Services in Pervasive Computing Environments: from Design to Delivery

par

Davide Carboni

thèse présentée au Département d'informatique

en vue de l'obtention du grade de docteur ès sciences (Ph.D.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, janvier 2005

Pagination irrégulière au début, mais ok tel quel.

III-1627

Summary

The work presented in this thesis is based on the assumption that modern computer technologies are already potentially pervasive: CPUs are embedded in any sort of device; RAM and storage memory of a modern PDA is comparable to those of a ten years ago Unix workstation; Wi-Fi, GPRS, UMTS are leveraging the development of the wireless Internet. Nevertheless, computing is not pervasive because we do not have a clear conceptual model of the pervasive computer and we have not tools, methodologies, and middleware to write and to seamlessly deliver at once services over a multitude of heterogeneous devices and different delivery contexts. Our thesis addresses these issues starting from the analysis of forces in a pervasive computing environment: user mobility, user profile, user position, and device profile. The conceptual model, or metaphor, we use to drive our work is to consider the environment as surrounded by a multitude of services and objects and devices as the communicating gates between the real world and the virtual dimension of pervasive computing around us. Our thesis is thus built upon three main "pillars". The first pillar is an domain-object-driven methodology which allows developer to abstract from low level details of the final delivery platform, and provides the user with the ability to access services in a multi-channel way. The rationale is that domain objects are self-contained pieces of software able to represent data and to compute functions and procedures. Our approach fills the gap between users and domain objects building an appropriate user interface which is both adapted to the domain object and to the end user device. As example, we present how to design, implement and deliver an electronic mail application over various platforms.

The second pillar of this thesis analyzes in more details the forces that make direct object manipulation inadequate in a pervasive context. These forces are the user profile, the device profile, the context of use, and the combinatorial explosion of domain objects. From the analysis of the electronic mail application presented as example, we notice that according to the end user device, or according to particular circumstances during the access to the service (for instance if the user access the service by the interactive TV while he is having his breakfast) some functionalities are not compulsory and do not fit an adequate task sequence. So we decided to make task models explicit in the design of a service and to integrate the capability to automatically generate user interfaces for domain objects with the formal definition of task models adapted to the final delivery context.

Finally, the third pillar of our thesis is about the lifecycle of services in a pervasive computing environment. Our solutions are based upon an existing framework, the Jini connection technology, and enrich this framework with new services and architectures for the deployment and discovery of services, for the user session management, and for the management of offline agents.

Acknowledgments

Now, the time has come for the acknowledgements[1]. This means that I made it, or almost. Over the last two and a half years, I felt alternatively enthusiastic and discouraged, happy and irritated, involved and apathetic, determined and unsure, satisfied and disappointed, tired and regenerated. At last I do not know whether I'll feel happy and releived, but I'll certainaly consider myself lucky. Lucky for having had the opportunity to undertake this modest adventure which took me back and forth between Canada and Sardinia, which gave me the opportunity to meet many new people, to know better people I already knew, which gave me the opportuntity to receive the support of my colleagues, of my friends, of my family, my parents and my in-laws.

Many thanks to Sylvain Giroux. Everything started on the day I met Sylvain while he was getting ready to leave Sardinia to teach in a Canadian University. I was thinking that it was a pity that he should leave and that it would be nice to continue collaborating with him, perhaps through a doctorate. I did not have to formulate my thought: he offered me to prepare a PhD under his supervision. This spontaneous demonstration of trust has always been one of the major factor which sustained me in this adventure and a powerful incentive to go ahead. Thank you Sylvain, not only for guiding me through scientific and technical choices but also for treating me on an equal footing, generously receiving me in his family, and sharing his life experience with me.

[1] A special thank you to Helga Wilson for having reviewed my use of English in this section.

Thank you to Pietro Zanarini and Gavino Paddeu because going back and forth between Canada and Sardinia for this doctorate would have been much more difficult without the support of CRS4, support that they obtained and sustained.

Thank you to Stefano Sanna and Andrea Piras, for their friendship and also for their great help in implementing with me important parts of the architectures presented in this thesis.

Thank you to Claude Moulin whom I have had the great pleasure to work with and who has been a key person in the development of the E-Mate project.

Thank you to Marc Frappier, one of the most friendly and kind persons I met during my brief experience at the Sherbrooke University, for having accepted to review this dissertation.

Thank you to Jean François Perrot, whom I had the great plausure to meet at Crs4, for having accepted to review this dissertation.

Thanks also to all the CRS4 colleagues involved in the E-Mate project, fighting against an instable code, with ill-working libraries and undefined requisites. Thanks to them because in the most intense moments of the E-Mate project, I had the rare and precious feeling of being part of a team.

Thanks to Angela for her patience and her daily support, for her catchy optimism, in a few words for her love.

At last, thanks to Riccardo, born during the first year of my PhD and who grants us a great happiness each time he runs, laughs, speaks and does all those things children do that remind us not to be afraid of the future.

Ringraziamenti

Sono arrivato ai ringraziamenti e questo vuol dire che ce l'ho fatta, o quasi. In questi ultimi due anni e mezzo mi sono sentito entusiasta e scoraggiato, contento e irritato, impegnato e indolente, determinato e dubbioso, soddisfatto e deluso, affaticato e rigenerato. Alla fine non so se mi sentirò felice e sollevato ma sicuramente mi riterrò fortunato. Fortunato per aver avuto l'opportunità di vivere questa piccola avventura che mi ha portato avanti ed indietro tra il Canada e la Sardegna, nella quale ho conosciuto tante nuove persone, ho conosciuto meglio le persone che già conoscevo e ho ricevuto l'appoggio dei miei colleghi, dei miei amici, della mia famiglia, dei miei genitori e dei miei suoceri.

Grazie a Sylvain Giroux. Tutto iniziò il giorno che incontrai Sylvain mentre si preparava a lasciare la Sardegna per il Canada ad insegnare all'università. Pensai con dispiacere alla sua partenza e che sarebbe stato bello continuare a collaborare con lui, magari tramite un dottorato. Non feci in tempo a sviluppare l'idea che lui, precedendo i miei pensieri, mi propose appunto di iniziare un dottorato sotto la sua supervisione. Questa manifestazione di fiducia spontanea è stata sempre uno degli stimoli più importanti per andare avanti in questa avventura. Grazie a Sylvain non solo per avermi guidato nelle scelte tecniche e scientifiche, ma anche per avermi trattato alla pari, generosamente, aprendomi la sua casa, la sua famiglia e la sua esperienza di vita.

Grazie a Pietro Zanarini e a Gavino Paddeu, perché obiettivamente questo dottorato tra Canada e Sardegna sarebbe stato molto più difficile da percorrere senza l'appoggio del CRS4 che loro hanno ottenuto e difeso. Grazie a Stefano Sanna e ad Andrea Piras. Amici soprattutto ma anche bravissimi colleghi che hanno implementato insieme a me una parte importante delle architetture presentate in questa tesi.

Grazie a Claude Moulin con il quale ho avuto il piacere di lavorare e che è stato uno dei pilastri del nostro gruppo al Crs4 durante il progetto E-Mate.

Grazie a Marc Frappier, una delle persone più amichevoli e gentili che ho conosciuto nella mia breve esperienza canadese, per aver accettato di far parte del jury.

Grazie a Jean François Perrot, che ho avuto il grande piacere di incontrare e conoscere al Crs4, per aver accettato di far parte del jury.

Grazie a tutti gli altri colleghi del CRS4 che hanno lavorato agli scenari E-Mate, spesso lottando con codice instabile, librerie mal-funzionanti e requisiti poco chiari. Grazie soprattutto perché nei momenti più intensi del progetto E-Mate ho avuto la rara e bella sensazione di far parte di un gruppo.

Grazie ad Angela per la sua pazienza, per il suo sostegno quotidiano, per l'ottimismo che mi ha trasmesso. In una parola per il suo amore.

Infine, grazie a Riccardo che è nato durante il primo anno del mio dottorato e che oggi ci regala immensa gioia ogni volta che corre, che ride, che parla e che fa tutte quelle cose che i bambini fanno per ricordarci che non bisogna mai avere paura del futuro.

List of Abbreviations

- API: Application Programming Interface. An interface defining methods/function signatures that allows a client code to invoke functionalities on another piece of code. The client has normally no access to the underlying implementation of the invoked functionalities. An API builds an abstraction that new programs can use to develop and extend existing programs.

- CORBA: Common Object Request Broker Architecture. Distributed architecture where object communication is mediated by a system called broker. It allows to extend the object oriented paradigm to distributed system and in multi-platform environments.

- EJB: Enterprise Java Beans. A multi-tier architecure for enterprise applications based on Java. Applications are deployed in servers which provide system services such as security and transaction management.

- HCI: Human Computer Interaction.

- HTTP: Hyper Text Transfer Protocol[3] It is the protocol designed and implemented in the early days of the Web which specifies requests and transmission of HTML documents between a Web client and a Web server.

- IDE: Integrated Development Environment. A program aimed at developers which integrates a suite of tools such as code editors, build tools, repository access, debugging mode, library management, online documentation, class browsers and so forth. Examples of IDEs are: Eclipse, IDEA, Netbeans.

- J2ME: Java 2 Micro Edition. The Java platform for devices with small footprint. It comprises a subset of the standard Java API and a new API for user interfaces, network protocols, and application lifecycle.

- MORE: Multi-channel Object REnderer. It is the system that automatically generates user interfaces for domain objects both adapted to the current platform and to the domain object currently viewed by the user.

- PLANES. Prototyping LANguage for Embedded Systems. A simple task models definition tool used to generate user interface code for J2ME applications. The core of PLANES has been extended to be integrated with MORE for the multi-channel generation of user interfaces over multiple platforms.

- POTS: Plain Old Telephony System. A traditional wired phone line that uses analogic signal transmission between the user phone and the network exchange, differently from the modern ISDN which is a digital subscriber line with two 64Kbps channels for voice and data communications.

- RMI: Remote Method Interface. The mechanism used in the Java Virtual Machine to implement an object oriented remote procedure system. Differently from CORBA, both clients and servers must be written in Java.

- SMS: Short Message Service. A relaying service for text based messages of maximum 160 characters availble in GSM networks.

- WAP: Wireless Application Protocol. A protocol stack developed by mobile phone vendors to deliver minimalist Web services to mobile phones.

- WIMP: Windows Icons Mouse Pointing. Abbreviation used in HCI literature to indicate a diffuse paradigm for graphical user interfaces

Table of Contents

List of Figures

Introduction

The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.[71]

0.1 Mobile, Personalized and Location-Based Services

Hitherto, personal computing has been confined in a unique paradigm of interaction: the user sitting in front of his screen eventually connected to the Internet by means of a cable. The invention of laptop/palmtop computers and the development of wireless data connections remove some technological constraints and leverage the use of computers in domains different from the traditional office productivity such as business and leisure on-the-road.

Nevertheless, hardware and network technologies are not sufficient for computing to scale up to a new level of full interoperability of devices, programs, and data. There is the need for organization, system architectures, and metaphors in order to make computing aware of the user profile, mobility, and user location.

0.1.1 Personalization

Personalization enables the system to provide the right information at the right time both in synchronous manner - i.e. when the user is connected and browsing between services - and in asynchronous manner, i.e. some agents keep searching any relevant data while the user is disconnected.

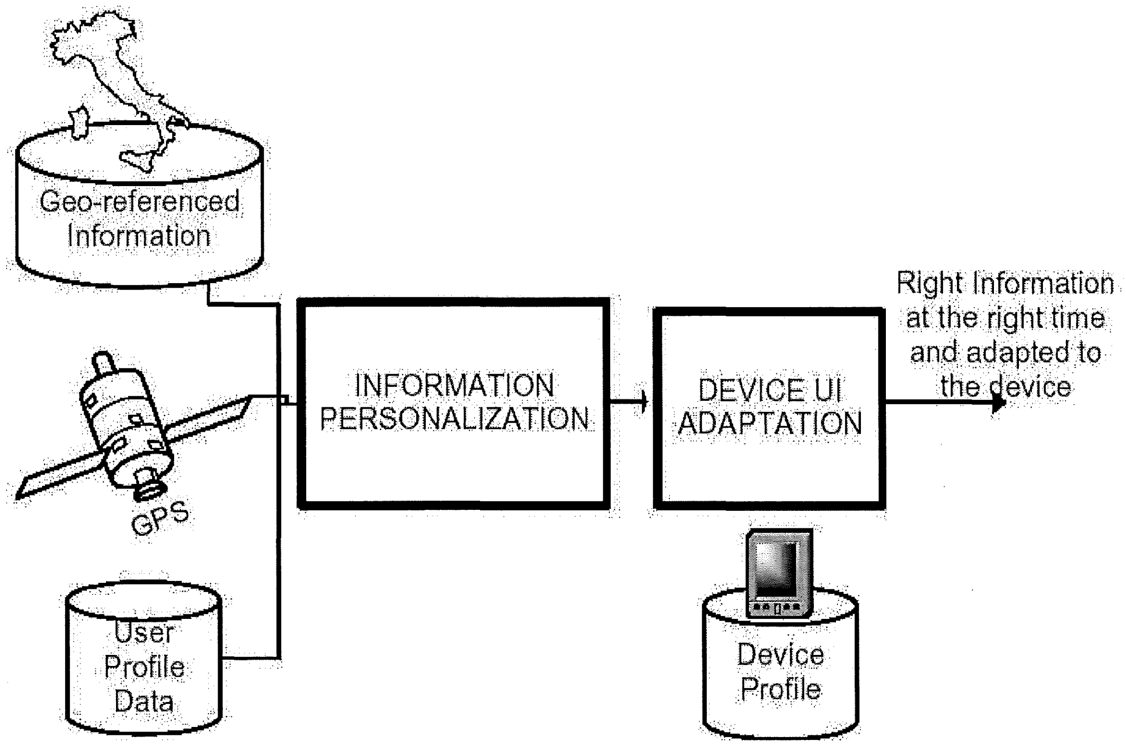

Geo-referenced Information
GPS
User Profile Data
INFORMATION PERSONALIZATION
DEVICE UI ADAPTATION
Device Profile
Right Information at the right time and adapted to the device

Figure 1: Information and device UI personalization process

The personalization process (Figure 1) is a two steps process. First, it integrates information from sources as diverse as localization technologies, user models, and GIS systems. Then, the personalized information must be presented in such a way that the human computer interacion is the easiest for the user. Thus, the personalization process must be followed by a device-adaptation process based on the device-profile of the user terminal.

User terminals may differ considerably from each other: some of them can load and run mobile code from the network, other can be programmed to communicate with online services exchanging data but not code, and many of them cannot be programmed but can access to information by means of pre-installed micro-browser or via voice-interfaces.

A wide variety of applications can better serve users if they adapt to user wants and needs. The key to enable the adaptation and personalization of services are the *user models.* User models can be successfully applied in recommendation systems [57], adaptive information retrieval systems [6] and systems for coaching/teaching users.

0.1.2 Mobility

In a pervasive computing environment the user will continuously switch from connected to disconnected states and may use a wide variety of heterogeneous devices. The mobility of users implies that services and information systems should be available from a number of different channels. The same information should be accessible from a Web page, from a WAP site or from an Interactive Voice Responder (IVR) machine. Besides, a given channel can provide many interaction modalities. For instance, a Windows-Icons-Mouse-Pointing (WIMP) desktop application can greatly improve usability by providing text-to-speech and a voice recognizer system for text dictation.

With respect to these implications, a service supporting mobility of users should be designed in a way such that the following principle will stand: design once, move anywhere. Nevertheless, multi-modality and multi-channel access are only the first step toward the result. In fact, even though a service is available for any device, the user can feel the interaction with the system rather uncomfortable. This occurs when the network is too slow, scarcely reliable and disconnections frequently occur. Moreover, some tasks can take long before their completion and waiting on-line for the results is useless and expensive. In such a situation the user experiences the system as a hostile environment to work with.

To improve user-system interaction, one should be able to interact directly with the service by means of an appropriate user interface in synchronous manner and to obtain immediately a result. On the other hand, the user would appreciate the possibility to schedule some kind of work, i.e. a search on the web, and to retrieve the results of such a job in a later time. The latter is an asynchronous way to interact with the service in which the user engages one or more agents to perform some tasks while the user is off-line.

0.1.3 User Position

The user position is an important parameter in computing. In fact, this parameter can be used to provide the right information at the right moment by location aware systems [52]. The precision required may vary according to the objectives. In fact, if we are designing an agent aimed to search cultural events such as concerts or football matches, it is important to screen out all events that will not be held in the city where the user is. So the precision required is at city-level. But if the agent task is to notify the user whenever his position is nearby a monument or a museum the precision required should be about 100 meters.

0.2 Ubiquitous Computing at PARC

Marc Weiser is the author of the seminal paper *The computer of 21st century* [71] and is worldwide considered as the father of Ubiquitous Computing. According to his definition Ubiquitous Computing is the third wave in computing after the main-frame era and the invention of the personal computer.

The main purpose of Ubiquitous Computing is to go beyond the way people have used computers till now. Today, a personal computer is basically a box which completely absorbs attention distracting the user from the surrounding environment. If one thinks of the computer as a tool to accomplish a given job, most of the time is spent trying to using the tool rather than solving the problem.

On the other hand, a "profound" technology is one that does something useful without drawing user's attention too much. In terms of metaphors, people currently think of personal computers as boxes containing a desktop. Such a metaphor is probably inadequate for computers of the future, and new metaphors should be devised. Weiser says that a good tool is often invisible [70], not in a physical meaning, but in the sense that it does not draw user's attention letting people focus on the problem. An example of such a tool are the eyeglasses, one use them to look around and not to look at the eyeglasses.

The Ubiquitous Computing era is still to come, but today technology has probably the means to realize what only ten years ago appeared to be visions of a science fiction writer. Today, most people use a computer at home and another at work, computers are easier to use and not only for experts as in the past, moreover beside personal computers, most people own palm size computers, laptop computer and programmable cellular phones. The problem here is that all these devices need a minimal set of configuration, data need to be kept synchronized among several devices, batteries need to be recharged, and different devices need to interact with each other through a large number of possible interfaces. The desktop metaphor still resists and we have many mini-desktop to administrate. Once again, the tools we use draw part of our energies and of our attention, the more is complex the technology the larger is the amount of time we spend to dominate such a complexity: this is not a "pervasive technology", but rather a "invasive technology".

According to [71] the Ubiquitous computer must have three characteristics:

- pervasiveness, it must be composed by a big number of hardware and software elements cooperating each other and small enough to be embedded in every day objects or in the human body;

- mobility, users must be able to move when using the computer, thus some elements of the Ubiquitous computer must move with the user;
- invisibility, the computer need to be invisible in order to draw our attention as less as possible.

Of course, pioneers researcher in Ubiquitous Computing were at PARC: Weiser and his colleagues. Their implementation consisted in a set of applications and devices to be used in work group environments. They built small palm sized computers to be used instead of notes and post-it, tablet computers to be used instead of paper and notebook, and board computers to be used instead of traditional writing boards (Figure 2). The device was in a infrared network. Their results was more or less satisfactory, they got a certain degree of pervasiveness, a certain degree of invisibility and mobility.

0.3 Metaphor: Devices are Portals

A device is a portal into an application/data space, not a repository of custom software managed by the user. An application is a means by which a user performs a task, not a piece of software that is written to exploit a device's capabilities.[2]

Computer technology is already a pervasive technology: processors, memories, displays and communication networks are massively deployed around us. Twenty years ago the only CPU in our house was probably the one inside our Commodore 64 or Sinclair Spectrum. Today one is likely to use a PC at home and one at office, a laptop or/and a PDA, a mobile phone with Java and so forth. Besides, domestic appliances have evolved: washing machines have embedded processors, the interactive digital TV is a new channel to access on-line services, a photo camera can build a Web site in its internal memory card, and so forth. There is rather a lack of ubiquity in software applications: a processor

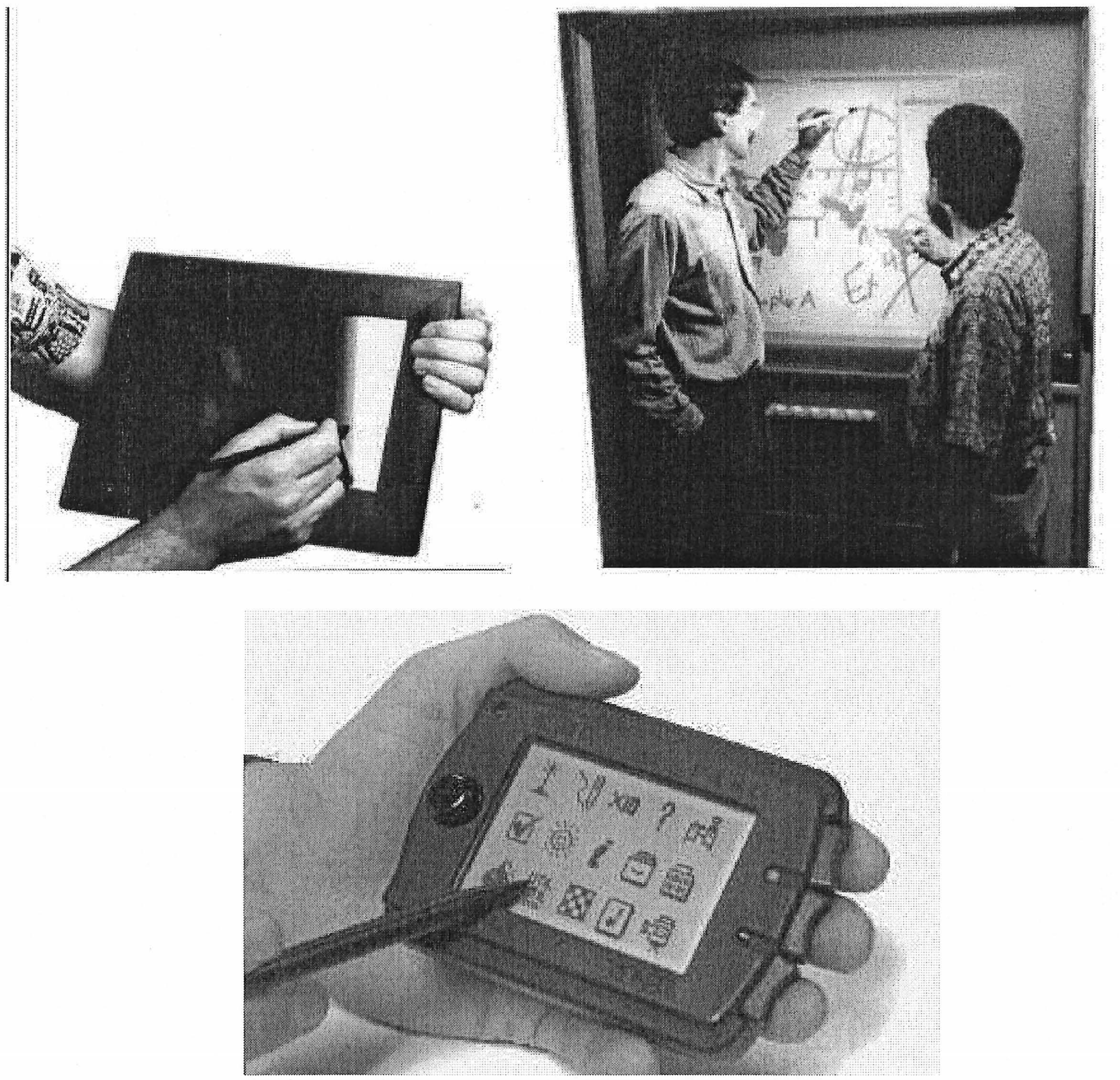


Figure 2: Tablet, board, and palm devices built at PARC
(Pictures from http://www.ubiq.com/hypertext/weiser/UbiHome.html).

is dedicated to a specific goal, and the whole set of computers around us do not perform as one personal, ubiquitous and inter-operable computing environment. In other words, there is a lack of organization and synergy.

It is desirable to evolve from an environment where dozens of computers around us have their own small computing context, to an environment where all computing devices are simply gates to a shared computing space (Figure 3). Interactions with computers, where computers are no longer boxes but gates to a pervasive computation, can be both explicit: the user sends a command and waits for feedback; and implicit, input/output devices disseminated in the real world gather information about our habits by means of sensors and such information are used to infer new behaviors and new tasks to accomplish. One must no longer think of computers as computing devices containing data and programs, but rather as part of an environment where users live and move around.

The old hype slogan "The Net is the Computer"[60] could be rewritten as "The World is the Computer". Thus, the Ubiquitous computer components, such as software and hardware, must follow users in move. Actual computing devices will simply become temporary hosts for whole or parts of Ubiquitous applications. The end user must be allowed to start a given task in a device and seamlessly migrate its work to another as soon he finds more serviceable to change. Thus, applications must be available from several channels such as web browsers, mobile phones and old Plain Old Telephone System (POTS) via voice interaction.

Applications enhance physical surroundings giving the user the perception to move in an empowered environment: the computing environment is the user's information-enhanced physical surroundings, not a virtual space that exists to store and run software [2]. So, virtual spaces can coincide with physical spaces.

According to this vision, phones, printers, and sensors become simply applications and services available from any device the user wants to use like a terminal. Moreover, the surroundings can provide more powerful terminals to access ambient applications - for instance, a broadband-wired connection instead of a wireless narrow-band one. Thus, terminals should be considered as any other services: they are subject to discovery and their availability should be notified whenever the task carried out with the actual device requires them. These requirements emphasize, among the others, the concept of position and the techniques to get the user position inside and outside buildings: *Ubiquitous computers must know where they are* [71].

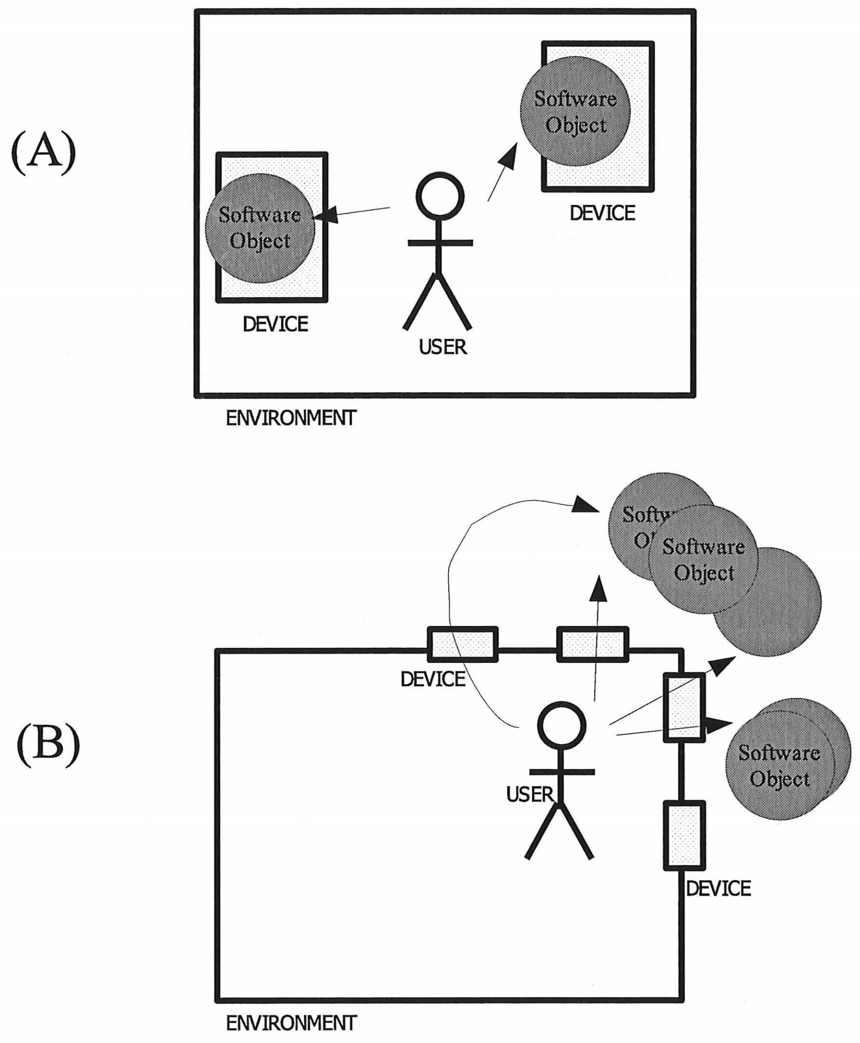


Figure 3: (A) Computing devices today: devices contain data and programs. (B) Devices in computing-pervaded environment are gates to another dimension: the dimension of computing where software objects live and are accessed through devices in the gate metaphor.

