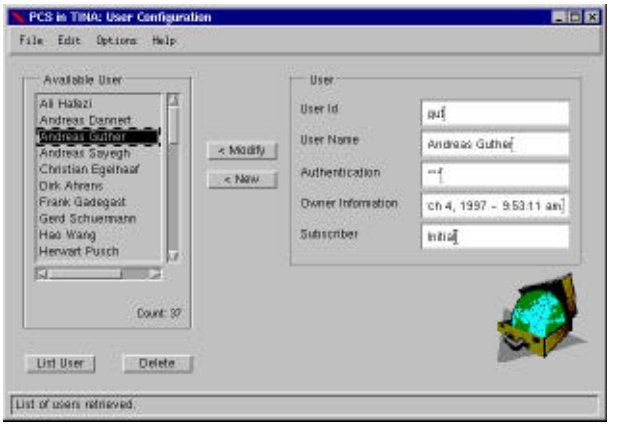


Figure 20-1. Main Window 'User Configuration'

20.2.1 How to Start

To start the User Data Management application:

1 Edit the script file to fit your personal environment.

This step is only needed if you start the User Data Management Application for the first time.

1. An example script file is presented in Section 26.4 on page 130.

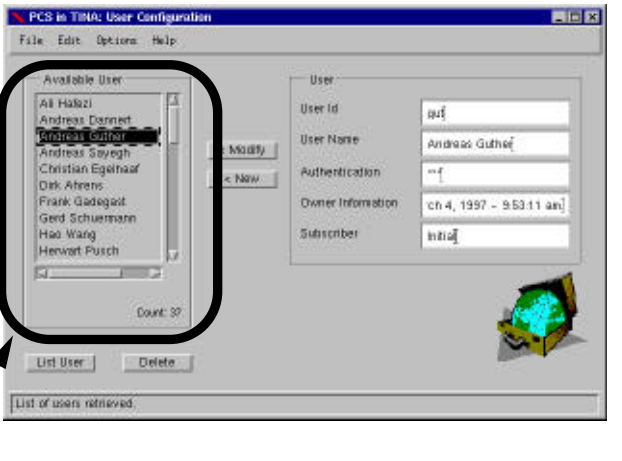
2 Start the script file.

After you have started the script file, the dialog for management and configuration of the user data as shown in Figure 20-1 will appear.

20.2.2 Listing of all Available Users in the System

After starting the application, the list of available users is empty. To retrieve a list of all available users in the system:

Press the ‘List User’ button.



The list of ‘Available Users’ in the system is empty after starting the application.

20.2.3 Getting User Data

To retrieve user data:

1 First list all the users in the system by pressing the List Users’ button.

2 Choose a user in the list with the mouse or the keyboard.

3 Press spacebar on the keyboard or click with the mouse on the selected user from the list.

User data will be shown in the right “User’ group.

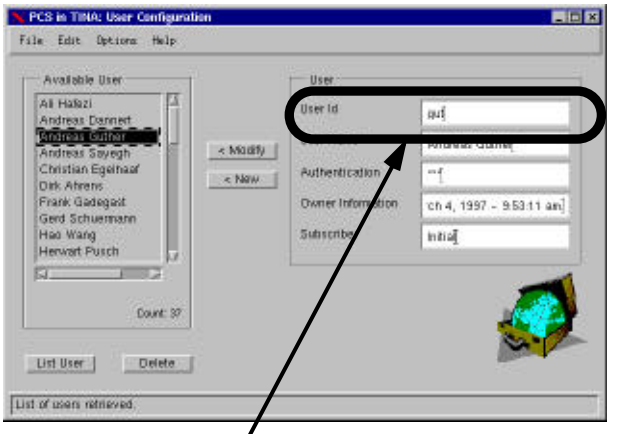
20.2.4 Creating a New User Agent

To create a new user, you have to choose a unique user ID.

1 Insert a unique user id in the ‘User Id’ field.

2 Edit the other fields appropriate to the new user’s data.

3 Press the ‘New’ button.



To create a new User Agent, you need a non existing User ID.

20.2.5 Modify User Data

To store modified user data select the button ‘Modify’.

1 List all available users.

2 Select a user from list.

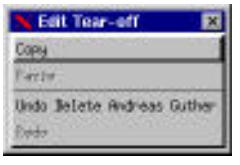
3 Change user data.

4 To save changes, press the ‘Modify’ button.

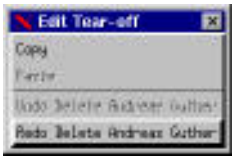
20.2.6 Undo and Redo

The User Data Management application provides an unlimited undo and redo command. Each available undo and redo command is available under the menu entry ‘Edit’. See also Figure .

The ‘Edit’ menu is a ‘Tear-Off’ menu: that means you can tear off the menu from the main menu and put it anywhere on your desktop. Tear-off menus are only available under Sun Solaris systems.



Undo, available after deleting a user entry.



Redo, available after undo of previous deletion.

Figure 20-2. Menus ‘Undo’ And ‘Redo’

20.2.7 Logging

The User Data Management application provides a logging function. Verbose information about program actions can be sent either to:

- Standard output,
  or
- An arbitrary logfile.

To change the logging output

1 **Select Options from the menu and then select ‘Logging’.**

2 **Set the desired options in the dialog shown in Figure 20-3.**

While starting the application, you can set the logging options using the DEBUG mode flag (see the example script file shown in Figure 26-1 on page 131).

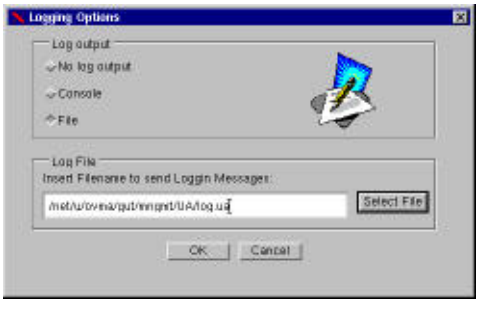


Figure 20-3. Dialog ‘Logging Options’

20.2.8 Dialog About

Every Management Application has an ‘About’ dialog to display information about the application.

Figure 20-4. Dialog ‘About’

21 Terminal Equipment Management

This chapter describes the usage of the management toolset application for managing Terminal Equipment Agents.

21.1 The PCS Terminal Equipment Agent

A Terminal Equipment Agent (TE-A) represents a user system within the provider domain. A TE-A maintains minimum information on resource configuration of a user system, e.g. access points, user applications, stream interfaces and Generic Session Endpoints. A TE-A is one of the key computational objects for mobility, since it keeps track of associations between terminals and access points in the provider domain[1].

The Terminal Management Application makes possible the creation and configuration of the PCS enhanced TINA Terminal Equipment Agent (PCS-TE-A). It includes several specific pieces of information for modelling an end user system.

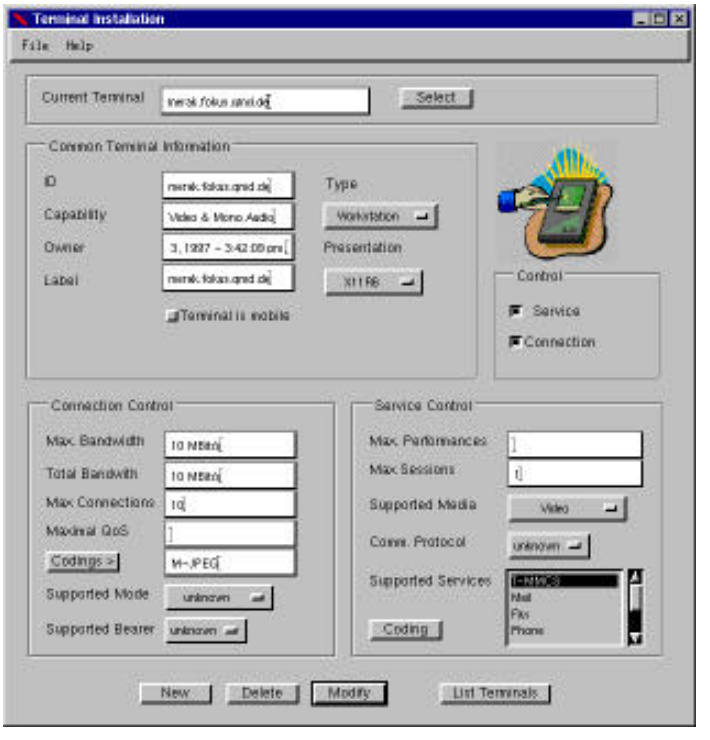


Figure 21-1. Main Window 'Terminal Management Application'

1. [Berndt+95]

21.2 Usage

The Terminal Management application is located in the package **de.gmd.fokus.ice.pcs.mngmt.uap.appTEA** and is named TEA. A script file to start the Terminal Management application is also provided[2].

21.2.1 How to Start

If you start the Terminal Management application for the first time, you may have to edit the start script file.

1 Edit the start script file for the Terminal Management application.

This step is only needed if you start the Terminal Management Application for the first time.

2 Start the Terminal Management application with the start script file.

21.2.2 Listing of all Available Terminals of the System

In order to list all available terminals of the system, you must open the ‘Terminal Selection’ dialog.

1 Press ‘List Terminals’ button.

The button is located at the bottom of the application. After pressing the button, the dialog window shown in Figure 21-2 will open.

2 Press ‘List’ button.

If you have opened the dialog for the first time, the list will be empty. After you have retrieved the list, you should be aware that the list is always a snapshot of the system taken at the time of receiving the data.

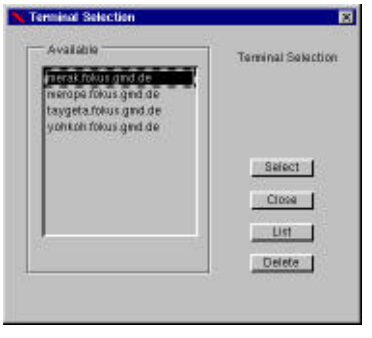


Figure 21-2. Dialog ‘Select Terminal’

21.2.3 Getting Terminal Data

To get the data of a special terminal, you have to either choose one from the list of available terminals or edit the ‘Current Terminal’ field.

2. An example script file is shown in Figure 26-1 on page 131.

Choosing a terminal from the list

To choose a terminal from the list of available terminals, you must list all available terminals of the system.

1 Press ‘List Terminal’ button in main application.

2 Press ‘List’ button in the dialog titled ‘List of available Terminals’.

3 Select a terminal from the list.

The terminal data will be updated automatically in the main application.

Insert a terminal name in the ‘Current Terminal’ edit field.

If you know the name of the terminal you want to display, you can insert the name directly into the ‘Current Terminal’ field and then activate the query for the terminal data.

1 Enter the name into the ‘Current Terminal’ field.

2 Query terminal data with the ‘Select’ button.

If you entered the name of an existing terminal, the terminal data will be displayed.

21.2.4 Creating a New Terminal Equipment Agent

To create a new Terminal Equipment Agent (TE-A) you need at least the unique label, which is the identifier for a TE-A in the provided system.

1 Enter a new label in the ‘Label’ field.

The field ‘Label’ is located in the ‘Common Terminal Information’ group (Figure 21-3).

2 Edit other terminal information data.

3 Press the ‘New’ button.

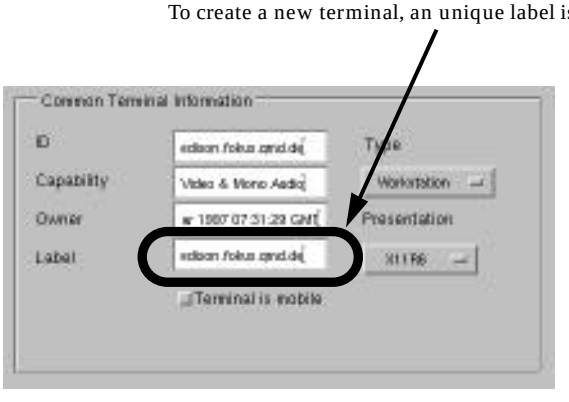


Figure 21-3. Group ‘Common Terminal Information’

21.2.5 Modifying Terminal Data

To modify terminal data:

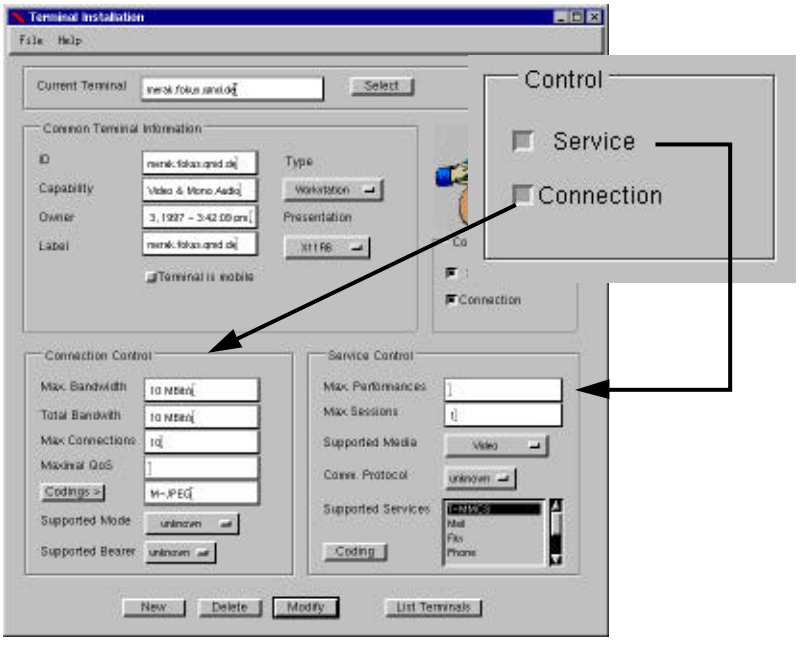
1 Select a terminal.

For how to select a terminal see Section 21.2.3.

2 Make changes to the terminal data.

3 Press the ‘Modify’ button.

If you do not press the ‘Modify’ button after changing the terminal data, all changes will be lost after selecting another terminal or closing the application.



With the ‘Control group the access to the groups ‘Service Control’ and ‘Connection Control’ can be toggled on or off.

Figure 21-4. Group ‘Control’

21.2.6 Set Codings of Connection Control

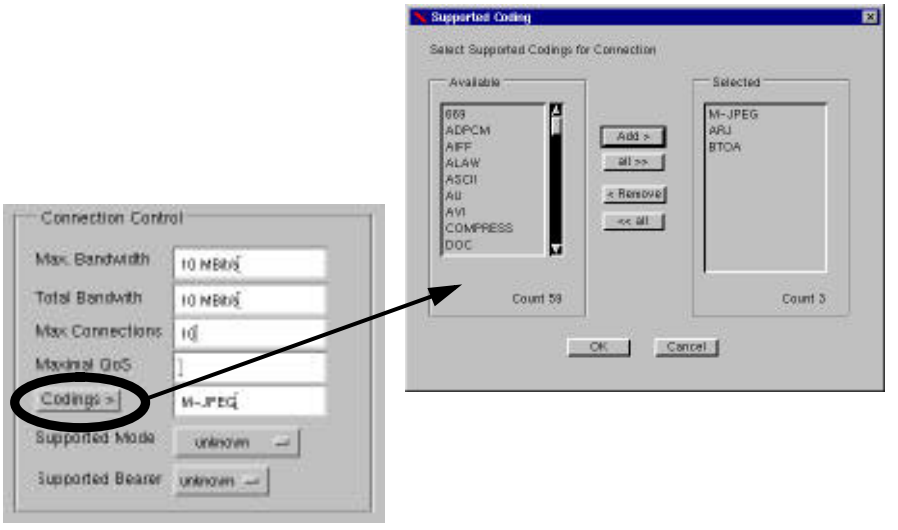


Figure 21-5. Dialog ‘Supported Codings’

For controlling the ‘connection control’, the Terminal Equipment Agent provides, among other things, a field called ‘Codings’. To insert a supported coding you can use the dialog title ‘Supported Coding’.

1 Select the ‘Codings’ button in the ‘Connection Control’ group.

If the button cannot be pressed, you have to check the group ‘Control’ and activate ‘Connection’ (Figure 21-4).

As a result, the dialog to select supported codings, shown in Figure 21-5, will be opened.

2 Change to the dialog titled ‘Supported Codings’.

3 Select codings from the ‘Available’ list.

Each selected item will be placed in the ‘Selected’ list. To de-select an item, click on it in the list ‘Selected’.

4 To insert the chosen codings into the ‘Codings filed from the ‘Connection Control’ group, press ‘OK’. To reject the selection, press ‘Cancel’.

21.2.7 Set Coding Quality of Service Control

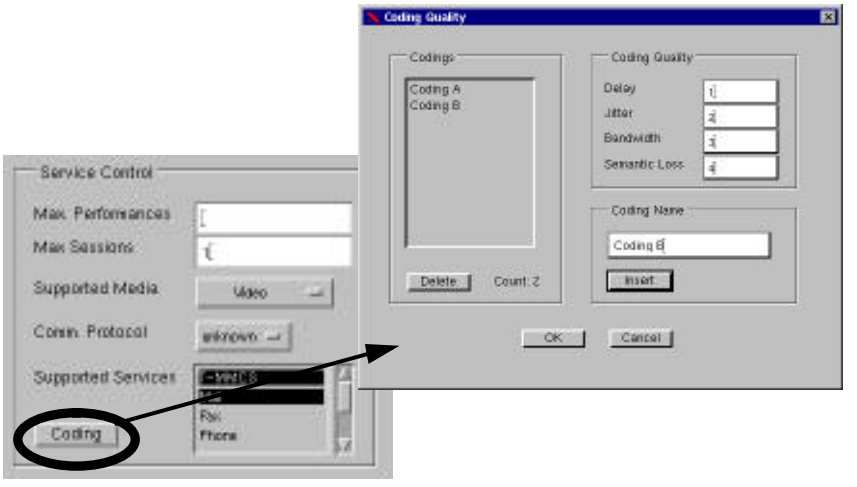


Figure 21-6. Dialog ‘Coding Quality’

The PCS enhanced TE-A provides the ability to set the codings and their quality parameter for the service control.

To set codings an their quality parameter, you must use the ‘Coding Quality’ dialog.

1 Press the ‘Coding’ dialog from the ‘Service Control’ group.

If the button cannot be pressed, check the group ‘Control’ and activate ‘Service’ (Figure 21-4).

The dialog to edit coding quality, shown in Section 21-6, will be opened.

2 Insert codings and their parameters.

3 Choose ‘OK’ to accept or ‘Cancel’ to dismiss.

Note that you have to save modification with the ‘Modify’ button before you select a different terminal or else all changes will be lost.

21.2.8 Undo and Redo

Undo and redo operations are not supported in this version of the Terminal Management application.

21.2.9 Logging

Logging options have to be set inside the start script file. For more information about parameters to set inside the start script file see Section 26.1 on page 129.

22 Location and Location Context Management

This chapter describes the usage of the management toolset application to manage the PCS Local Context.

22.1 The PCS Location

The knowledge about a local context supports the selection process of finding an appropriate terminal to address an end user, without forcing him to explicitly register at a terminal. Currently, TINA only supports the registration at terminals. TINA does not consider registrations at locations. Registration at terminals limits the set of usable terminals to those the user manual is registered with.

A location describes a distinguished region, like a room or a zone. Inside a location, a certain set of terminals can be located. The set of terminals associated with a location is called the local context, which are represented by the computational object LCxt. The management application for location and local context management provides facilities to manage those computational objects.

The PCS-enhanced Access Session supports user registration at locations. Through user registration at locations, the system associates users with specific well known locations, thereby minimizing the required user cooperation to keep the registration data up to date. Normally, a location can be mapped to a room, which is described by the surrounding walls. In some cases it may be desirable to use imaginary zones to describe a location.

22.2 The PCS Local Context

The PCS component Local Context (LCxt) associates a set of terminals with a location. This location capabilities describe the context of a user. Each Local Context represents a specific location in the user domain. A local context keeps the references to the terminal equipment of that location. Thus, the PCS-enhanced Usage Context object is enabled to find terminals related to these specific locations, which in turn enables it to select an appropriate usable terminal.

22.3 Usage

The Location Management Application user application is located in the package **de.gmd.fokus.ice.pcs.mngmt.uap.appLCxt** and should be started by using a script file. An example script file to start the LMA can be found in Section 26.4 on page 130.

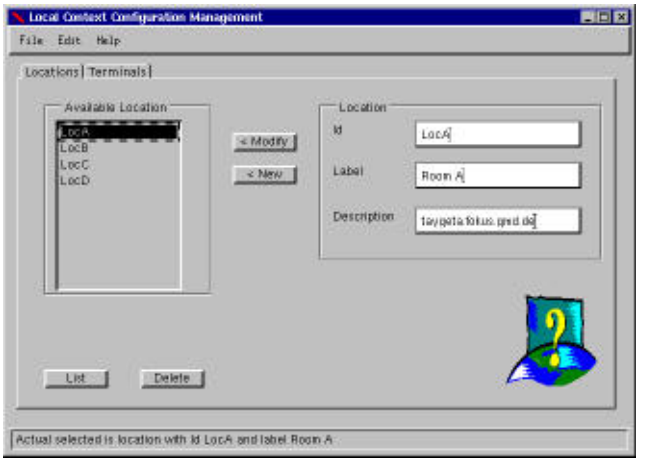


Figure 22-1. Main Window ‘Location Configuration Management’

22.3.1 How to Start

To start the Location Management application:

1 **Edit the script file to start the application concerning your personal environment.**

This step is only needed if you start the Management Application for the first time.

2 **Start the script file.**

After you have started the script file, the dialog, shown in Figure 22-1 will be displayed.

22.3.2 Listing of all Available Terminals in the System

To list the available locations in the system:

Select ‘List’ button.

22.3.3 Getting Location Data

To get the data of a location:

1 **List all available locations of the system.**

2 **Choose a location from the list.**

To retrieve the associated data press the space bar of your keyboard or click with the mouse on the desired location in the list.

22.3.4 Creating a New Location

To create a new location:

1 Select a location ID that is not used in the system.

2 Edit the location data.

3 Select button ‘new’.

22.3.5 Modifying a Location

To modify location data:

1 Get location data.

2 Edit location data.

3 To save modifications press ‘Modify’ button.

22.3.6 Deleting a Location

To delete a location:

1 Select a location from the list.

2 Press ‘Delete’ button.

Note: There is no warning before deletion. To undo a deletion, follow the steps listed in Section 22.3.8.

22.3.7 Configuring a Local Context

To configure a local context:

1 Select a location.

2 Change to the ‘Terminal’ dialog.

To change to the ‘Terminal’ dialog select the tabbed panel labelled ‘Terminals’. The dialog window depicted in Figure 22-2 will be displayed.

3 List all available terminals of the system.

To list all available terminals of the system, press the ‘List Available Terminals’ button.

4 To add a terminal to the location, select a terminal from the list of available terminals, and press the ‘Add’ button.

5 To remove a terminal from a location, select the terminal from the ‘Terminals in Location’ list and then press the ‘Remove’ button.

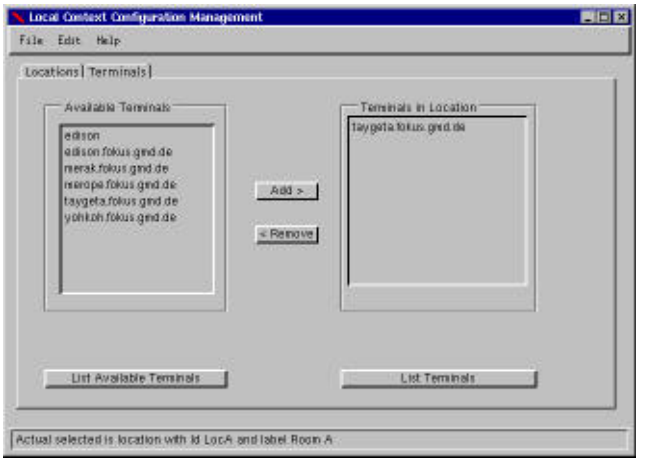


Figure 22-2. Dialog ‘Add Terminals to a Location’

22.3.8 Undo and Redo

The location management application provides an unlimited undo and redo command. Each available undo and redo command is available under the menu entry ‘Edit’. See also Figure on page 101.

The ‘Edit’ menu is a ‘Tear-Off’ menu. That means that you can tear off the menu from the main menu and put it anywhere on your desktop. Tear-off menus are only available under Sun Solaris systems.

22.3.9 Logging

The UCT provides a logging function. Verbose information about program actions can be sent either to

- Standard output,

  or

- An arbitrary logfile.

The logging option of the LMA user application can only be set using the start parameter ‘DEBUGMODE’. For more information about start parameter see Section 26.1 on page 129.

23 Registration Management

The Registration Management Application allows you to retrieve information about registered users at locations as well as to register users manually at locations. Furthermore, the Registration Management Application enables the you to delete all registrations in a specific time lapse.

23.1 The PCS Registration Server

User Registration is the activity allowing the user to register his current location within the Personal Communications Support in *PCS in TINA*. Thus enabling him system-wide personal mobility. The user can register either manually or alternatively automatically by the Registration Server that locates a user and keeps the registration information of the user updated, according to his position. In other words the PCS Registration server automatically keeps track of the location of a user and therefore enables the system to select an appropriate terminal next to the user's position.

23.2 Usage

The Registration Management Application is located in the package **de.gmd.fokus.ice.pcs.mngmt.uap.appRS** and is named **RegServer**. A script file to start the application is also provided[1].

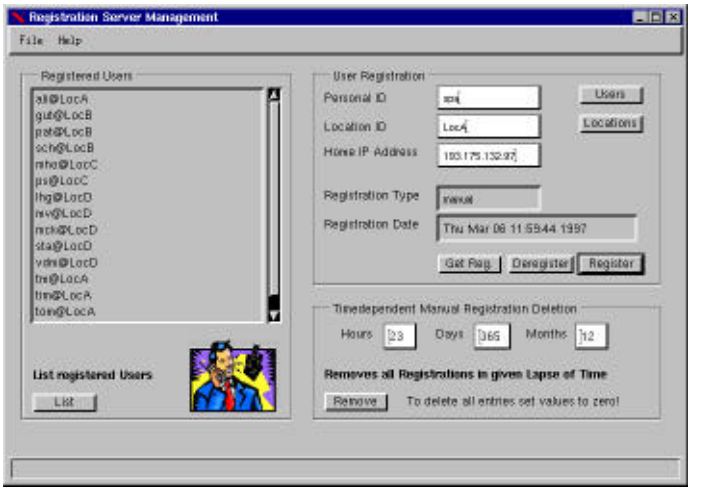


Figure 23-1. Dialog 'Registration Management'

1. An example script file is presented in Section 26.4 on page 130

23.2.1 How to Start

To use the Registration Server Management Application:

1 Edit the start script file to fit to your environment.

This step is only needed if you start the Registration Management Application for the first time.

2 Start the application with the script file.

After you have started the script file, the dialog for the management and configuration of the user registrations as shown in Figure 23-1 will appear.

23.2.2 Listing of all Registered Users

To list all registered users:

Press the 'List' button.

Figure 23-2. Group 'User Registration'

23.2.3 Registering a User

To register a user manually:

1 Press the 'Users' button in the 'User Registration' group.

A dialog with a list of users of the system will be opened.

2 Change to the 'Users' dialog.

If the list is empty, you have to query for the user with the 'List' button.

3 Select a user.

The selected user will be inserted automatically in the field 'Personal ID' in the main application.

4 Change to the main application.

5 Press the 'Locations' button in the 'User Registration' group.

A dialog with the list of all locations will be opened.

6 Change to dialog 'Locations'.

If the list is empty, you have to query for the locations with the ‘List’ button.

7 Select a location.

The selected location will be inserted automatically in the field labelled ‘Location ID’ of the main application.

8 Change to the main application.

9 Press the ‘Register’ button.

The registered user will be shown in the list of registered users.

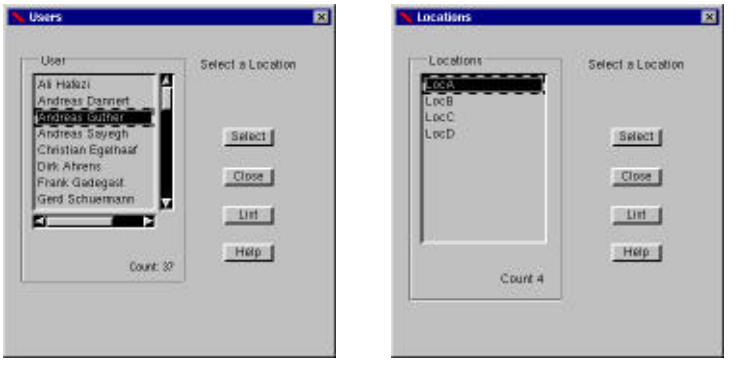


Figure 23-3. Dialogs ‘Users’ and ‘Locations’

23.2.4 De-register a User

To de-register a user

1 List all registered users.

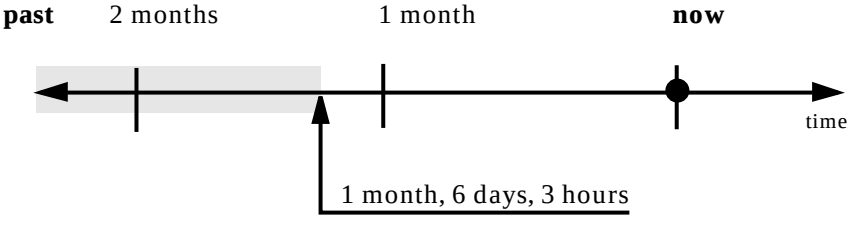
2 Select a user from the ‘Registered Users’ list.

3 Press the ‘De-register’ button.

23.2.5 Purge Registrations

To purge registration in a specific lapse of time:

1 Enter time from which to start purging.



Input: 1 month, 6 days and 3 hours; all manual registrations in the grey shaded part will be deleted.

You enter the time in the groupbox ‘Timedependent Manual Registration Deletion’ (see Figure 23-4). If you set all values to zero, all registration will be purged. If you enter 3 hours, 6 days and 1 month all registration older than one month plus one day plus 2 hour starting from the actual time will be purged.

2 Press button ‘Remove’.

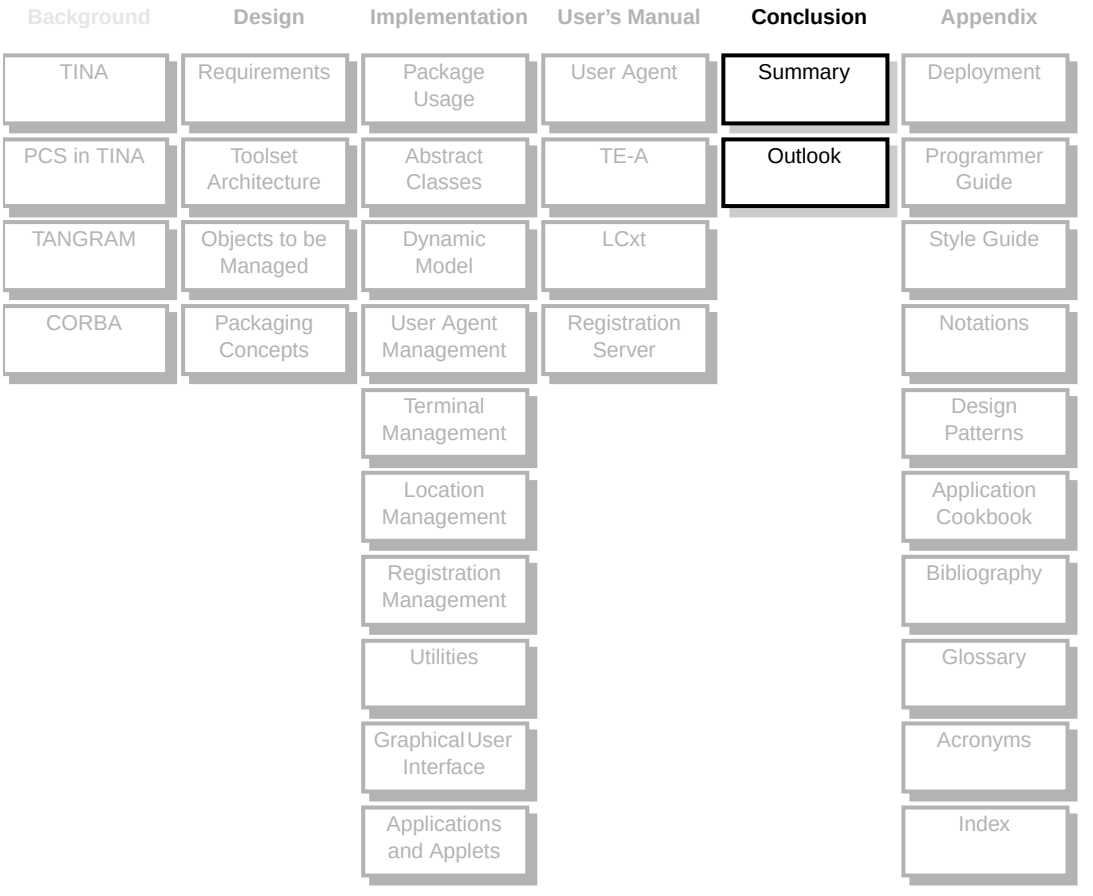
Note that there is no Undo for purging.

Figure 23-4. Group ‘Timedependent Deletion of Registrations’

Part 5

Conclusion

This part of the book gives a summary of the design and implementation of the management toolset. This includes a discussion about problems which occurred during the work and their solutions. This part closes with a look at ideas for further extensions of, and changes to the management toolset as it is presented for this project.


Background
Design
Implementation
User's Manual
Conclusion
Appendix
TINA
PCS in TINA
TANGRAM
CORBA
Requirements
Toolset Architecture
Objects to be Managed
Packaging Concepts
Package Usage
Abstract Classes
Dynamic Model
User Agent Management
Terminal Management
Location Management
Registration Management
Utilities
Graphical User Interface
Applications and Applets
User Agent
TE-A
LCxt
Registration Server
Summary
Outlook
Deployment
Programmer Guide
Style Guide
Notations
Design Patterns
Application Cookbook
Bibliography
Glossary
Acronyms
Index

24 Summary

This diploma thesis presents the design and implementation of a generic management toolset for a set of PCS components which are located in a distributed, TINA compliant processing environment. A set of four applications are implemented. Each application is able to manage a particular computational object of the *PCS in TINA* environment and is built using small, independent, pluggable, and reusable components. Some of those components are used in more than one application. This also demonstrates their reusability.

The management toolset is designed using a three layer architecture. This aids in building applications which are detached from their underlying platforms. Using this architecture also provides a simple way to exchange an API without affecting the main part of an application. A further advantage of the layering concept is that it permits the switching of access from one specific platform to another—during runtime. This is assuming that an equivalent API for the desired platform exists and a corresponding Service Access Manager is implemented.

The management applications have been available to other programmers of the *PCS in TINA* project since February, 1997, and have been used successfully since then.

24.1 Design

The design of this thesis project was highly influenced by the recent movement in design patterns. The design patterns have, in my opinion, helped to improve my design dramatically. Nevertheless, designing does not end with the start of the implementation phase; during the implementation phase, design flaws that have gone unnoticed often become visible. In every time-limited project, one has to draw a line when it comes to re-design decisions during the implementation phase; because there will always be *something* one can make better. A design is ever subject to improvement—and my design is no exception. In the following chapter, I will list some points where improvements could made.

In the words of Antoine de Saint Exupéry:

> You know you've achieved perfection in design not when you have nothing more to add, but when you have nothing more to take away[1].