0.4 Challenges

Building applications in computing-pervaded environments requires efforts at various levels [56]:

- wireless connectivity as indispensable support to obtain the mobility;
- network protocols able to handle user mobility seamlessly, the actual IP and TCP were implemented for a network model in which nodes were always connected and they could be inadequate to handle billions of mobile elements;
- mobile code and interactive elements able to adapt themselves to the device. Porting user interfaces code from a platform to another is hard, there are problems related both to the physical capabilities of devices (the presence of a keyboard and a mouse, screen size etc.) and to computational capabilities (yet there is no means to get code that can run on any device, even if it is written in Java);

0.5 Structure and Contribution of this Thesis

This thesis is built on three pillars addressing three main issues:

- how to provide user interfaces enabling users to manipulate objects from any device?
- how to provide a means to synchronize and constrain the interactions between a user and many objcets?
- how to deliver services to the "gates" of the pervasive space?

- **Chapter 1** addresses multi-channel delivery from the viewpoint of user interfaces. First, it describes an object-driven methodology for the design of application objects that allows programmers to disregard low-level details about the delivery of application to the end-users. The chapter proceeds with the details of the MORE framework which supports this methodology then we describe step-by-step the implementation of an example application and its deployment on different platforms: more precisely from a desktop computer to a voice browser[2];

- **Chapter 2** addresses the limits in object-driven interaction in the context of pervasive computing, then it proposes solutions to go beyond such limits enabling constrained interactions with many objects to perform a specific task. PLANES implements a framework to do so and an example is presented.

- **Chapter 3** focus is on service life-cycle. It contains an analysis of deployment, discovery and delivery issues. Then architectures to address these issues are presented: the deployment architecture deals with how a new service becomes available in the environment; discovery deals with how users can locate services; finally, delivery deals with how services become accessible from the end user device.

- **Chapter 4** describes how the results of this work have been inserted in larger research project called E-Mate, whose objective is the realization of a framework for the design, deployment and delivery of services which are platform independent, geo-referenced and user-tailored. This chapter sketches four applications used as test beds of the whole architecture: a ubiquitous travel assistant (Section 4.1), a mobile lesson management and deployment system (Section 4.2), a crisis management system (Section 4.4), and finally an investor decision support system called MKTS (Section 4.5).

- **Chapter 5** compares our results to related works on Ubiquitous Computing, interaction models and design methodologies for pervasive computing;

[2]The term *voice browser* includes a range of systems which allow a voice-based interaction such as Interactive Voice Responders (IVR), answering machines, and VoiceXML enabled browsers

- **Chapter 6** closes this work sketching promising future research directions.

0.5.1 Topics Beyond the Scope of this Dissertation

This work covers only few of the challenges reported in Section 0.4. Thus, all software, methods and solutions proposed here assume the availability of appropriate bandwidth, appropriate network protocols and appropriate distributed systems and frameworks for the construction of network distributed applications.

This work does not try to introduce new metaphors for blurring computers in the background and get them "invisible" to the user. This work only tries to implement the metaphor "device are portals" described in Section0.3 The focus of this work is on explicit user-computer interaction and not on unconscious interaction where user needs are anticipated by collecting data and inferencing habits. This research topic is investigated and addressed in other research projects, for instance in *Intelligent Habitats* [49] at Sherbrooke University.

This work does not deal with issues related to security of data, privacy of users, and ergonomics and usability of produced systems.

0.5.2 Code Contribution

The main code contributions related to this thesis are:

- MORE: a multi-channel Object REnderer system which generates views for Java objects allowing direct method invocation and object manipulation in several platforms such as Java Swing, Web pages, WAP pages, Voice XML browsers and J2ME phones.
- PLANES: a graphical tool and a language specification for task models definition.

- Libraries for shared data structures, service lookup, and application life-cycle management in application servers.

Chapter 1

Domain Objects and User Interfaces

Nowadays information systems have to spread on a wide variety of devices. Networks have enabled their deployment on very heterogeneous devices. Thus the design and implementation of interactive software in a platform neutral way have become a central concern. The application should be deployable in any device even those unknown at design time. The main issue is then how to generate user interfaces at runtime adapted to the end user device. Our solution relies on an object-driven design of the application domain based on some coding conventions. The reflective capabilities of the language are used to retrieve, from the code, any relevant information useful to build on-the-fly a user interface for objects of the application domain. Thus, one needs to focus only on the business logic while the user interface will be generated when needed.

As users are freed from technology concerns, designers should be able to design software abstracting from any technology details and to focus only on application domain concepts and their relations.

Some frameworks address general aspects of distributed applications development but none addresses aspects specific to pervasive computing environments. For instance, the Enterprise Java Beans (EJB) framework [63] aims to build applications by composing reusable software components which share a common runtime environment called "application server". The EJB architecture faces the management of transactions and provides data persistency. Although EJB have proven their efficiency in the development of back-end systems for many commercial Web applications, they do not address the issues related to mobility of users and adaptation to the end user device.

In general programmers must still deal with the details of the final delivery of their applications. Current Integrated Development Environments (IDE) aim to build applications whose delivery context is known a priori. For example, IBM Visual Age makes available a number of tools such as Java libraries, user interface visual composition, and stubs generation for Remote Method Invocation [59], but it assumes that the deployment will take place in a desktop environment or in a enterprise server. IDEs like Visual Age have not been devised for the development of applications for the J2ME profile, so the programmer must not use classes unloadable in small devices.

Although Java is aimed to be a platform-independent language the number of devices able to host a Java Virtual Machine is really small. Therefore, the slogan "Write once run anywhere" seems too optimistic, fails in its literal intent, and is meaningful only for classes of devices: an application written for the Java Standard Edition (J2SE) can be executed on Windows2K, Linux, or MacOS but cannot run on a palm-top or on a mobile phone. Thus, there is a need for languages and environments for device independent application development.

If we look under the hood of a software application we can notice that some parts such as the data/object model and the functions can be considered device neutral, while other parts such as the user interface are strictly device dependent. Abstracting the design from the device means allowing developers to focus solely on the device neutral part of the whole, spending the minimum effort for device dependent details. This chapter shows how to free the programmer from issues related to user interfaces for applications delivered in heterogeneous devices. The main idea is to extract information from the code of the application and to generate a user interface according to the device used.

In Section 1.1 we present the rationale of our approach, in Section 1.2 we show how reflection is useful for interacting with domain objects, in Section 1.3 we show, with some examples, how to write domain objects and the resulting user interfaces, and finally in Section 1.4 we describe the details of domain objects introspection and multi-channel user interface rendering.

1.1 An Object-Driven Approach

Objects are building blocks able to react and therefore to interact. Thus, if a system provides a viewing/controlling mechanism to automatically fill the gap between the users and the application objects allowing the direct interaction between them, and if this bridge is implemented for any type of terminal used by the user, then we can put in practice a design based solely on the definition of domain related objects and their relations.

Our first contribution is the definition of an object-driven approach for developing interactive applications in a platform-independent and device-independent manner. In Section 1.3 and Section 1.4 the details of such a mechanism and how it is implemented in Java are described.

The rationale behind this approach [8] is based upon the following considerations: any application refers to a set of domain object classes. Further on in the text, such objects are called "models", borrowing the naming from the object oriented design pattern Observer [24] based on the "Model View Control" [37] architecture. Examples of models for an application in the tourism domain are: *person, address, calendar, travel, event, etc.* Among these classes there are several relations such as: *which addresses are relevant to a given person; an event is something which happens in a given place and at a given moment; an appointment is a subclass of event which involves two or more persons and so forth.* In the object oriented paradigm any object is an instance of a given class[1]

[1]The concept of class has been introduced by Simula [16] and reimplemented by Smalltalk [35],

Objects are reactive entities which are activated whenever their methods are invoked. The reaction of an activated object is to produce a result object and, as a side effect, to change its internal state.

1.2 Interacting with Objects

In real world, interacting with an unknown object requires us to perform an inspection using our senses. This may be also true for software objects instantiated and running in a computer process.

Our goal is to allow users to interact with the domain objects which are in general not known beforehand. Thus, it is necessary to explore the domain objects in order to understand how they are done and what they can do for the user. Then, a user interface must be provided to allow users to manipulate and interact with such domain objects.

Objects are units of code loaded and running in the context of a computer program. If such a program wants to inspect how objects are made and wants to interact with their methods, then the program must be a reflective system.

1.2.1 Reflection in Programming Languages

A reflective system is a system which incorporates a self-representation. This self-representation makes it possible for the system to answer questions about itself and

and expresses the behavior of a set of objects which share the same semantics operating on the same attributes.

A class defines the internal structure and the behavior of its instances. The internal structure is defined by their instance variables and by their instance methods. Not all object oriented languages are built around the concept of class. In some languages such as ACT1 [39] and ThingLab [5] new objects are created cloning existing individuals and classes are not a language construct but only the result of grouping objects with similar characteristics. Further on, it is assumed for simplicity that an object-oriented language is a language where classes are a construct and where the concept of data type is uniformed to the concept of class. Notice that this last statement is not true for many languages. For instance, in Java or C++ data of type `int` are not instances of any class. To address this lack of uniformity, the Java API provides the class `Integer` which is a wrapper for a data of type int. Similar wrapper classes are available for other primitive types.

support actions on itself [40]. In other words, it performs a computation on a program.

The definition above does not assume a particular programming paradigm: reflective systems can be written in procedure-oriented, function-oriented, or rule-oriented languages. Nonetheless, object-oriented languages have been a rich[2] field of experimentation for reflective systems. Smalltalk80 [26] introduced the concept of meta-class and the unification of the concept of class and object: any class is an instance of a given meta-class, in this case meta-classes are the core of the self-representation for reflective systems. They allow the self-analysis of a system and any dynamic self-modification.

Further improvements towards complete reflection in programming languages are achieved in PLASMA [58], and OBJVLISP[14] where any element of the language is an object: classes, meta-classes, methods, variables and messages are all objects. This helps to remove boundaries between reflective code and application code.

1.2.2 Exploiting Reflection

The runtime generation of a graphical user interface can be described as follows: a class unknown at compile time, is loaded and instantiated. This instance is the object with whom the user wants to interact. Another object in this system performs an inspection of the first object in order to extract all the information relevant to the interaction: the names of the fields, the names of the methods, the type of the fields and the signature of the methods. Among other things, a third object which is the graphical user interface is instantiated by virtue of the information gathered from the object inspection.

The first object is the model, the second one acts as inspector and the third is the view (Figure 4). The system is reflective according to the definition because it collects information on itself, more precisely on a part of it, and performs some actions according to this information.

[2]Lisp has been the privileged language to experiment reflection however.

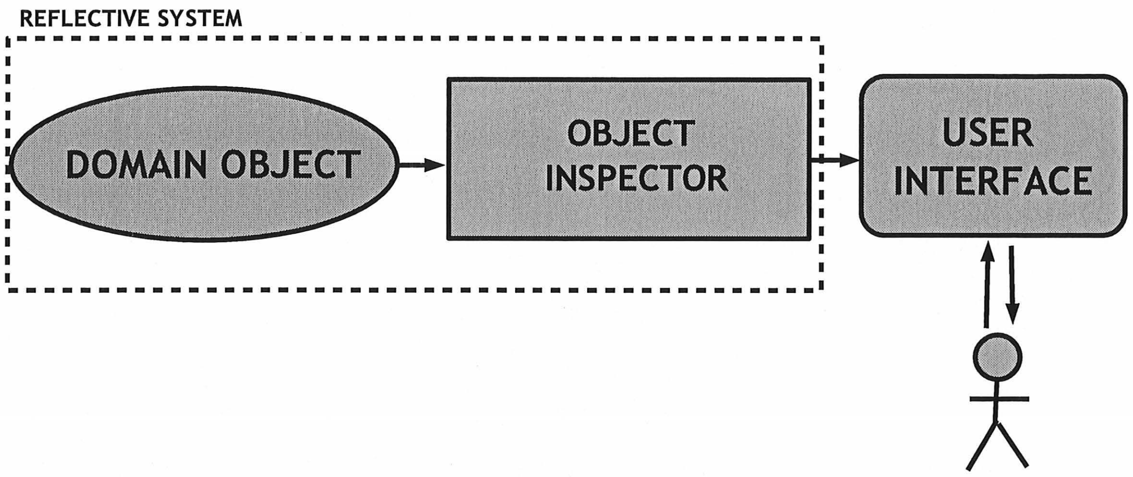


Figure 4: A reflective system used to inspect domain objects and generate a graphical user interface

1.2.3 Class Structure Requirements

It is not possible to generate an effective user interface for an arbitrary object without a set of rules and naming conventions. Rules and naming conventions [28] are the grammar and the syntax for a system to gather a minimal amount of information from an object. For instance, it can be inferred that an object encapsulates a field called `balance` of type `Number` by observing that the object has a method `Number getBalance()`. This is a straightforward consequence of a naming convention followed by both who writes the class and who inspects the code.

Many languages are provided with tools for the easy introspection of the running code. This thesis refers in all cases, unless declared differently, to the Java programming language. Although Java was not the first language to implement reflective tools, nor its reflective capabilities are the most interesting or the most developed, Java is the language choosen for the implementation of the code related to this thesis.

The reasons for the choice of Java can be summarized as follows:

- **general relevance reasons**

    - availability of best commercial and open source development environment
    - availability of testing and project management tools
    - huge availability of open source libraries for a large range of problems
    - code portability in different operating systems
    - excellent documentation
    - unified framework for application development
    - better performances if compared to fully interpreted languages such as Smalltalk, Python [64], et al.

- **reasons relevant to distributed systems and pervasive computing**
    - availability of a remote methods invocation (RMI) model
    - availability of a connection technology for the discovery, lookup and deployment of objects in a network (JINI)
    - availability of XML parsers and other XML related libraries
    - availability of Java Virtual Machine for small and embedded devices (J2ME) and personal digital assistants (Personal Java)
    - mobility of the code which allows objects to be serialized and deserialized in different running virtual machines with remote class loading performed via HTTP
    - robust and scalable security model based on policies
    - interface to legacy and non Java system via Java Native Interface (JNI).

In Java, the state of an object is the set of values of its instance variables that are normally encapsulated in the object and therefore not accessible. Nevertheless, such variables may often be accessed by means of specific access methods, called "accessors", when such methods exist. Accordin to naming conventions, accessors have the form: `xtype getX()`and `void setX(xtype newX)`. The former returns the actual value of the variable x of type `xtype`, the latter assigns `newX` as new value for this variable. In some cases, a variable can be accessed even if no accessors are provided. Moreover, acccess to instance variables may also be precisely controlled: in a Java class, a variable can be declared with private access, and therefore accessible only from the object code; with package protected access, which extends the access to any class belonging to the same package; with protected access, which extends the previous access to any other object whose class is a subclass; and finally with public access, which is extended to any other object. Further on, it is assumed that an object has a variable x if there exists an accessor `getX()` which returns a `xtype` object, and that a variable x is writable if there exists an accessor `setX(xtype newX)`. From a client-code point of view it is not relevant whether the result of `getX()` is the value of a real variable or it is computed on the fly. In both cases, for the client code it is the value of an internal property of the object.

1.2.4 Reflection in Java

The Java Reflection API is a part of the core definition of Java. It implements the following functionalities for reflective code:

- construct new class instances and new arrays;
- access and modify fields of objects and classes;
- invoke methods on objects and classes;
- access and modify elements of arrays;

Such functionalities are exploited in sophisticated applications that need to discover at runtime methods and fields of an object. Among these applications, there are object inspectors, interpreters and class browsers. The Java Reflection API is affected by some important limitations. In fact, although classes such as `Class`, `Field`, `Constructor`, `Array`, `Method` and `Modifier` are defined to manipulate the metalevel concepts, the metalevel is not modifiable by user code. The only metaclass is the class `Class` which is declared `final` and thus cannot be extended.

```
public final class Class
extends Object
implements Serializable

public Object invoke(Object obj,
                     Object[] args)
              throws IllegalAccessException,
                     IllegalArgumentException,
                     InvocationTargetException
```

Given that neither `Class` nor `Method` can be extended, the user code cannot change the core method invocation mechanism. In other words, the Java Reflection API allows user code to inspect and to use dynamic invocation but it does not allow to modify the default implementation of such mechanisms. The reasons behind this design are related to security.

The constraints in Java reflection described here have no serious impact on the implementation of the code related to this dissertation. The simple "passive" reflection is sufficient for our requirements.

1.2.5 The Javabeans API

The Java Development Kit includes a library called the "Javabeans API" [28]. A Javabean is a class which conforms against a given set of rules and naming conventions but, apart from that, it is a simple user-defined class and it does not extend any special-purpose system class.

The Javabean framework has beed devised to define classes that can be manipulated by automatic tools, for instance, builder tools that compose Javabeans at design-time and runtime environments for Javabeans instances.

An automatic builder tool is able to infer from a Javabean code field names, field types, method signatures, and the events fired by the bean. In this way, a builder tool can generate the code of an application by the composition of a certain number of Javabeans.

Thus on the one hand, the Javabeans API allows systems to be flexible enough to gather information about the intimate structure of running object instances in order to invoke methods, read/write fields, and to listen to events; on the other hand, the Javabeans API allows programmers to specify additional information for reflective systems by means of special purpose classes called BeanInfo(s).

For instance, if one wants a precise enumeration order of methods for a given class, then the bean developer can implement a BeanInfo class where the enumeration of methods is explicitely coded and not dependent to the actual implementation of the Java Virtual Machine.

The code related to this dissertation largely exploits the reflective capabilities of the Java programming language and of the Javabeans API for the implementation of a reflective system for the automatic generation of user interfaces called MORE (Multiplatform Object REnderer).

1.3 Overview of the Automatic User Interface Generation

MORE builds on-the-fly a graphical user interface for any given object. Widgets such as text fields, radio buttons, and more sophisticated controls, are deployed as placeholders for the data they refer to: strings, numbers, boolean and more complex data types like object arrays and collections. Complex models are viewed as composite graphical objects and the rendering process is applied recursively till atomic interactors are obtained. The process of inspecting composite models to generate a graphical user interface can be summarized by the algorithm described in the Figure 5.

```
Render(Domain Object)  --> View

Create a container for the actual platform
//e.g. a JFrame in Java Swing toolkit.
//e.g. a Web page in HTML
//e.g. a form in J2ME

For each "displayable" field of the model object:
  Look-up for a run-time editor for the field.
  If found: add the editor to the container.
  Else: add a link (e.g. a hypertext link, a button, etc. )
        to the field

  For each method of the object:
    Add a trigger, e.g. a button, to execute the
    corresponding method.
Return the container
```

Figure 5: The skeleton of rendering algorithm implemented in MORE.

A run-time editor is a component aimed to write/read the value of an object field. For instance, if our model contains a field of type `java.lang.String`, the above algorithm looks up a run-time editor for such a type, that will likely be a text field graphic component. To foster the development of new applications, MORE provides run-time editors for main basic Java types: `java.lang.String, java.util.Number, java.lang.Boolean`, and `java.util.Calendar`. Besides, it provides run-time editors for composite types like `java.util.Collection`, and `java.lang.Object[]`. For those types, the editor allow users to select, display, modify, remove, and add elements to the collection or to the object array. Finally, during the development of multi-media and geo-referenced applications [4], emerged the need for some new Java types: `Situated` and `MultimediaResource`.

`Situated` is a Java interface that provides information about the physical position of an object either in a Cartesian format (latitude and longitude) or in a topological format (city, street, number, etc.). A `Situated[]` component is displayed as a map centered on the barycentre[3] of the array, in which the user can zoom in, zoom out, select an object and display it in a separate view. The `MultimediaResource` type is basically a wrapper to a multimedia file; supported types are JPEG, GIF, MP3, wav, and mpeg.

1.3.1 Example 1: A Trivial Calculator Machine

To provide a better grasp of how generating a user onterface on-the-fly for a given object, we use a trivial calculator machine as illustration. The calculator is implemented in Java and it is able to perform basic numerical operations (Figure 6).

The following code snippet computes 5*6 using this object:

```
TrivialCalc c=new TrivialCalc();
c.setA(new Double(5.0));
c.setB(new Double(6.0));
Double result=c.multiplication();
```

[3]The barycentre of the points defined by geographic coordinates $(lat_1, lon_1), (lat_2, lon_2), ..., (lat_n, lon_n)$.

In general, a human programmer easily figures out how to use a given object by the structure of its code. Similarly, a reflective system which loads a TrivialCalc instance, infers the existence of two numerical fields, `A` and `B`, and of two operations `sum` and `multiplication`, since Javabeans coding conventions are respected. After this simple analysis, it is possible to build a user interface for inserting numerical values and for invoking operations. Figure 7 shows the graphical user interface generated on-the-fly for a PC.

```
package crs4.more.testobjects;

public class TrivialCalc {
    Double a=new Double(0.0);
    Double b=new Double(0.0);

    public Double getA() {
        return a;
    }

    public void setA(Double a) {
        this.a = a;
    }

    public Double getB() {
        return b;
    }

    public void setB(Double b) {
        this.b = b;
    }

    public Double sum(){
        return new Double(a.doubleValue() + b.doubleValue());
    }

    public Double multiplication(){
        return new Double(a.doubleValue() * b.doubleValue());
    }

    public String toString() {
        return "I'm a trivial calc";
    }
}
```

Figure 6: Java code for the TrivialCalc example

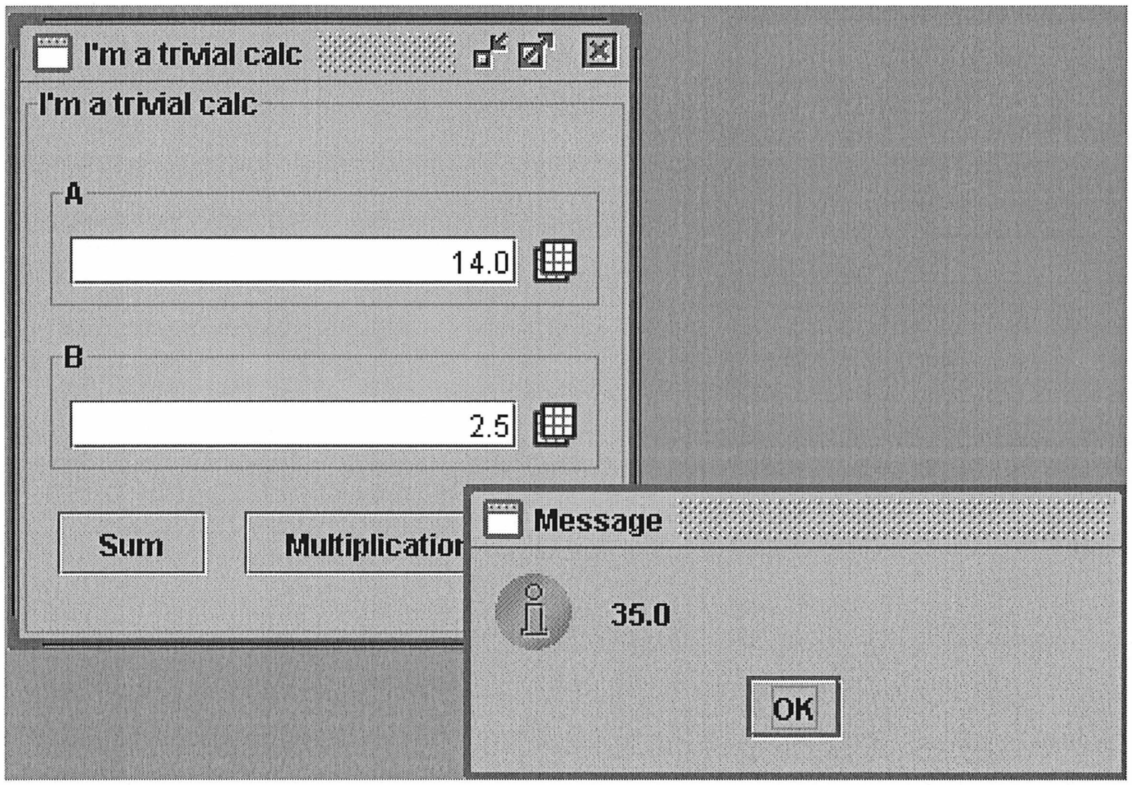


Figure 7: Graphical user interface generated on-the-fly for a TrivialCalc object with Java Swing widgets.

1.3.2 Coding Rules and Legacy Systems

The `TrivialCalc` code (Figure 6) is not the most intuitive that a programmer could write. In fact, a client program would expect methods with parameters:

```
TrivialCalc c=new TrivialCalc();                //(1)
Double result=c.multiplication(5,6);            //(2)
```

Furthermore, the parameters makes the internal registers A and B inside the object useless. Then in this case, instantiation (1) is also useless, and in fact a waste of space. The code could be redesigned to be fully procedural and class methods used instead of instance methods:

```
Double result=TrivialCalc.multiplication(5,6);
```

From the code above one can raise the question whether or not it is possible to generate a user interface from the sole knowledge of the signature of a static method with parameters. The answer is yes for a code as simple as the example. But more in general, given arbitrary legacy code written with no naming conventions, no fixed idioms, and no style rules, is it possible to write a reflective system able to provide the user with a proper user interface in order to effectively interact with the code? We do not answer this, but we suppose that the cost of implementing such a system would be really big[4]. On the other hand, a more pragmatic approach is to rule code-writing with few simple guidelines. The resulting code becomes easily interactive in a reflective system with only a reasonable burden put on programmers' shoulders.

The rules MORE is relying on are the following:

[4]There are some commercial products which provide a semi-automatic integration of legacy systems such as IHC [33]. In most cases the integration of legacy system is a manual task which requires the design and implementation of a middleware to wrap the legacy system to be incorporated in a Web based service architecture.

- If a field X is expected to be visible to the user then write an read accessor `public getX()`
- If a field X is expected to be writable by the user then write a write accessor `public setX(XType x)`
- If a method is expected to be directly invoked by the user then the method must be public and without parameters

The last rule sets a strong constraint, but the example of TrivialCalc shows that the state of an object can be used instead of parameters to perform the computation. If we remove the last rule, at the moment of the invocation of a method the system should ask the user to insert the parameter values. This task is quite simple for strings or numerical values but could be more complicated for object references.

As final consideration, legacy code could be wrapped inside a well-written code in order to become directly manipulable by the user.

1.3.3 Example 2: The Pocket Calculator

The `TrivialCalc` (Figure 1.3.1) does not fit the usual mental model of pocket calculators. We can easily write a class `Calc` (Appendix A) which is a closer metaphor for real world pocket calculators. It contains a field `display` of type `String`, ten methods (one for each digit) which append the corresponding digit to the display, one method for each operator, and one to show the result.

The class `Calc` is not a user interface, but rather a model for which it is possible to attach a view generated in an automatic manner by a rendering mechanism. This model of calculator performs the same interactions as real world calculators does, with no use of graphical components and thus, implementing the interactions with platform independent code. Figure 8 shows the pocket calculator in various platforms.