However, I think that the design of this thesis project presents a valuable architecture which can be applied to future management applications.

1. Found in [Gosling95]

24.2 Implementation

During the implementation phase, the efficiency of the design became obvious. For instance, less and less time was needed for implementing applications with every new application which could make use of the existing toolset components. The implementation of the Registration Server management application, which was the last one, took only a week.

Impacts on Implementation Cycles

The first application to be built was the User Agent management application. All possible aspects of the design were applied to this application. For example, the User management application is the only one which allows the changing of logging output during runtime. Note that logging or redo and undo functionalities were not part of the requirement specifications for this thesis project.

The second application to be executed was the Terminal Equipment management application. Due to its complex data structure, the implementation of this application was very time consuming. With the time limitations on the thesis project in mind, I made the decision not to implement redo and undo functionalities. Even though the Terminal Equipment Agent management application was more complicated to implement; it took, relatively speaking, less time to finish than the User Agent management application.

The third program to be implemented was the Local Context management application and finally, the Registration Server management application.

The Impact of Java

The programming language I used, Java, had not been available for more than a year at the start of this project. In the course of working on the project, I rapidly became acquainted with this object oriented programming language, which also serves certain classes that support design patterns. Java helps programmers to avoid flaws and errors in a very clever way. Regardless of the fact that Java is an interpreted language and is therefore significantly slower than the object oriented language C++, it pays to use Java and to wait for new versions which are promised to be much improved in performance.

Usage of Java's Automatic Source Code Documentation

For the source code documentation of the sources of this project, I used Java's tool *javadoc*, which generates an automatic source code documentation in the HTML format. The usage of this tool helps—among other things—to improve implementation decisions in the usage of keywords like public or protected. In using this tool, the impact of such keywords are revealed to the programmer.

Usage of CORBA

In addition to Sun's Java Development Kit, I used Visigenic's Visibroker for Java. This ORB implementation can be quickly comprehended and used after a very short lead time to get familiar with it. Its usage is quite simple and implementing a server takes very little effort.

24.3 Experiences, Problems and Recommendations

24.3.1 Flaws in Java

It has been stated that Java is a very recent programming language. Due to its youth, it has many areas which could stand improvement and it even has a certain amount of bugs. For example, after several hours of testing I became aware of some limitations and flaws in the Java Date class. Another weakness is apparent in the unpredictable behavior of the expansion of lists. In order to better regulate the height of the lists, I put them in the more disciplined panel object, which is capable of maintaining a size limitation on the lists.

Recommendation: It is very important to keep track of the bugs with the bug list, provided by Sun over the internet.

24.3.2 Converting Applications to Applets

As mentioned in a previous chapter[2], the security restrictions of the Netscape browser present some problems to be solved when trying to use a Java application as an applet. With the first attempts to convert my applications, it became obvious that more time was needed for preparation, testing and designing than I had previously estimated, and than was available for this thesis project. Netscape security restrictions make it necessary to build a special server for providing access to others hosts than the one first contacted.

While defining CORBA interfaces in IDL for specifying a server, I discovered to my surprise, that the IDL does not support derived exception classes—to which category all of my classes belong. This makes it necessary to re-write all exceptions classes used in the management toolset, as far as they are to be used on the server side.

24.3.3 Performance

To test my applications, I built a list of locations with more than two hundred entries and saved them in the TANGRAM platform. This uncovered a performance fault in the system; retrieving the locations list took two to three minutes—an absolutely unacceptable length of time.

The source of the problem was in using the Smalltalk ORB to the C++ ORB. The Smalltalk ORB builds the list while using special function calls of the C++ ORB. To build a single entry, several remote calls have to be executed which results in the long waiting time.

24.3.4 Using a Graphical User Interface Builder

For the design of the graphical user interface, I used a special GUI builder, RogueWave's JFactory. All in all, I was very disappointed with this tool, which had such promise as a a project supporting tool. Making changes in the code is very awkward. The code itself generated by JFactory is full of avoidable comments. The

2. See "Problems While Using Applets" on page 95.

source code builder also writes more than one class in a class file. This is not permitted, and makes it impossible for other classes to reuse them without extracting those classes into separate files which would allow them to be declared as public.

25 Suggestions for Future Extensions

25.1 Towards TINA Service Architecture 4.1

The actual implementation of this project is based on the TINA Service Architecture 2, which is the Service Architecture that the TANGRAM platform was based on at the end of 1996. While I'm finishing this diploma thesis in April 1997, the TANGRAM team and the *PCS in TINA* team are working on a migration from the Service Architecture 2.0 to 4.1.

As soon as the specification for the new platform is released, this management toolset should be converted to the new Service Architecture as well.

The place to apply changes in the toolset is the API package—as long as no changes are made to the structure of the computational objects.

25.2 Management as TINA Service

The actual management applications contain the full access functionalities to the TANGRAM platform. This makes the applications pretty 'fat'. It should be considered weather or not to implement a management service which is located inside of the TINA Architecture.

25.3 Security

At the moment, the management applications are only used by a small set of developers, all of whom are trusted members of the *PCS in TINA* project. When making the management toolset available to a broader set of people, security measures should be considered, like login procedures while starting an application or changing a platform.

25.4 Logging

Using the Command Processor patterns does not only allow the application to provide undo and redo operations, it also allows it to trace all user-actions which concern the modification of computational objects. All management applications provide the ability to turn on or off logging and debugging information; an attribute that is of more interest to programmers than to end users.

Using the Command Processor to trace user actions could provide the following features:

- Saving all changes during a session into a file.
- Allowing undo of a complete session.
- Allowing undos of previous sessions.

25.5 Performance

The performance of the applications is pretty good, considering that Java is an interpreted language. While starting the application, the user has to wait a short, but acceptable time until the application is loaded. This is partly a problem of the Java Abstract Window Toolkit (AWT) which is kept very generic so that it is platform independent.

The bottleneck of performance lies in the usage of different ORB implementations, which serve different needs. The management applications contact a Smalltalk ORB, which contacts, for selected purposes the C++ ORB implementation. Lists, like a list of users or terminals, are created on the Smalltalk server side. The Smalltalk server contacts the C++ server to retrieve information about, e.g. each single user. To retrieve information about single users, several remote procedure calls have to be made. This is highly expensive, not desirable and results in extremely long waiting times for lists that contain 50 items or more.

There are two possible ways to get rid of that bottleneck:

1. The C++ server implementation provides operations to retrieve lists. This is obviously the best solution.
2. The management toolset is optimized to avoid waiting times.

The first point is not in the range of this project and therefore only the second one can be taken into consideration. Possible ways to optimize the implementation are:

- Multithreading
- Callbacks

While multithreading is a precondition for callbacks[1], multithreading without callbacks could be a first step in the right direction. The idea is to implement a thread that is contacting the server automatically while the application is started. When the user decides to query for a list, the thread could already have accomplished the biggest part of the task. The waiting time could be shortened.

Secondly, using callback mechanisms, the client could inform the server—once a list was retrieved—that the client is interested in upcoming changes on the server side and wants to be informed when ever a change in the data structure of a special kind of computational objects occurs. So every time, an other user has changed data, the client will be informed by the server and therefore only has to query for changes in that particular object. After retrieving changed data, the client only has to update it's internal list.

25.6 Usage of Different Platforms

The management toolset provides facilities to change from one platform to another without changing the implementation of the application layer. However, this project has access to only one platform; the TANGRAM DPE. As soon as the ORACLE Java Server is available, access to that platform should also be implemented and provided.

1. [Orfali+97]

25.7 Integration of Authoring Components

Simultaneously with the design and implementation phases of this project, research was done to develop concepts for retrieving data, stored in an arbitrary text file. Those concepts are based on the Shopper / Provider pattern and were done under the label of authoring components. A first implementation is already available. Due to some changes of interfaces during the implementation of this management toolset, both implementations are not, at the moment, one hundred percent compatible. Some changes must first be applied to the authoring components.

The advantage of authoring components lies in their capability to allow data accessing from any kind of database without the need for specialized database engines. The only precondition is the availability of facilities for writing data to a text file from inside of a database.

25.8 Applets in a Netscape Browser

Applets in a Netscape browser (Navigator version 3[2]) are not authorized to contact an other host than the one an applet is loaded from. Since the TANGRAM platform uses different ORB implementations, neither of which is located on the Web server host, changes have to be made on the management toolset in order that applications can be run as applets:

1. A gateway to other hosts has to provide all operations which need contact with other hosts. This is done by a specialized server implementation.
2. For a server implementation the critical operations have to be located.
3. The operations have to be defined in IDL.
4. Exceptions thrown by the API layer have to be rewritten since CORBA does not allow extensions of the exception class in the way it is provided by the management toolset.

Visigenic provides a gateway tool[3] which could also be a solution to the problems described above.

25.9 Extended Usage of Factories

The factory method concept is very useful for moving the responsibility and knowledge of creating an object out of the framework[4]. The management toolset only uses factories to create service access manager. For more flexibility it is imaginable to implement factories for widgets as well as for other components.

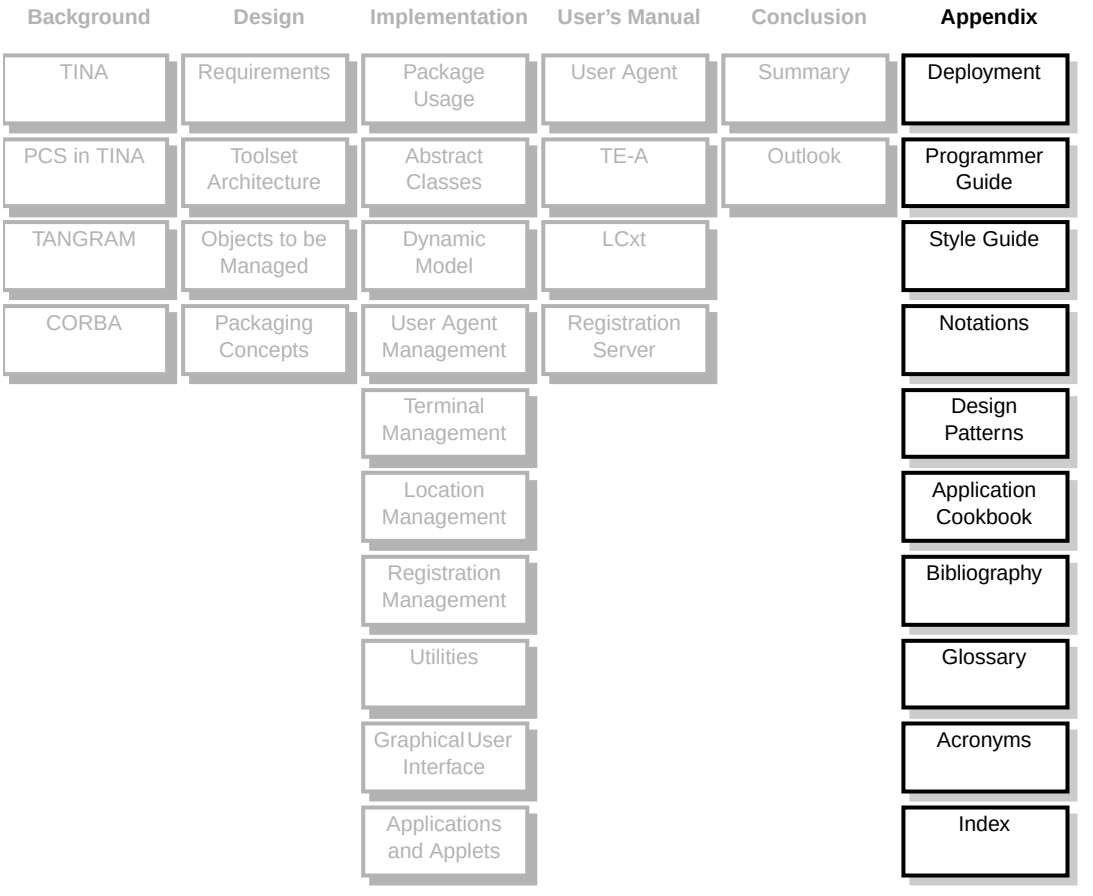
2. Netscape's browser 4.0 will be renamed to Communicator.
3. IIOP Gateway. [Visigenic96a]
4. [Gamma+94]

Part 6

Appendix

This last part contains additional information on various topics, with the aim of giving a better overall understanding of this diploma thesis project. It begins with some hints for installation, followed by a programmer's guide, which could be helpful in adding further extensions. A style guide is included which lists the rules for programming I followed during the implementation phase. Then, there is a chapter which lists and explains all used notation used in this thesis. Next, a chapter follows that explains all the design patterns used in the project and can be seen as the pattern catalogue of this thesis project. The design pattern chapter is followed by an application cookbook which describes the steps needed to implement a portable client.

The book closes with a bibliography, a glossary, a list of acronyms and an index.


Background
Design
Implementation
User's Manual
Conclusion
Appendix
TINA
PCS in TINA
TANGRAM
CORBA
Requirements
Toolset Architecture
Objects to be Managed
Packaging Concepts
Package Usage
Abstract Classes
Dynamic Model
User Agent Management
Terminal Management
Location Management
Registration Management
Utilities
Graphical User Interface
Applications and Applets
User Agent
TE-A
LCxt
Registration Server
Summary
Outlook
Deployment
Programmer Guide
Style Guide
Notations
Design Patterns
Application Cookbook
Bibliography
Glossary
Acronyms
Index

26 Deployment

This chapter presents useful information on how to successfully start the management application.

There are several parameters which must be passed to the application during start-up. These parameters are listed and described in Table 26-1. The following section discusses the software needed to access the Object Request Broker. Next, a list of necessary software with their minimum required version numbers is given.

This chapter ends with sample start script files for a UNIX system which show possible entries for each management application.

26.1 Start Parameter for the Applications

There are several options that can be set while starting one of the management applications. Although not all applications support option dialogs, all do support the setting of options with start parameters. A list of available start parameters is given in Table 26-1. For examples of how to use the start parameters, see the following example start script files.

Table 26-1. List of Available Start Parameter

Parameter Name	Possible values	Description
OSAGENT_ADDR	The address of your OSAGENT for CORBA access	sets the address which the application needs to access servers in a distributed environment via a CORBA ORB.
NS_IOR_URL	The location of the Naming Service IOR	Needed for communication between different ORB implementations.
GSEPIOR	The location of the Generic Session Endpoint IOR	Needed for communication between different ORB implementations.
DEBUGMODE	0 for no Output 1 for debug output on console	See Chapter on debugging and logging.
IMAGEPATH	The path to the images used in the application	If the image directory is not located in the same directory from which your application was started, you need to set that path.

26.2 Using the Object Request Broker

The provided TANGRAM Service Access Manager needs an access point to an Object Request Broker (ORB). Access to an ORB is granted while a so called osagent is running on your system. If no osagent is running on the host from

which you started the management application, you need to specify the host that is running an osagent. For more information about osagents, please see Visigenics ‘Programmer’s Guide Version’ [Visigenic96a].

26.3 Packages Needed to Run the Applications

The Java Packages needed to run a management application of the management toolset are listed in Table 26-2.

Table 26-2. Packages Needed to Run the Management Toolset

Package Name	Version	Provider
java	JDK 1.0.2	JavaSoft, Sun Microsystems
com.roguewave.widgets	JFactory 1.1.0	Roguewave
CORBA	Visibroker 1.2.0	Visigenic
pomoco	Visibroker 1.2.0	Visigenic
de.gmd.fokus.ice.pcs.mngmt	1.0	GMD Fokus and TU Berlin

26.4 Script Files to Start the Applications

This section provides some sample script file for starting the management applications. All examples given are for a UNIX system but are easily adaptable to other operating systems.

User Data Configuration Tool

```
#!/bin/sh
# OSAGENT_ADDR is needed by the Visigenic ORB
OSAGENT=OSAGENT_ADDR=marconi.fokus.gmd.de
# Where to find IOR
IOR_HOME=file:/internet/shannon/home/hw/TestImage
#
# NS_IOR_URL is needed by Naming Service class NS_Adaptor
IORURL=NS_IOR_URL=${IOR_HOME}/ns.ior
# GSEP_IOR_URL is needed by class GSEP_Adaptor
GSEPIOR=GSEP_IOR_URL=${IOR_HOME}/gsep.ior
# DEBUGMODE: if not set or set to 0, no debugging information will be shown
DEBUG=DEBUGMODE=1
#
IMAGES=IMAGEPATH=/internet/shannon/home/gut/PCSinTINA/Management/images
# Where to find the Java interpreter:
JAVAHOME=/net/ice/java/java-jdk-1.0.2
# CORBA ORB for Java
BROKER_HOME=/net/ice/java/unsupported/visibroker-java-1.2.0
BROKER_CLASSES=${BROKER_HOME}/classes
PACKAGES=/net/u/ovma/gut/mngmt/PCSinTINA.zip
JAVACLASSES=${JAVAHOME}/lib/classes.zip

echo Running User Configuration Tool
echo Using ORB ${BROKER_HOME}
${JAVAHOME}/bin/java \
 -D${OSAGENT} \
 -D${IORURL} \
 -D${GSEPIOR} \
 -D${DEBUG} \
 -D${IMAGES} \
 -classpath .:${JAVACLASSES}:${BROKER_CLASSES}:${PACKAGES} \
 de.gmd.fokus.ice.pcs.mngmt.uap.appUA.UserConfigurationTool &
```

Figure 26-1. Script File to Start the User Data Configuration Tool

Terminal Data Configuration Tool

```
#!/bin/sh
# OSAGENT_ADDR is needed by Blackwidow ORB
OSAGENT=OSAGENT_ADDR=marconi.fokus.gmd.de
# Where to find IOR
IOR_HOME=file:/internet/shannon/home/hw/TestImage
# NS_IOR_URL is needed by Naming Service class NS_Adaptor
IORURL=NS_IOR_URL=${IOR_HOME}/ns.ior
# GSEP_IOR_URL is needed by class GSEP_Adaptor
GSEPIOR=GSEP_IOR_URL=${IOR_HOME}/gsep.ior
# DEBUGMODE if not set or set to 0, no debugging information will be shown
#DEBUG=DEBUGMODE=0
#
IMAGES=IMAGEPATH=/internet/shannon/home/gut/PCSinTINA/Management/
images
JAVAHOME=/net/ice/java/java-jdk-1.0.2
# CORBA ORB for Java
BROKER_HOME=/net/ice/java/unsupported/visibroker-java-1.2.0
ORB_CLASSES=${BROKER_HOME}/classes
#
PACKAGES=/net/u/ovma/gut/mngmt/PCSinTINA.zip
JAVACLASSES=${JAVAHOME}/lib/classes.zip

echo Running User Configuration Tool
echo Using ORB ${BROKER_HOME}
${JAVAHOME}/bin/java \
 -D${OSAGENT} \
 -D${IORURL} \
 -D${GSEPIOR} \
 -D${DEBUG} \
 -D${IMAGES} \
 -classpath .:${JAVACLASSES}:${ORB_CLASSES}:${PACKAGES} \
 de.gmd.fokus.ice.pcs.mngmt.uap.appTEA.TEA &
```

Figure 26-2. Script File to Start the Terminal Data Configuration Tool

Location Data Configuration Tool

```
#!/bin/sh
# OSAGENT_ADDR is needed by Blackwidow ORB
OSAGENT=OSAGENT_ADDR=bell.fokus.gmd.de
#
# Where to find IOR
IOR_HOME=file:/internet/shannon/home/hw/TestImage
#
# NS_IOR_URL is needed by Naming Service class NS_Adaptor
NSIORURL=NS_IOR_URL=${IOR_HOME}/ns.ior
# GSEP_IOR_URL is needed by class GSEP_Adaptor
GSEPIOR=GSEP_IOR_URL=${IOR_HOME}/gsep.ior
# DEBUGMODE if not set or set to 0, no debugging information will be shown
#DEBUG=DEBUGMODE=0
IMAGES=IMAGEPATH=/internet/shannon/home/gut/PCSinTINA/Management/images
JAVAHOME=/net/ice/java/java-jdk-1.0.2
# CORBA ORB for Java
BROKER_HOME=/net/ice/java/unsupported/visibroker-java-1.2.0
ORB_CLASSES=${BROKER_HOME}/classes
#
PACKAGES=/net/u/ovma/gut/mngmt/PCSinTINA.zip
JAVACLASSES=${JAVAHOME}/lib/classes.zip

echo Running Local Context Configuration Tool
echo Using ORB ${BROKER_HOME}
${JAVAHOME}/bin/java \
 -D${OSAGENT} \
 -D${NSIORURL} \
 -D${GSEPIOR} \
 -D${DEBUG} \
 -D${IMAGES} \
 -classpath .:${JAVACLASSES}:${ORB_CLASSES}:${PACKAGES} \
 de.gmd.fokus.ice.pcs.mngmt.uap.appLCxt.LCxtUap &
```

Figure 26-3. Script File to Start the User Data Configuration Tool

Registration Management Tool

```
#!/bin/sh
# OSAGENT_ADDR is needed by Blackwidow ORB
OSAGENT=OSAGENT_ADDR=marconi.fokus.gmd.de
# Where to find IOR
IOR_HOME=file:/internet/shannon/home/hw/TestImage
# NS_IOR_URL is needed by Naming Service class NS_Adaptor
IORURL=NS_IOR_URL=${IOR_HOME}/ns.ior
# GSEP_IOR_URL is needed by class GSEP_Adaptor
GSEPIOR=GSEP_IOR_URL=${IOR_HOME}/gsep.ior
# DEBUGMODE if not set or set to 0, no debugging information will be shown
#DEBUG=DEBUGMODE=0
#
IMAGES=IMAGEPATH=/internet/shannon/home/gut/PCSinTINA/Management/images
JAVAHOME=/net/ice/java/java-jdk-1.0.2
# CORBA ORB for Java
BROKER_HOME=/net/ice/java/unsupported/visibroker-java-1.2.0
ORB_CLASSES=${BROKER_HOME}/classes
PACKAGES=/net/u/ovma/gut/mngmt/PCSinTINA.zip
JAVACLASSES=${JAVAHOME}/lib/classes.zip

echo Running User Configuration Tool
echo Using ORB ${BROKER_HOME}
${JAVAHOME}/bin/java \
 -D${OSAGENT} \
 -D${IORURL} \
 -D${GSEPIOR} \
 -D${DEBUG} \
 -D${IMAGES} \
 -classpath .:${JAVACLASSES}:${ORB_CLASSES}:${PACKAGES} \
 de.gmd.fokus.ice.pcs.mngmt.uap.appRS.RegServer &
```

Figure 26-4. Script File to Start the Registration Management Tool

27 Programmers Guide

With the goal of aiding programmers who want to use or even to extend this toolset, this chapter describes the environment in which the management toolset was developed.

27.1 The Programming Environment

This section describes the programming environment that I installed and used to implement the management toolset.

Java Development Kit

For development, the Java Development Kit, version 1.0.2 as released in May 1996 was used to compile the sources.

Widgets

In addition to the Java Development Kit, I used the RogueWave graphical user interface builder JFactory 1.1.0 and the with this tool provided packages.

CORBA Broker Packages

To run the binaries, Visigenics Visibroker for Java, version 1.2.0 was used additionally.

27.2 Generating Java Binary Code

Make Files

For each package, I wrote a makefile which provides tags for each class file in the directory in which it is located. To be included in each makefile, a central makefile.inc is also provided. The path for this latter file can be found in each ‘normal’ makefile.

27.3 Running and Testing the Binaries

Test Script Files

Each main program is located in a separate directory and follows the naming conventions as described. In every directory where there is a main program, a ‘run’ file can also be found. To execute the software, just type run. Environment etc. will be set inside of that runfile which means it will be necessary eventually to edit the runfiles to fit to your special environment.

27.4 Source Code Documentation

Sources are documented in two ways. I followed the' javadoc' guidelines[1] and added javadoc comments as much as was possible. I generated HTML documents which can be displayed using a WWW browser. The actual WWW address can be found in the README file in the package root directory.

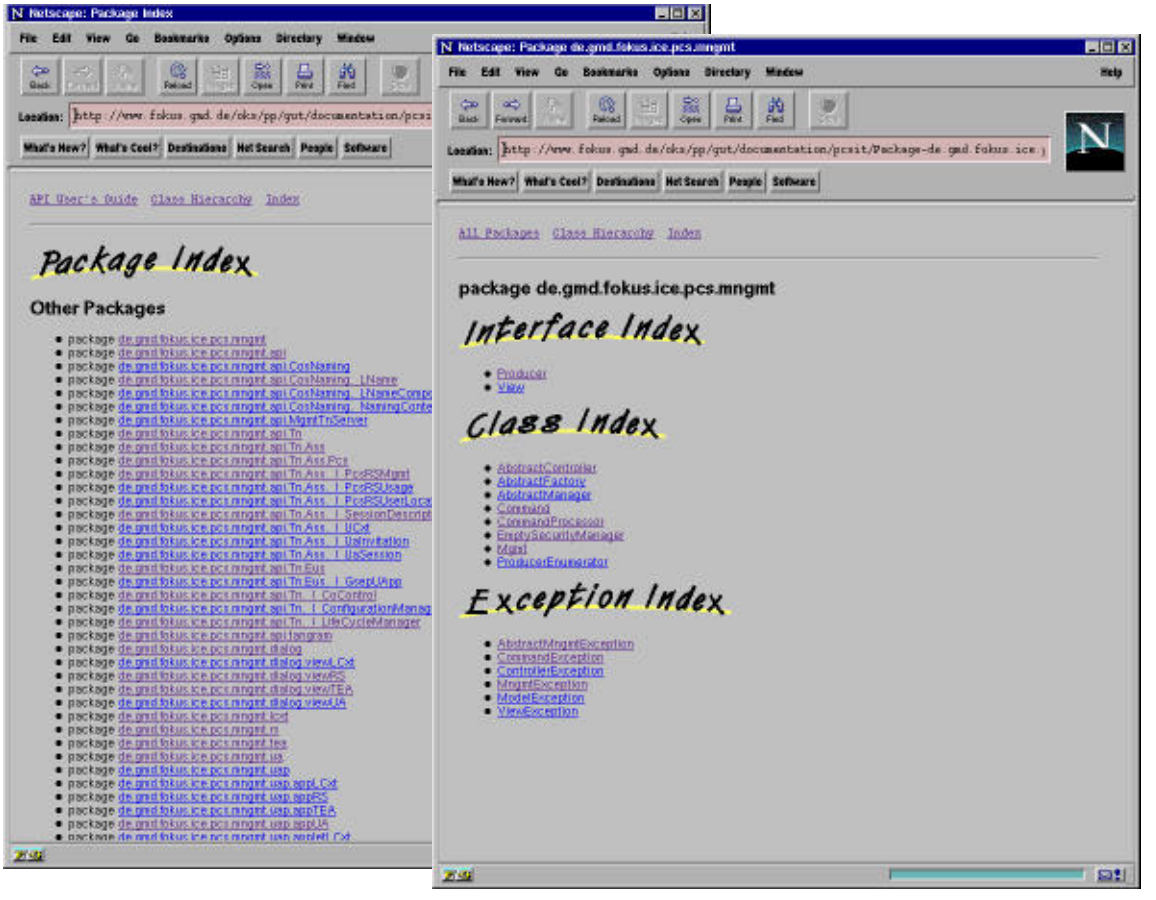


Figure 27-1. Source Code Documentation Available with a WWW Browser

1. [Aitken96], [Friendly95or96]

28 Style Guide

This chapter lists the programming conventions for the implementation part of this project. Naming conventions are declared as well as programming guidelines for writing methods and naming them.

28.1 Structure and Documentation

Packages

For each self-contained project or group of related functionalities, a new java package was created. Each package is hosted in its own directory according to the java package guide lines.

Class Files

Each class is placed in a separate file.

Comments

Java allows C-like comments. A special formatting of comments allows the Java tool *javadoc* to create online-documentation in HTML from the source files. Each class and each method should be commented with a javadoc comment.

28.2 Naming Conventions

Packages

All packages designed for the scope of this project are written in lower case letters (de.gmd.fokus.ice.pcs.mngmt.ua).

Classes

Class names start with a capital letter, followed by mixed letters (UserManagerFactory)

Exceptions

Exception class names end with Exception (ThisShouldNeverOccurException)

Methods

Methods start with a lower case letter, followed by mixed letters (getAllUserNames)

Fields

Fields start with a lower case letter, followed by mixed letters (private String **ownerInformation**;)

Constants

Constants are written in uppercase letters (MAX_WIN_WIDTH).

28.3 Access to Class Fields

All class fields should be declared as private. Private fields are not visible from the outside. Access to fields are granted with access methods; where necessary.

Setting the Value of a Class Field

Methods to set a field start with the prefix **set**.

Example: Field: private String **ownerInformation**;

public void **setOwnerInformation**(String ownerInfo) …

Reading the Value of Class Fields

Methods for reading a field value start with prefix get.

Example: Field: private String **ownerInformation**;

public String **getOwnerInformation** () …

28.4 Recommendations

The following recommendations are taken out of a paper from Doug Lea [Lea97]. This paper, available via WWW, concerning the optimization of Java sources, presents some valuable starting points for Java programmers.

import

Minimize global import of a package by using the asterisk (*). Be precise about imported classes. Check that all declared imports are actually used. Otherwise readers of the code will have a hard time understanding its context and dependencies.

null references

Assign n u l l to any reference variable that is no longer being used. This includes, especially, elements of arrays. This enables garbage collection.

Instance Variable

Never declare instance variables as public.

29 Notations

This chapter introduces the notations used in this book.

29.1 Computational Objects

The notation of computational objects is used in the TINA-C documents.

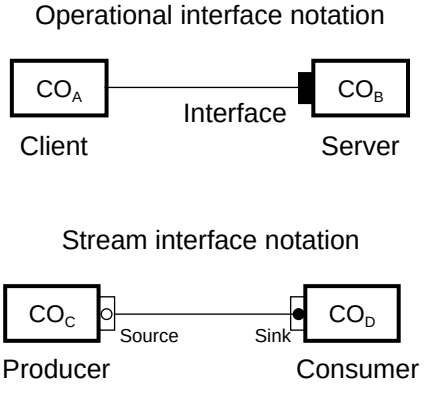


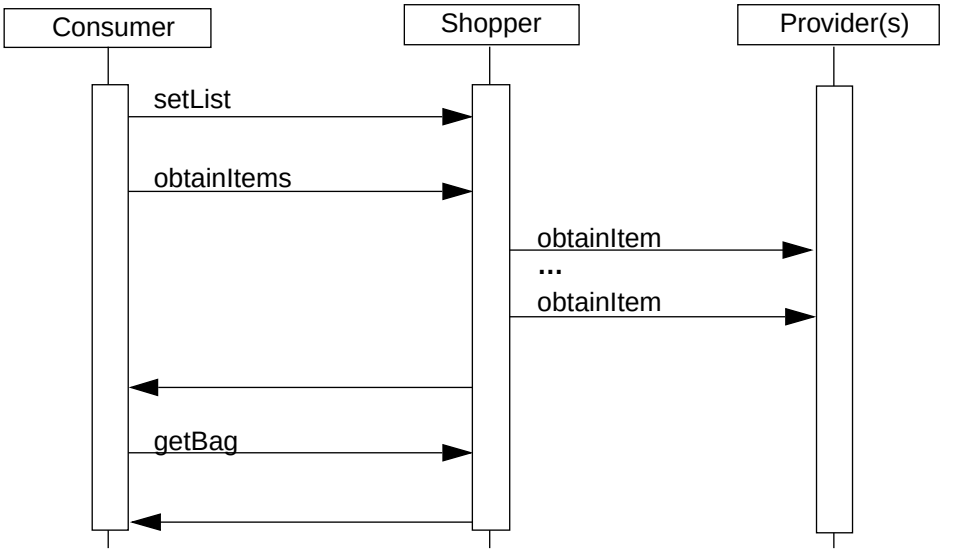
Figure 29-1. Computational Object Graphical Description

29.2 Engineering Objects

The notation of engineering objects is also used in the TINA-C documents.

Figure 29-2. Engineering Computational Object Graphical Description

29.3 Interaction Diagrams

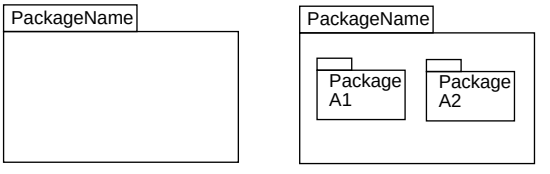


Vertical blocks denote the lifetime of an object. This notation is derived from the Object Sequence Message Charts that can be found in [Buschmann+96].

Figure 29-3. Interaction Diagram Notation

29.4 Packages

The package notation is from the Unified Modeling Language.



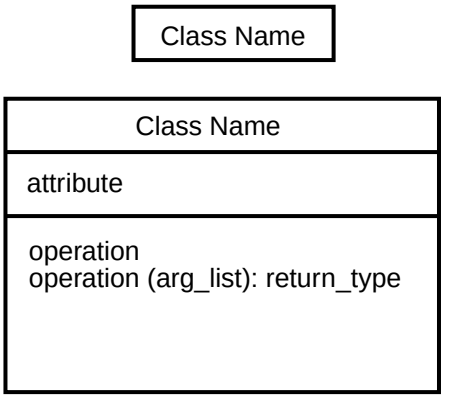
The left notation stands for a package, the right one is a package with nested packages. Relations between packages can be expressed using dashed arrows. More information can be found in [Rational97a].

Figure 29-4. Package Notation

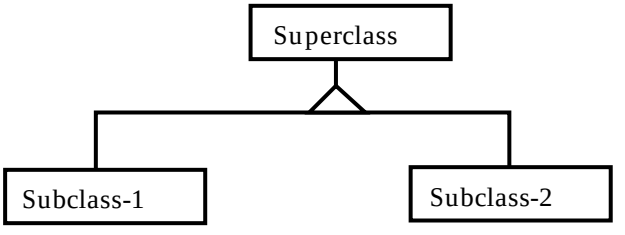
29.5 OMT Notation

The OMT notation provides a rich variety of symbols to express relations between objects. Here I will list only those notations used in my work.