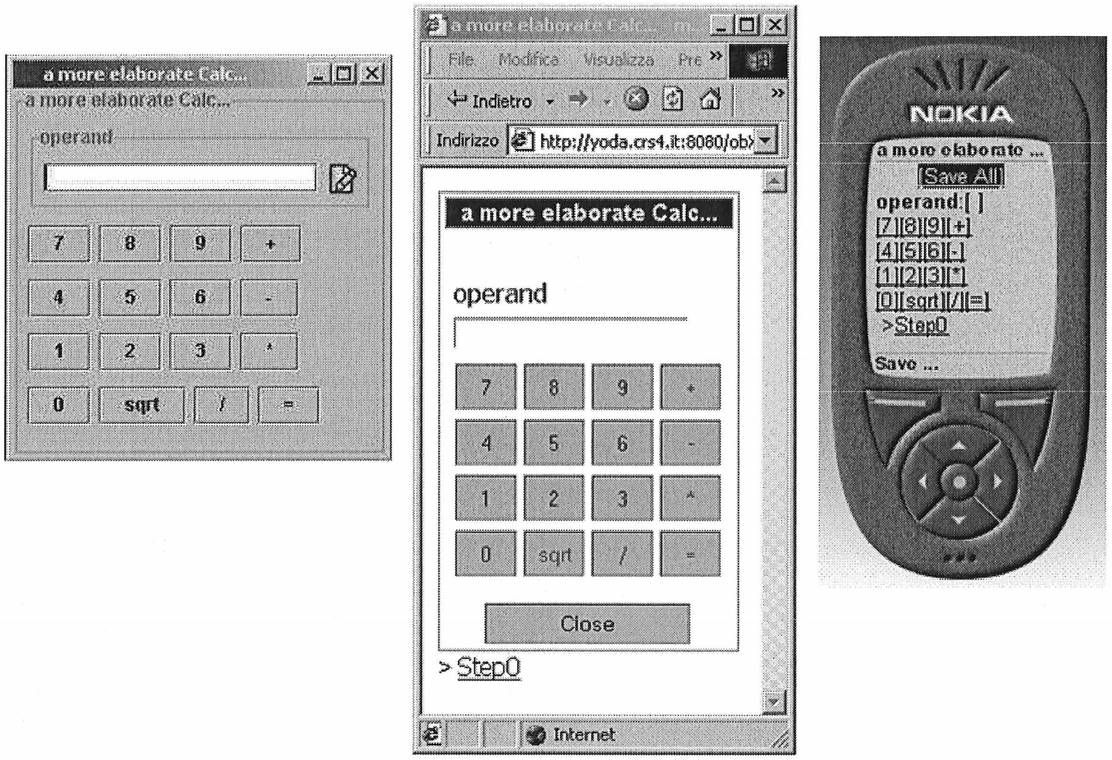


Figure 8: Three views automatically generated by the object renderer from the calculator machine model. From left to right: a Swing view, an HTML view, a WAP view.

The client program using a `Calc` instance to compute the sum $34 + 33$ would be as follows:

```
Calc c=new Calc();
c._3_();
c._4_();
c.sum();
c._3_();
c._3_();
```

```
c.result();
String result=c.getOperand();
```

It is like a macro where any single instruction corresponds to an action on the actual user interface.

1.3.4 Delivery Architecture

Application delivery is a fundamental aspect in the life cycle of a software application. The complete description of deployment, discovery, lookup and delivery of software objects in a pervasive computing environment is detailed in Chapter 3. In this section only the delivery concepts strongly related to the automatic generation of user interfaces are introduced.

At this stage, we make a rough classification of devices according to the respective location of models and views. The architecture where models and views are both instance running in the same virtual machine is denoted as Fat Client; the architecture where models are instances running in a remote host and views are instances running in the user device is denoted as Thin Client; the architecture where both models and views are instances running in a remote host and the user device is simply a browser for HTML, WML, VoiceXML documents is denoted as HTTP-only Client.

Next section describes the implementations of MORE for these three architectures: Fat Client, Thin Client, e HTTP-only Client.

1.4 MORE: the Rendering Engine

This section describes the main elements of MORE. Further on, the following terminology is used:

- *the user* is the human who interacts with objects by means of appropriate user interfaces generated by MORE.

- *the developer* is the human who writes the domain-objects (also called *models*), and other classes relevant in the application such as preconditions, layouts and mates which will be explained later in the text.

- *the system* is the set of classes performing models introspection, user interfaces generation and interaction events handling. The word "system" here does not include all other mechanisms such as deployment, discovery, and delivery needed to handle the entire lifecycle of applications (see Chapter 3 for details).

  The classes and the concepts introducted here are: `Model` (Section 1.4.1), `Session` (Section 1.4.2), `AbstractView` (Section 1.4.3), `Mate` (Section 1.4.4), and `RuntimeEditor` (Section 1.4.5).

1.4.1 Model

Models are the domain objects involved in the interaction with users at runtime. They are self-contained bunches of data and code. Their code does not deal with user interface aspects. The code is organized in methods returning values which can be either `void`, other models or basic types values such as strings, dates, numbers and so forth.

A model class is not required to extend some framework superclass, it simply must conform to coding conventions explained in Section 1:

- If a field X is expected to be visible to the user then write an read accessor `public XType getX()`

- If a field X is expected to be writable by the user then write an accessor `public setX(XType x)`

- If a method is expected to be directly invoked by the user then make the method public and without parameters[5].

Model classes are neither required to extend any framework class nor to fire any kind of framework event. This is a vital feature to avoid event handling code in domain object classes.

1.4.2 Session

In this section the term session is used to denote the interaction session. More precisely a session is instantiated in the end user device when the user starts interacting with models. Such session is transient and not persistent over devices and connections[6].

The functionalities provided by the interaction session are:

- to create views and controllers for a given model;
- to maintain a model-view mapping;
- to maintain a register of instantiated models;
- to allow user to instantiate new models or to retrieve already existing models;
- to allow developers to register in the session the classes the user will instantiate;
- to allow the developer to associate specialized mates to domain object classes;

A session often takes the form of a desktop with a toolbar for creating new instances of registered model classes. The instances are showed to the user by means of views. Figure 9 shows a session in the JVM J2SE platform where user interfaces are made of Swing components.

[5]In Section 1.3.2 we explain why methods must be parameter-less.

[6]It should be not confused with the session described in Section 3 which is called User Session and is a server-side container which stores user related objects when the user is offline.

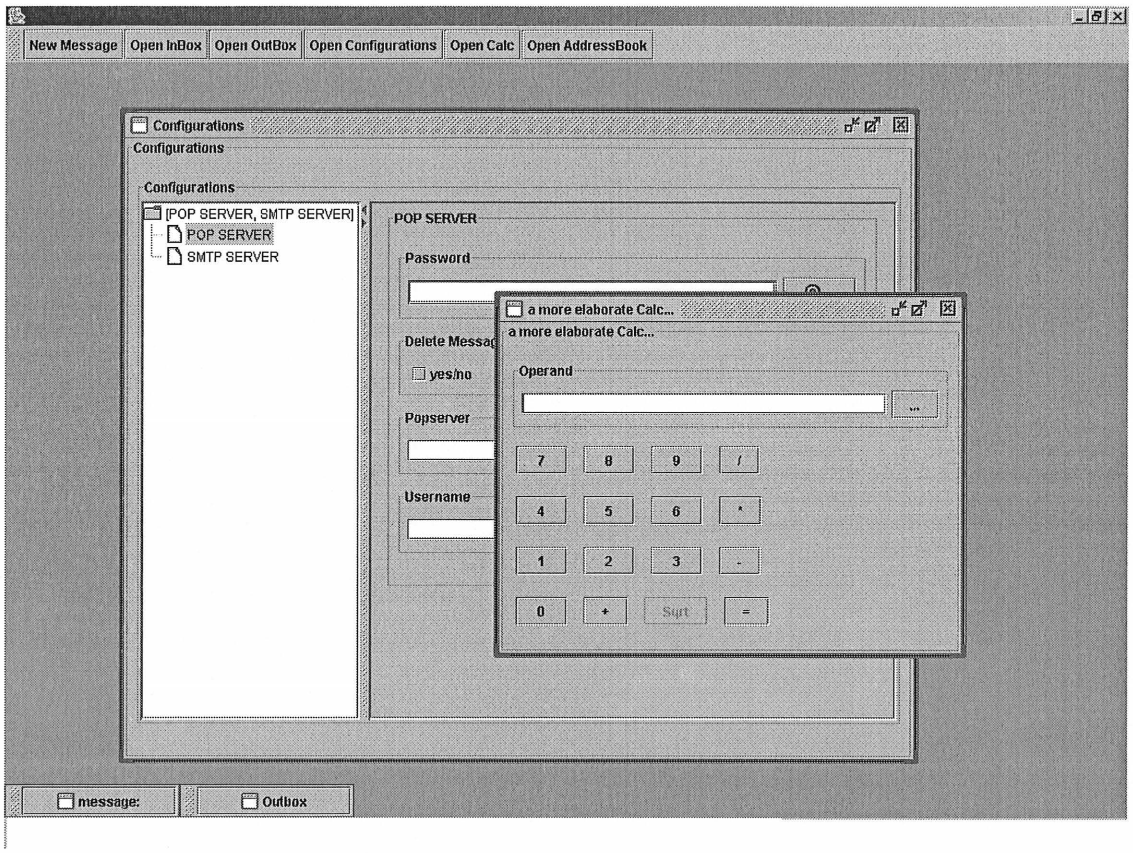


Figure 9: A session displayed with the Java Swing library

1.4.3 Abstract View

MORE is built on top of a core implementing basic features common to all possible delivery context and user platforms. For instance, the algorithm in Section 1.3 is platform-independent and its implementation relies on method invocation on abstract classes.

The `AbstractView` class defines and implements the rendering algorithm in the `render()` method which is a template [24]. It is up to subclasses to implement the platform-dependent steps inside the `render()` operation. The appropriate subclass is loaded depending on the information collected from the delivery context. The rendering algorithm is reported in the Figure 10

The `AbstractView` is a container of user interface components which are independent from GUI toolkits or mark-ip languages. At runtime the concrete view is finally instantiated from the most appropriate library for the actual delivery context. Concrete components are deployed for displaying fields and methods of the model. The `AbstractView` is a system object so application developers are not expected to extend this class. System implementations are already provided for Java Swing, J2ME, AWT and markup languages clients such as Web browsers, VoiceXML browsers and WAP phones. New implementations of this class are necessary only in the case one wants to port the system to a new platform, but in this case it is the system developer task and not the application developer one.

Usually, when speaking of MVC, views are expected to be components belonging to a given library such as Swing, SWT, or AWT in the case of Java. Such components delimitate a portion of the screen and contain other components such as panels, buttons, labels and so forth. In terms of design, this is exactly a Composite pattern where containers and components are defined and where the following relations are assumed: a container is a component; a container contains zero, one or more components. In the case of an platform independent MVC implementation, one cannot assume at design time that components belong to a given library, in fact at runtime they could be implemented in Swing, AWT, WAP, HTML or other component families. Thus the AbstractView is a graphical container abstraction where one can add new abstract components placed according to a given layout. Whether or not this component will appear as a desktop frame, as a Web page, or as a Wap card is a fact depending of the actual Delivery Context (Section 3.3).

```
1:    public void render() throws Exception {
2:        String last = null;
3:        Mate mate = getSession().getMate(getModel());
4:        Iterator sequence = mate.getLayout().layoutIterator();
5:        while (sequence.hasNext()) {
6:            String memberName = (String) sequence.next();
7:            if (mate.isVariable(memberName)) {
8:                Object value = mate.getValue(memberName);
9:                if (value != null &&
10:                     mate.isRenderableFeature(memberName)) {
11:                     renderFeature(value,
12:                                mate.getDescriptor(memberName));
13:                     last = memberName;
14:                 }
15:            } else if (mate.isMethod(memberName) &&
16:                     mate.isRenderableFeature(memberName)) {
17:                 renderOperation
18:             (mate.getDescriptor(memberName).getOperationPane());
19:                 last = memberName;
20:            } else if (memberName == Layout.NEWLINE) {
21:                 if (mate.isVariable(last))
22:                     getFieldsPane().addNewLine();
23:                 else if (mate.isMethod(last))24:
24:                     getOperationPane().addNewLine();
25:            }
26:        }
27:
28:         getOperationPane().addNewLine();
29:    }
```

Figure 10: The method render in the `AbstractView` class. In line 3 we obtain an instance of `Mate` which contains all meta-information about a model. In line 4 we get a sequence of member names and from line 5 to line 25 they are laid out in the AbstractView by means of either `renderFeature()` or `renderOperation()` depending on we are rendering a field or a method. The features of class `Layout` are described in E

1.4.4 Mate

In MVC the view keeps a reference to the model for reading model status and invoke model methods. This is a straightforward architecture when views are tailored by hand for a given model. In the case of automatic generation of views some issues arise:

- if an application requires a given method to be invoked only if a pre-condition is met where is defined the pre-condition? Who evaluates the pre-condition at runtime?
- If an application requires an object to be displayed with a specific layout who stores the layout information?
- If the view requires a field to be displayed with a phrase like "Please, insert your PIN" the developer should define a field with a name "pleaseInsertYourPIN" which is not exactly an solution. Alternatively, who can store alias for methods names and field names?
- Last, assuming that method invocation or value setting must be somehow filtered for a special purpose in your application, who perform as filter?

The solution is to place a mediator (called *mate*) between the view and the model and that such mediator are "stackable" in a way that their filtering functions can be combined, juxtaposed or superposed recalling the idea of Decorator [24] pattern.

The resulting architecture is showed in Figure 11.

1.4.5 Runtime Editor

Section 1.4.3 describes the `AbstractView` as a container of user interface components which are independent from GUI toolkits or mark-ip languages. At runtime the concrete view is finally instantiated from the most appropriate library for the actual delivery

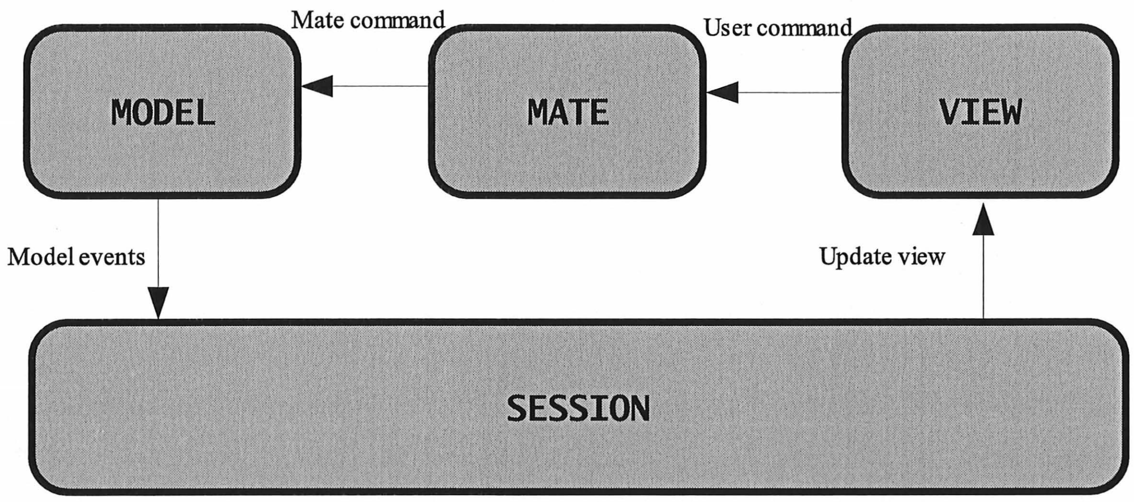


Figure 11: MVC modified with `Mate` insertion

context. Concrete components are deployed for displaying fields and methods of the model.

Concrete components are instances of Java classes extending the superclass `RuntimeEditor` which in turn extends the class `AbstractComponent`. At runtime, the system loads a table where types are associated to the appropriate editor. For instance, if the user is running the application in a J2SE platform on a desktop computer, then for a field of type `Number` the appropriate `NumberEditor` is loaded and instantiated.

The table is structured as a simple Java property file and contains entries such as the following:

```
field type = runtime editor class type
```

The mapping between editor types and value types follows the inheritance rules such that if the system needs the editor for a value of type `T` and no editor is available then the editor for the superclass of `T` is searched and so forth up on the inheritance tree until the class `Object`. If no editor is found the value is represented by a navigable link to another object.

The editor table is loaded at the beginning of the interaction session, and if one wants to change it at runtime it should be done by means of mate classes.

For instance, if an application requires that in model `M1` fields of type `Number` are mapped to editors of type `E1` and in model `M2` fields of type Number are mapped to editors of type `E2`, then the developer must implement the mate classes in order to specify that. At coding level, it means overriding the `Mate.getRuntimeEditor()` method.

MORE provides runtime editors for different platforms and for the basic types listed below:

```
java.lang.Number
java.lang.String
java.lang.Boolean
java.util.Calendar
java.util.Date
java.util.Collection
java.lang.Object[]
```

It is up to application developers to implement new runtime editors for other types relevant to applications. For instance, in the E-Mate project (Chapter 4) additional editors have been implemented for geo-referenced objects and other multi-media resources that are shared by most of services developed within the E-Mate framework.

1.5 MORE architecture and Javabeans API

The entire MORE API could be indeed considered as an extension of the Javabeans API aimed at the multi-channel delivery of services. For instance, the information contained in a `Mate` is similar to the additional information a `BeanInfo` stores for a `Javabean`. Nonetheless, a `Mate` covers other aspects not fully addressed by a BeanInfo:

- The mapping between beans and infos is at class level, i.e. for a given bean class we have a `BeanInfo` class. In MORE, instances of the same model can have as mates instances of different `Mate` classes. Moreover, a `Mate` can be coupled and decoupled to a model at runtime changing dynamically the behaviour of the application.

- The `Mate` programming interface is richer than the `BeanInfo` one. It directly handles pre-conditions, layouts and it simplifies the access to fields values and the invocation of model methods.

- Javabeans are *reusable software component that can be manipulated visually in a builder tool*, while models are domain objects involved in the interaction with the user at runtime by means of an appropriate user interface generated automatically. So the aim of the two is quite different: Javabeans define a complete component model for software application in general with a special focus on visual composition at design time while MORE API implements an architecture for the delivery of services on any device. Nonetheless, the Javabeans API could be exploited to provide some of the features described in this section. For instance, the class `Introspector` is used by MORE classes to get objects metadata at runtime.

- In the MORE architecture, a `Mate` is an intermediate layer between the view and the model. Any action on the view triggers a method of the `Mate` which filters the user action in the most convenient way. In other words, if a view wants to read/write a given model variable, or invoke a model method, or gather meta information about which methods and which fields are available in the model, all this information is conveyed by the `Mate`.

1.6 The Maildemo Application

1.6.1 Introduction

This section shows how to develop a simple application for reading and writing mail messages and that make them available on multiple heterogeneous platforms. First, model classes are designed and implemented against coding conventions defined in Section 1.3.2. Then, their implicit information is used to generate the user interface for a desktop computer, a Web browser, and a cellular phone. We show also how auxiliary objects such as mates, pre-conditions, and layouts may be integrated in the application to provide a better user experience and a better control to preserve model integrity.

1.6.2 Design of the Model Classes

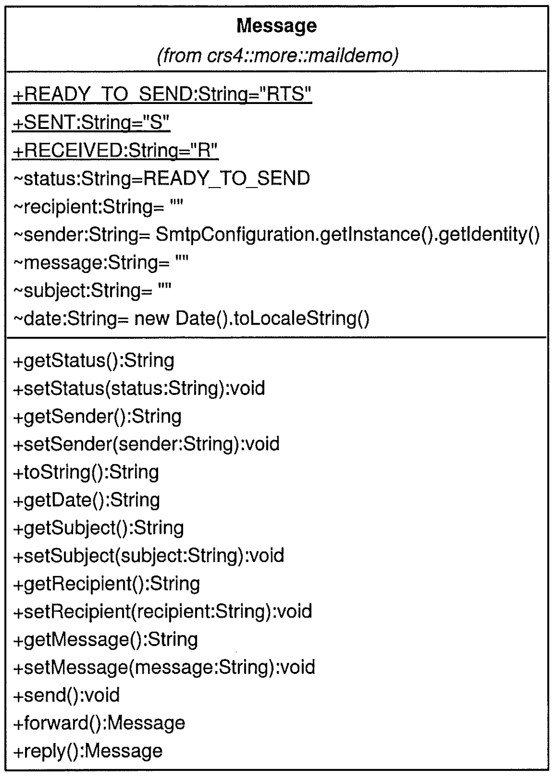
The Maildemo application relies on the following classes: `Message`, `MailBox`, `InBox`, `OutBox`, `Configurations`, `PopConfiguration`, `SmtpConfiguration`. In this section we will describe each of them.

The Message Class

The class `Message` (Figure 12) defines objects that are created, read, modified, and processed in the application.

A glance through the code shows that, apart from `set/get` accessors, the class `Message` declares methods: `toString`, `send`, `forward` and `reply`. They are all parameterless and the method `toString` is redefined to allow the system to display a frame title containing the subject of the message.

- `send()` appends the message to the `OutBox` (a message in the `OutBox` is considered as sent)

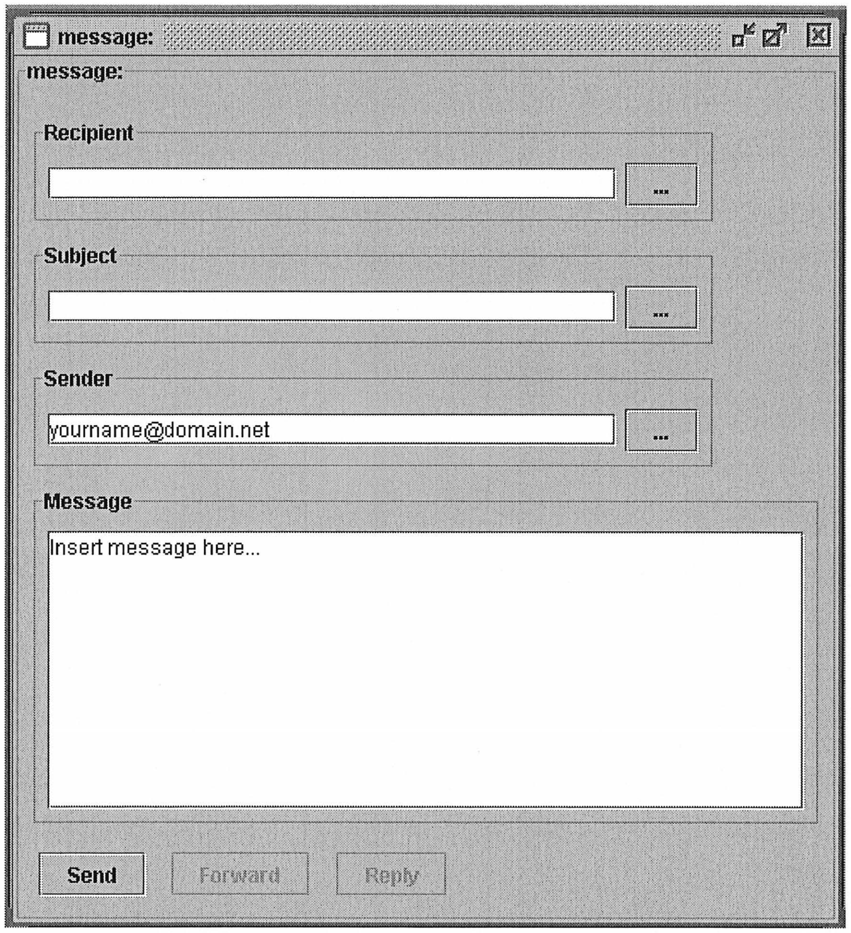


```
Message
(from crs4::more::maildemo)

+READY_TO_SEND:String="RTS"
+SENT:String="S"
+RECEIVED:String="R"
~status:String=READY_TO_SEND
~recipient:String= ""
~sender:String= SmtpConfiguration.getInstance().getIdentity()
~message:String= ""
~subject:String= ""
~date:String= new Date().toLocaleString()

+getStatus():String
+setStatus(status:String):void
+getSender():String
+setSender(sender:String):void
+toString():String
+getDate():String
+getSubject():String
+setSubject(subject:String):void
+getRecipient():String
+setRecipient(recipient:String):void
+getMessage():String
+setMessage(message:String):void
+send():void
+forward():Message
+reply():Message
```

Figure 12: UML for class `Message`

- `forward()` returns a new message whose subject is the result of prepending the literal 'Fw:' to the current `this.subject`
- `reply()` returns a new message whose subject is the result of prepending the literal 'Re:' to the current `this.subject`. Furthermore, the recipient of the new message is set equal to the sender of the `Message.this`.

With no further information, MORE would render a message instance with all methods exposed at the same time. However, for a brand-new message, it is a nonsense to invoke `reply` and/or `forward`. They should be disabled. For a message coming from the `InBox` queue is a nonsense to invoke the method `send()`. Methods expected to be enabled should be `reply()` and `forward()` only. This is a typical situation where a pre-condition can be defined on methods in order to constraint their activation according to the state of the message object.



yourname@domain.net

Insert message here...

Figure 13: A view for a message with pre-conditions and layout assigned

The first step to define a pre-condition on a method is to extend the class `Mate` with a new class `MessageMate`. Let us assume that the field member `status` may only assume one of the following values:

$$status \in \{READYTOSEND, SENT, RECEIVED\}$$

.

The MessageMate class has to check the following conditions to the execution of message methods:

- pre-condition on send: $status == READYTOSEND$;
- pre-condition on forward: $status == RECEIVED \vee status == SENT$;
- pre-condition on reply: $status == RECEIVED$;

The Figure 14 shows the Java code assigning preconditions and Figure 15 shows an implementation of the precondition enabling the send method.

```
public MessageMate(Object model, Session session) {
  super(model, session);
  addPrecondition("forward",new ForwardReplyPrecondition(this));
  addPrecondition("reply",new ForwardReplyPrecondition(this));
  addPrecondition("send",new SendPrecondition(this));
  [ ...snip...]
```

Figure 14: The auxiliary class MessageMate assigns pre-conditions to allow the invocation of specific methods of the model class Message.

In this way the MessageMate enables or disables methods in the user interface depending on the message state.

A second issue regarding the message view is the following: once the user invoke send on a message, the proper behaviour would be to close the message view in the screen. Closing the message user interface in the screen provides a feedback to the user that the

```
public class SendPrecondition extends Condition {
        public boolean isTrue() {
        try {
                return mate.getValue("status").
                equals(Message.READY_TO_SEND);
        } catch (IllegalAccessException e) {
        e.printStackTrace();
        } catch (InvocationTargetException e) {
                e.printStackTrace();
        }
                return false;
        }
        [... snip ...]
```

Figure 15: Precondition implementation for method `send()`.

task is done. Such a behaviour cannot be implemented in the `Message` class, nor in the view since it is generated by the system. Thus the proper place is the mate. (Figure 16)

A third issue is about presentation order of fields in the view. With no additional information, members would be laid out in the panel in an unpredictable order. To constraint the presentation to a specific order a `Layout` object must be provided (Figure 17).

As a result, a view of a message enables methods according to pre-conditions and lays out field members respecting a specific order (Figure 13).

Finally, before sending a message we must assess that `subject` and `recipient` are not null fields. In that case, the method should throw an exception. Then, the mate handles exceptions like any other model objects to be rendered and an appropriate runtime editor is loaded and presented to the user.

```
public void invokeMethod(String name) throws Exception {
    super.invokeMethod(name);
    if(name.equalsIgnoreCase("send"))
        session.closeView(model);
}
```

Figure 16: The invokeMethod method overrides the default behaviour inserting the instruction for closing the view when the invocation of send is successfull.

The MailBox Class

The `MailBox` class defines the common structure to all mail boxes (Figure 18). Specifically, it defines a field `envelopes` of type `Message[]`. The class `Inbox` extends `Mailbox` and defines the method `checkNewMessages()`. The class `OutBox` defines the method `addEnvelope(Message m)` which appends a message to the outgoing queue. Notice that the method `addEnvelope(Message m)` is not showed to the user for it is not a zero-parameter method.

Figure 19 shows how the `InBox` is rendered on PC screen. The message array is rendered by MORE in the Swing platform as a tabular component, more precisely a `JTable` instance from the Swing API is used. With this kind of editor the user is allowed to scroll the message list, to sort them with respect to a field, to remove a message and to open the selected message in a separate view.

To show how the mate can filter and dynamically modify the behaviour of the application, we wrote a `MailBoxMate` able to observe the user behaviour and switch from *novice* to *expert* mode the interaction with the `InBox`. In the novice mode the operation on the mail box are shielded by a confirm dialog in order to avoid unexpected deletion of data. After a given number of interactions with the mail box object, the user is considered expert and the dialog is not popped any more.

```
    public MessageMate(Object model, Session session) {
        super(model, session);
        [...snip...]
        Layout l=new Layout();
        l.addMember("recipient");
        l.addNewLine();
        l.addMember("subject");
        l.addNewLine();
        l.addMember("sender");
        l.addNewLine();
        l.addMember("message");
        l.addNewLine();
        l.addMember("send");
        l.addMember("forward");
        l.addMember("reply");
this.setLayout(l);
[...snip...]
```

Figure 17: Definition and assignation of a `Layout` object in the body of `MessageMate`

Although it is a simple example of dynamic change of behaviour of an application, it highlights the mechanisms that may provide for personalization. To do so the method `MailBoxMate.setValue(String variable, Object value)` overrides the `Mate.setValue(String variable, Object value)` (Figure 20). After N invocation of any operation which sets the fields of the mail box object, a default mate without user confirmation, replaces the current one for the mailbox and the user is notified that from now on he is considered an expert user.