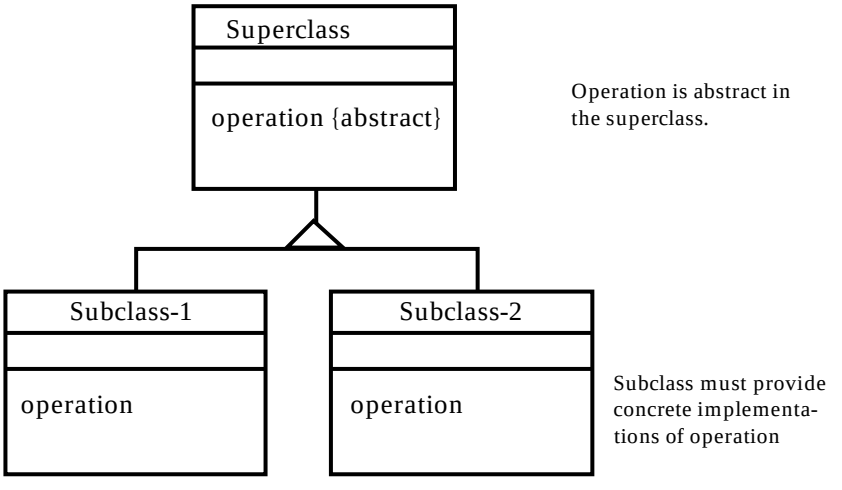
Class



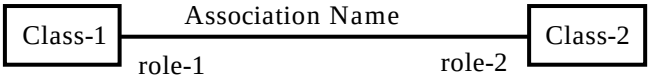
Generalization (Inheritance)



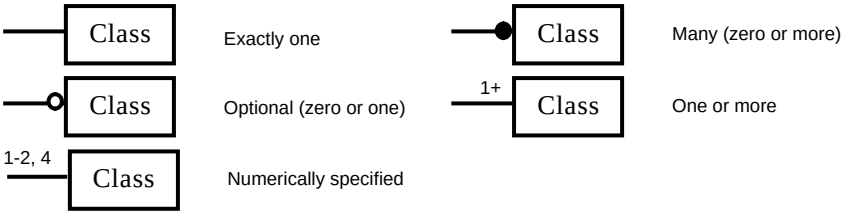
Abstract Operation



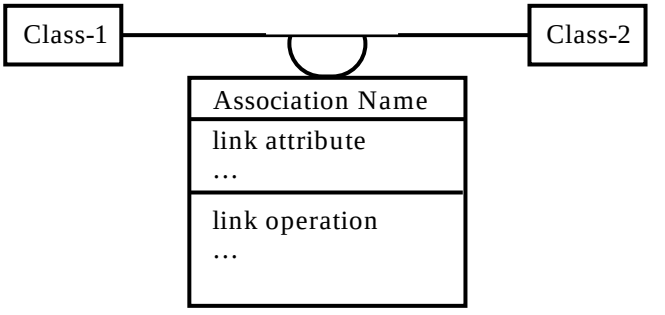
Association



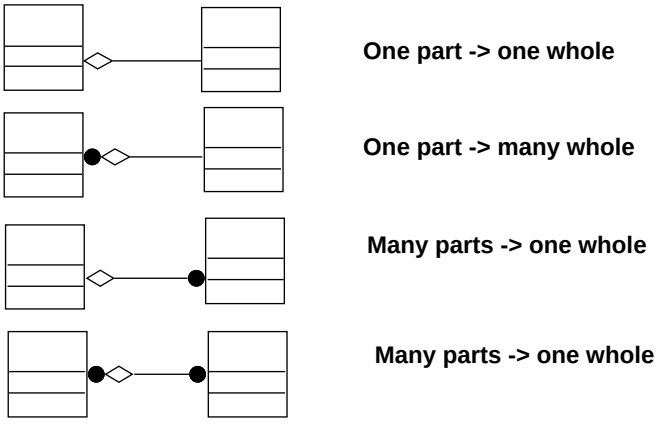
Multiplicity of Associations



Association as Class



Aggregation



30 Catalog of Applied Design Patterns

Software design patterns are an emerging tool for guiding and documenting system design[1]. Design patterns provide a solution for a given problem in a specific context. They are a valuable aid for documenting and communicating software design experience. The sum of all patterns applied to a given domain, build a specific pattern language.

To a great extent, design patterns are used during the design and implementation phase of this management toolset. Knowledge of and familiarity with design patterns are of crucial importance for the better understanding of this project. This introduction is nevertheless just that. The reader who would like to go into more depth then the glance at each pattern that this introduction offers, is referred to the literature listed in the appendix at the end of this book.

30.1 What are Design Patterns for?

In his article 'What is a method'[2] James Rumbaugh says about patterns:

> Over time many people have observed that there are good solutions to certain problems that come up repeatedly in good designs. Over the years in any creative craft, the practitioners learn good ways to solve certain kinds of problems, and the good solutions tend to be reused. Rather than create a brand new design from first principles each time, expert designers save a lot of work by incorporating these 'canned solutions' into their own designs. Home builders have conventional solutions to building windows, running plumbing, or laying out kitchens. These standard solutions may not be optimal in every respect, but they are good, serviceable designs that have been validated by years of experience. These 'solutions waiting for problems' are called patterns. Architects, tailors, cabinetmakers, and other craftsmen have traditionally kept pattern books that showed good solutions to many design problems. Novice craftsmen could learn from the experts by examining their patterns.

James O. Coplien describes pattern in the following way:

> The term pattern, as adopted by contemporary software designers exploring its benefits, is both part of a system and a description of how to build that part of the system. Patterns usually describe software abstractions used by designers and programmers in their software. As abstractions, patterns commonly cut across other common software abstractions like procedures and objects, or combine more common abstractions in powerful ways. [...] The term pattern applies both to the thing (for example, a collection class and its associated iterator) and the directions for making a thing. In this sense,

1. [Coplien94] Software Design Patterns: Common Questions and Answers
2. [Rumbaugh95] pp 14

> software patterns can be likened to a dress pattern: the general shape is in the pattern itself, though each pattern must be tailored to it's context.[3]

Christopher Alexander, the 'father' of the pattern movement and an professor of architecture says in his book 'A Pattern Language'[4]:

> In short, no pattern is an isolated entity. Each pattern can exist in the world, only to the extent that is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it.
>
> This is a fundamental view of the world. It says that when you build a thing you cannot merely build that thing in isolation, but must also repair the world around it, so that the larger world at that one place becomes more coherent, and more whole; and the thing which you make takes its place in the web of nature, as you make it.
>
> […] Each solution is stated in such a way that it gives the essential field of relationships needed to solve the problem, but in a very general and abstract way—so that you can solve the problem for yourself, in your own way, by adapting it to your preferences, and the local conditions at the place where you are making it[5].

The *Design Patterns* movement in computer science is still young. It got its big 'kick' with the book that has already become a classic, 'Design Patterns, Elements of Reusable Object-Oriented software'[6]. This book is also called a catalog of patterns. The fact is, none of the patterns in the book were invented by the authors. The innovation lies mainly in the fact, that never before has a handbook for programmers and system designers existed, that lists a catalog of problems and their solutions given in various contexts.

Since then, several books about patterns have been published and the movement continues to excite a growing interest in the field of computer science. The Impact of design patterns speaks for itself, in that the design of the 'Internet programming language' Java, was heavily influenced by design patterns, as well as was the design of the Common Object Request Broker Architecture (CORBA).

30.2 How to Read this Chapter

The patterns used in this work are mostly taken out of two books: 'Design Patterns, Elements of Reusable Object-Oriented Software'[7] and 'A System of Patterns, Pattern-Oriented Software Architecture'[8].

The description of each pattern given here is very brief and should be seen as an introduction for a reader not familiar with the design pattern movement.

The introduction to each design pattern follows, of course, a specific pattern. It starts with a short description of the intent of each pattern, mostly quoted from the book which the pattern was taken from. The intent section is followed by a description of problems and their solutions that the patterns provide. The compo-

3. [Coplien94]
4. [Alexander+77]
5. [Alexander+77], p. xiii
6. [Gamma+94]
7. [Gamma+94]
8. [Buschmann+96]

sition of the pattern is sketched in the structure section which is followed by a benefits and a liabilities section. This section describes the possible consequences of using each pattern. Each pattern description ends with a section describing the applicability of the pattern relative to this project.

30.3 Layers

The *Layer*s pattern is a widely known and used concept in design software. It is used for example, in the OSI 7-Layer Model, the Java Virtual Machine and in Application Programming Interfaces. The following introduction is an adoption from the book 'A System Of Patterns'[9].

Intent

The *Layers* architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction[9].

Problem

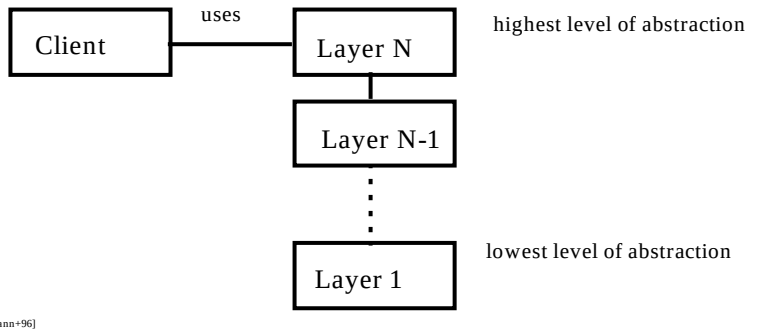
Imagine a system with a mix of low- and high-level issues, where high-level operations rely on the lower ones. The system requires, that late source code changes should not ripple through the system, that interfaces should be stable, and that parts of the system should be exchangeable. Furthermore, similar responsibilities should be grouped to improve comprehension and maintenance.

Solution

Structure your system into an appropriate number of layers and place them on top of each other.

Structure

The main structural characteristic of the Layers pattern is that the service of one layer is only used by the layer above. There is no other dependency between layers. Buschmann et al. compares this with a stack or an onion where each individual layer shields all lower layers from direct access by higher layers.



[Buschmann+96]

9. [Buschmann+96] pp 31

Benefits

Several benefits of the Layers pattern are named by the authors:

- Reuse of layers: If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts.
- Support for standardization: Clearly-defined and commonly-accepted levels of abstraction enable the development of standardized tasks and interfaces. Different implementations of the same interface can then be used interchangeably.
- Dependencies are kept local: Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed.
- Exchangeability: Individual layer implementation can be replaced by semantically-equivalent implementations without too great an effort.

Liabilities

Liabilities are also listed in the description of the Layers pattern.

- Lower efficiency: A layered architecture is usually less efficient than, say, a monolithic structure or a ‘sea of objects’.
- Unnecessary work: If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer, this has a negative impact on performance.
- Difficulty of establishing the correct granularity of layers: A layered architecture with too few layers does not fully exploit this pattern’s potential for reusability, changeability and portability. On the other hand, too many layers introduce unnecessary complexity and overheads in the separation of layers and the transformation of arguments and return values.

Applicability

The architectural pattern *Layer* is used as an overall architecture for this project. Each toolset is divided into three layers:

- The top layer, the *Application Layer* is responsible for handling and displaying the data of the managed objects.
- The middle layer, the *Service Access Management Layer*, allows a platform independent access to services and decouples the application layer from the bottom most layer.
- The bottom layer, the *Application Programming Interface Layer*, regulates the access to a specific platform.

30.4 Observer

A comprehensive description of the *Observer* behavioral pattern can be found in the ‘Design Patterns’ catalogue[10]. The following introduction is mainly based on that description. The Observer pattern is also known as Publisher-Subscriber pattern and can also be found as part of the Model-View-Controller pattern[11].

10. [Gamma+94], Observer(293)
11. see page 146

Intent

The *Observer Pattern* defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically[10].

Problem

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. To achieve consistency by making the classes tightly coupled is not always wanted, because this reduces their reusability.[10]

Imagine an application with different dialogs showing different aspects of the same context. Each view of that context should be informed automatically after a change has occurred.

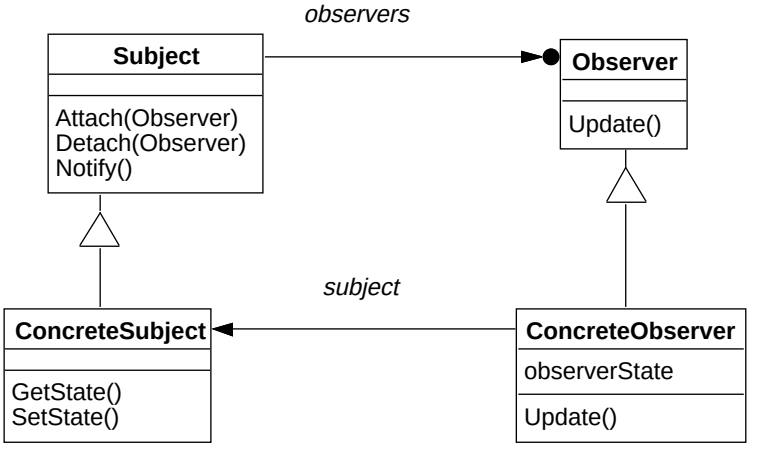
Solution

Design a system where an object which holds data that will be used by several, is called the subject, and the objects interested in the data are called observers. The subject holds a list of all observers and informs them after a change has occurred. Each observer decides if and how it will react to that notification.

Structure

The participants of this pattern are the subject and its observer. A subject can have an unlimited number of observers. An observer attaches (subscribes) to the subject. The subject notifies, after a change of its core data, all observers calling their update method. The observers are free to determine if they want to query the subject for changed data.

The informational model consists of two abstract classes: the *Subject* and the *Observer* (cf. Figure 30-1). The Observer defines an updating interface for Objects that should be notified of changes in a Subject. The Subject provides an interface for attaching and detaching Observer objects. The *ConcreteSubject* implements methods to query and set the core data which are used by the *ConcreteObserver* to retrieve the desired information or change the state of the ConcreteSubject.



[Gamma+94]

Figure 30-1. Structure of the Design Pattern ‘Observer’

Benefits

- Subjects and observers can vary independently.
- Subjects can be reused without reusing their observers, and vice versa.
- Observers can be added without modifying the subject or other observers.
- Subject and object are coupled in an abstract way. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract *Observer* class. The subject is unaware of any feature of the observers except the update method.
- Support for broadcast communication is provided.

Liabilities

- Unexpected updates. Because the subject does not know which of the observers is interested in what information change, it informs all observers after each change. This can cause unwanted updates.

Applicability

The Observer pattern is used in the Model-View-Controller pattern (see below).

30.5 Model-View-Controller

The *Model-View-Controller* pattern (MVC) serves as a pattern to design interactive applications with a flexible human-computer interface.

The MVC pattern is also known as *Publisher-Subscriber* or *Observer-and-Observable*. This pattern is actually an extension of the *Observer* pattern, which is described above.

The programming language Java provides a class *Observable* and an interface *Observer* to support the implementation of the MVC pattern.

The introduction to this pattern is based on the ‘A system of Patterns’ catalog.[12]

Intent

‘The Model-View-Controller architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.'[12]

12. [Buschmann+96]; Model-View-Controller(125)

Problem

Imagine an application where different users place conflicting requirements on the user interface. A typist enters information into forms via the keyboard. A manager wants to use the same system mainly by clicking icons and buttons. Consequently, support for several user interface paradigms should be easily incorporated.

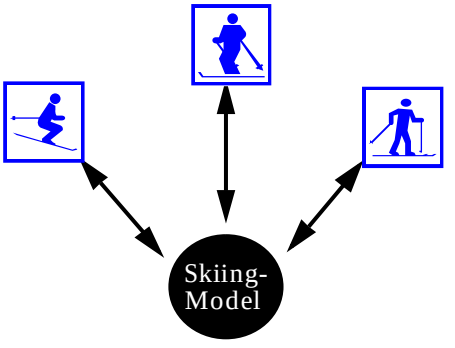


Figure 30-2. Multiple Views of the Same Model

Solution

Use the Observer pattern and add a control instance to handle user input.

Structure

The Model-View-Controller design pattern consists of three components.

The *model* component contains the functional core of the application. It encapsulates the appropriate data, and exports procedures that perform application-specific procession. Controllers call these procedures on behalf of the user. The model also provides functions to access its data that are used by view components to acquire the data to be displayed.

The *view* component is responsible for the representation of the model's data. A view might only display the focus on a very specific part of the models core. Different views might display the same focus in completely different ways. A view registers with the model which informs all registered views through a common interface about changes in the core. This is called the change-propagation mechanism. The propagation is normally done through invoking the update method of the observer. The observer decides after being notified, which methods of the model have to be invoked to retrieve the changed data needed in that specific view.

The *controller* components accept user input as events. The controller components then analyze this input, and decide who—model or view—to inform about the specific user demand. A controller can also be registered as an observer with the model and can therefore participate in the change-propagation mechanism.

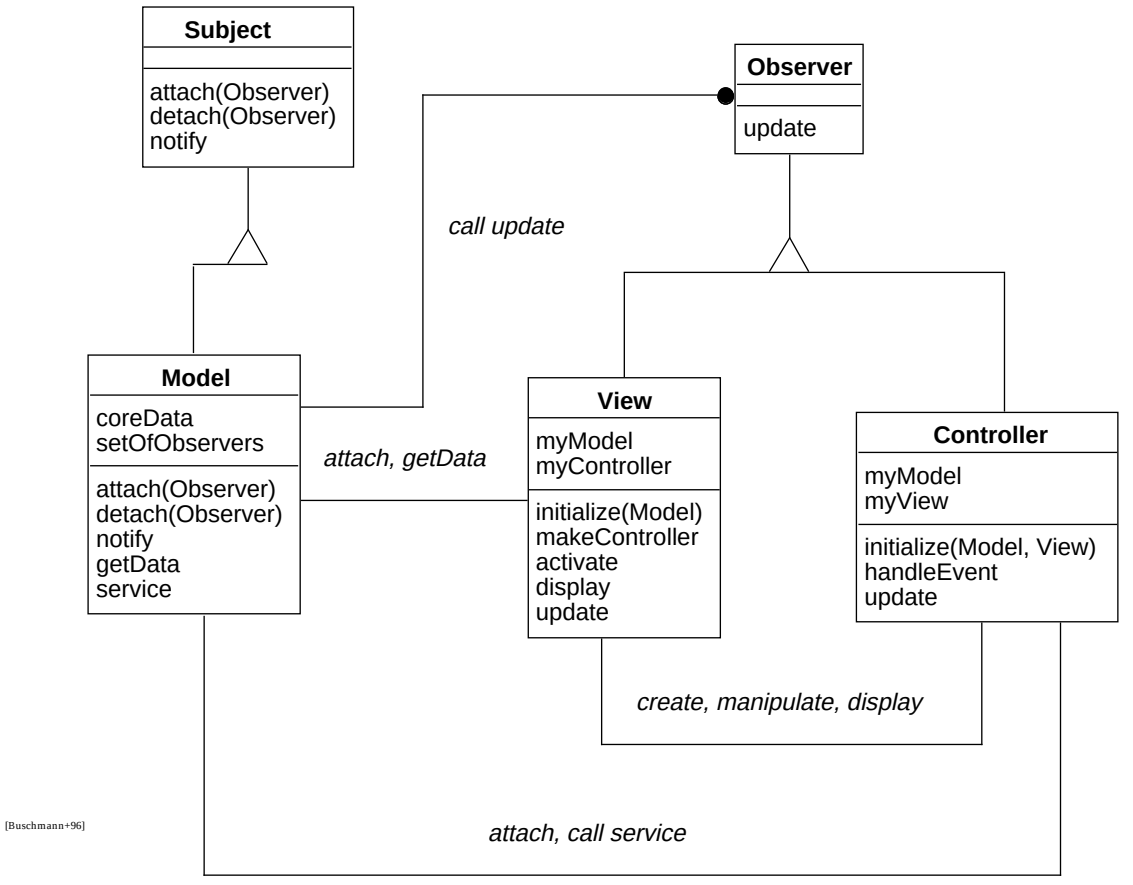


Figure 30-3. Structure of the Design Pattern 'Model-View-Controller'

Benefits

- Multiplicity: Multiple views of the same model.
- Synchronized views: Through the change-propagation mechanism, views are notified upon changes inside the model at the correct time. The model decides when to inform the observers.
- 'Pluggable' views and controllers: A strict separation of the model from the user-interface components allows an easy exchange of views without affecting the model. User interface objects can even be substituted at run-time.
- Exchangeability of 'look and feel': For the same reason, different views for different platforms can be implemented without affecting the implementation of the model.

Liabilities

- Increased complexity: It is possible to have more than one controller for a view. More components to maintain often leads to a more complex implementation which may be harder to understand.
- Potential for an excessive number of updates: Each time a change occurs in the model, all registered observers are notified even if most of them are not interested in that specific change. To avoid this, a message should be sent with each change-propagation to inform observers about the kind of change. Then it is up to the observer to decide whether to contact the model or not.

- Intimate connection between view and controller: Controller and view are separate but closely related components. This hinders their individual re-use.
- Inefficiency of data access in the view component: Depending on the interface of the model, a view may need to make multiple calls to obtain all its display data. Unnecessarily requesting unchanged data from the model weakens performance if updates are frequent. Caching of data within the view improves responsiveness.

Applicability

All management toolsets in this project are designed using the Model-View-Controller pattern. Each graphical user interface is composed of different views which can be exchanged and plugged into other applications. However, I have used the *Controller* pattern in a slightly different way than described in this section. For more details see section “Layer for Application” on page 46.

30.6 Command

The object behavioral pattern *Command* is described in detail in the ‘Design Patterns’ catalog[13]. The *Command* pattern is also known as *Action and Transaction* and is used and extended in the pattern *Command Processor*[14]. The pattern eases implementation of the undo and redo functionalities in an application.

Intent

The Command pattern encapsulates a request as an object, thereby permitting clients with different requests to parameterize, queue or log requests, and support undoable operations.

Problem

Sometimes it is necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. Graphical user interfaces often provide different possibilities for executing a command. Take, for example, buttons and menu entries. It is not desirable to implement the request explicitly in the button or the menu entry.

Solution

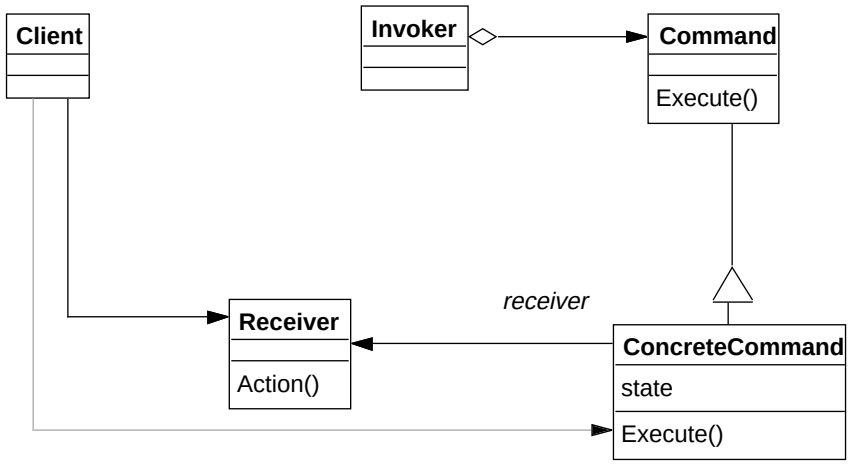
‚The *Command* pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object. This object can be stored and passed around like other objects’[15].

Structure

The Command pattern consists of five participants (cf. Figure 30-4). The command declares an interface for executing an operation. A ConcreteCommand defines the binding between a receiver object and an action. It implements the ‘Execute’ by invoking the corresponding operations on the receiver. A client creates a *ConcreteCommand* object and sets its receiver. The Invoker, a button for

13. [Gamma+94]; Command(233)
14. see page 150
15. [Gamma+94]

example, asks the command to carry out the request. The receiver, e.g. the *Model* of the Model-View-Controller pattern, knows how to perform the operations associated with carrying out a request.



Gamma+94]

Figure 30-4. Structure of the Design Pattern ‘Command’

Benefits

The benefits, listed in the ‘Design Patterns’ catalog are as follows:

- The operation is decoupled from the object that invokes it.
- Command objects are easy to manipulate and to extend.
- Commands can be assembled into a composite command, e.g. a macro command which is a sequence of command objects.
- New commands are easy to add, because existing ones do not have to be changed.
- Undo and redo operations are easy to implement.

Liabilities

For liabilities see the discussion in the *Command Processor* pattern below.

Applicability

The *Command* pattern is used to support the operations create, modify and delete and to support the undo and redo functionalities.

30.7 Command Processor

The *Command Processor* pattern extends the Command pattern. It specially focuses on how to implement undo and redo functionalities. A profound description of the Command Processor pattern can be found in the ‘A System of Patterns’[16] catalog on which this introduction is based on.

16. [Buschmann+96]; Command Processor(277)

Intent

The Command Processor design pattern separates the request for a service from its execution. A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.

Problem

How to implement a flexible and extensible service related functionality in an application and support undo and redo functionality.

Solution

Buschmann et al describe the solution in their book as follows:

> The Command Processor pattern builds on the Command design pattern in [Gamma+94]. Both patterns follow the idea of encapsulating request into objects. Whenever a user calls a specific function of the application, the request is turned into a command object.

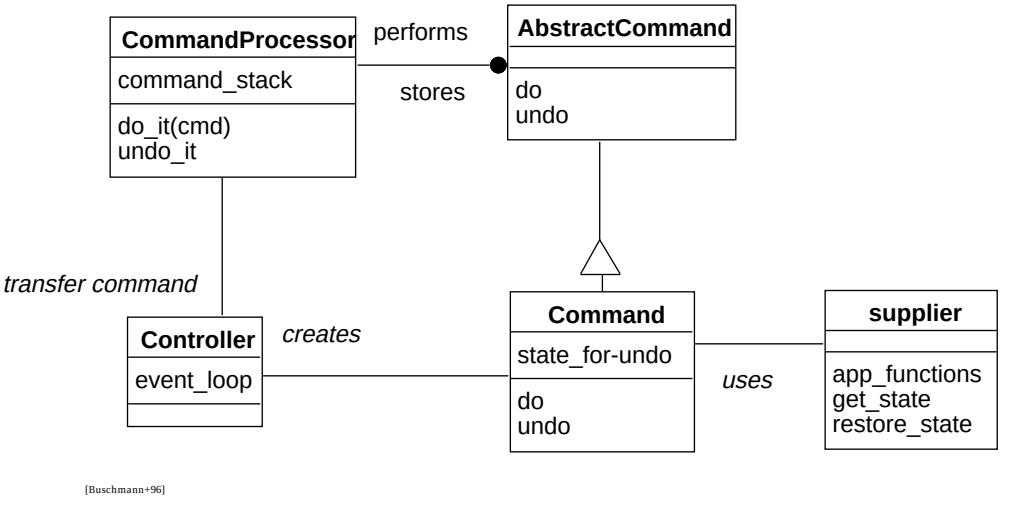


Figure 30-5. Structure of the Design Pattern ‘Command Processor’

Structure

The design pattern is structured into four parts.

- An abstract command defines the interfaces of all commands, e.g. execute, do and redo.
- A controller represents the interface of the application. It accepts requests such as create or delete and creates the corresponding command objects.
- A command processor manages command objects, schedules them, and starts their execution.
- The supplier component provides most of the functionality required to execute concrete commands.

Benefits

- Flexibility in the way in which requests are activated: Different user interface elements for requesting a function can generate the same kind of command object, e.g. a menu item and a button execute the same functionality.
- Flexibility in the number and functionality of requests: Changing the implementation of a command or introducing new command classes does not affect the command processor or other unrelated parts of the application.
- Programming execution-related services: An advanced command processor can log or store commands in a file for later examination or replay.

Liabilities

- Efficiency loss. Decoupling of components costs storage and time. A controller that performs a service request directly does not impose an efficiency penalty.
- Potential for an excessive number of command classes. An application with rich functionality may lead to many command classes.
- Complexity in acquiring command parameters. Some command objects retrieve additional parameters from the user prior to, or during their execution. This situation complicates the event-handling mechanism, which needs to deliver events to different destinations such as the controller and some activated command objects.

Applicability

I used the Command Processor pattern in creating the management toolset for the purpose of providing undo and redo commands and the Model-View-Controller implementation to fulfill the role of the supplier.

30.8 Factory Method

The creational pattern Factory Method is described in the ‘Design Patterns’ catalog[17]. The following description is mainly based on that catalog entry.

Intent

The *Factory Method* defines an interface of a complex object, but lets subclasses decide which class to instantiate. The Factory Method lets a class defer instantiation to subclasses.

Problem

How to enable the user to choose, during runtime, from different classes which class he wishes to instantiate.

Solution

The *Factory Method* pattern eliminates the need to bind application-specific classes into the code. The code only deals with a standardized interface normally defined within an abstract class— therefore it is independent.

17. [Gamma+94]; Factory Method(107)

Structure

The creational design pattern Factory Method is comprised of four participants (cf. Figure 30-6).

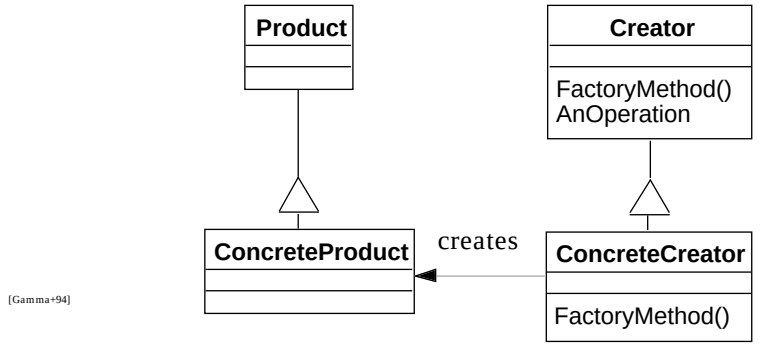


Figure 30-6. Structure of the Design Pattern 'Factory Method'

- The *Product* class defines the interface of objects which the factory method creates.
- The *ConcreteProduct* class implements the *Product* interface.
- The *Creator* class declares the factory method, which returns an object of the type, Product. The Creator class can also define a default implementation of the factory method that returns a default *ConcreteProduct* object. The *Creator* may call the factory method to create a *Product* object.
- The *ConcreteCreator* overrides the factory method to return an instance of a *ConcreteProduct*.

Benefits

The catalog lists the following benefits:

- Products can be added and removed during runtime.
- New objects can be specified by varying values and not by defining new classes. New kinds of objects are effectively defined by instantiating existing classes.
- New objects can be specified by varying the structure.
- Subclassing can be reduced.
- Applications can be configured with classes dynamically.

Liabilities

- The flexibility is limited: A *Factory Method* hard-codes the *Product* class in the *Creator* class. The hard coding is only transferred from the application class to an 'external' class and therefore still exists.

Applicability

I have used the *Factory Method* pattern to instantiate platform managers. Thus, the user can decide during runtime, on which platform he wants to mange objects. He may, in fact, change platforms at any time as long as they are supported by corresponding layers.

30.9 Singleton

The Singleton object creational pattern is part of the 'Design Pattern' catalog[18]. The singleton pattern describes how to ensure that a class has only one instance during the lifetime of an application.

Intent

The *Singleton* pattern ensures that a class only has one instance at runtime, and it provides a global point of access to that instance.

Problem

How to keep a specific class from having more than one instance at runtime.

Solution

The *Singleton* pattern hides its constructors and provides a static method, which controls instantiation of the class and returns a reference to the instance.

Structure

The *Singleton* pattern defines an instance operation that lets clients access its unique instance. The instance method of the *Singleton* is protected and not accessible from outside the class. The *Singleton* has a protected field that holds the reference to the invoked instance. It provides a static method that creates one and only one instance of the singleton at runtime and returns a reference to this field.

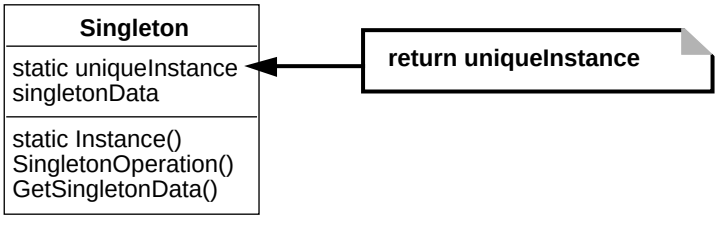


Figure 30-7. Structure of the Design Pattern 'Singleton'

Benefits

The catalogue lists several benefits:

- Controlled access to sole instances can be achieved.
- Name space can be reduced.
- The *Singleton* pattern permits refinement of operations and representation.
- The *Singleton* pattern permits a variable number of instances.
- The *Singleton* pattern is more flexible than class operations.

Liabilities

The catalogue lists no liabilities. Nevertheless, there is some discussion in the pattern literature about the problems of how to delete a Singleton [19].

18. [Gamma+94] Singleton(127).
19. [Vlissides96-W3]

Applicability

I used the *Singleton* pattern to assure that only one instance of a model exists.

30.10 Facade

The *Facade* object structural pattern is listed in the 'Design Pattern' catalog[20]. This pattern helps to structure a system into subsystems with the goal of reducing complexity. The *Facade* object pattern provides a single, simplified interface to the more general facilities of a subsystem.

Intent

The *Facade* pattern provides a unified interface to a set of interfaces in a subsystem. The *Facade* pattern defines a higher-level interface that makes the subsystem easier to use.

Problem

The structural pattern *Facade* addresses the problems of how to make a subsystem easier to use and how to shield clients from the internal structure of the system.

Solution

Design subsystem interfaces for external clients.

Structure

The *Facade* pattern consists of the *Facade* object, which knows its subsystem classes and also knows which one of them is responsible for a request (cf. Figure 30-8). The *Facade* object delegates client requests to appropriate subsystem objects.

The Subsystem class implements the subsystem functionality and handles work assigned by the *Facade* object. The subsystem has no knowledge of the *Facade* object.

Benefits

The 'Design Pattern' catalogue lists the following benefits:

- A *Facade* pattern shields clients from subsystem components. This reduces the number of objects that clients deal with and makes the subsystem easier to use.
- The *Facade* pattern promotes a weak coupling between the subsystem and its clients.
- The *Facade* pattern doesn't prevent applications from using subsystem classes if they need to. Thus a one can choose between ease of use and generality.

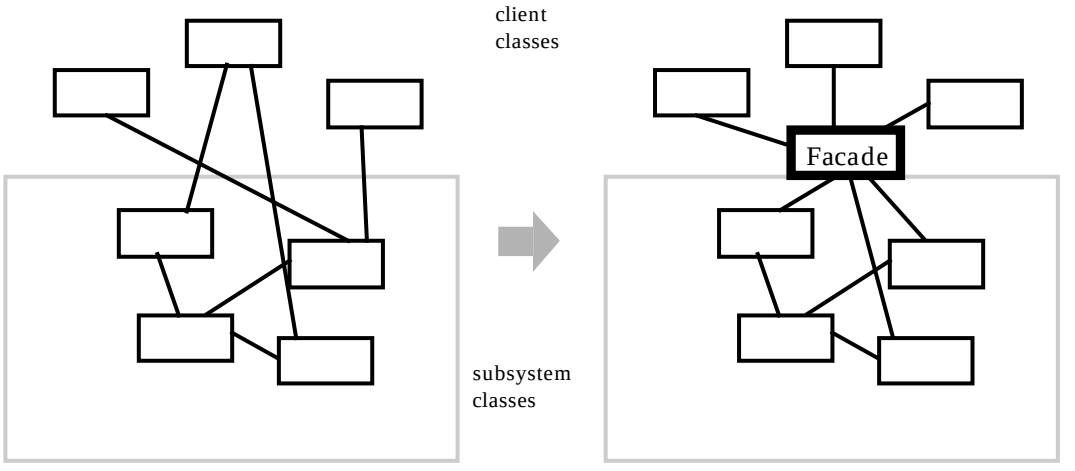
Liabilities

No liabilities are listed in the catalogue.

20. [Gamma+94], Facade(185)

Applicability

I have used the *Facade* pattern in the design of the Service Access Manager Layer. I have also used it in designing the model. The model consists of a set of classes and provides an interface for operation on the represented business model which, in turn, is represented by the model.



[Gamma+94]

Figure 30-8. Structure of the Design Pattern ‘Facade’

30.11 Mediator

The behavioral pattern *Mediator* helps to manage objects from a central point. These objects need only to know about the existence of the mediator and not of any other object which could be influenced by an object’s change. The *Mediator* pattern is listed in the ‘Design Patterns’ catalog[21].

Intent

The *Mediator* pattern defines an object that encapsulates how a set of objects interact. It also promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Problem

The Mediator pattern addresses the design problem of how to design complex object collaboration while avoiding a strong coupling between those objects. For example, pressing a menu entry could result in the disabling of several other menu entries. The menu entry itself should not be aware of all possible changes to be expected and it should not be in the responsibility of the menu entry to change the state of the other entries.

Solution

Use one object to manage the behavior of a set of objects.