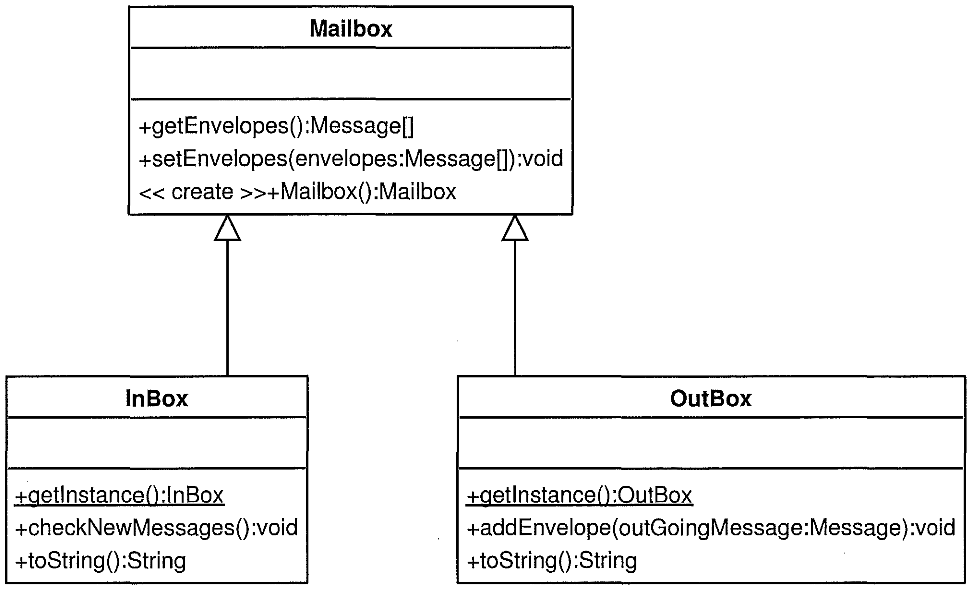
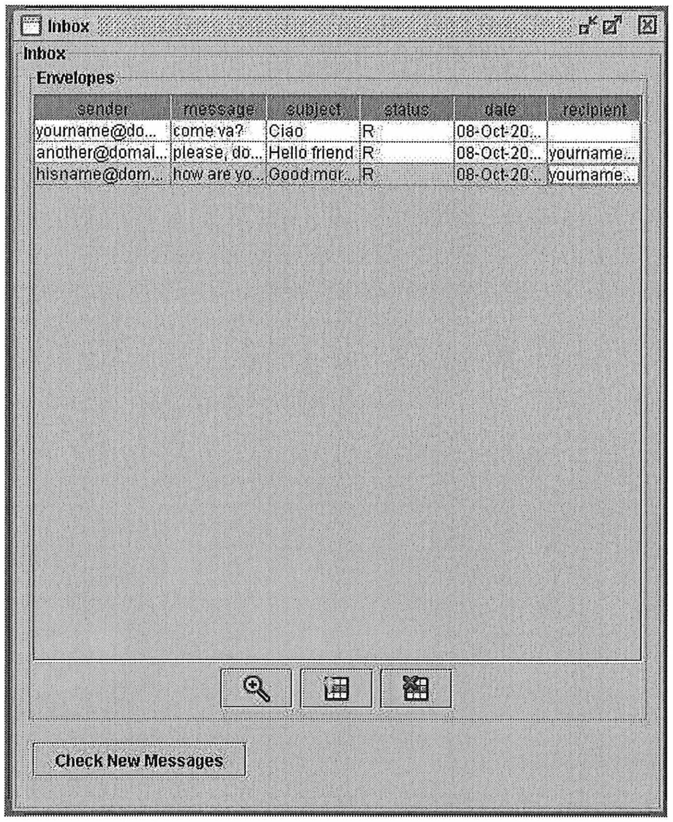

Mailbox
+getEnvelopes():Message[]
+setEnvelopes(envelopes:Message[]):void
<< create >>+Mailbox():Mailbox
InBox
+getInstance():InBox
+checkNewMessages():void
+toString():String
OutBox
+getInstance():OutBox
+addEnvelope(outGoingMessage:Message):void
+toString():String

Figure 18: UML diagram for classes `MailBox`, `InBox` and `OutBox`

Preferences and Set-up

By means of this tool the user is enabled to configure his mail identity, the outgoing mail SMTP server , the incoming mail POP3 server , and other security relevant information such as username and password. The classes involved are `PopConfiguration`, `SmtpConfiguration`, and `Configurations` (Figure 21).

The class `Configurations` is simply a container of objects which defines a field `configurations` of type `Vector`. When MORE runs in the J2SE platform and the delivery context is based on Swing, a `Configurations` instance appears as in Figure 22. The view allows the user to navigate a tree on the left panel, and to select an object



sender	message	subject	status	date	recipient
yourname@do...	come va?	Ciao	R	08-Oct-20...	
another@domai...	please, do...	Hello friend	R	08-Oct-20...	yourname...
hisname@dom...	how are yo...	Good mor...	R	08-Oct-20...	yourname...

Figure 19: An `InBox` instance as displayed by MORE on the J2SE platform with Swing API.

which is rendered on the right panel.

1.6.3 Multi-platform access to the MailDemo application

Previous sections sketched the general principles for generating dynamically a user interface for a PC. This section depicts how user interfaces are dynamically generated for different platforms, namely a Web page and a cellular phone

```
public void setValue(String variable, Object value)
throws ...{
  int k = session.showShield("Want to keep your changes?");
  System.out.println(k);
  if (k == 0)
     super.setValue(variable, value);
     counter --;
     if(counter == 0){
       session.setMate(model,new Mate(model,session));
       session.showInfo("Now you are expert, ...");
     }
}
```

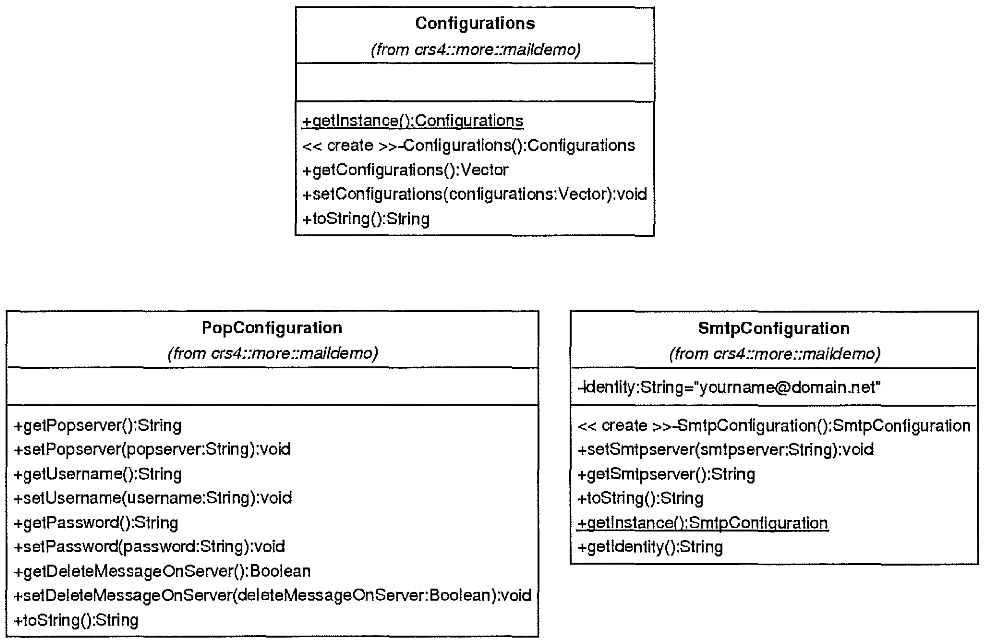
Figure 20: `MailBoxMate` code overrides the default `setValue` method. Initially, a confirmation is expected from the user. When counter equals zero a default mate, with no such a policy, is assigned to the model object.

Access to MailDemo from a Web Page

The MailDemo application may be rendered by MORE when the user connects through a Web browser (Figure 23 and Figure 24). Any object is represented by a view which is a Web page dynamically generated. Any view shows also a toolbar with a button for any class registered in the session.

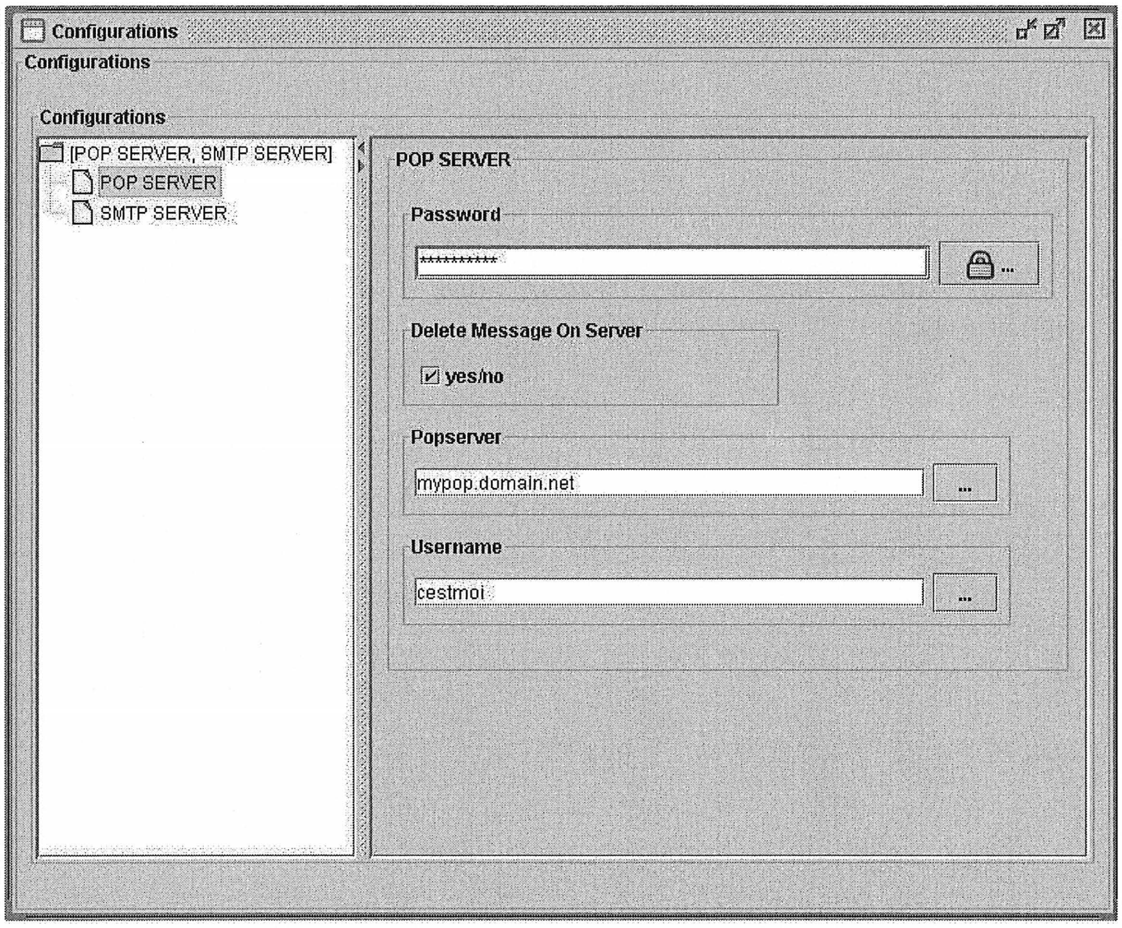
Access to MailDemo from a Cellular Phone

Maildemo model objects can also be delivered to a cellular phone by MORE (Figure 25). Here the delivery device is a Nokia 3510 Mobile Phone simulator using WAP browsing mode. The simulator has been choosen instead of the real device to easily grab images of the running application instead of photos.



Configurations
(from crs4::more::maildemo)
+getInstance():Configurations
<< create >>-Configurations():Configurations
+getConfigurations():Vector
+setConfigurations(configurations:Vector):void
+toString():String
PopConfiguration
(from crs4::more::maildemo)
+getPopserver():String
+setPopserver(popserver:String):void
+getUsername():String
+setUsername(username:String):void
+getPassword():String
+setPassword(password:String):void
+getDeleteMessageOnServer():Boolean
+setDeleteMessageOnServer(deleteMessageOnServer:Boolean):void
+toString():String
SmtpConfiguration
(from crs4::more::maildemo)
-identity:String="yourname@domain.net"
<< create >>-SmtpConfiguration():SmtpConfiguration
+setSmtpserver(smtpserver:String):void
+getSmtpserver():String
+toString():String
+getInstance():SmtpConfiguration
+getIdentity():String

Figure 21: UML diagram for `Configurations`, `PopConfiguration`, and `SmtpConfiguration` classes



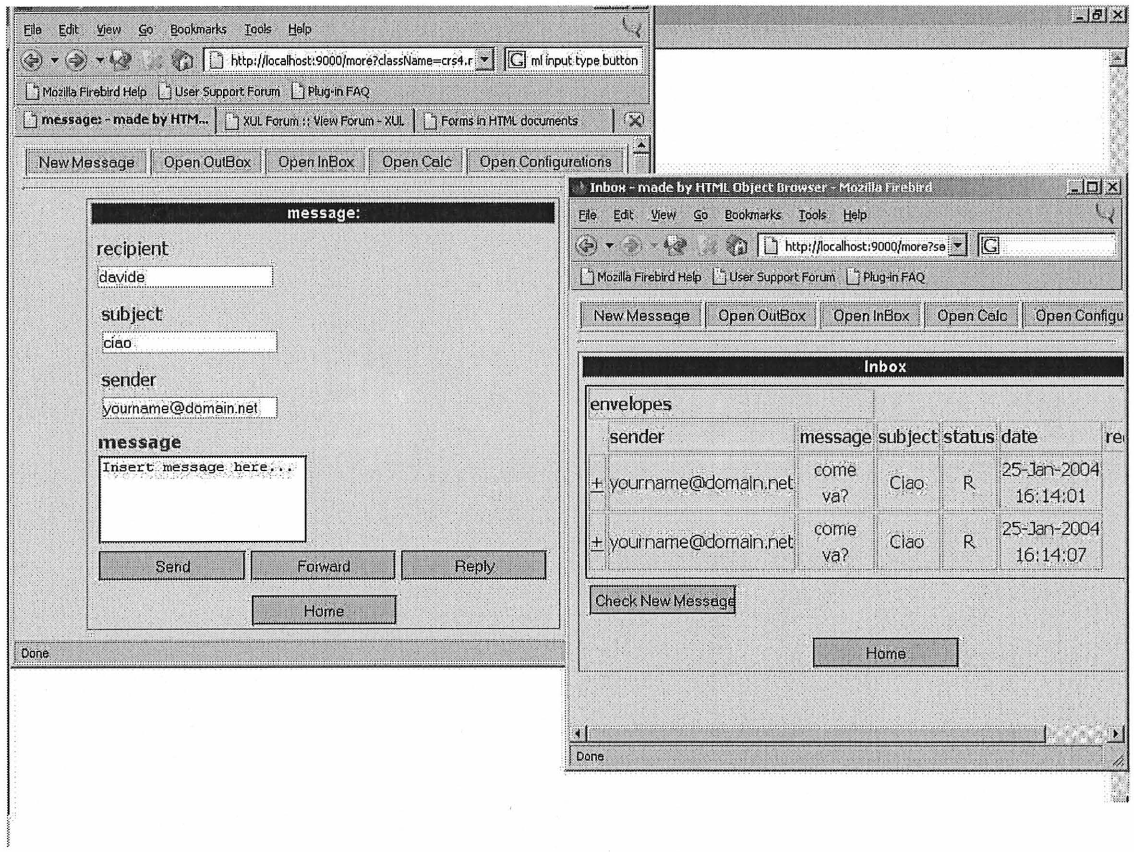
Password: **********

Delete Message On Server: [✓] yes/no

Popserver: mypop.domain.net

Username: cestmoi

Figure 22: View for a `Configurations` instance containing the mail settings for the J2SE platform with Swing user interfaces.


File Edit View Go Bookmarks Tools Help
http://localhost:9000/more?className=crs4.r
ml input type button
Mozilla Firebird Help
User Support Forum
Plug-in FAQ
message: - made by HTM...
XUL Forum :: View Forum - XUL
Forms in HTML documents
New Message
Open OutBox
Open InBox
Open Calc
Open Configurations
Send
Forward
Reply
Home
Done
Inbox - made by HTML Object Browser - Mozilla Firebird
http://localhost:9000/more?se
Open Configu
Check New Message

message:

recipient: davide

subject: ciao

sender: yourname@domain.net

message:

```
Insert message here...
```

Inbox

envelopes

	sender	message	subject	status	date	re
+	yourname@domain.net	come va?	Ciao	R	25-Jan-2004 16:14:01	
+	yourname@domain.net	come va?	Ciao	R	25-Jan-2004 16:14:07	

Figure 23: HTML pages generated by MORE. The generated Web pages are linked to model objects of type `Message` and `InBox`

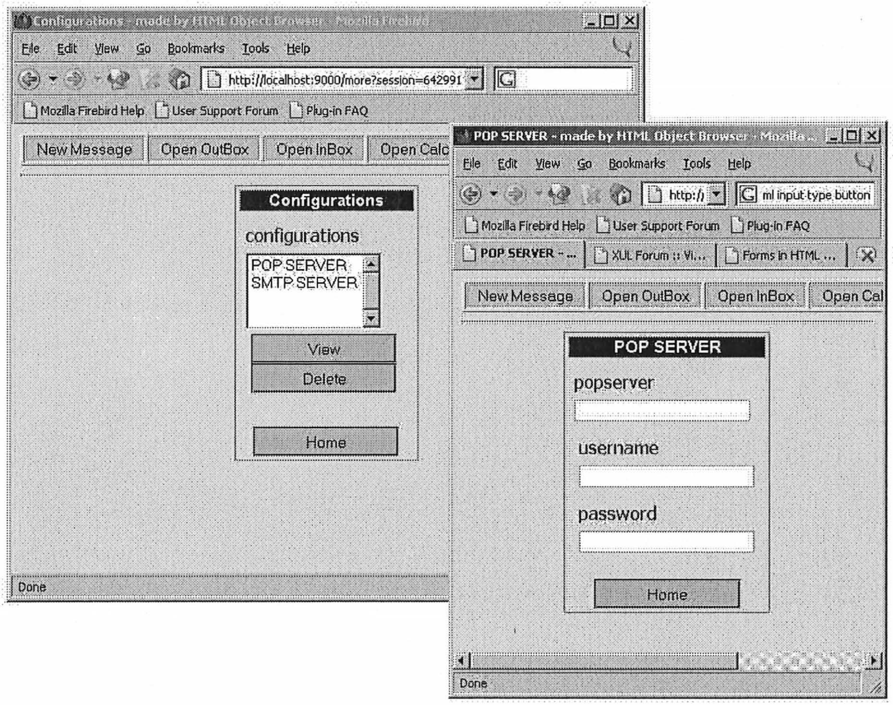
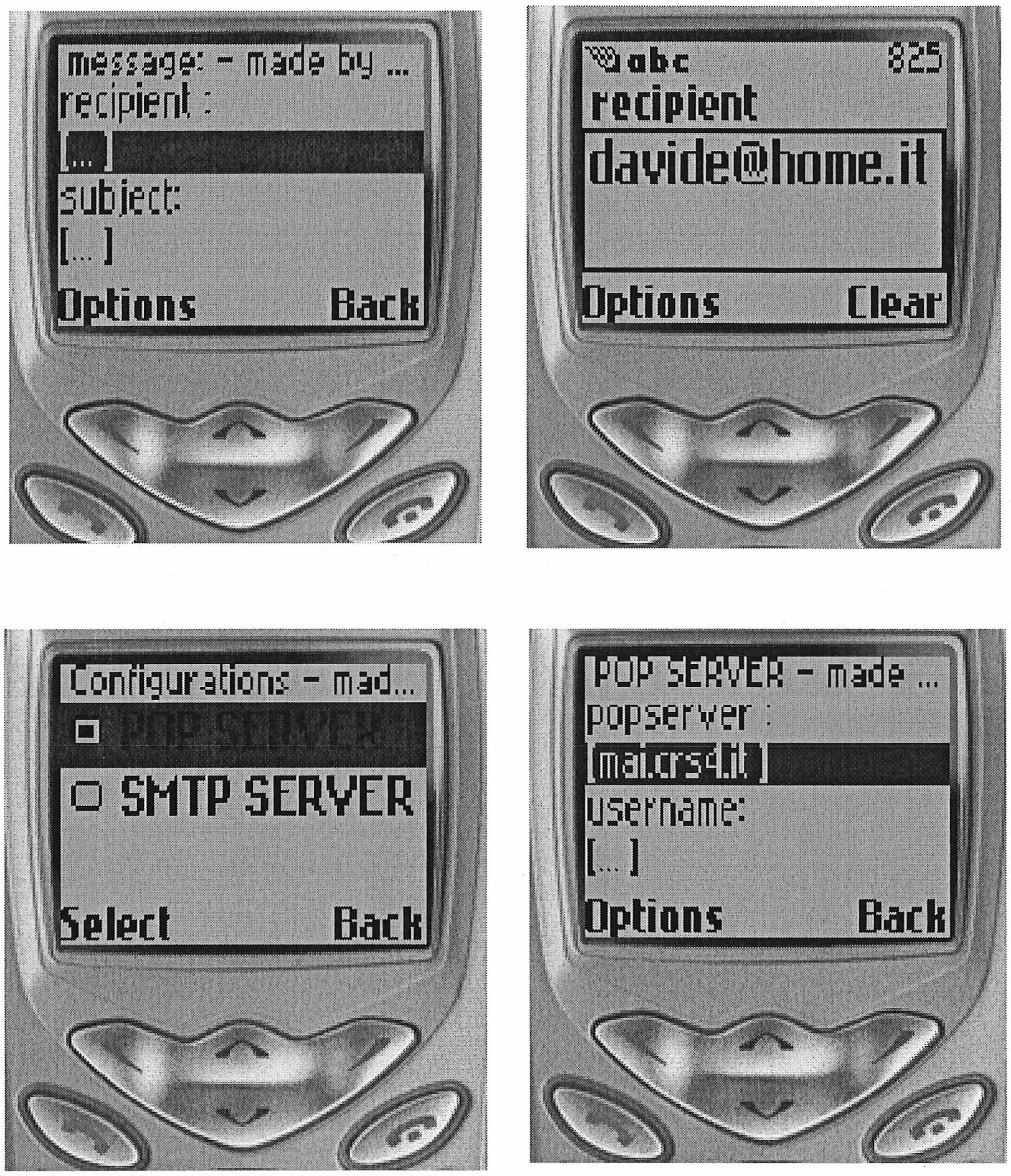

Configurations - made by HTML Object Browser - Mozilla Firebird
File Edit View Go Bookmarks Tools Help
http://localhost:9000/more?session=642991
Mozilla Firebird Help User Support Forum Plug-in FAQ
New Message Open OutBox Open InBox Open Calc
Configurations
configurations
POP SERVER
SMTP SERVER
View
Delete
Home
Done
POP SERVER - made by HTML Object Browser - Mozilla
File Edit View Go Bookmarks Tools Help
http:/)
ml input type button
POP SERVER - ... XUL Forum :: Vi... Forms in HTML ...
New Message Open OutBox Open InBox Open Cal
POP SERVER
popserver
username
password
Home
Done

Figure 24: HTML pages generated by MORE. The generated Web pages are linked to model objects of type `Preferences` and `PopPreferences`


message: - made by ...
recipient :
[...]
subject:
[...]
Options Back

abc 825
recipient
davide@home.it
Options Clear

Configurations - mad...
POP SERVER
SMTP SERVER
Select Back

POP SERVER - made ...
popserver :
[mail.crs4.it]
username:
[...]
Options Back

Figure 25: WML pages generated by MORE. (Top-left) Generated WML page for a `Message` instance. (Top-right) The generate edit dialog for the recipient field member of `Message` class. (Bottom-left) Generated WML page for the `Configurations` instance. (Bottom-right) Generated WML page for the `PopConfiguration` instance.

Chapter 2

BEYOND OBJECT-DRIVEN INTERACTION

The design approach described in Chapter 1 has the advantage of a pure object-based definition of applications where users access data and functions directly interacting with application's first class citizens, the objects. From a programmer point of view this is a straightforward way to implement in a platform independent manner all the functionalities letting user interfaces and platform specific deployment details to be automatically handled by the system.

Such an approach, which provides multi-channel access with no wrinkles, is somehow fascinating for whom does not want to deal with user interface issues related to design and usability. Programmers do not necessarily feel confortable to work side-by-side with usability experts because in many cases they assume to have the authority to tell programmers what they have to do. For their part, managers simply want to save as much money as they can. From a Java programmer point of view there is another advantage: he/she can design, implement, and deploy an application working exclusively with Java code. No Javascript, HTML, XML or whatever else have to be used.

However, object-driven and direct-manipulation systems have received strong and merciless critics from usability experts. For example, Constantine in [15] writes a very negative review of the work described by Pawson in [48]. The two authors starts from totally different assumptions and reach opposite conclusions about the opportunity to adopt object-driven interactions in software applications.

The subject of this chapter is not to examine all the reasons from both sides. We rather want to start from the experience gathered during the development and the experimentation of the MORE framework to assess the limits of object-driven interactions in the context of pervasive computing (Section 2.1). Then we go beyond these limits and propose in PLANES (Section 2.3) a possible solution. The next step is to integrate PLANES and MORE in a complementary approach (Section 2.4). Tools (Section 2.5) are designed and implemented to help working with PLANES. Finally, the MailDemo application is revisited to show the adaptation obtained with the new approach (Section 2.6).

2.1 Limits of the Object-Driven Interaction

In this section we examine three different forces that make pure object-driven interactions inadequate in the context of pervasive computing. Such forces are the user profile (Section 2.1.1), the device profile (Section 2.1.2), the context of use (Section 2.1.3), and the combinatorial explosion of objects (Section 2.1.4).

2.1.1 User Profile

The user profile is an important factor in interacting with computers both in traditional and pervasive computing. Let limit the attention on the experience of the user[1]. Although the direct interaction with domain objects seems to be straightforward for an expert computer user, it is not the case for a novice one: we cannot assume a novice user to

[1]The study of a complete user profile is far beyond the scope of this work.

understand the metaphor behind the object-driven interactions. Therefore, assistance from the system will be often necessary to reach his goal.

Moreover, even a computer literate can find useful a guidance when he faces an application in a complex or critical domain. I was personally involved in using a software for the production of Modulo 740, an Italian form stating annual incomes, real estate ownership, medical tax discounts and other tax relevant information. The production of such a module was in the past really a hard task, even for acconting and finance experts. In 1992 it has been defined "lunare"[2] by the former president of Italy, Francesco Cossiga. Then a specialized software tool was developed to guide the user and help him to fill the form. Users accustomed with tax regulation could just pick the relevant forms and print it with no additional works. For non-expert users in tax laws and accounting jargon, the tool provided a guided tour of the Modulo 740, asking the user punctual informations about his incomes, real estate and so forth.

Assisting the user often means constraining his interaction in a precise sequence of tasks and sub-tasks. This approach is exactly the opposite of the direct manipulation of model objects. In other words, novice users need to focus their attention on a precise goal in a step-by-step process. The possibility to interact with the complete set of model objects at the same time can lead the novice to confusion. Consequently, the model objects must blur themselves in the background letting tasks and goals emerge.

In terms of Interaction Patterns [73], it is the right place to apply the Wizard pattern[3].

[2]From the moon; something really mazed or mad.