21. [Gamma+94]; Mediator(273)

Structure

The *Mediator* pattern consists of three participants. The *Mediator* itself acts as a dialog director and defines an interface for communication with colleague objects.

- The *ConcreteMediator* implements cooperative behavior by coordinating colleague objects.
- The *ConcreteMediator* knows and maintains its colleagues.
- The *Colleague* class, e.g. menu entries, buttons etc., know their *Mediator* object. Each of them communicates with the mediator whenever they would have otherwise communicated with another colleague.

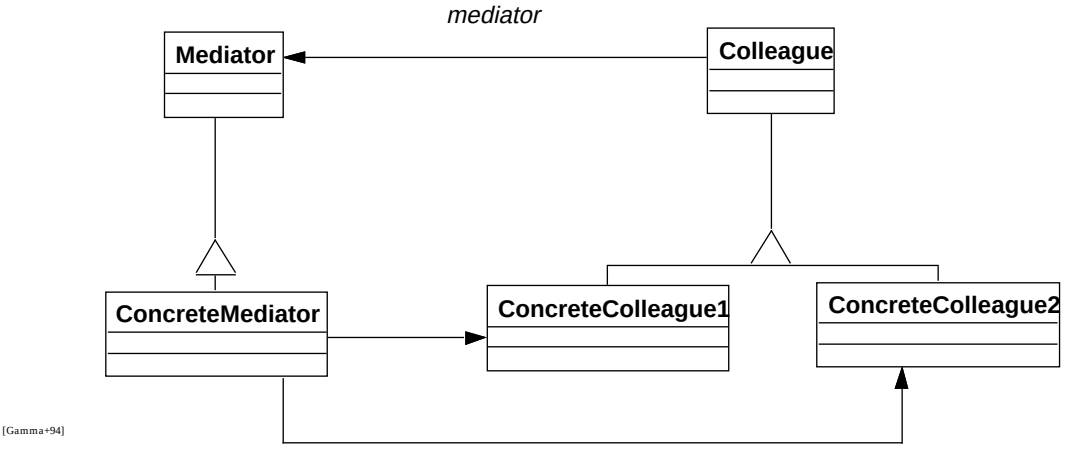


Figure 30-9. Structure of the Design Pattern ‘Mediator’

Benefits

The catalog lists several benefits:

- The *Mediator* pattern limits subclassing by localizing behavior that otherwise would be distributed among several objects. Changing the behavior requires subclassing the *Mediator* only; Colleague classes can be reused as they are.
- The *Mediator* pattern allows loose coupling between colleagues.
- The *Mediator* pattern simplifies object protocols by replacing many-to-many interactions with one-to-many interactions.
- The encapsulation of an interaction between objects into on object can help clarify how objects interact in a system.

Liabilities

- The Mediator pattern centralizes control. When protocols are encapsulated into one object, this object can become more complex than any individual colleague. Subsequently, the mediator itself may become a monolith that is hard to maintain.

Applicability

I used the *Mediator* pattern in the *Terminal Installation* application.

31 A Cookbook for Portable Clients—A Pattern System

As a functional summation of this diploma thesis, this chapter presents a 'cookbook' which describes how to implement a portable client for distributed environments. This cookbook recapitulates the results achieved with this project.

This chapter introduces the pattern system *Portable Client*, which I have based on already existing design patterns. All of the implemented management applications in this project were accomplished using this pattern system.

This cookbook's intension is to present a recipe which can be used for the implementation of further management applications.

31.1 Portable Client

Context

You want to implement an application with a graphical user interface. This application must manage data which is described in a specific business model. The data is located on a particular platform. The application should be scalable to different levels of user experience. Undo and Redo operations should be available.

Problem

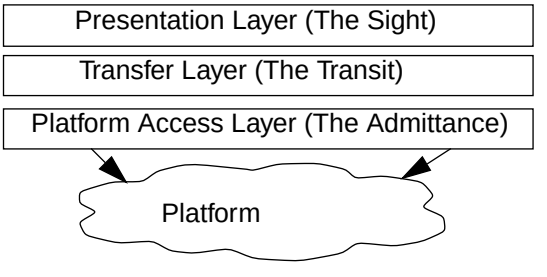
You want to build the application in such a way that it can access an arbitrary amount of platforms.

Solution

Divide your application into three layers:

- **The Sight**: an application layer to present and administer the data.
- **The Admittance**: an application programming interface layer for attaining access to a specific platform.
- **The Transit**: a data transfer layer, which deals as a translator and mediator between the application layer and the application programming interface layer.

Structure



Collaboration

The presentation layer is responsible for presenting the data to the user. It keeps track of the current state of the presented data. It allows the data to be manipulated, to be retrieved from the storage base (the platform), and to be saved in the storage base. The platform access layer knows how to communicate with a specific platform. In this layer is located the application programming interface (API) for a specific platform. The transfer layer hides the complexity of the API from the presentation layer. It marshals the data structure of the API into the format that the presentation layer demands. In reverse, it transforms the data format of the presentation layer into the format of the platform access layer. The transfer layer communicates with the presentation layer with a standardized interface in which the specific functionality of the platform access layer is wrapped.

Consequences

A specific platform and the access to that platform is transparent to the presentation layer. The presentation layer knows only the standardized interface of the transfer layer and is unaware of the functionality of the platform access layer. This allows for the exchanging of access form one platform to an other without affecting the presentation layer.

31.2 The Sight

Context

You want to implement a graphical user interface to represent different aspects of an underlying data base. To present the data, different views should be provided. Different types of users should have different access rights for the modification of data (read only, read and write, create, delete). The graphical user interface should provide a way to undo changes to the data base.

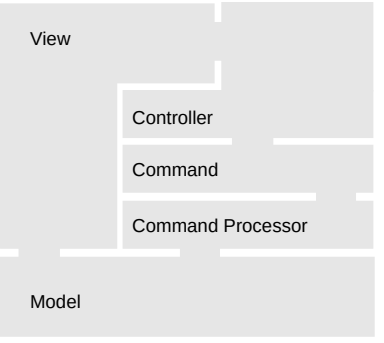
Problem

The data core should be unaware of the views. Changes in the views should not affect the data core. An update mechanism should provide a consistent state of the views regarding changes in the data core.

Solution

To present and administer the data, use the Model-View-Controller pattern. Redefine the Controller in such a way as to allow different views to apply different access rights. To support undo and redo functionalities, use the Command and the Command Processor pattern. Implement the undo and redo functionality in the Command class. To avoid inconsistency, use the Singleton pattern to make sure that only one instance of the model is used by the application.

Structure



Collaboration

The Sight pattern consists of five participants:

- The model maintains the current state of a data entity.
- The view represents the data to a user as retrieved from the model. The view is able to query data directly from the model. A variation on this, could be to implement the controller as a buffer between model and view, so that the view can only communicate with the model by using the controller.
- The controller checks the user input and creates an appropriate command.
- The command processor receives a command from the controller.
- The command processor stores the command for later undo or redo operations and invokes the command's execution (do) operation.

31.3 The Transit

Context

You want to implement an application which uses an application programming interface to access data which is stored on a specific platform. A migration from one platform to another is targeted. Therefore, you need to exchange one API with another.

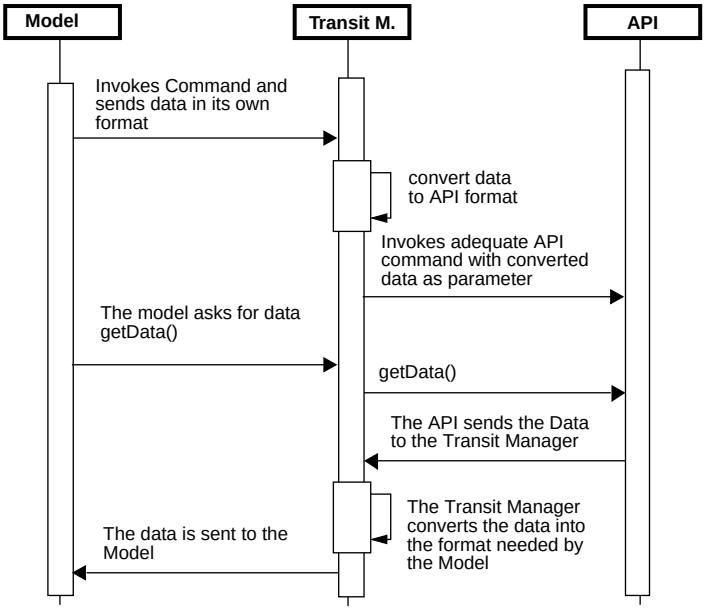
Problem

How to design the application in such a way that neither changes in the API nor the exchange of the API could result in changes in the presentation part of the application.

Solution

Install, between the presentation layer and the API layer, a layer to mediate between both layers. The layer provides a standardized interface towards the presentation layer transferring all data between the two layers and wrapping all API commands of the API with the standardized operations. Use the Facade pattern to wrap the complexity of the API. Use the Factory Method pattern to create different Transit (-Managers) during runtime.

Structure



Collaboration

The model uses a standardized command of the Transit manager, e.g. modify. Such a command usually needs certain parameters, e.g. the data to be modified. The transit manager converts the retrieved parameters into the data format needed by the API. It then invokes the adequate command of the API including the converted data.

Consequences

The usage of the API is transparent to the model. The API can be exchanged without affecting the application layer.

31.4 The Admittance

Context

Provide an access to a system.

Problem

How to hide a complex access to a system.

Solution

Provide an application programming interface.

Also known as

Application Programming Interface (API).

32 Bibliography

TINA-C

Abarca+97 — Abarca, C. et al: TINA-C: **Service Architecture 4.0**, TINA-C Deliverable, TB_RM.001_4.0_96; C. Abarca, P. Farley, J. Forslöw, T. Hamada, P.F. Hansen, H.Hegeman, S.Hogg, H.Kamata, K.Kiwata, L. Kristiansen, C. Licciardi, M.Mampaey, R.Minetti, H.Mulder, S.Pensivy, E.Utsunomiya, M. Yates; TINA-C: 28 October 1996.

Bagley96 — Bagley, Mark: **The Market for Information Services and its demands on TINA-C** (TINA-C Enterprise / Business Model), Version 2.0; TB_MB.001_2.0_95; March 5, 1996.

Berndt+95 — Berndt, H. et al.: **Service Architecture, Version 2.0**, Document No. TB_MDC.012_2.0_94; H. Berndt, C. Kim, S. Kim, H. Kobayashi, R. Minerva, K Ohtsu, J. Pavon, F. Ruano, M. Wakano, H. Yagi; TINA-C, March 1995.

Brown+94 — Brown, Dave and Stefano Montesi: **Requirements upon TINA-C architecture**; TINA Baseline TB_MH.0002_2.0_94; February 17, 1994.

Chapman+95 — Chapman, David and Stefano Montesi: **Overall Concepts and Principles of TINA**; TINA Baseline, Document No. TB_MDC.018_1.0_94; February 17, 1995.

Christensen+95 — Christensen H.:TINA-C: **Information Modeling Concepts**, TINA-C Deliverable, Version 2.0, April 1995.

Farley+97 — Farley, Patrick et.al: **Service Architecture, Version 4.1**; TINA-C Technical Report; TR_PFH.01_4.1_97; January 17, 1997.

Fuente+94 — Fuente, L.A. de la and Tony Walles: **Management Architecture**; TINA Baseline Document No. TB_GN.010_2.0_94; December 1994.

Graubmann+94 — Graubmann, P. et al: TINA-C: **Engineering Modeling Concepts (DPE Architecture)** / P. Graubmann, W. Hwang, M. Kudela, K. MacKinnon, N. Mercouroff, N. Watanabe; TINA Baseline TB_NS.005_2.0_94; December 1994.

Handegård+96 — Handegård, Tom (ed): TINA-C: **Computational Modeling Concepts**, TINA Baseline, TP_HC.012_3.2_96, 17 Mai 1996.

Mulder97 — Mulder, Harm: **TINA-C Glossary of Terms**; Version 2.0; TINA-C Overall Deliverable; BL_HM.001_2.0_970107; January 7; 1997.

Leydekkers+95 — Leydekkers, P. et al: TINA-C: **TINA Distributed Processing Environment (TINA-DPE),** Version 1.0 / P. Leydekkers, K. McKinnon, N. Mercouroff; TINA Stream Deliverable, TB_PL.001_1.0_95, August 2, 1995.

Leydekkers+95a — Leydekkers, P. et al: TINA-C: **TINA Distributed Processing Environment (TINA-DPE)**, Version 1.3 / P. Leydekkers, K. MacKinnon, N. Mercouroff; TINA-C Core Team reviewed; Document No. TR_PL.001_1.3_95; December 21, 1995.

Parhar96 — Parhar, A: **Object Definition Language Manual Version 2.3**, TINA-C Stream Deliverable, TR_NM.002_2.2_96,July 22, 1996.

Abarca+96 Abarca, Chelo et al: TINA-C: **Comments on PCS auxiliary project** *Report Version 1.0 (Draft)* / Chelo Abarca, Hans Hegeman, Lill Kristiansen, E. Utsunomiya; TINA-C Engineering Note, November 1996.

Takita+97 Takita Watura and Takashige Hoshiai (Editors): **Computational Modeling Concepts**, Version 3.3.0; TINA-C Draft Deliverable TDB_WTTH.001_3.3.0_97; 21st February 1997.

Yates+97 Yates, Martin et al: **TINA Business Model and Reference Points** (formerly TINA Reference Points), Version 4.0 / Martin Yates, Wataru Takita, Laurence Demoudem, Rickard Jansson, Harm Mulder; TINA-C Baseline; February 25, 1997.

Java

Aitken96 Aitken, Gary: **Automatically Generating Java Documentation**: javadoc and the doc comment; Dr. Dobb's Journal, July 1996.

Arnold+96 Arnold, Ken; James Gosling: **The Java Programming Language**; Addison-Wesley 1996.

Flanagan96 Flanagan, David: **Java in a Nutshell**: A dEsktop Quick Reference for Java Programmers; O'Reilly & Associates, Inc; 1996.

Geary+97 Geary, David M. and Alan L. McClellan: **Graphic Java: Mastering the AWT**; SunSoft Press; 1997.

Gosling95 Gosling, Arnold: The Java Language Environment: A White Paper; Sun Microsystems Computer Company; October 95.

Lea96 Lea, Doug: **Concurrent Programming in Java**: Design Principles and Patterns–The Java Series; Addison-Wesley Publishing Company; 1997.

Niemeyer+96 Niemeyer, Patrick; Joshua Peck: **Exploring Java**; O'Reilly & Associates, Inc; 1996

Oaks+97 Oaks Scott et.al: Java Threads; O'Reilly & Associates, Inc; 1997.

Papurt+96 Papurt, David M. and Jean Pierre LeJacq: **Design with Java**: Design aspects of the Standard I/O Library; Journal of Object-Oriented Programming; pp 6; November-December 1996.

Wayne96 Wayner, Peter: B**etter Java Programming**; Knowing how Java's dynamic linking works can help you improve a program's performance. BYTE: September 1996; page 63.

Java Coding Style Guidelines

Friendly95or96 Friendly, Lisa: **The Design of Distributed Hyperlinked Programming Documentation**; Sun Microsystems Inc.; no year (approximately 1996).

Lea97 Lea, Doug: **Draft Java Coding Standards**; http://gee.cs.oswego.edu/dl/html/javaCodingStd.html; February 11, 1997.

Naval96 Naval Postgraduate School: **Java Style Guide**; http://dubhe.cc.nps.navy.mil/~java/course/styleguide.html; July 8, 1996.

Sandvik96 Sandvik, Kent: **Java Coding Style Guidelines**; http://reality.sgi.com/sandvik/JavaGuidelines.html; 1996.

Davis97 Davis, Mark: **Java Cookbook**: Porting C++ to Java; Taligent; available as Acrobat File via WWW: http://www.taligent.com/Technology/WhitePapers/PortingPaper/index.html; January 23, 1997.

PCS in TINA

Arbanowski+96 Arbanowski, S. et al: **TINA-C Auxiliary Project: Personal Communications Support in TINA, Report No. 1**, Version 1.0 / S. Arbanowski, T.Eckardt, P. Kielhöfer, T. Magedanz, U. Scholz, S. van der Meer, M. Vetter, H. Wang; GMD FOKUS, Berlin; June 28, 1996.

Eckardt+96a T. Eckardt et al.: **TINA-C Auxiliary Project: Personal Communications Support in TINA, Report No. 2**, Volume 1,Version 1.0; T. Eckardt, A. Guther, L.Hagen, P. Kielhöfer, U. Scholz, H. Wang; GMD FOKUS, Berlin; December 19, 1996.

Arbanowski+96a Arbanowski, S. et al: **TINA-C Auxiliary Project: Personal Communications Support in TINA, Report No. 2, Volume 2: ODL Specifications**, Version 1.0/ S. Arbanowski, P. Kielhöfer, S. van der Meer, H. Wang; GMD Fokus, Berlin; December 19, 1996.

CORBA

Eckert95 Eckert, Klaus-Peter: **The Object Management Group - OMG Concepts, Architectures and Experiences**; Recommendations for the Migration of Multimedia Teleservices to the world of Distributed Object Systems; Release 1.0; BERKOM-Project: “Multimedia Teleservices (MMTS)”; GMD-FOKUS; Deliverable January 31, 1995.

HP95 Hewlett-Packard. HP Distributed Smalltalk: **HP Distributed Smalltalk 4.0**, White Paper, January 1995.

HP95a Hewlett-Packard. HP Distributed Smalltalk: **HP Distributed Smalltalk Release 5.0** Release Notes, October 1995.

HP95b Hewlett-Packard: **HP-DST Programmer’s Reference Guide**, Release 5.0

IONA94 ONA Technologies Ltd.: **Designing and Building Distributed Applications with Orbix and CORBA**, Tutorial Material, 1994.

IONA95 IONA Technologies Ltd.: **The Orbix Architecture**, November 1995.