[3]The Wizard pattern was proposed by Martijn van Welie. It can be summarized as follows:

- Problem: The user wants to achieve a single goal but several decisions need to be made before the goal can be achieved completely, which may not be known to the user
- Principle: User Guidance (Visibility)
- Context: A non-expert user needs to perform an infrequent complex task consisting of several subtasks where decisions need to be made in each subtask. The number of subtasks must be small e.g. typically between 3 and 10.
- Forces: The user wants to reach the overall goal but may not be familiar or interested in the steps that need to be performed. The tasks can be ordered but are not always independent of each

It is an example of a guidance for novice users that need an explicit definition of task and subtasks. This is not easy to implement in our current object driven system. Forcing an object driven system like MORE to produce a Wizard interaction raises interesting issues: objects in a Wizard are not really domain objects but rather steps and they should have methods like next, back and exit that link a step to the other. Such step objects have no meaning in an object oriented domain model, they would be at least a sort of anomaly. So modelling a Wizard in a object driven system is not a viable solution. A more appropriate solution is to define task models.

2.1.2 Device Profile

The end user device is another critical factor that must be taken into account when designing user interaction. In a pure object driven system object are accessed by means of automatically generated views even for devices with poor input/output capabilities.

---

other i.e. a certain task may need to be completed before the next task can proceed. To reach the goal several steps need to be taken but the exact steps required may vary because of decisions made in previous steps.

- Solution: Take the user through the entire task one step at the time. Let the user step through the tasks and show which steps exist and which have been completed.

When the complex task is started, the user is informed about the goal that will be achieved and the fact that several decisions are needed. The user can go to the next task by using a navigation widget (for example a button). If the user cannot start the next task before completing the current one, feedback is provided indicating the user cannot proceed before completion (for example by disabling a navigation widget). The user is also able to revise a decision by navigating back to a previous task.

The users are given feedback about the purpose of each task and the users can see at any time where they are in the sequence and which steps are part of the sequence. When the complex task is completed, feedback is provided to show the user that the tasks have been completed and optionally results have been processed.

Users that know the default options can immediately use a shortcut that allows all the steps to be done in one action. At any point in the sequence it is possible to abort the task by choosing the visible exit. The navigation buttons suggest the users that they are navigating a path with steps. Each task is presented in a consistent fashion enforcing the idea that several steps are taken. The task sequence informs the user at once which steps will need to be taken and where the user currently is. The learnability and memorability of the task are improved but it may have a negative effect of the performance time of the task. When users are forced to follow the order of tasks, users are less likely to miss important things and will hence make fewer errors.

Does it make sense to access domain objects and their methods in a 1,8 inch display with no mouse and keyboard? Does it make sense to perform interactions in parallel as one does on a desktop computer in a totally different device. Using a modern cellular phone, it is unlikely for the user to start writing a message, suspend such task to configure its mail account, then navigate in the message list and finally resume writing the suspended message. That is what we do on a desktop computer. When a device has a small display, no mouse and a 12-keys keyboard even if the software allows parallel interaction, the bottleneck is input/output capability yielding to interactions that are strongly serialized. Yet in this case it seems interactions must be simple and focused on a precise goal, thus a guidance through a task model is likely to be a suitable solution.

2.1.3 Context of use

The device is just one variable in the whole delivery context. There are other factors related to place and time: when and where the interaction occurs. For instance, a user at office has the time and the hardware for coordinating different concurrent interactions with the computer. On the other hand, a user during a travel would usually access programs and data through the Web because Internet browsers are available for most platforms. Someone having a breakfast in his kitchen would check his mailbox without turning on the computer but rather by means of his interactive TV. So tasks models should be tailored according to the context of use, e.g. place, time, and device available.

2.1.4 Towards a Combinatorial Explosion of Objects

In Chapter 1, objects are described as interactive and self-contained computing elements. They allow users to perform activities related to objects such as: invoke a method, assigning a value to a field, select an object in a list, and so forth. Assuming a pervasive computing environment like a place enhanced with thousands of objects then it would be hard for the user to handle such a complexity with a pure object driven interaction system. The next section introduces task models, explains how to define tasks and subtasks using a visual tool and how to integrate a task model with the object model in a pervasive computing environment.

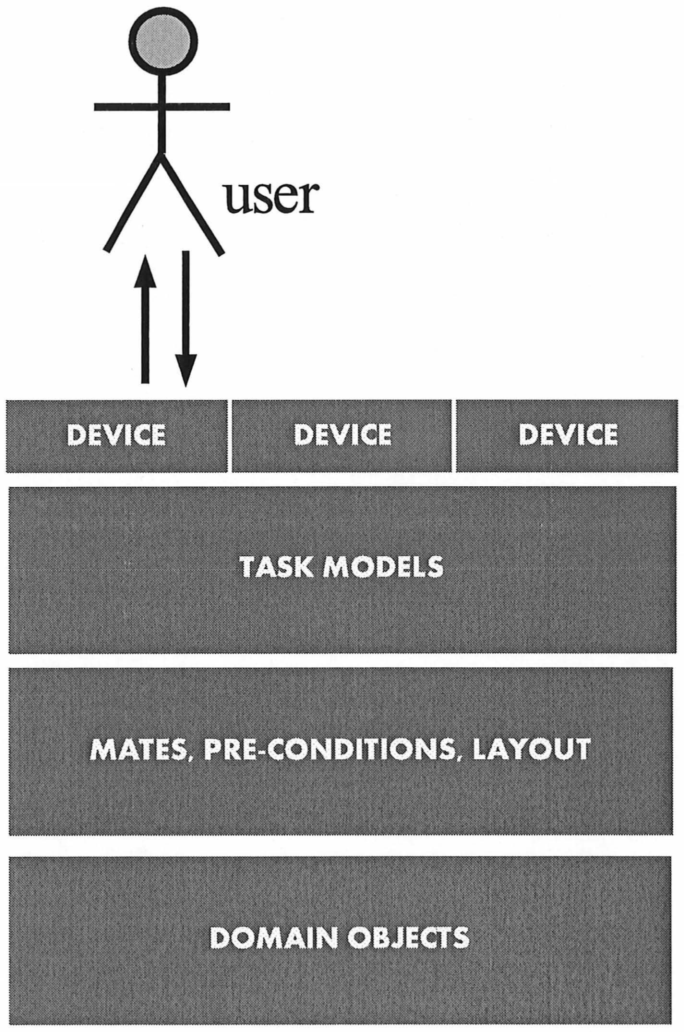


Figure 26: Layers between the user and domain objects

Our approach clearly becomes a layered one (Figure 26). The object model contains all domain related objects of the application and thus all data and logic. Mates, preconditions and layouts builds a layer wrapping objects. The wrapping layer adds application specific and device specific behaviours as well as other presentation relevant information. The task models layer hides underlying layers to provide users with guided, structured, and personalized user experience.

2.2 On Task Models

Task models capture the flow of activities that must be carried out to reach a given goal. Task modelling involves a high-level description of choices and sequences of activities. Tasks represent activities at different levels of abstraction. For instance, a task representing the insertion of a new appointment into a personal agenda must be considered an abstract task which will be reified by one or more concrete tasks.

Task-driven design assumes an abstraction on technology and implementation as well as object-driven design does. Tasks are first-class citizens and operate on data, the user is led through sequences or choices of actions to the accomplishment of the main task, while in object-driven systems the user acts much more as a problem solver. The advantages of task-driven systems are many. First, it eases the job of users. Second, it provides a clear guided path. For instance, a user interacting with a mobile application may likely prefer to know exactly what to do next instead of figure out how to exploit the user interface. Furthermore, a task-driven system could be easily personalized according to the previous activity. For instance, assume that while interacting with an answering machine a user must choose among "save" and "exit" and that the actual user often does "save" and then "exit", the system could infer that a better menu for the actual user is: "save and exit", "save", "exit" composing and reordering the options given.

2.3 PLANES: Prototyping LANguage for Embedded Systems

The in-depth exploration of task driven systems is far beyond the scope of this thesis. Our focus is rather on the experimentation of task-driven design and the integration with the automatic user interface generation provided by MORE for developing services for pervasive computing environments.

Many languages and environments for task driven design of interactive systems had been proposed in the past but none of them emerged as a standard tool or standard suite (although some tools such as ConcurrentTaskTrees (CTT) [46] are gaining a wide adoption from HCI experts). The reader could find several examples of task modelling tools in literature. [4],[31],[29],[46],[50],[61] address task modelling in all its aspects.

Since none of the above task driven approaches has reached the completeness and the easyness-to-use for a commercial licensing, we at CRS4 designed and implemented a simple task-driven design tool for mobile devices. The input of the tool, called PLANES [7], is a task model defined with a visual palette of tasks, the output is a set of Java classes for the J2ME platform. The original experiment was to design a personal agenda system. It was limited to a pattern of applications that can be defined ChooseTask-FillTheForm-Confirm-Submit(Figure 27 and Figure 28). The Java code generated by the tool can be manually refined in order to cover specific presentation requirements.

One of the objective of our work is the integration of the task driven design of application provided by PLANES with the capacity to generate user interfaces for any device provided by MORE. In other words, how to exploit the abstraction and the expressiveness of task driven design in pervasive computing where multiple platforms such as desktop computers, noteboks, Web pages, WAP decks, Java phones and VoiceXML enabled browsers must be addressed.

The elements of a PLANES model[4] have the following properties:

- any model can be represented by a N-nary tree.
- Nodes in the tree are tasks.
- Nodes can be either concrete tasks or composite tasks.
- A composite task is composed by one or more subtasks either composite or concrete.
- There are four different type of composite tasks: SEQUENCE, CHOICE, INTERLEAVING, and REPEAT.
- A SEQUENCE node is a composite task which constraints its subtasks in a sequential execution. Once the last task in a sequence is performed then the sequence is over and returns the control to its ancestor task. Tasks in a sequence must communicate data between them. For this purpose, they share a common registry called $TaskContext$ which can contain an object reference.

  $SEQUENCE(T_1, T_2, \ldots, T_n)$ means execute T_1 then T_2 ... then T_n
- A CHOICE node is a composite task which contains one or more mutually exclusive subtasks.

  $CHOICE(T_1, T_2, \ldots, T_n)$ means user has to choose to execute T_1 or T_2 ... or T_n
- An INTERLEAVING is a composite task whose subtasks are all allowed to be performed by the user concurrently. In this case more "Task Context", one for each interleaved task, are created in order to ensure consistency between concurrent activities.

  $INTERLEAVING(T_1, T_2, \ldots, T_n)$ means T_1 and T_2 ... and T_n are performed in parallel way

[4]The exact specification of the text notation used in this section is reported in appendix C.

- A REPEAT node is a container which repeats its component tasks arbitrary N or infinite times according to its definition.

  $REPEAT(T, N)$ means repeat task T N times

  $REPEAT(T, FOREVER)$ means repeat task T forever

The integration of PLANES tasks in the MORE framework required to introduce some concrete tasks related to object instantiation, methods invocation and get/set of object properties. These concrete tasks are:

- INSTANTIATE: it is a system task that creates a new instance of a given class. The new instance is saved in the task context in order to be retrieved by next task in a sequence.

  $INSTANTIATE(ClassName)$

- INPUT: this is a user task where the user is required to input the value of one or more properties of the current model object contained in the task context. The type of dialog presented to the user depends on the type of value that must be inserted and on the platform.

  $INPUT(x, y, z)$ opens a dialog for the insertion of x,y,z. Once the user submits the form, methods setX, setY and setZ are invoked on the model object at the top of the context[5].

- SELECTION: a user task to select one or more model objects.

  $SELECTION$ read an array from the task context, waits the user selection and writes the selected item into the task context.

- SHIELD: a dialog between user and system to prevent accidental changes in data. The user must choose among going further in a sequence, aborting the sequence or

[5]To be more precise, methods are not directly invoked by the task but by the mate which always acts as mediator between domain objects and clients.

going one step back. This task does not modify the context. If the user chooses to abort, all the following tasks are simply skipped and the entire sequence is considered done. Any change in the task context or any other side effect caused before the abortion is not rolled back.

- INVOKE: this is a system task which invokes the specified method on the object contained in the task context[6]. The result is written in the task context.

2.4 PLANES and MORE Integrated

The integration of PLANES tasks in the MORE framework relies on the integration of different models:

- Delivery Context: the delivery context contains device and context related information. (See section 3.3 for a detailed description)

- Task Models: the application developer writes a set of contextual task models. For instance, the task models can be associated to the actual end user device. For a desktop PC with mouse, 102 keys and large screen the task model will be different than the one for a 2 inches display, 12 keys mobile phone.

- Mate Model: the application developer writes a set of classes which stand aside the domain object. Such classes filter invocations, define layouts, define and check preconditions on methods and fields of the underlying domain object model.

- Object Model: the object model defines the domain of the application. They are self contained pieces of data and behaviour.

[6]Also in this case any method invocation is proxied by the mate of the domain object

Once the application developer defines the Object Model and the Mate Model, depending to the actual Delivery Context, the appropriate Task Model is loaded and executed during service delivery.

2.5 PLANES: The Graphical Editor

The graphical editor for PLANES allows composition, file saving/loading, and deployment of task models (Figure 28). A palette of tasks is presented to the developer in order to insert and edit tasks in the tree. The tool is written in Java and relies on Swing user interfaces. The format for saving/loading PLANES models is XML (See Appendix D for an example of XML output). Other formats such as LOTOS [4] or CTT [46] could be implemented in future if requested.

We choose to serialize the task tree in XML instead of Java binary because saving Java objects in the native binary format is not version-resistant: for any change in the implementation of a class, all previously saved models cannot be reloaded by the editor.

2.6 Task Models for the Maildemo application

The code of Maildemo application 1.6 was made of the following classes: an object model which contains message, inbox, outbox and preferences classes; a mate model containing preconditions, special behaviours and layouts. This section shows how and put on the top of the previous models (Figure 26).

As examples, three different delivery contexts are depicted: Delivery Context on a PC (Section 2.6.1), Delivery Context on a cell phone (Section 2.6.2), Delivery Context on voice responder (Section 2.6.3).

2.6.1 Delivery Context on PC

In the first scenario the user works with his personal computer. More than one activity can be going on at the same time. Interactions are interleaved and performed in parallel. Such activities are: instantiate classes, edit fields of instances, invoke methods and so forth (Figure 29).

```
agenda:=REPEAT(CHOICE(addAppointment,modifyAppointment),FOREVER)

addAppointment:=SEQUENCE(instantiateAppointment,
                         fillAppointmentForm,
                         SHIELD,
                         invokeWriteOnDisc)

instantiateAppointment:=INSTANTIATE("Appointment")

fillAppointmentForm:=INPUT("when", "why", "whom", "where")

invokeWriteOnDisc:=INVOKE("save")

modifyAppointment:=SEQUENCE(selectAppointment,
                            CHOICE(editAppointment,
                            removeAppointment))

editAppointment:=SEQUENCE(fillAppointmentForm,
                          SHIELD,
                          invokeWriteOnDisc)

removeAppointment:=SEQUENCE(SHIELD,INVOKE("remove"))

selectAppointment:=SEQUENCE(INVOKE("listAppointments"),SELECTION)
```

Figure 27: Prototype of an Agenda application designed with PLANES for a J2ME powered mobile phone.

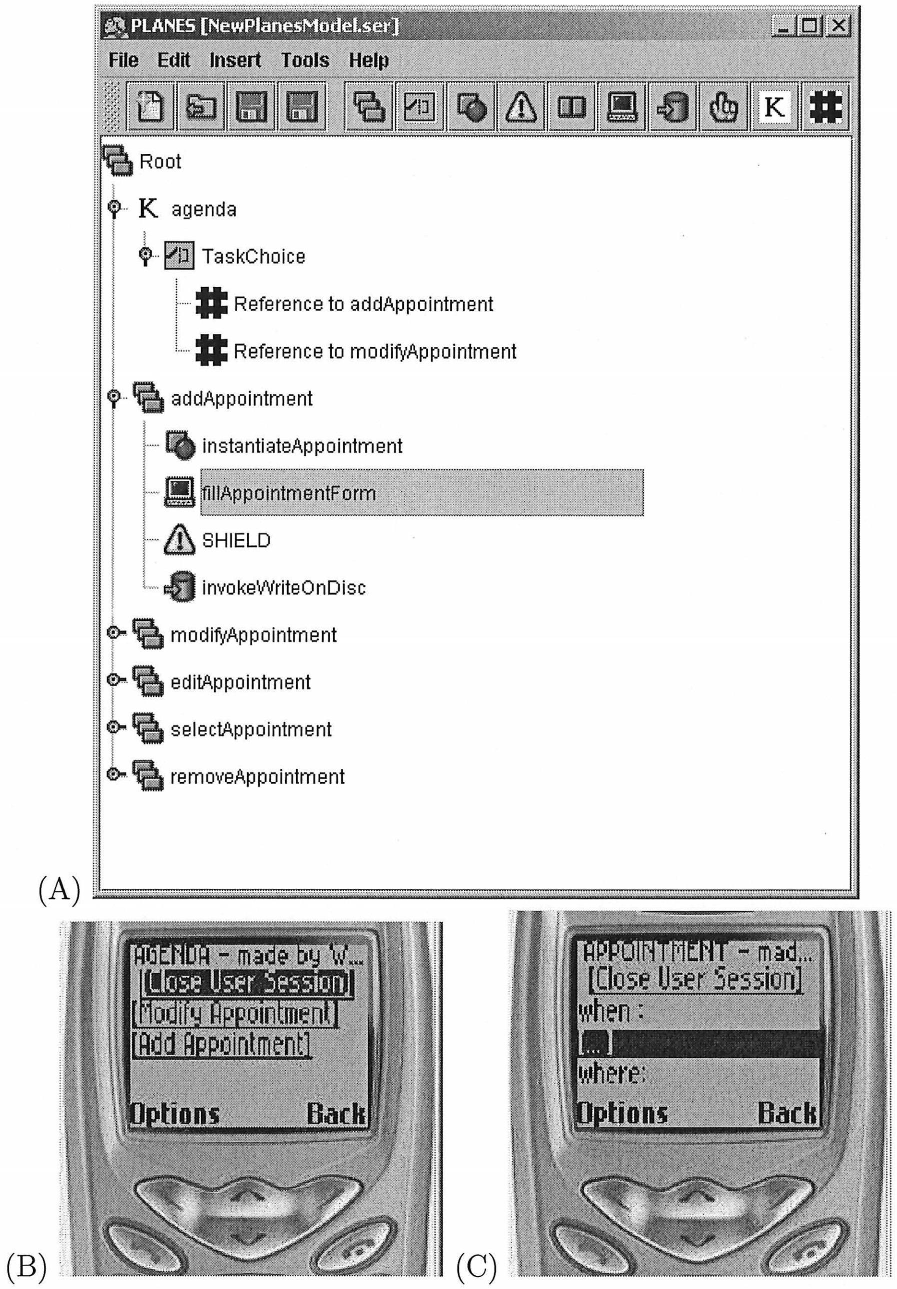


Figure 28: PLANES: the graphical tool in action. The task model in the picture (A) defines tasks and subtasks for an Agenda application. In (B) and (C) screenshots of the user interfaces related to tasks of the Agenda.

```
mailer:=INTERLEAVING(
              REPEAT(sendMessage,FOREVER),
              REPEAT(checkIncomingMail,FOREVER),
              REPEAT(manageSentMessages,FOREVER),
              REPEAT(manageInbox,FOREVER),
              REPEAT(configure,FOREVER))

sendMessage:=SEQUENCE(INSTANTIATE(Message),editMessage,INVOKE(send))
editMessage:=INPUT(recipient,subject,message)
checkIncomingMail:=SEQUENCE(INSTANTIATE(InBox),
                            INVOKE(checkNewMessages),
                            manageInbox)

manageInbox:=SEQUENCE(INSTANTIATE(InBox),
                      INVOKE(getEnvelopes),
                      SELECTION,
                      editMessage,
                      CHOICE(replyMessage,forwardMessage,
         deleteMessage))

replyMessage:=SEQUENCE(INVOKE(reply),editMessage,INVOKE(send))
forwardMessage:=SEQUENCE(INVOKE(forward),editMessage,INVOKE(send))
manageSentMessages:=SEQUENCE(INSTANTIATE(OutBox),
                             INVOKE(getEnvelopes),
                             SELECTION,
                             editMessage,
                             CHOICE(forwardMessage,deleteMessage))

configure:=CHOICE(configurePOP, configureSMTP)
configurePOP:=SEQUENCE(INSTANTIATE(PopConfiguration),
                       INPUT(username,
                             password,
                             popServer,
                             deleteMessagesOnServer))

configureSMTP:=SEQUENCE(INSTANTIATE(SmtpConfiguration),
                        INPUT(identity,smtpServer))
```

Figure 29: Task model for the Maildemo application delivered on a PC.

2.6.2 Delivery Context on a Cell Phone

In this delivery context the user connects to the Internet with his mobile phone. The connection is slow, the display is 2 inches monochrome screen and the user agent is a WAP browser. In this context some activities can be performed while others cannot (Figure 30). One can check new messages but not retrieve the list of old message for the slow connection and the limited amount of memory. He can write and send a new message but he cannot open the configuration panel. All parameters must be set in another delivery context.

The main difference with the delivery on a PC are:

- the task `mailer2` is not a composition of interleaved subtasks;
- the task `manageInbox2` differs from the task `manageInbox` because it invokes the method `getNewEnvelopes` instead of the method `getEnvelopes` thus limiting the number of items displayed to the user. However, the class `Inbox` does not implement a method `getNewEnvelopes`. We should not have to modify the domain object to fit the task model. It would not be coherent with the approach underlying our thesis. To address this issue, a mate for the Inbox is implemented. This mate filters new messages when the method `getNewEnvelopes` is invoked by a task.

```
mailer2:=REPEAT(CHOICE(sendMessage,checkIncomingMail2);FOREVER)

checkIncomingMail2:=SEQUENCE(INSTANTIATE(InBox),
                             INVOKE(checkNewMessages),
                             manageInbox2)

manageInbox2:=SEQUENCE(INSTANTIATE(InBox),
                       INVOKE(getNewEnvelopes),
                       SELECTION,
                       editMessage,
                       CHOICE(replyMessage,
                              forwardMessage,
                              deleteMessage))
```

Figure 30: Task model for the Maildemo application delivered on a cell phone.

2.6.3 Delivery Context on Voice Responder

In this delivery context the user connects to the application by means of an Interactive Voice Responder (IVR). He wants to keep his phone call really short and he is interested in newly incoming mail only. A further limited set of tasks are allowed (Figure 31). It is allowed only to read new messages:

The task `manageInbox3` reduces the `manageInbox2` by preventing the user to reply or forward messages.

```
mailer3:=REPEAT(CHOICE(checkIncomingMail3);FOREVER)

checkIncomingMail3:=SEQUENCE(INSTANTIATE(InBox),
                             INVOKE(checkNewMessages),
                             manageInbox3)

manageInbox3:=SEQUENCE(INSTANTIATE(InBox),
                       INVOKE(getNewEnvelopes),
                       SELECTION,
                       editMessage)
```

Figure 31: Task model for the Maildemo application delivered on a interactive voice responder.

Chapter 3

LIFECYCLE OF APPLICATIONS

In Section 1.4 is introduced and explained an object-driven methodology for the design and implementation of domain classes and the subsequent automatic user interface generation which allows the direct interaction between users and domain objects. The Section 2.3 analyzes issues in direct user-object interaction and highlights the need for a task-centered design. This chapter discusses the lifecycle of objects and services once design and implementation have been successfully accomplished. The main design goals are: how to deploy services making them accessible in the surrounding environment? How to discover them? Finally, how to deliver them to end user in a pervasive computing context?

3.1 Lifecycle

Once the application developer writes the classes of the domain model and provides a set of task models, objects should be instantiated in order to be accessible to end users. A simplistic solution is to instantiate objects on an arbitrary machine in the network. Nevertheless, this solution is not viable in a pervasive computing environment because the existence of brand new objects would be not known to anybody. This kind of discovery issue is often encountered in distributed system, and even in the Web: how can someone connect to a Web page if no search engine has indexed it? *When the user knows the host name and the directory path of the page.* But in general users cannot be expected to know beforehand the Internet address of a new object.

A step further is to deploy objects and to advertise them in a dedicated and well known registrar. Many systems adopt this technique: RMI uses the RMI Registry, Jini uses the Lookup service, CORBA uses the ORB and finally Web pages try to be indexed by search engines such as Google and Altavista.

Discovering services is not the only problem in distributed and pervasive computing contexts, the following items describe and summarize other important design goals:

- The application developer must be able to deploy his services with the confidence that they will be made accessible to end users.
- The deployment architecture must be flexible enough to handle load balancing in a transparent way both for the user and for the developer.
- A search mechanism must be available for those who seek services.
- Objects must be able to move and reach the end user device, or at least to reach a proxy or a message relay connected to the device. Instances must be serialized and deserialized in different machines, therefore remote class loading is required.
- Objects must be made persistent in order to be recovered after a system crash or when they cannot all be stored in the physical memory.
- The user should be allowed to seamlessly change device while he is interacting with a service.
- In some applications (Section 4.1) it compels to perform some jobs while the user is disconnected. For instance, to search the Web, to check for incoming appointments

and so forth. A framework is necessary to build off-line and active operations[1].

Moreover, beyond deployment and discovery issues there are issues related to delivery. The questions are:

- when the device is turned on, what is the entry point to the system?
- How is the delivery context modeled and how applications are affected by it?
- How is the user interface adapted to the actual device?

In the following subsections possible solutions are described to address the issues and the design goals enumerated in this introductory analysis.

3.1.1 Deployment Service and Load Balancing

A straightforward approach to service deployment is based on special system services called application servers. An application server is committed to instantiate services, to allocate required resources, to publish service references in network registrars or distributed index, to provide persistence facilities and to handle transactions. The use of a deployment service could be summarized as follow: a developer connects to the deployment service and puts packaged code that an application server will use to deploy the service later.

We assume that in general more than one application server is available to deploy services. Therefore, the application server architecture must take into account the cooperation between multiple application servers and load balancing. Rather than letting the developer choose which application server will deploy his service, a master/worker architecture can be put at work. Services are packed into entities called service packs. These service packs are inserted into a distributed queue. Any application server having resources available, connects to the queue, gets a service pack and deploys it (Figure 32).

[1]Further on in the document we refer to off-line agents.

When an application server has not sufficient resources to run a service then the service is repackaged and inserted back into the queue. This distributed algorithm is already used in distributed operating systems and multicomputers [62]. In our approach, the queue itself is a service that can be discovered in the network and that can be used to coordinate deployment by clients through application servers.

A positive aspect of this algorithm is that when the load in an application server is high it does not pick up new packs from the queue. On the other hand, an application server with many resources available will connect frequently to the queue. Alternative algorithms without queue coordination may also be implemented: for instance, application servers can measure their workload against a given metric and then when the workload reaches a given threshold they connect to another application server asking this one to take some of the workload. If the contacted server has not enough resources the request is denied and then another server is contacted. After N failures the server stop contacting the other application servers. This algorithm was first proposed in [21] for multi-computer load balancing. Compared to the distributed queue it has the drawback that network traffic increases at the least favorable moment: when the average workload in servers is high.

Application servers can provide extra services such as publishing, persistence and transaction management.

3.1.2 Service Discovery

Service discovery is a mechanism that allows clients to find services. Clients can be either end user clients or other online services. The simplest solution is to publish a reference to a service in a publicly available registrar or index. From the user point of view, we want to implement the metaphor "devices are portals": then using any device, when the user connects to the registrar he should be able to see the list of available services, select a service, load it and start the execution.

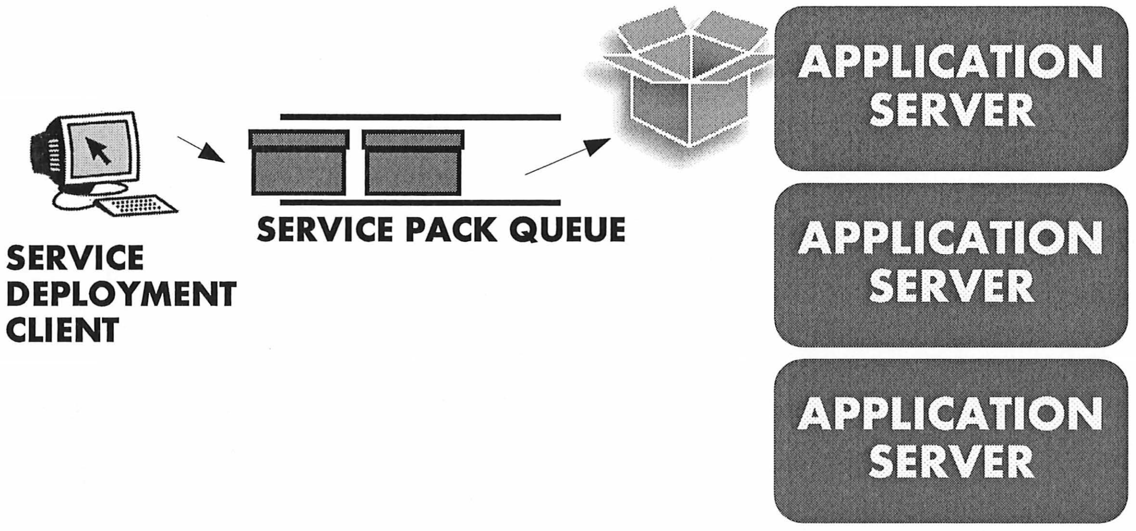


Figure 32: A distributed queue contains service packs to be deployed by application servers.

Several architectures implement such a discovery mechanism: CORBA [66], RMI [59], Jini [1], UPnP [69], Globe [65], JXTA [45] and Salutation [55]. The implementation of the work related to this thesis relies on the Jini connection technology. This section explain in details this choice and how it has been deployed to implement the lifecycle of services in a pervasive computing context.

3.1.3 Delivery

In this document we denote by the word *delivery* the set of actions and mechanisms by which applications reach users. Delivery enables the user-service interaction. Given that in a pervasive computing environment no prior assumption can be made on the capabilities of the end user device. Hence, the main issues are related to code mobility and to the adaptation of user interface to the end user device.

The direct approach is to move objects from the application server to the end user device. If we analyze how object mobility is implemented in the Java Virtual Machine, we see that for each serialized object sent on the network all its referenced objects must move too, unless they are declared `transient` or `Remote`. Thus, it is an entire graph of objects which moves.

When both JVMs share the same classes, they are pre-loaded from their local disk but in a real-world distributed system, the class bytecode must be loaded from a remote server with the HTTP protocol. The recipient JVM can load remote classes and can perform the instantiation of deserialized objects. This mechanism works fine when sender and recipient JVM share the same "profile". However, in general it is not possible to serialize an object in a J2SE virtual machine and move it to a J2ME one.

3.2 Implementation Details

Previous section defined requirements to support the lifecycle of services in an environment computationally pervaved and highly dynamic. In this sections we first present and justify the choice of Jini for the spontaneous networking architecture (Section 3.2.1). Then we sketch the implementation of the distributed queue (Section 3.2.2). We explain details of the service model in Section 3.2.3. Finally a concrete example is sketched (Section 3.2.4).

3.2.1 Jini as Connection Technology for Pervasive Computing

Jini extends the classic object oriented paradigm to distributed system in a more robust, fault-tolerant and dynamic way than other technologies such as RMI or CORBA. By means of discovery, lookup and join protocols becomes possible building service networks with no setup overhead. Jini is deeply woven with RMI for method invocation and code mobility. The evolution of object-oriented distributed systems has been toward a uniform programming paradigm both for *local* method invocations and *remote* method invocations.S

A Jini service network (or service federation) relies on some fundamental system services:

- Lookup service, which supports the registration of services and the search from clients.

- Transaction manager service, which handles transactions with multiple participants.

- Javaspaces services: they are used to decouple applications and enable them to exchange data and to synchronize operations in similar way as Linda systems [12].

These system services are built like all other Jini services and while Transaction managers and Javaspaces can be considered optional, an instance of Lookup service is compulsory to boot a network.

The Lookup service stores objects that can themselves be whole services or simple proxies to remote servers. These objects are stored in form of byte arrays and are not instances when in the Lookup service. The discovery protocol in Jini is used to localize a Lookup service in the network without knowing its IP number and port number. Such protocol uses IP multicast packets to find Lookup servers in the network, and therefore is not suitable for a large Internet environment. In fact, IP multicast packets are special purpose packets where the IP destination does not correspond to a real machine but stands as the address of a group of services. Any device can join and leave the group and therefore receive the packets. This works straightforward in intranets but is not suitable in inter networking context like the Internet because packets have a TTL (Time to live) field which is decremented whenever the packet is relayed by a router from a physical network to another. The TTL field is used to prevent infinite loop propagation of multicast packets.

Once a Lookup service is discovered, the device willing to register a service receives an instance of an object implementing the Java interface `ServiceRegistrar` which implements the discovery, join and lookup protocols. The main methods of this interface are `lookup` and `register`. The method `lookup` implements a search in the set of already registered objects. The matching criteria can vary as follows:

- an object O matches the query Q if Q refers to a Java interface that O implements;
- an object O matches the query Q if O was registered together with a tuple of attributes $(a_1, a_2, ..., a_n)$ and Q contains a tuple $(b_1, b_2, ..., b_n)$ where b_i are either equal to null or equal to a_i.
- an object O matches a query Q if Q contains a service ID and O was registered with that ID.

The method `register` perform the registration of an object in the Lookup service (Figure 33). Such a registration is a resource-consuming operation. To correctly handle resources, the Lookup service uses a lease mechanism to prevent the waste of resources. For each resource-consuming operation a Lease is created and returned to the requestor. When the lease expires the resource is freed. The requestor may renew the lease with an appropriate request.

Methods `lookup` and `register` are woven with the underlying code mobility mechanism of Java. In fact, in a Jini federation of services, objects are mobile and when classes cannot be loaded locally they are loaded from a remote machine by means of network protocols. Invoking `register` inserts an object in the Lookup service. The object is serialized, stored in the Lookup service and then deserialized by the client. This object can be a self-contained service or simply the mobile part of a two-tiered or N-tiered service. In this last case, it performs as a proxy as described by the Proxy design pattern [24]. Jini does not make assumptions about the communication protocol between proxies and

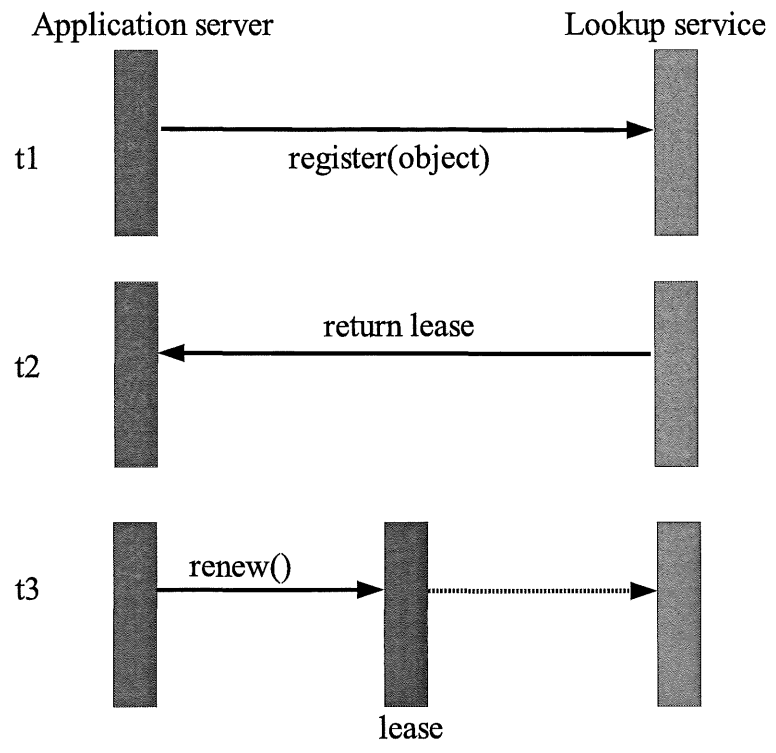


Figure 33: Registration of an object in the Jini Lookup service. At time t1 the Application server invokes register on the Lookup service. At time t2 the Lookup service performs the registration and returns a lease object to the Application server. At time t3, the Application server renews the lease.

back-end tiers. Any protocol is good, because it is deployed in a private communication where the proxy is a mobile client fully implementing this protocol. The proxy will be loaded, linked and executed at runtime in the recipient virtual machine.

The last consideration about object mobility is that Lookup service stores objects in a byte array form. In other words, while stored they are not instances of their own class but simply instances of `MarshalledObject` class. This fact lays a constraint on Jini architectures: all virtual machine must be capable to load the `MarshalledObject` class. This cannot be granted in some Java virtual machine profiles such as PersonalJava and J2ME.

Provisioning of class binaries to requesting virtual machines is performed by means of the underlying RMI implementation and no further API is added by Jini. Application developers would be expected to install jar files in HTTP servers. The architecture described in this chapter, make this installation automatic in application servers that are committed to serve classes via HTTP.

3.2.2 Implementation of the Service Queue

Our implementation of the ready-to-run service queue is based on Javaspaces [22]. A Javaspace is a Jini service which allows applications to coordinate each other via tuple-objects exchange. The Javaspace interface is quite simple and provides methods to read, write and take tuples[2] from the space:

```
Entry  read(Entry tmpl, Transaction txn, long timeout)
Entry  readIfExists(Entry tmpl, Transaction txn, long timeout)
Entry  take(Entry tmpl, Transaction txn, long timeout)
Entry  takeIfExists(Entry tmpl, Transaction txn, long timeout)
Lease  write(Entry entry, Transaction txn, long lease)
```

Writing a tuple means to insert a local instance of the tuple in the space. The method `read` takes a template tuple as argument and blocks until a tuple matching the argument is inserted by someone else in the space (Figure 34). A tuple T matches an template tuple Q if given $T = (t_1, t_2, ..., t_n)$ and $Q = (q_1, q_2, ..., q_n)$ where q_i are either equal to null or equal to t_i. When `read` completes it returns the tuple without removing it form the space. On the other hand, the method `take` behaves as `read` does but it alse removes the tuple from the space.

The method `readIfExist(tuple)` and `takeIfExist(tuple)` are non-blocking methods performing reading/taking operations. They return a null value if no tuple matches

[2]More precisely, the class implementing tuple in Javaspaces is called `Entry`.

the argument.

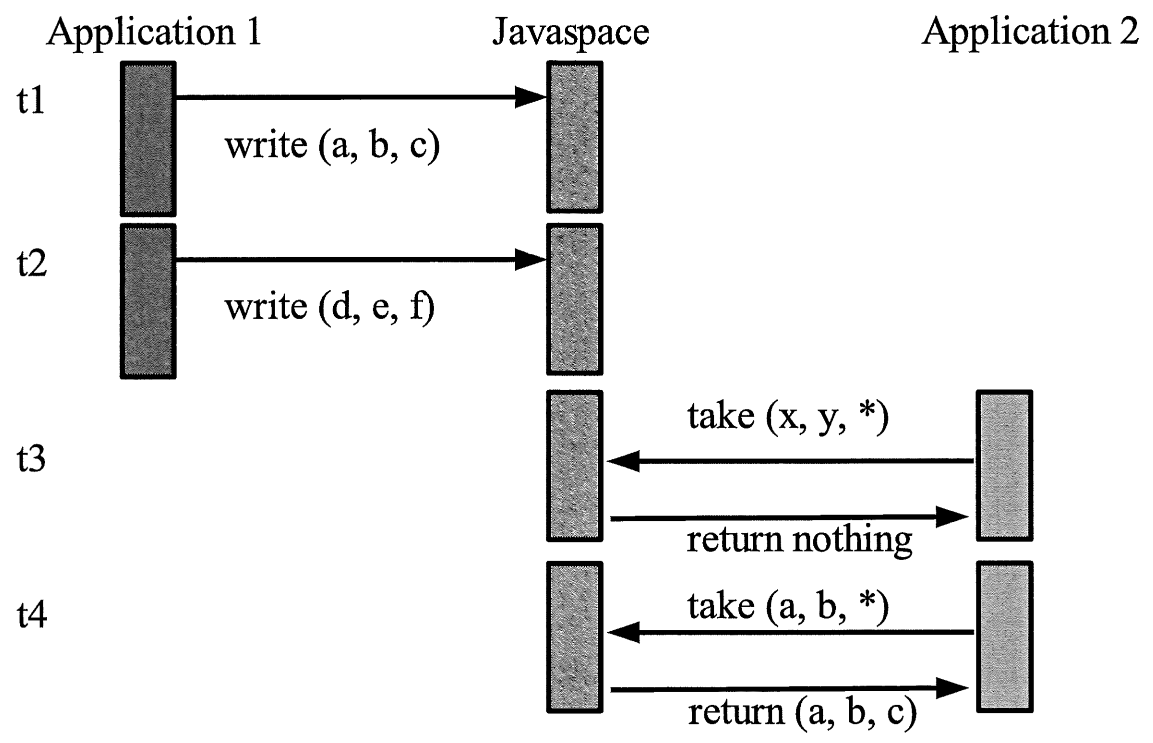


Figure 34: Tuple Matching in Javaspaces. At time t1, Application 1 writes a tuple (a, b, c). At time t2, Application 1 writes a second tuple (d, e, f). At time t3, Application 2 fails to take a tuple. At time t4, Application 2 succeeds to take a tuple.

The Javaspaces specifications do not define the behavior of the space when more than one tuple match the template.

Javaspaces interface is really simple and does not support the management of queues. Some algorithms have been proposed [32] to build more structured data types on the top of Javaspaces architecture. Our queue implementation is based on them. It implements methods to put objects in the end of the queue and to pick up objects from the front. Differently from Javaspaces, the `DistributedQueue` interface works with any subclass of `Object` and does not require they are tuples.

3.2.3 Service Model

We can say that a service model is the mix of various ingredients: domain objects, mate models, task models, and others. We will introduce them in this section. Some of them have to move to reach the user while other will run remotely. Mobile components are put into a delivery unit called `Servicelet`. A `Servicelet` recalls the concept of Applet but with some special features. Communications between the mobile Servicelet and the components that reside in the application server occurs by means of RMI.

The service model is composed of the following components:

- User interface: the service user interface is expected to be platform independent as it is unknown beforehand which device the user will use to access the service.
- Domain objects: they are the common objects shared by more applications in the same domain (i.e. music) and users are supposed to interact with them via user interface.
- Back-end methods: they implement the server-side behavior which cannot move to the user client device. For instance, an electronic payment service must reside server-side.
- User workspace: this is memory space allocated to store domain objects. It can be fault-tolerant if a underlying database is used to provide persistence. Objects stored in a workspace remain instantiated and running while the user is offline.
- Agents: in some applications the user can schedule some operations to be performed when he is disconnected. For instance, in a personal agenda an agent is committed to check incoming events and to notify them somehow (by email or by SMS). Such agents share the user workspace. They are not mobile agents for they always run in the application server.

Figure 35 shows components of a service and how they are partitioned among local and remote tiers.

We claim that user experience is greatly improved through the management of user sessions and the scheduling of off-line agents. A session is a container created the fist time a user connects to a service. The session contains the following elements: a workspace to store domain objects instances; connectors to external services like databases, mail exchanger, or any type of business methods; finally, agents that can manage objects in the session and invoke methods on connectors (for instance, an agenda agent can read user appointments and invoke an external short message service). Sessions can be joined and left any time and by means of any device. Thus, a session created with a mobile phone can be joined in a later time with a palm computer or with a desktop-PC. Agents may be created, associated to sessions and committed to perform some tasks such as content-searches or cultural events notification when the user is off-line. We believe that the possibility to create agents and to join and leave sessions using different devices in different moments can really improve the interaction between the user and a distribute information system.

The next section makes a step-by-step description of the deployment of a personal agenda service.

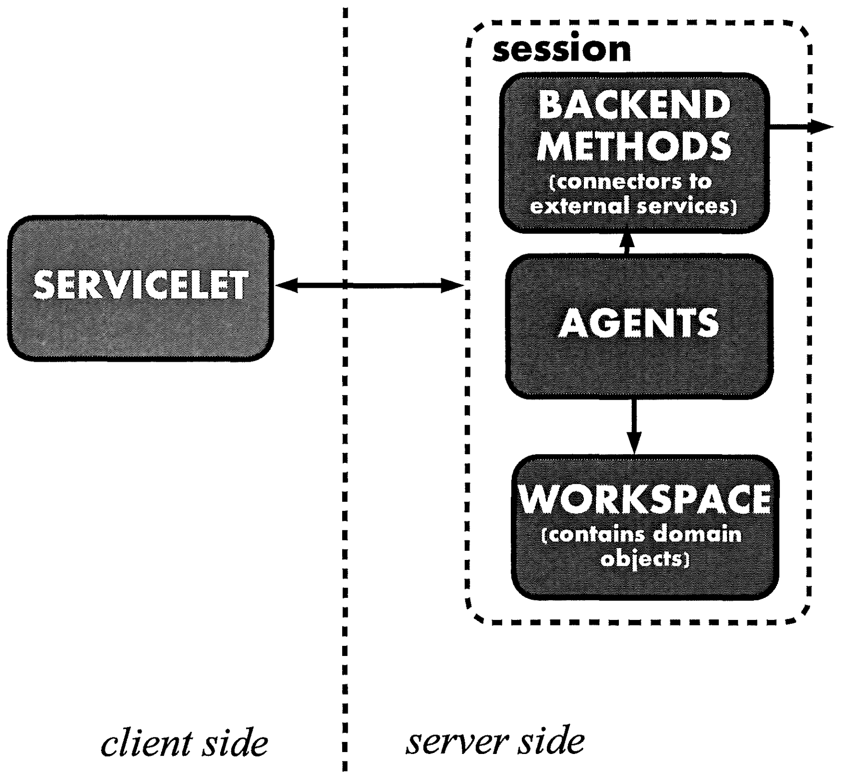


Figure 35: Service components partitioned between local and remote tiers.

3.2.4 Example: Personal Agenda service deployment

- *Service Pack.* The first step is to put into the queue the service pack released from the developer. The service pack contains a Java archive that stores classes and resources required to run the service such as configuration and multimedia files. The service pack remains in the `DistributedQueue` until an application server extracts it to perform next steps in deployment.

- *HTTP class publishing.* Once the service pack is opened by an application server all service classes expected to be loaded from remote clients are published in a HTTP server embedded in the application server.

- *Service session creation and initialization.* When an application server opens a service pack, it creates an instance of `ServerSession`. The `ServerSession` instance is a factory which creates a new instance of `UserSession` each time a new user connect to the service. So any user has a session which persist between connections. The `ServerSession` initialization is performed by the application server running the method `initializeService()`. Such a method is defined by the service developer, and in the case of the Agenda example, this method creates the backend methods[3], loads the agent classes, and defines two workspaces called `AGENDA` and `ADDRESSBOOK` to store appointments and contacts.

```
public void initializeService() throws Exception {

    //init of business method prototypes
    AgendaPrototype proto = new AgendaPrototype();
    Prototype[] protoArray = new Prototype[] { proto };
    getServerSession().setPrototypes(protoArray);

    //registering off-line agent classes
    getServerSession().registerAgentClass
              (AgendaAgent.class,"AgendaAgent");

    getServerSession().createWorkspace("AGENDA");
    getServerSession().createWorkspace("ADDRESSBOOK");


}
```

Figure 36: Initialization of the server-side part of Agenda Service

[3]Backend methods are implemented in form of Command pattern [24]. A Command is an object instantiated when the service is initialized and then stored in the server session. Once a new user session is created, a deep-copy of this Command is created by cloning and copied in the user session. So all user session have their own instance of back-end methods which are all clones of the original Command.

- *Servicelet publishing.* Once the service session has been created and initialized, the application server discovers the Lookup service and register the `Servicelet` that is the mobile part of the service. To do so, the `Servicelet` is first serialized. Then the resulting byte code is put into a `MarshalledObject` instance which is marked with the URL of the HTTP class server. This tag allows remote class loaders to correctly unmarshal object instances.

- *UserSession creation and initialization.* When a user connects to the Agenda service he/she uses an appropriate application called *ServiceViewer* which is fully described in Section 3.3.6. The viewer receives the mobile part of the service (an instance of the `Servicelet`). If the user connects for the first time, a new user session is created and initialized in the application server, otherwise he/she simply reconnects to his/her already instantiated user session. Hence, the user reconnects to edit/modify appointments already inserted in the workspace with another device. Any user session replicates the structure defined in the service session which generated it. Therefore it contains workspaces, back-end methods and offline agents.

  Speaking the design pattern jargon, we can say that a service session is the factory of all the user sessions.

- *Workspace creation.* When a new user session is created, its workspaces are also created.

- *Back-end methods.* As stated before, some features cannot be implemented in the mobile part of the code of the service. In the case of the Agenda, the user needs to send messages to some of his/her contacts from the address book. The code implementing message relaying is based on standard Internet protocols such as SMTP and it is implemented in the server-side of the service. `MessageRelay` is the class implementing the mail protocol. A new `MessageRelay` object is instantiated for each user session and stored in the user session itself. In this manner, any

`MessageRelay` can be configured with user specific settings. The Figure 37 shows the `MessageRelay` interface. `MessageRelay` methods must be remotely invokable, therefore the `MessageRelay` interface must extends the Java `Remote` interface.

```
public interface MessageRelay {
        void relay(Message m) throws Exception;
}
```

Figure 37: MessageRelay interface

- *Agenda Agent.* When the user is offline, the `Agenda` service keeps running. Every five minutes, the agent checks future appointments and if one is imminent it sends to the user a message. Of course, this simple behaviour may be enhanced.

3.3 Delivery to any Device

3.3.1 Introduction

Once services have been deployed they must be delivered to users. Our goal is to allow users to pick up the most convenient device and connect to the desired service. In the user interface we can hide the concept of service and let the user focus on tasks, but even in this case the objects implementing the required functionalities must be searched and loaded in order to start the interaction. Furthermore, the user could switch to another device as soon he/she has at hand a more appropriate device, and then he/she keeps working on the same task.

3.3.2 Delivery Context Analysis

One can think of delivery context as a vector

$$C = (f_1, f_2, \ldots, f_n)$$

where the single f_k is a feature in the context. Examples of feature are the RAM memory or the disk space. Basically, features are related to hardware/software capabilities of current device but they can be related to other environmental and user specific variables as well.

In fact, cultural features such as the user language or physical features such as the geographic position in longitude/latitude coordinates, the surrounding temperature and the air humidity are examples of delivery context variables that may affect the behavior of an application.

A unique definition of delivery context seems to need more than the definition of a static and fixed set of variables because given a class of applications some features are supposed to affect the computation while the others are totally irrelevant.

Nevertheless, as this work is devoted to the delivery of applications to any device, we can make a bare classification based on hardware/software features and on the user location:

- hardware features: CPU, clock, memory, disk, screen, vocal interface, keyboard, mouse.
- Software features: OS, markup language, markup language version, java version.
- Cultural features: language.
- Geographic features: longitude, latitude, altitude, speed, timezone.

Several methods can be used to code context relevant information in the body of a communication protocol. For instance, in the HTTP protocol such data can be inserted in the headers. The HTTPExt [68] framework defines how to extend the HTTP headers. Another solution for including device capabilities and user preferences in HTTP requests is the Composite Capability/Preferences Profiles (CC/PP) [53] framework where context relevant information are coded by means of XML/RDF.

3.3.3 Exploitation of the Delivery Context

Once the delivery context has been defined and gathered the question is "how to use it?". Applications can modify their behaviour according to one or more features. For instance, one of the most used cultural features is the language: once a Web service realizes that your browser is sending the header

$$Accept - Language : en - us, en; q = 0.5$$

it may send you English translated pages rather than original French or Italian pages.

Another possibility is to infer from which geographical region the user is connecting by the analysis of IP numbers. For instance, there are online databases which map IP-to-Location [43] in the range of a metropolitan area.

If one thinks of f_k as the coordinates of a point $C = (f_1, f_2, \ldots, f_n)$ in a hyperspace, C is one of the possible values for the context. An application can adapt itself with respect to its user interface, its server-side computation, its partitioning among different tiers and so forth. Some adaptation are strictly related to the application other can be common to more applications. For instance, the adaptation of the user interface to the end user device is a common adaptation.

In the case of pervasive computing, services must be delivered seamlessly to any kind of device. If applications are not able to adapt their user interface they become barely unaccessible to users. In terms of regions of the context hyperspace, a device neutral application could be delivered effectively in all regions of the hyperspace. In real world systems, it is not possible to cover all the possible delivery context regions but only a limited subspace, limited both in dimensions[4] and in range.

In our case, the MORE delivery architecture takes into account the following features:

- Remote classloading capability: values = (YES, NO)
- Markup Language: values = (HTML, WML, VoiceXML, HTML + Javascript, XML)
- Java: values=(J2SE, J2ME, PersonalJava, None)
- User Location expressed as $(longitude, latitude)$.

This set does not contains information such as the CPU power, memory, disk, screen size, voice I/O and so forth because the choice is made upon the following assumptions:

- If a remote classloading is available then the client is supposed to run on an high end computer with at least J2EE or J2SE and full graphic user interfaces such as the Java Swing toolkit. This assumption is not always true, in fact a host can be able to load remote class but also to provide other types of user interfaces such as vocal interfaces. Anyway we make this assumption for sake of simplicity and we will refine later the context analysis in case for the voice interface delivery.
- If remote classloading is not available, but the device has a pre-installed virtual machine then it is assumed to be a low-end device such as a Java enabled PDA computer or a Java enabled smart phone with Java.
- If Markup Languages are HTML, WML, VoiceXML, then the device is hosting a browser but it is not running a JVM (or the user decided not to use the JVM to connect the service). Then the service must be delivered across a proxy able to generate a user interface based on the appropriate mark-up language.

[4]If N is the number of features in the delivery context only $k < N$ are taken into account.

Yet these assumptions may seem too simplistic but they allow to design and implement a delivery architecture able to cover the following platforms:

- Desktop PC with J2SE;
- Desktop PC with Internet browsers such as Mozilla, Miscrosoft IE and Opera;
- Mobile phone with WAP browser;
- Mobile phone with J2ME;
- Mobile phone or PDA with HTML browser;
- PDA with J2ME or Personal Java;
- Interactive TV with HAVi [38] components;
- ISDN/PSTN phones or other voice devices connected via VoiceXML browsers;

When speaking of context/device neutrality we assume it as a good point for applications in pervasive computing. However, it is desirable for an application to be context dependent if from this dependency an added value can be achieved. For instance, if the device is a mobile phone then a service could exploit the ring tone to make the user aware that something is going to happen.

With respect to geo-referenced applications, the position variable is something expected to enhance the computation. For instance, a search engine could give higher priority to search results somehow "near" to the user location [9].

Task Models and Delivery Context

One good point in task model specification is that the application designer can provide, together with the service, one or more models and decide at design-time which model must be used to interact with the user in a given delivery context.

3.3.4 Issues Related to Java Classes and Code Mobility

The JVM is the running core of any Java program. This core is built under an API which is the same for any operating system and is literally a framework whence any application can be considered an extension. Since Java is a general purpose language, the Java API is quite large and it contains classes and interfaces for almost any topic related to data manipulation such as simple file and console I/O, complex data structures, XML processing, RPC and CORBA services and so forth. It is impossible to write a new Java program without dependencies to the Java API, in fact even if your class does not extend any other class in the API it must at least extend the root of the hierarchy tree: the class `Object`.

Therefore, loading the class `Object` also implies loading class `Class` and class `String` because `Object` has the methods `Object.getClass()` and `Object.toString()`. Then, it implies that also `Classloader`, `InputStream`, `Field`, `Constructor` and so forth are loaded as they are either return values types or argument types for methods of class `Class`. The propagation of dependencies spreads along all the Java API, then the simplest Java program causes huge classloading when it is launched.

Although Java is supposed to be a cross-platform language, this is true only if all dependencies of a program are correctly satisfied in the runtime environment. Java is neutral with respect to operating system but it is not neutral with respect to platform profiles. In fact, there is more than one Java API specification , or edition, depending on the device type. The most common specification is the Java Standard Edition (J2SE) which is targeted for desktop and laptop computers. The standard edition enriched with components for server side applications is called Java Enterprise Edition (J2EE). The Personal Java edition and the Java Micro Edition(J2ME) address the needs of small devices. Finally the Javacard edition is so called for it is aimed at the development of applets in smart cards.

This heterogeneity of profiles is an issue when we want our code to move from a device to another, in fact we cannot assume in general that the edition of the sender JVM is compatible with the edition of the recipient JVM. Unfortunately, compatibility is not ensured, as one could suppose, from smaller to bigger profile, in fact a class written for the J2ME platform simply does not work in J2SE.