IONA95a IONA Technologies Ltd.: **Programming Guide Orbix 2**, Release 2.0, November 1995.

Kitson95 Kitson, B.: CORBA and TINA: **The Architectural Relationships**, Proceedings of the TINA’95 Conference, Melbourne, Australia, February 1995.

Mowbray+95 Mowbray, J. Thomas and Ron Zahavi: **The Essential CORBA**: Systems Integration Using Distributed Objects; John Wiley & Sons, Inc; 1995.

OMG:ORB95 Object Management Group (OMG): **The Common Object Request Broker**: *Architecture and Specification*, Revision 2.0, July 1995.

OMG:Services95 Object Management Group (OMG): **CORBAservices***: Common Object Services Specification*; Revised Edition, March 1995, OMG Document Number 95-3-31.

OMG:Mgmt95 Object Management Group (OMG): **Object Management Architecture Guide**, Revision 3.0, June 1995.

OMG:Facilities95 Object Management Group (OMG): **Common Facilities Architecture**; Revision 4.0, OMG Document 95-1-2, January 3, 1995.

Orfali+96 Orfali, Robert; Dan Harkey, Jeri Edwards: **The Essential Distributed Objects Survival Guide**; John Wiley & Sons, Inc.; 1996.

Orfali+97 Orfali Robert and Dan Harkey: **Client/server programming with Java and CORBA**; John Wiley & Sons, Inc.; 1997.

Vinoski97 Vinoski, Steve: CORAB: **Integration Diverse Applications Within Distributed Heterogeneous Environments**; IEEE Communications Magazine, pp 46-55; February 1997.

Visigenic96a Visigenic: **Programmer's Guide Version 1.0**: VisiBroker for Java; 1996.

Visigenic96b Visigenic: **Reference Guide Version 1.0**; VisiBroker for Java; 1996.

TANGRAM

Durmosch+97 Durmosch, M. Khayrat and Klaus-D. Engel: **The TANGRAM DPE: A Distributed Processing Environment in A Heterogeneous CORBA 2 World**; HICSS-30, Hawaii, January 1997.

Egelhaaf96 Egelhaaf, Chr.(Editor): **Recommendations for a CORBA Based Distributed Processing Environment**; Provisional Report; Release 1.0; TINA-C auxiliary project; GMD-FOKUS; March 31, 1996.

Eckert+95 Eckert, Klaus-Peter et al: **Open Distributed Processing Platforms for Support of Telecommunication Applications and their Management** / Klaus-Peter Eckert, Peter Schoo, Gerd Schürmann; GMD-FOKUS; June 1995.

Schoo95 Schoo, Peter (Editor): **Tangram - A Visionary Project on Information Networking**, Release 1.2; BERKOM Project TANGRAM; GMD-FOKUS; September 30, 1995.

Schoo+96 Schoo, Peter and Klaus-Peter Eckert (Editors): BERKOM Project TANGRAM: **Experiences with TINCA-C Results and CORBA 2 Products**; *Deliverable for the 4th milestone of the TANGRAM Project*; Edited by P. Schoo and K.-P.Eckert; Release 1.3; August 31, 1996.

Eckert96 Eckert, Klaus-Peter (Editor): **Specification of the TANGRAM Framework**, *Final Version - Deliverable for MS 2, Release 1.6*, BERKOM Project TANGRAM; GMD-FOKUS; January 1996.

Object-Oriented Programming

Budd91 Budd, Timothy: **An Introduction to object oriented programming**; Addison-Wesley, April 1991.

Booch91 Booch, Grady: **Object oriented design with applications**; The Benjamin/Cummings Publishing Company, Inc.; 1991.

Jacobson+92 Jacobson, Ivar et al: **Object-Oriented Software Engineering**: A Use Case Driven Approach; Addison-Wesley; 1992, reprint from 1995.

Khoshafian+95 Khoshafian, Setrag and Razmik Abnous: **Object Orientation**, Second Edition; John Wiley & sons, Inc.; 1995.

Webster95 Webster, Bruce F.: **Pitfalls of object-oriented development**: a guide for the wary and the enthusiastic; M&T Books; 1995.

Design Patterns

Alexander+77 Alexander, Christopher et al: **A Pattern Language**: Town, Building, Construction / Christopher Alexander, Sara Ishikawa, Murray Silverstein with Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel; New York, Oxford University Press, 1977.

Brown+95 Brown Kyle and Bruce G. Whitenack: Crossing Chasms: **A Pattern Language for Object-RDBMS Integration**; “The Static Patterns”; Technical Journal; available via http://www.ksccary.com/; no year, approximately 1995.

Coplien94 Coplien, James O.: **Software Design Patterns**: Common Questions and Answers; AT&T Bell Laboratories, Software Production Research Department; no year but approximately 1994.

Coplien97 Coplien, James O.: **Idioms and Patterns as Architectural Literature**; IEEE Software; pp 36; January 1997.

Beck+94 Beck, Kent: **Patterns Generate Architectures**; Kent Beck and Ralph Johnson; European Conference for Object Oriented Programming; ftp://st.cs.uiuc.edu/pub/patterns/papers/patterns-generate-archs.ps.

Buschmann+96 Buschmann, Frank et.al: **A System of Patterns**: Pattern-Oriented Software Architecture; Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal; Chichester, New York a.o.: John Wiley & Sons; 1996.

Fowler97 Fowler, Martin: **Analysis Patterns**: Reusable Object Models; Addison-Wesley Publishing Company; 1997.

Gamma+94 Gamma, Erich et al.: **Design Patterns**: elements of reusable object oriented software; Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Addison-Wesley Publishing Company; 1994 (Fifth printing December 1995).

Helm95 Helm Richard: Patterns & Software Design: **Observations on Observer**; Dr. Dobb’s Journal on CD ROM; \DDJ_CD\1995_2.HW4; 1995.

Kerth+97 Kerth, Norman L. and Ward Cunningham: **Using Patterns to Improve Our Architectural Vision**; IEEE Software; pp 53; January 1997.

Krasner+88 Krasner, Glenn E. and Stephen T.Pope: **A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80**; Journal of Object-Oriented-Programming, August/September 1988.

Mellor+97 Mellor, Stephen J and Ralph Johnson: **Why Explore Object Methods, Patterns, and Architectures?**; IEEE Software; pp 27; January 1997.

Mowbray+97 Mowbray, Thomas J. and Raphael Malveau: **Corba design patterns**; John Wiley & Sons, Inc, 1997.

Monroe+97 Monroe, T. Robert et al: **Architectural Styles, Design Patterns, and Objects**; Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan; IEEE Software; pp 43; January 1997.

Nielsen+96 Nielsen, Mark and Nick Abdo: **Applying Design Patterns to PowerBuilder**; The Observer pattern provides a window communication mechanism; Dr. Dobb's Journal, June 1996.

Riehle96 Riehle, Dirk: **Describing and Composing Patterns Using Role Diagrams**; Ubilab, Union Bank of Switzerland; 1996.

Riehle96a Riehle, Dirk: **The Event Notification Pattern–Integrating Implicit Invocation with Object-Orientation**; To be published in Theory and Practice of Object systems 2, 1 (1996); available via ftp from Ubilab, Union bank of Switzerland; 1996.

Riehle+96 Riehle Dirk and Heinz Züllighoven: **Understanding and Using Patterns in Software Development**; To be published in Theory and Practice of Object Systems 2, 1 (1996); available via ftp from Ubilab, Union Bank of Switzerland.

Riehle97 Riehle, Dirk: **A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose**; Ubilab Technical Report 97-1-1; Union Bank of Switzerland; 1997.

Pree95 Pree, Wolfgang: **Design Pattern for Object-Oriented Software Development**; Addison-Wesley Publishing Company; 1995.

Vlissides+96 Vlissides John M. et al. (Editors): **Pattern Languages of Program design 2**; edited by John M. Vlissides, James O. Coplien, Norman L. Kerth; Addison-Wesley Publishing Company; 1996.

Xin96 Shu, Xin: **Fitting Design Patterns into Object-Oriented Methods**; Doctoral Thesis; University of Illinois at Chicago; 1996.

Design Patterns in the Internet

Portland-W3 **Portland Pattern Repository**: www.c2.com / ppr; This is a large collection of Web sites about Design Patterns.

Schmidt-W3 Schmidt, Douglas: **Design Patterns and Pattern Languages**: www.cs.wustl.edu / ~schmidt / patterns.html

UIUC-W3 University of Illinois: **Patterns Homepage**; st-www.cs.uiuc.edu / users / patterns; Another good collection of Web sites about Design Patterns.

Vlissides96-W3 Vlissides John: Pattern Hatching: **To kill a Singleton**; www.sigs.com / publications / docs / cppr / 9606 / cppr9606.c.vlissides.html; 1996.

Modeling and Design

Balzert94 Balzert, Helmut: **Von OOA zu GUIs – das JANUS-System**; OBJEKTspektrum 4 / 94; pp 43; 1994.

D'Souza96 D'Souza, Desmond: **Interfaces, subtypes, and frameworks**; Journal of Object-Oriented Programming; pp19; November-December 1996.

D'Souza97 D'Souza, Desmond: **Collaborations: Behond subtypes**; Journal of Object-Oriented Programming; January 1997.

Jacobson+95 Jacobson, Ivar et al: **Modeling With Use Cases**: Using contracts and use cases to build plugable architectures; Ivar Jacobson, Stefan Bylund, Patrick Jonsson, Staffan Ehneboom; Journal of Object-Oriented Programming; pp 18; March 1995.

Meyer94 Meyer, Bertrand: **Reusable software**: The Base object-oriented component libraries; Prentice Hall International (UK) Limited; 1994.

Rational97a Unified Modeling Language: **Notation Guide**; Version 1.0; Rational Software Corporation; January 13, 1997.

Rumbaugh94 Rumbaugh, James: **Eine Betachtung der Architektur Model-View-Controller (MVC)**; in OBJEKTspektrum 3/94; pp 49; 1994.

Rumbaugh+91 Rumbaugh, James; Micahel Blaha, William Premerlani, Frederick Eddy, William Lorensen: **Object-Oriented Modeling and Design**; Prentice-Hall Inc.; 1991.

Rumbaugh95 Rumbaugh, James: **What is a method?**; Journal of Object-Oriented Programming; pp 10; Vol 8, No 6; October 1995.

Rumbaugh95a Rumbaugh, James: Driving to a solution: **Reification and the art of system design**; Journal of Object-Oriented Programming; pp6, Vol 8, No 4; July/August 1995.

Rumbaugh96a Rumbaugh, James: **Packaging a system**: Showing architectural dependencies; Journal of Object-Oriented Programming; pp11; November-December 1996.

Rumbaugh96b Rumbaugh, James: **Layered additive models**: Design as a process of recording decisions; Journal of Object-Oriented Programming; pp 21; March-April 1996.

Shaw+96 Shaw, Mary and David Garlan: **Software Architecture**: Perspectives on an Emerging Discipline; Prentice Hall; 1996.

Tepfenhart+97 Tepfenhart, William M and James J. Cusick, AT&T: **A Unified Object Topology**; IEEE Software; pp 31; January 1997.

General Programming

Arthur93 Arthur, Lowell Jay: **Improving Software Quality**: An Insider's Guide to TQM; John Wiley & Sons, Inc.; 1993.

McConnell93 McConnell, Steven C: **Code complete**: a practical handbook of software construction; Microsoft Press; 1993.

Maguire93 Maguire, Stephen A.: **Writing Solid Code**: Microsoft's Techniques for Developing Bug-Free C-Program; Microsoft Press; 1993.

Graphical User Interface

Cooper95 Cooper, Alan: **About Face**: The Essentials of User Interface Design; IDG Books Worldwide, Inc.; 1995.

Horton94 Horton, William: Das Icon-Buch: Entwurf und Gestaltung visueller Symbole und Zeichen (American issue: **The Icon book**: visual symbols for computer systems and documentation by John Wiley & Sons, Inc. 1994); Addison-Wesley (Deutschland) GmbH; 1994.

Howlett96 Howlett, Virginia: **Visual Interface Design for Windows:** Effective User Interfaces for Windows 95, Windows NT, and Windows 3.1; John Wiley & Sons, Inc.; 1996.

Weinschenk+95 Weinschenk, Susan and Sarah C. Yeo: **Guidelines for Enterprise-Wide GUI Design**; John Wiley & Sons, Inc; 1995.

Miscellaneous

Kitson95a Kitson, B.: **PLATyTools and ODL**, Proceedings of the TINA'95 Conference, Melbourne, Australia, February 1995.

Magedanz+96 Magedanz, Thomas and Radu Popescu-Zeletin: **Intelligent Networks**; International Thompson Computer Press; 1996.

33 Glossary

Common Object Request Broker Architecture	CORBA—An architecture which specifies a system which provides interoperability between objects in a heterogeneous, distributed environment. Its design is based on the OMG Object Model.
Design Pattern	Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice (Alexander+77 p. X)
Generic Session End Point	GSEP—A service independent computational object which models the minimum set of capabilities as an end-point of an Access Session and (service) session control. As an end-point of Access Session, a GSEP interacts with a User Agent to access services. As an end-point of (service) session control, a GSEP interacts with User Service Session Managers to invoke operations for (service) session control. (Berndt+95 p 6-4).
Hypertext Markup Language	HTML—The document description language used in WWW browsers to display documents
Interface Definition Language	IDL—A declarative interface definition language which is programming language independent. IDL has a syntax resembling that of C++.
Information Specification	An information specification is a description of a structure that models the information in a system (domain of discourse, or problem domain) in terms of information bearing entities, relationships between the entities, and constraints and rules that govern their behavior, including creation and deletion (Christensen+95, p. 2-3).
Information objects	An Information object is an object that occurs in an information specification. Information objects model the basic information entities in an information specification. Henceforth whenever we write ‘object’ without any qualifier, we mean “information object”. Each object has an identity, which is an intrinsic, invariable part of the object. Although two objects are otherwise equal, they are considered as separate objects if they have different identifies (Christensen+95, p. 3-1).
Interoperable Object Reference	IOR—An object reference for objects within a distributed ORB system. CORBA 2.0 defines, that vendors must use IORs to pass object references across heterogeneous ORBs.
javac	A compiler for generating Java binaries
javadoc	A compiler for generating HTML online documentation. To generate online help files, javadoc uses special comments in Java source code files.
Location Computational Object	A Location computational object, within the definitions of the *PCS in TINA* project, is normally a room or a zone with given borders in which a set of terminal is located.
Local Context	LCxt—The Local Context computational object within the definitions of the *PCS in TINA* project describes a set of end user terminals located in a specific location.

Object Reference	An unique name or identification for an object within a distributed ORB System.
Object Request Broker	ORB—commercially known as CORBA; the ORB is the communication heart of the standard. It provides an infrastructure allowing objects to converse, independent of the specific platforms and techniques used to implement the objects. Compliance with the Object Request Broker standard guarantees portability and Interoperability of objects over a network of heterogeneous systems.
Pattern Catalogue	A pattern catalog is collection of related patterns (perhaps only loosely or informally related). It typically subdivides the patterns into at least a small number of broad categories and may include some amount of cross referencing between patterns.
Pattern Language	Pattern languages define the patterns germane to a given domain and the ways in which they should be combined. Ideally, a pattern language shows all of the ways to build all ‘good’ architectures within a domain.
Pattern System	A pattern system is a cohesive set of related patterns which work together to support the construction and evolution of whole architectures. Not only is it organized into related groups and subgroups at multiple levels of granularity, it describes the many interrelationships between the patterns and their groupings and how they may be combined and composed to solve more complex problems.
Registration Server	It provides automatic or manual registration of a person at a location.
Service Access Layer	The Service Access Layer is the layer in which the API functionality is located.
Service Access Management Layer	SAM—The Service Access Management Layer is the layer between the Application Layer and the Service Access Layer. It serves as a mediator between applications and the API.
TANGRAM DPE	The TANGRAM DPE is a TINA compliant platform which was developed at GMD FOKUS to evaluate the TINA Service Architecture.
Terminal Equipment Agent	TE-A—A Terminal Equipment Agent represents a user system within the provider domain. A TE-A maintains information on resource configuration of a user system, e.g. access points, user applications, stream interfaces and Generic Session End-points.
User Application	UAP—A User Application is defined to model a (variety) of service applications(s) in a user system. It acts as an end-point of a Service Session. Zero or more stream interfaces (i.e. end-point of Communications Sessions) can be attached to a UAP. The stream interface can be bound to those in other user systems and/or those in the provider domain (e.g., those attached to video servers) by Communication Session Managers (Berndt+95 p 6-4).
User Agent	UA—A User Agent is a computational object that represents a user in the provider domain. Operations supported by a UA are service independent (Berndt+95 p6-4).

34 Acronyms

AIN	Advanced Intelligent Network
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
AWT	Abstract Window Toolkit
B-ISDN	Broadband-Integrated Service Digital Network
BOA	Basic Object Adapter
CORBA	Common Object Request Broker Architecture
CPE	Customer Premises Equipment
DPE	Distributed Processing Environment
ESIOP	Environment-Specific Internet-ORB Protocol
GIOP	General Inter-ORB Protocol
GMD	German National Research Center for Information Technology
HTML	Hypertext Markup Language
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IN	Intelligent Network
ISDN	Integrated Service Digital Network
JDK	Java Development Kit
LCxt	Local Context
NAP	Network Access Point
NCCE	Native Computing and Communication Environment
ODL	Object Definition Language
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
PCS	Personal Communications Support
PPrf	Personal Profile
PSTN	Public Switched Telephone Network

TCP/IP	Transmission Control Protocol / Internet Protocol
TE-A	Terminal Equipment Agent
TINA	Telecommunications Information Networking Architecture
TINA-C	Telecommunications Information Networking Architecture Consortium
TMN	Telecommunication Management Network
UA	User Agent
UCxt	Usage Context
UPT	Universal Personal Telecommunications
USM	User Session Management
WWW	World Wide Web

35 Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W