3.3.5 Delivery Architecture

As described in the Section 3.3.2 the delivery context can be categorized according to the physical characteristics of devices. For instance, devices can be divided according to their computation power or according to their network interface type (Ethernet, PPP, etc.), or according to their input/output modalities (keyboard, touch screen, mouse, etc.). If one wants to take into account all the existing possibilities the number of cases to face up becomes unmanageable.

In our delivery architecture, the first step is to group devices by the capability to dynamically load and execute the mobile code of a service. We distinguish three fundamentals categories of clients:

- Fat client: the device can download and execute mobile code. This category includes devices such as desktop-PC, Unix workstations and laptops.
- Thin client: the device can execute local code and can download/upload data. Smart-phones and PDA programmable in Java fall into this category.
- HTTP-only: the device can only download/upload data and can visualize the information by means of some pre-installed applications like a browser. VoiceXML browsers, WAP phones and WEB kiosks fall into this category.

3.3.6 The Service Viewer

Users need an entry point to access online services. This entry point may change depending on the end user device. For instance, it could be a URL in a HTTP-only device or a stand-alone application in the Fat client architecture. Our goal is to provide the user with a tool which performs the discovery of available services and allows the user to access them.

We called this tool *Service Viewer*. Any Service Viewer is committed to the following tasks:

- user authentication,
- loading of the device profile,
- discovery of portals and lookup of services,
- dynamic loading and execution of mobile code,
- generation on-the-fly of the user interface.

With respect to these requirements, the building blocks of any Service Viewer are:

- Device Profile: this module contains any device relevant information like keyboard/pen availability, display size, color depth and user position. Once the device profile is loaded, it is stored in the execution context. Depending on the actual device profile, the Service Viewer should look up in the service if a specific task model definition has been provided for that device. In this case, the task model definition should be used to present the user interfaces.
- Lookup Component: this module searches all portals available across the network by either multicast or unicast discovery protocols and allows the user to access services published in the portals.

- Object Renderer: at runtime the Object Renderer first loads the domain objects. Then it generates a user interface for them. Finally, it links the domain objects and their user interfaces.

In the E-Mate project (Section 4) we have developed one implementation of Service Viewer for each category of devices. Next sections will discuss these implementations.

Service Viewer for Fat Clients

In the Service Viewer for Fat Clients the mobile part of the service runs on the terminal and the device profile is loaded from a local file. The devices that fall into this category normally can display services with WIMP user interfaces and connect to GPS via serial interface. The Service Viewer is then a local application containing all the building blocks cited above. It generates a concrete user interface based on windows, buttons and keyboard/mouse input modality. Figure 38 shows how the Service Viewer components and the application server components are partitioned in the architecture. Follows a description of Service Viewer components in this architecture

- The Lookup component is implemented and embedded in the Service Viewer. It is based on the underlying Jini API.

- The Object Renderer component is the implementation of MORE for the current platform. As said before, we assume that in a fat client we have a fully WIMP user interface, therefore the Object Renderer implementation generated Swing based user interfaces.

- The Device Profile is managed by a Java library which sends to the User Session information related to the device. If a GPS antenna is connected to the device, an appropriate Java library gathers this information through the serial port and adds it to the device profile. This allows geo-referenced services to adapt their computation to the end user position.

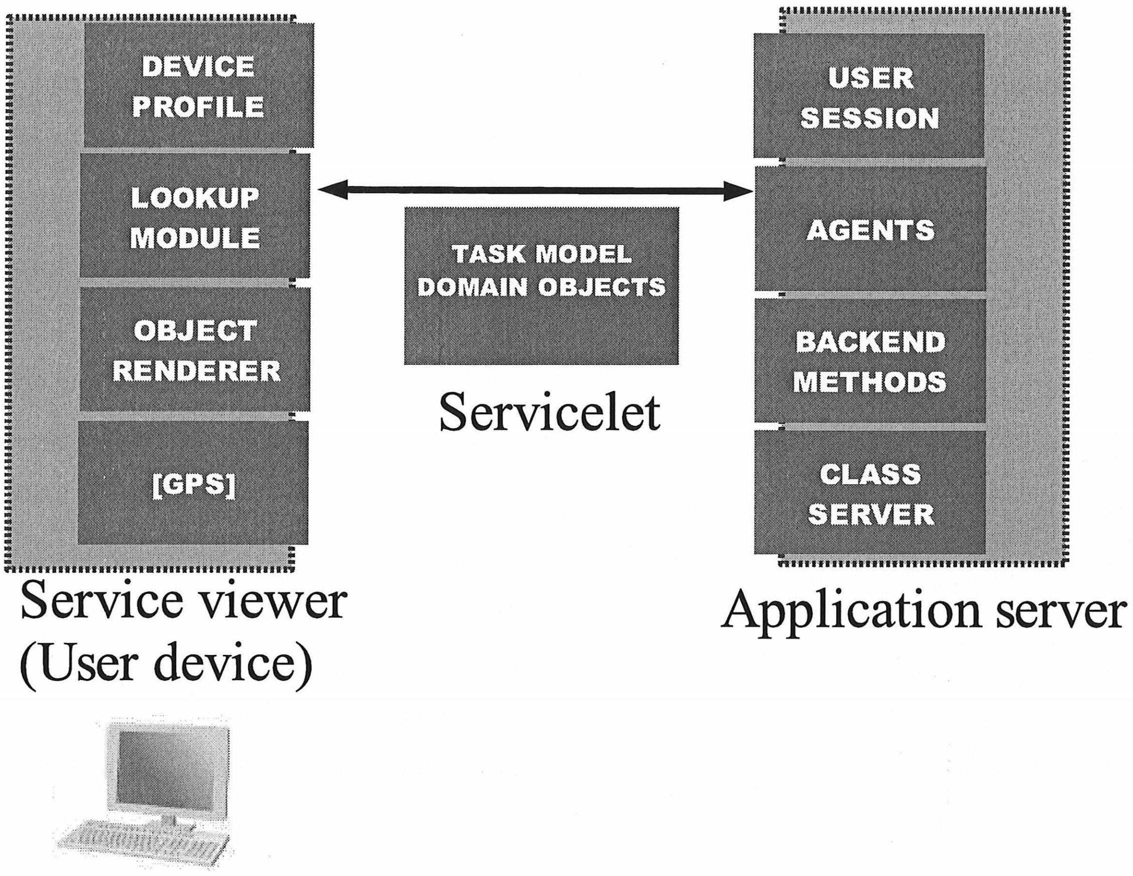


Figure 38: Delivery architecture for fat clients. The arrow represents a push/pull network connection and the Servicelet is the mobile code which moves at run-time to the client.

Service Viewer for Thin Clients

Thin Client devices cannot run mobile code and cannot directly connect to the Jini lookup nor to services, therefore the Service Viewer must be two-tiered. The local tier is the part of the viewer that must be installed in the user device beforehand. It loads the device profile and connects to GPS via serial interfaces. It is also provided with an object renderer implementation that generates at runtime the user interface. The remote tier of the Service Viewer runs on another machine (not the user device and not the application server). It performs the discovery of portals, the lookup of services, the downloading and the execution of the mobile code related to domain objects, task models and control of the application. The communication between local tier and remote tier of the Service Viewer uses XML as application level protocol and HTTP as the transport protocol. The communication is push/pull in the sense that the local tier can be triggered by events from the remote tier and the local tier can also request information to the remote tier.

Follows a description of Service Viewer components in this architecture

- The Lookup component is exactly the same component used in fat clients. It is deployed in the remote tier.
- The Object Renderer component is split in two subcomponents: the remote sub-component uses reflection to inspect domain objects at runtime and generate a description which is encoded in XML. On the user device, the local sub-component receives this XML description and instantiates local widgets. The XML communication protocol between local tier and remote tier is specified in appendix B by means of XML schema definitions.
- The Device Profile is managed by a Java library which sends to the User Session information related to the device. In the case of fat clients and thin client the Device Profile is computed in the end user device and sent to the User Session. If the platform allows Java to read from serial ports, then GPS coordinates are sent to the the User Session. The access to serial ports is a low-level feature that is implemented in C and linked to the JVM through JNI (Java Native Interface) [34]. Unfortunately, the JNI extension of JVM is available in the PersonalJava profile but not in the J2ME. This implies that nowadays a GPS can be connected to a PDA or to an interactive TV (if such a thing would make any sense) but not to a smart phone.

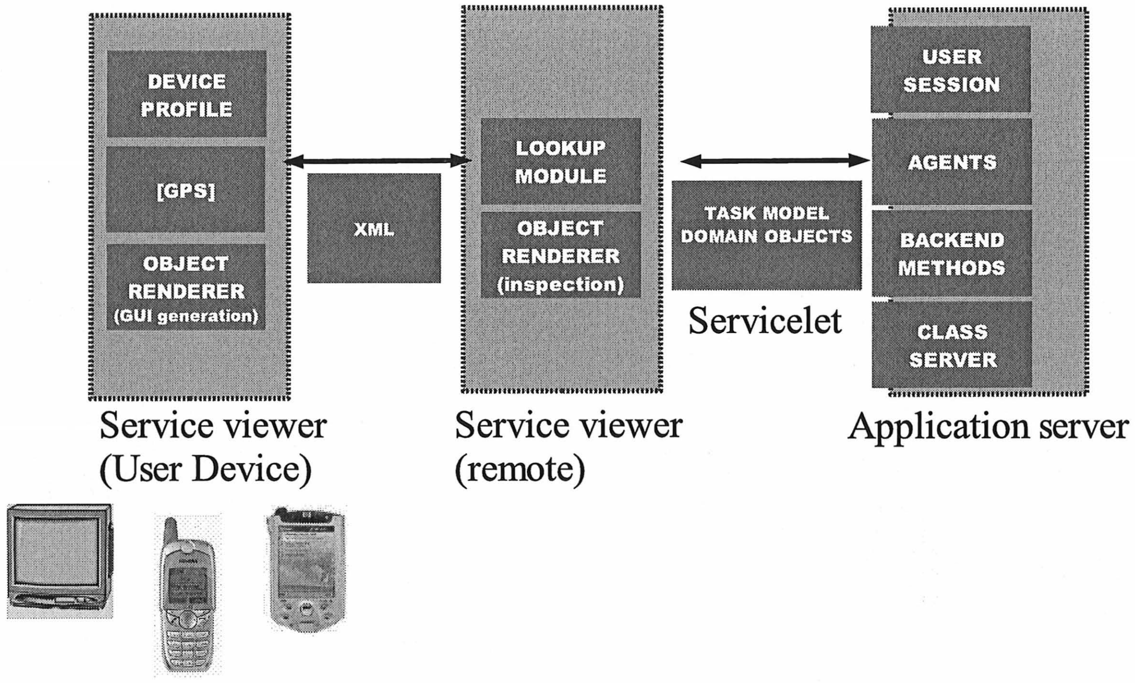


Figure 39: Delivery architecture for thin clients. The arrow represents a push/pull network connection and the Servicelet is the mobile code which moves to the remote tier of the Service Viewer.

ServiceViewer for HTTP-only Clients

In a HTTP-only Client the developer can install none of the Service Viewer modules so all the Service Viewer building blocks are remote (Figure 40). A pre-installed browser must be available. It is a third-party application bundled with the terminal that is not part of E-Mate. This pre-installed browser queries the Service Viewer via HTTP. The Service Viewer translates its response into the appropriate markup language. This is done on-the-fly by means of translation processors. This category of devices can usually not connect to a GPS system or a location service. In most cases, the device profile must be built by the remote Service Viewer using the HTTP header data or other relevant user agent information.

Follows a description of Service Viewer components in this architecture

- The Lookup component is exactly the same component used in fat and thin clients. It is deployed in the remote tier.
- The Object Renderer component entirely runs in the remote tier. The local tier receives a XML document already adapted for the current delivery context.
- The Device Profile is managed by a Java library which runs in the Web server. This component collects the client request and from the analysis of HTTP headers builds the device profile. This profile deeply affects the type of XML translation performed. In fact, depending on the device profile an appropriate translator is loaded. The translation is performed by means of XSLT [36]. Regarding the GPS, it is not assumed that coordinates are sent from the device because the Internet browser is a third party pre-installed application. Nevertheless, it is possible under certain conditions, to load a special add-on in the browser and connect to the GPS hardware if this is available[5]. The Internet browser can insert the current GPS coordinates in the header of HTTP requests. This is possible if the browser is extensible, a GPS device is connected and the user performs a minimal set of installation and configuration tasks.

[5]An example of Internet browser extension for GPS management is available in [27][9].

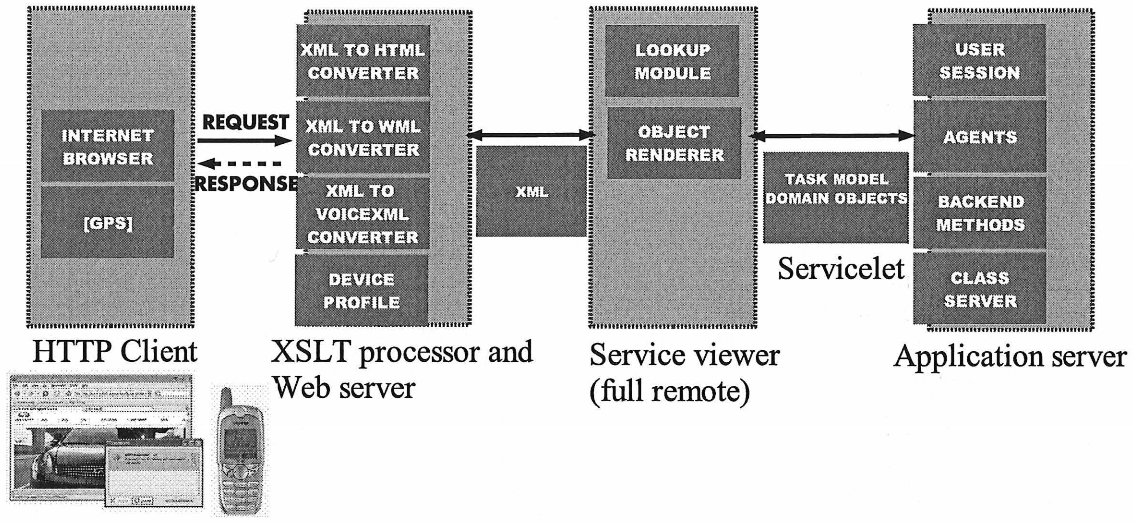


Figure 40: Delivery architecture for HTTP-only clients. Between the end user client and the Service Viewer there is a Web server which computes the current device profile and perform the appropriate XML to XML translation. In our case, we provide three possible translation: XML to HTML, XML to WML, and XML to VoiceXML.

3.3.7 Position Analysis

There are several ways to obtain the position of the user. The most known is the Global Positioning System (GPS) [30] that is based on the device capability to connect via radio links to a network of satellites. Other systems are specifically based on mobile phone networks such as Enhanced Observed Time Difference (E-OTD) and Cell Global Identity (CGI). The precision of those systems can range from 1-10 meters, in the case of GPS, to 30 km in the case of CGI. See [42] for a complete discussion about positioning for mobile phones. In our experiments only the GPS position was used. Therefore, the position collection is affected by two important limits: it is available only outdoor because the GPS works only on direct visibility between satellites and device; even outdoor, it is limited by buildings and reliefs because satellites has equatorial orbits and therefore they are close to the horizon in most of developed countries. Many technologies development are underway. For instance, the European Union is planning to launch its own satellite based global positioning system called Galileo. Other enhancements are on going to obtain indoor position by the combination of the GPS signal with the double integration of acceleration collected through inertial system, this kind of devices are still under experimentation.

A GPS peripheral is commonly connected to a computer through a serial port. A special string protocol called NMEA 0183 [44] is transported through the port at bit-rate of 4800 BPS. So collecting GPS data is merely a matter of string parsing. For this purpose, we use a Java library called Gipsy (Figure 41).

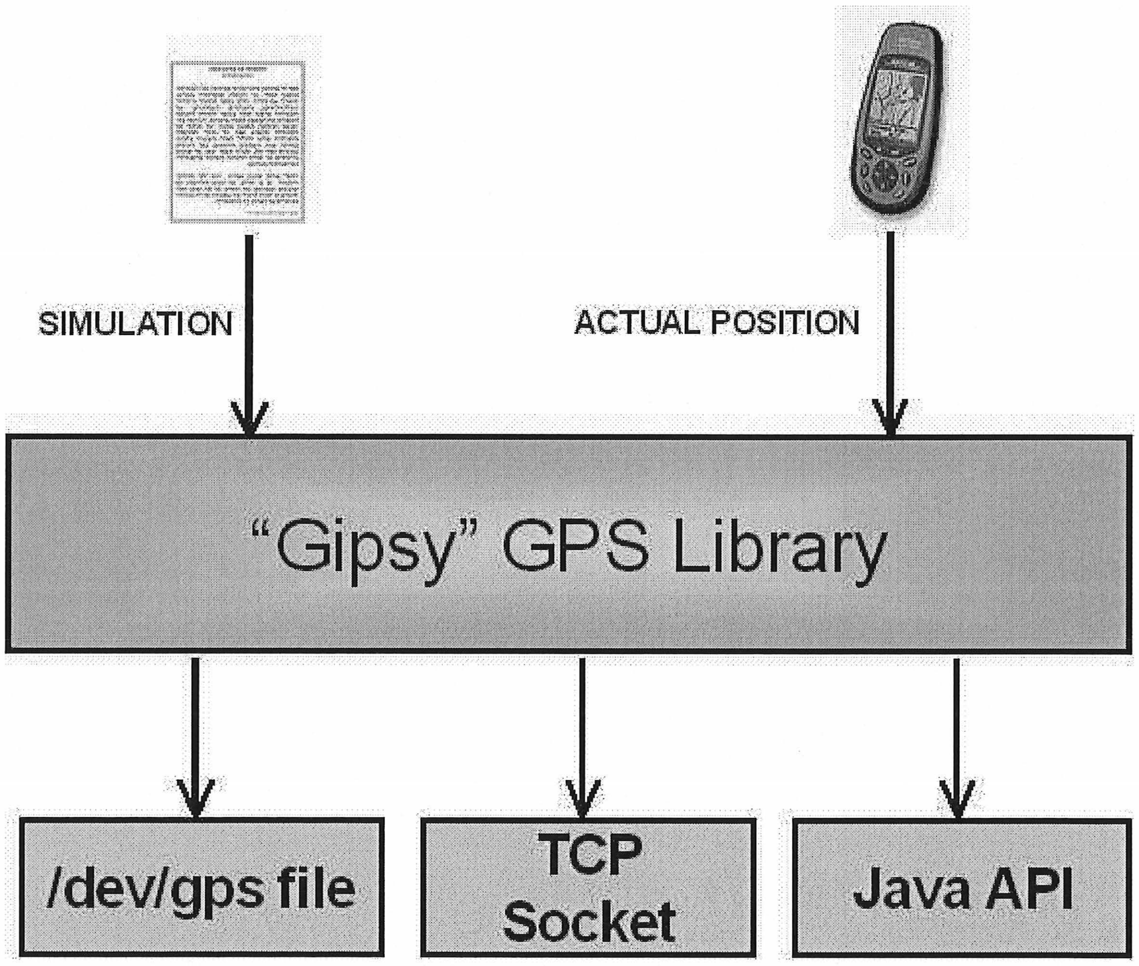


Figure 41: Gipsy is a Java library based on JavaComm. It can run as daemon or can linked as a library. Gipsy parses the output of a GPS device connected to a serial port. Programs can receive GPS coordinates either by direct method invocation, or by TCP/IP, or by file based data exchange

Chapter 4

E-MATE: AN OPEN ARCHITECTURE TO SUPPORT MOBILITY OF USERS

Solutions, architectures, and code presented in the previous chapters have been conceived to address specific issues such as the delivery on any device, the service lifecycle, and the task-driven and object-driven design approach. This chapter shows how these solutions were used to develop prototyes of significant size for real world applications. Most of these prototypes were done in the context of the E-Mate project [10]. E-Mate was a 24-month research program which has people with different competencies: technicians, programmers, researchers, teachers, students and GIS experts. Therefore, it contains more features than the ones described in this dissertation. Particularly, E-Mate focus is on three scenarios in the field of multi-modal, mobile, personalized and location-based software applications. Such scenarios were defined to elicit the technical requirements for the framework of E-Mate.

This chapter describes four applications. It shows that our work sets the foundations for real-life applications. These applications are: an Ubiquitous travel assistant

(Section 4.1), a mobile lesson management and deployment system (Section 4.2), a crisis management system (Section 4.4), and finally an investor decision support system called MKTS(Section 4.5). The chapter is closed by a section describing models, design goals, and issues about the composition of different E-Mate services.

4.1 A Travel Assistant: T-MATE

4.1.1 Introduction

In this section an Ubiquitous travel assistant called t-MATE is described. The application addresses three use cases: user profiling, macro-planning and micro-planning. Each case focuses on a specific type of devices and on a different delivery context. In the range of devices involved in this scenario are included Internet-digital TV, desktop computers, cellular phones and personal digital assistant (PDA). The assistance to a traveler is personalized and position-aware. The variety of tasks and configurations is luxuriant. Some are performed when the user is disconnected. Some require him to be on-line. Some require thin-clients, others are better suited for fat-clients. Some involve the cooperation between distributed remote services. Other are stand-alone services written from scratch. This scenario was implemented as a whole but we divide the presentation in three parts and for each we sketch the status of the implementation.

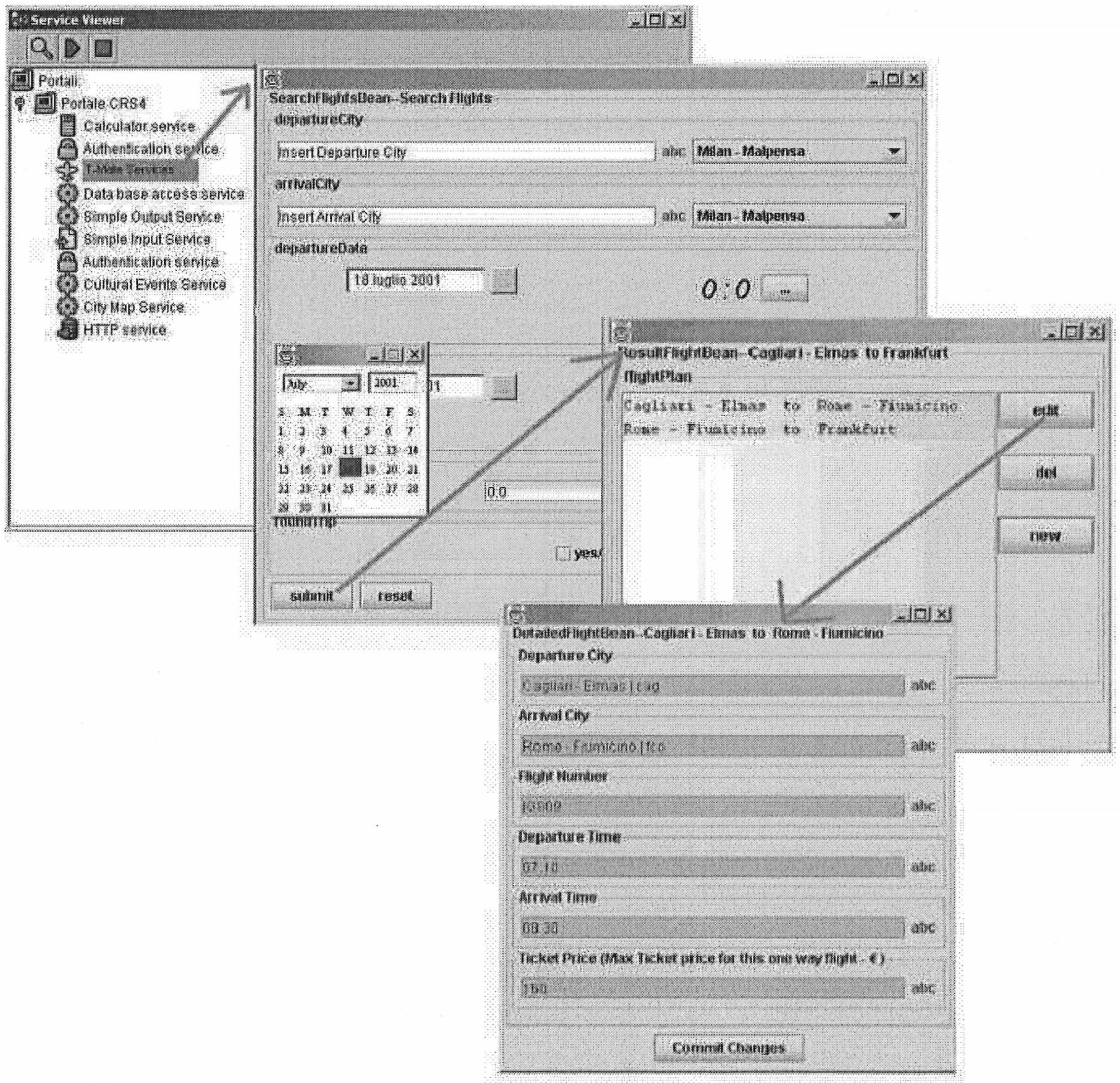
4.1.2 User Profiling

The first use case relies on profiling the user when he is listening to interactive-TV. An off-line agent, called "Profile Manager", is loaded on the set-top box. This agent gathers information and refines continuously the user profile. The user profile structure is defined according to services ontologies and requirements. User profiles are refined either automatically in the background and/or at the explicit request of the user. The automatic profiling system makes inferences on preferences and interests according to the semantic information encoded in the video stream (MPEG4 and MPEG7 [13]). For instance, if the user spends most of the time watching TV series on angling, the system should refine the profile with this information. On the other hand the user can explicitly assign a score to a video content; for instance, if during a television documentary about Kenya the user marks the program with the maximum score the system can infer interest about a Kenya sojourn. Thus, the user profiling is refined gradually over time. The Profile Manager connects periodically to the service servers to update the user profile available for a service. Once ready to go on vacation, the customer requests proposals to his travel agency service. To build these proposals, the agency will use his profile built while he was listening to the TV. Obviously profiles may be built using information coming from multiples sources, electronic journals, web browsing, etc. The profiling engine indeed needs only a semantic encoding of information and a function that computes the interest of a topic selected in the ontology.

4.1.3 Macro-Planning

The second use case focuses on the macro-planning of the journey. It mainly involves personal computer, browsers and e-mails. During this phase the user connects to the planning agent service in order to plan a trip and submits, by means of the user interface of the service (Figure 42), his requirements about the travel such as the total budget, the preferred hour of departure and so on. Hence, the planning agent will use its knowledge of the travel industry to plan an appropriate itinerary.

The planning agent should also arrange hotel accommodation for the duration of the traveler's stay, and should also provide a selection of local attractions that might interest the traveler during his stay.



departureCity: Insert Departure City abc Milan - Malpensa

arrivalCity: Insert Arrival City abc Milan - Malpensa

departureData: 18 luglio 2001 0 : 0

July 2001

0.0

yes

flightPlan:

```
Cagliari - Elmas  to  Rome - Fiumicino
Rome - Fiumicino  to  Frankfurt
```

Departure City: Cagliari - Elmas | cag abc

Arrival City: Rome - Fiumicino | fco abc

Flight Number: 10909 abc

Departure Time: 07.10 abc

Arrival Time: 08.30 abc

Ticket Price (Max Ticket price for this one way flight - €): 150 abc

Figure 42: User interface of the t-MATE service accessed by a PC. The user builds a trip plan submitting requirements like the maximum budget, the preferred hour of departure and so on.

4.1.4 Micro-Planning

The last use case occurs while the user is on the road. Once the plan has been approved by the user, the planning agent keeps working off-line to handle any event or change that might affect the plan such as adverse weather forecast, strikes and so forth. The main devices involved are PDAs and cellular phones.

During the night, the traveler's personal agent achieves a micro-planning for the next day. For instance, the planning agent works off-line searching cultural events in order to fill the tomorrow agenda of its user. It uses weather forecasts to select the activities, museum if rain is expected, beach otherwise. As far as personalization is a concern, the agent needs to know the user preferences (user profile). Events are screened out by inquiring the profile manager and those matching the user profile are stored in the user personal agenda as appointments. The personal agenda notifies the user whenever an event is imminent by means of SMS messages. The personal agenda is a self-contained service accessible from any device with respect to multi-channel capabilities of E-Mate. Therefore, the user can connect to his agenda with his cellular phone and manage his appointments while he is in line for visiting a museum or if he's moving through the city by train. Furthermore, the agent can act as a tourist guide, providing information according to the physical user position; for instance, the traveler may ask: *what is the name of the church I am in front of?*. The traveler may point new interests to his agent. The agent will then search information related to these newly expressed interests and adapts the user's agenda accordingly. The travel agency may offer last minute promotions for shows to its customer through the customer's t-MATE.

4.1.5 Implementation Details

To implement the *User Profiling* use case we have developed an E-Mate service called `TV-Log` (Figure 43) to be delivered in the TV set-top box. Our set-top box was provided

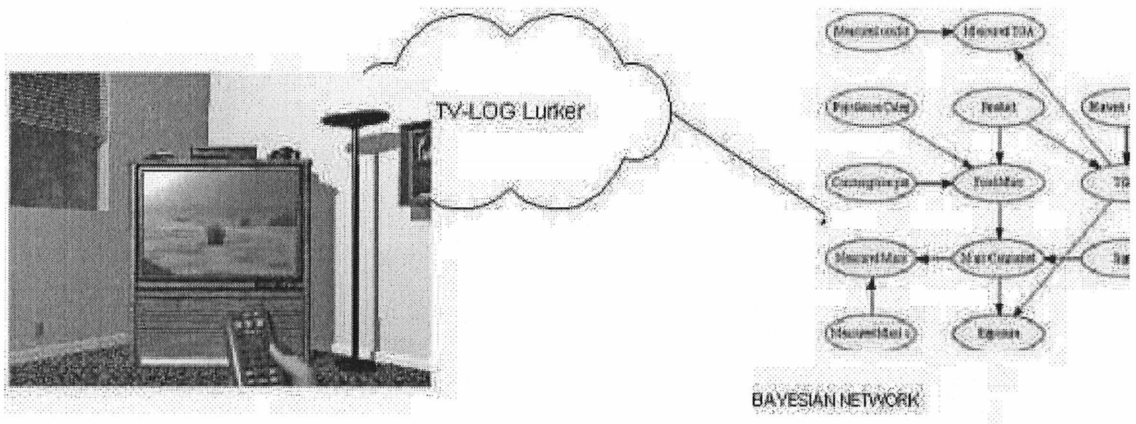


Figure 43: The TV-Log is an E-Mate service which is delivered in the digital TV and registers information about user preferences. This spying activity is used to populate a bayesian network which is the knowledge based used to personalize service delivery to users. This architecture is simply a proof-of-concepts and personalization issues are beyond the scope of this thesis.

with a special distribution of Linux, a JVM, and a Multimedia Home Platform (MHP) implementation consisting in layers HAVi [38] and DAVIC [17]. The HAVi layer is a Graphical User Interface toolkit while the DAVIC API is an interface to internal features of the digital TV such as tuning, channels, and other low-level controls.

To implement the *Macro planning* use case we used a fake database containing only a out-of-date subset of flights sold by real companies. The interface with the online flight reservation system was out of the scope of this work.

The current implementation of the planning algorithm is summarized as follows: (i) search the minimum route, (ii) repeatedly make queries to the database looking for the corresponding flights, (iii) remove the unavailable routes, and (iv) return the minimum-cost route. The first phase is realized using an A*-like algorithm, the result of this phase is the list of cities and airports involved. The agent repeatedly makes queries to the database in order to find all the possible flights. Among the resulting flights the agent has to screen out flights that are unavailable at the given departure date. Finally, the agent sorts the flights with respect to the cost and returns the minimal cost itinerary to the user.

To implement the *Micro planning* use case we developed various modules: a EventAgent is committed to seeks cultural events from various datasources such as online databases and Web sites. Another module is the Agenda which is accessed by the EventAgent to insert new cultural events. Finally, an interface to a Short Messaging System (SMS) has been implemented to communicate incoming events to the user when he/she is disconnected.

Figure 44 shows correlations between modules in the t-MATE scenario.

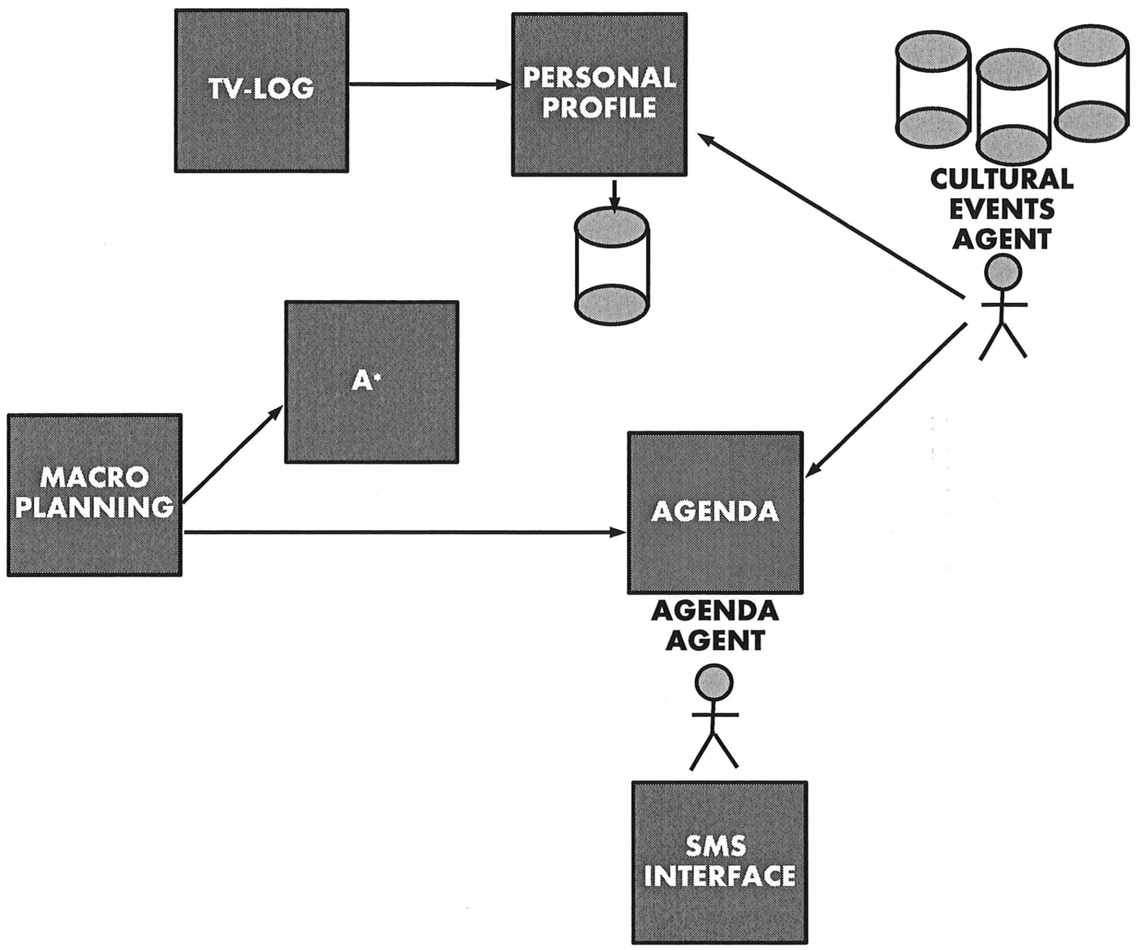


Figure 44: User activities during TV-watching are sent from the `TV-Log` to the Profile Manager which populates a bayesian network. The Macro planning module provides a GUI for creating trip plans. Trip plans are inserted in the Agenda. During Micro planning, an Event Agent seeks new cultural events co-located with the user. Events are filtered by the Personal Profile manager and only those matching the user profile are inserted in the Agenda. The Agenda is equipped with an Agenda Agent which notifies imminent events to the user by means of SMS messages.

4.2 Mobile Lesson

A mobile lesson [25] is an educational activity where teachers prepare questions relevant to places or monuments the students have to find. The questions are asked to students at the very moment the monument or the place is reached. This section describes both how such a lesson was experienced by students and teachers and how the system was implemented.

4.2.1 Experimentation

Teachers and students of an high school in Sardinia (Italy)[1] experimented a mobile lesson for the archaeological site of Nora. This site is very interesting from an historical perspective because it contains both Punic and Roman ruins. The lesson was performed in June 2001 with a class of 12-13 years old students.

Nora was chosen because the pedagogical program of these students included the knowledge of the architecture of cities in the Ancient Rome.

From a technological point of view, the requisite was a software application for both the edition of the lesson content and the management and monitoring of the students work on the field.

While preparing the mobile lesson, teachers first identified the zones of interest, then they selected a set of hot spots for each zone (Figure 45). Hot spots correspond to precise location. Knowing the exact position of a student enables to ask questions on what she can see. So teachers first went to the chosen site with a GPS system and pointed out the coordinates of each hot spot. The objective was to bring students to discover these points once on the field. The hot spot GPS position is a 2-tuple like East, 38 59 3,80 ; North, 09 00 59,72.

[1]The project would not have been possible without the full support and participation of the Scuola Media Statale no 2, Assemini, Italy. In particular, the collaboration of the teachers Mrs Maria-Cristina Sanna and Mrs Giovanna Arru was invaluable.

Each hot spot was then associated with other information. First a label like the *roman theater* gives the name of the hot spot. Obviously, the students have to find a significant place, for instance the theatre, and not just a GPS position. But this is not enough, because the theater is indeed a squared area whose side is more than forty meters long. The students must find the right position picked up by the teacher, near the theater. Why the teacher chose this precise point is a question they have to answer. When some difficulties occur to find the right place, explanations, help and hints are supplied progressively. Their form may be as wide-ranging as the teachers imagination can sustain: charade, riddle, description, etc.

Once on the field, teams of two or three students using laptops connected to a GPS system had to discover the significant hot spots previously identified by the teachers. They can wander wherever they want. Since the site is fenced, letting them free was not an issue. When students thought they were at the right location, the one identified by teachers, they asked the MobileLesson for confirmation. The current GPS position is then compared to the GPS position taken by the teacher. If the current GPS position is close enough, students have access to questions related to it. They may be general questions about the reached place but often they are questions about what the students can see from this precise location. Students have only to turn or move slightly to observe the place and discover or infer the right answers. A score was associated to each hotspot and each questions to motivate the students and to give them the feeling of a game. If the position was wrong, more information about the place is supplied. Receiving it, students have to move and ask again the software if they are at the right place or not. Latitude and longitude were used to represent a GPS position. For convenience, it might be possible to transform them into UTM coordinates [11] expressed in meters using a specific algorithm but, for the purpose of the experiment described here, it was not necessary. Many tests have been carried about the precision of GPS data and ten meters uncertainty was considered acceptable.

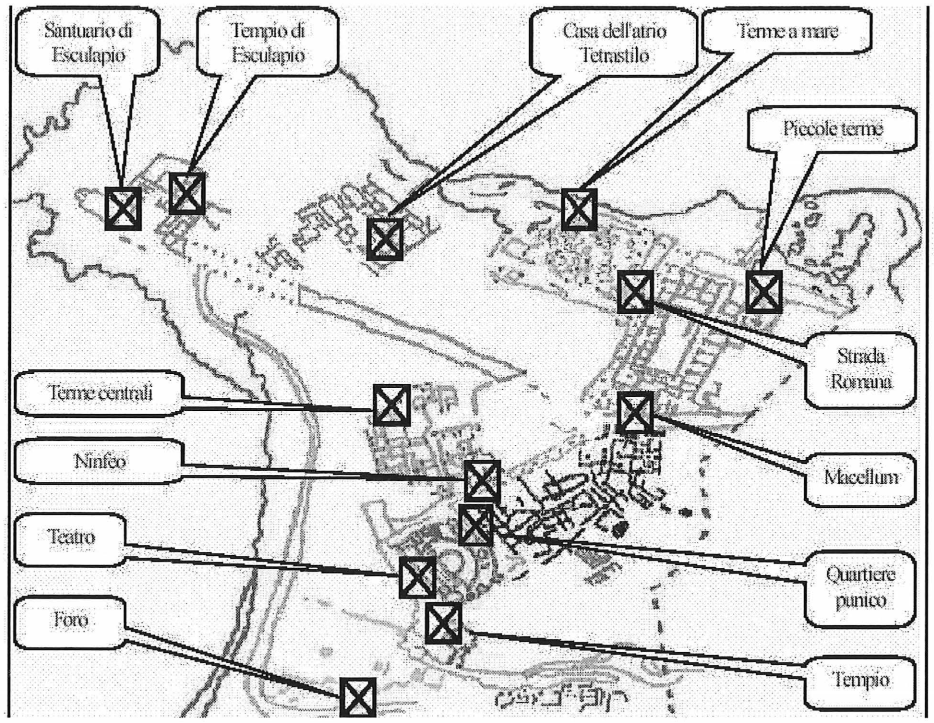


Figure 45: Hotspots chosen by teachers for the Nora mobile lesson

4.2.2 The Technology

The first version was a stand-alone application. Then it was re-designed as a service based on the the E-Mate platform where domain objects are accessed by means of user interfaces automatically generated by MORE (Figure 46 and Figure 47). It uses the general infrastructure described in this dissertation concerning design, delivery and deployment of services in a pervasive computing context. Software and devices used on the field are completely different but the content itself of the lesson remains the same. We also integrated results of the experimentation and comments of all actors involved in the process to improved the way of presenting data and we added modules not implemented previously. Students on the area of the lesson, are now using a PDA. Only a part of software is installed on PDA and assures all connections with the school web site and the remote part. The PDA is equipped with a GPS system that automatically gives its position coordinates. This position is transferred to the remote architecture and so the service can detect if a student is facing an hot spot of the lesson or not.

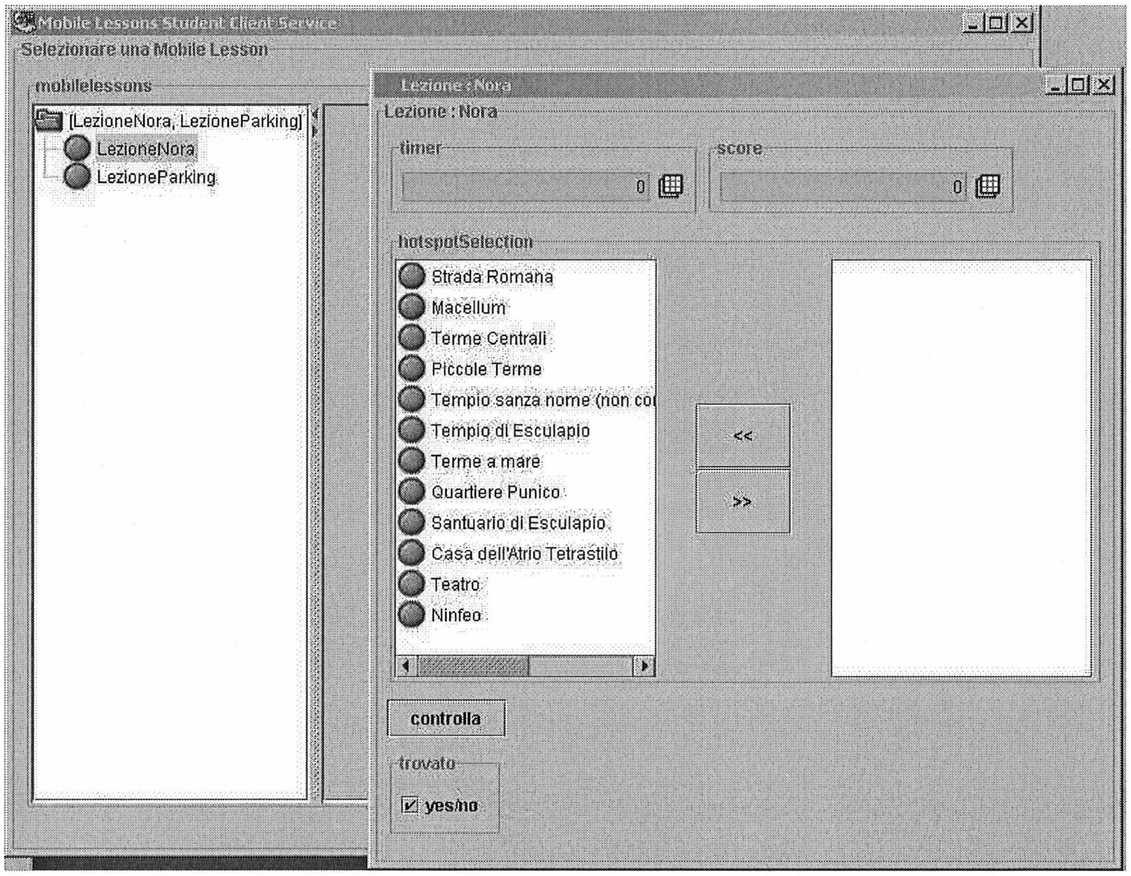
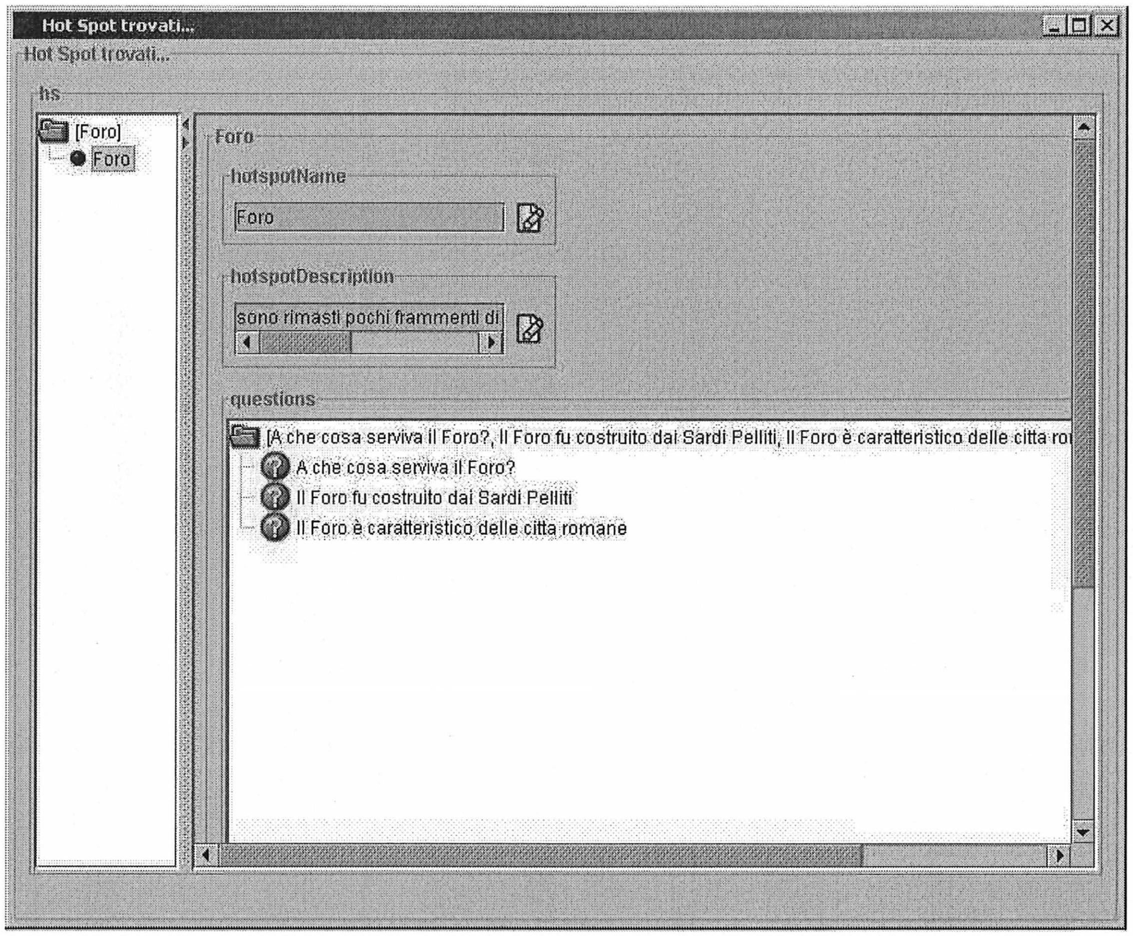


Figure 46: The right panel shows the lessons published in the system. Any lesson can be selected and the set of hot spots is presented to the student. On the field, the student can try to guess which hot spot he/she has reached.



[Foro]
Foro

Foro

sono rimasti pochi frammenti di

[A che cosa serviva il Foro?, Il Foro fu costruito dai Sardi Pelliti, Il Foro è caratteristico delle citta ro
A che cosa serviva il Foro?
Il Foro fu costruito dai Sardi Pelliti
Il Foro è caratteristico delle citta romane

Figure 47: Once the student has guessed the hot spot the system asks some questions relevant to the hot spot.

4.3 Evacuation Plan

4.3.1 Application Description

The mobile lesson application is a distributed software application that delivers an adapts geo-referenced information on request. Figure 48 gives the architecture of the system that supports the mobile lesson. Only a part of the service is loaded on it, the terminal tier. As several PDA are simultaneously connected an HTTP server is required to manage for each device the remote tier of the service. It find its resources in the application server that deployed it. The HTTP server itself finds the remote tier of services required by the users in the service portal where the application server has published them after their deployment. Data exchanged between PDA devices and the HTTP server is contained in an XML document.

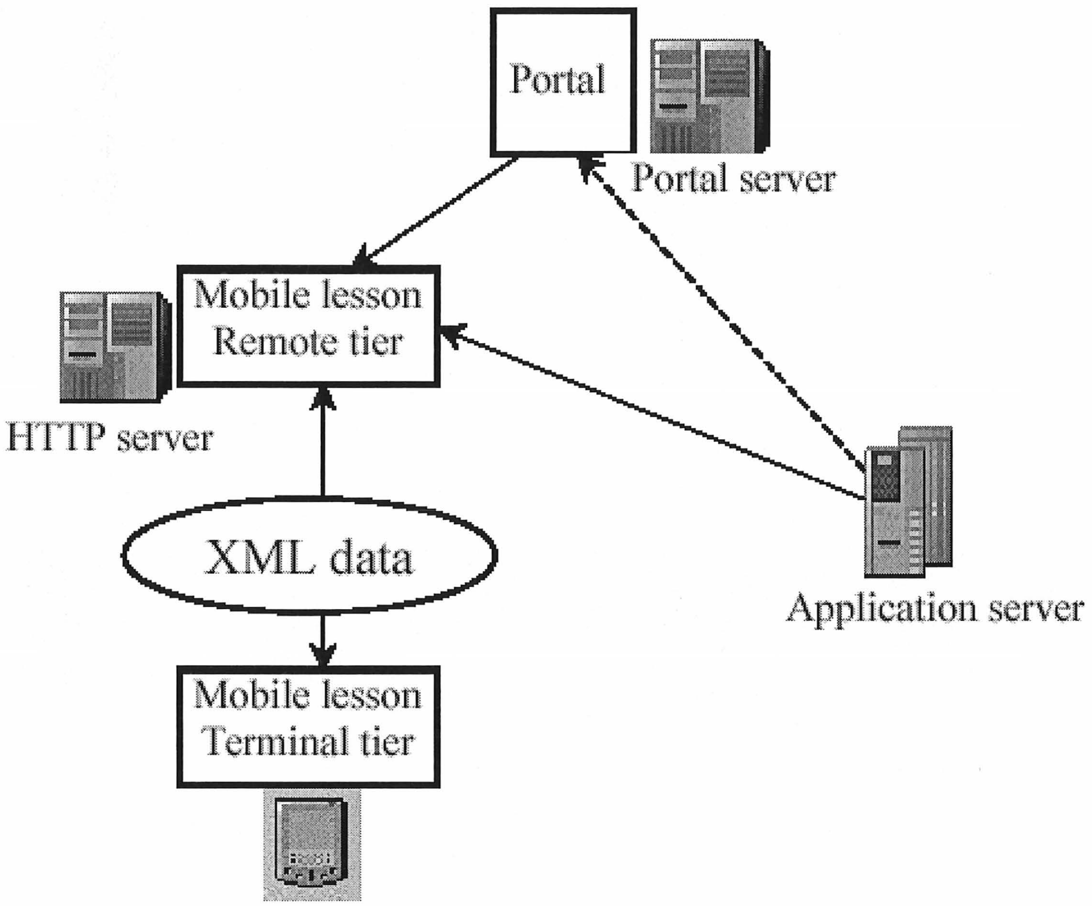


Figure 48: Communication in the case of distributed architecture. Notice how mobile lesson maps to general delivery architecture for thin clients.

4.4 Evacuation Plan

This application [19] refers to a hypothetical oil explosion, having its epicenter in Monte Urpino, the "green lung" in the town of Cagliari, occurred on a winter weekday between 8,00 and 9,00 a.m. Also, it is assumed that the zone to be cleared falls inside a circle area having its center point in the explosion epicenter, and radius of about 1 km.

4.4.1 Scenario Description

The considered area is approximately 220 ha, one third of which are occupied by the park. The rest is characterized by residential and commercial neighborhoods, where some schools, the main police department, the Court, and a hospital reside. The zone surrounding the Court is densely populated and characterized by a high traffic density, with congestion phenomena, especially in the early morning. The road network of the zone, about 27 km in length, is a urban type, classified in primary roads, district and inter-zone roads, and local streets for door-to door traffic. Approximately, one third of the roads is one-way. Parking along the lanes is allowed almost everywhere. Six intersections are signal controlled. Traffic is mostly composed by car; however, public transport services operate in the main roads of every neighborhoods. All the streets include sidewalks.

4.4.2 System Description

For the evacuation application, services have been designed by taking into account the different figures who can be involved during an emergency. Services are implemented as E-Mate services and they rely on the architecture for service deployment, service discovery and delivery-on-any-device described in Section 3.

Evacuation plan related entities are:

- the citizens, considered as vehicles, that have to evacuate the area;
- the decision makers (or coordinators), that supervise and co-ordinate the operations;
- several aid forces (police, red cross, fire fighters, etc.), that have to operate in the area of the disaster under the control of the coordinators;
- the traffic inspectors (or observers), who have to cover predefined neighborhoods to oversee and constantly send data (with their PDA) for updating the traffic conditions;
- the traffic directors (or actors), who have to stand in given junctions and follow the coordinators instructions to direct cars toward the egress routes.

The system is composed by the following services:

- Coordinator Service: it is the decision makers main interface, and the core of the system. The coordinators have the possibility to visualize all the information concerning the ongoing situation, to communicate with the other services of the system, with the observers, the actors, and the aid forces;
- Database Reader Service: it is a data source that loads, memorizes and manages the data related to the observers, actors and to their position, and to the characteristics of the road network, via a DBMS (Data Base Management System) connection;
- Operational Research Service: it executes the transportation algorithms that initialize the traffic flow on the network, that update the changed traffic conditions, and that calculate the shortest path;
- GIS Layer Service: it is a data source that builds the required layers and their attributes (Open Map Graphiclist), after verifying the properties of the necessary shapefiles or raster images by reading an XML metadata file;

- Map Server: it returns different maps such as map representing the traffic conditions at any time, map showing the observers position, the shortest path, etc;
- Device Registration Service: it registers each device present on the net and memorizes the position of the observers on the field;
- Device Register Messages: it manages the observers communication providing the Database Reader Service with the observers information;
- Traffic Flow Simulator: it is a multi-agent system that simulates the evacuation scenario. Given the impossibility to test the system in effective emergency condition, tests used a multi-agent system integrating several human behavioral parameters and fortuitous factors, such as different types of accidents. The agents represent the road lanes (containing groups of cars), the observers and the actors. As perceived by the agents, the information is sent to the Device Register Messages Service. In order to simplify the simulation, it has been assumed that the aid forces had limited discretionary power, having just to follow the coordinator instructions, and, therefore, they have not been identified by agents.

4.5 MKTS

The objective of the MKTS project was the application of mobile and Ubiquitous Computing technologies in the domain investments planning. The services developed in the MKTS project are E-Mate services, therefore rely on the distributed architecture for the deployment, discovery and delivery on any device. GIS products have been integrated in order to provide geo-referenced data.

4.5.1 Features of MKTS

The features implemented in MKTS are:

- decision support system
- ATECO [2] based classification of enterprises;
- funding and tax-exemption knowledge base for investors.

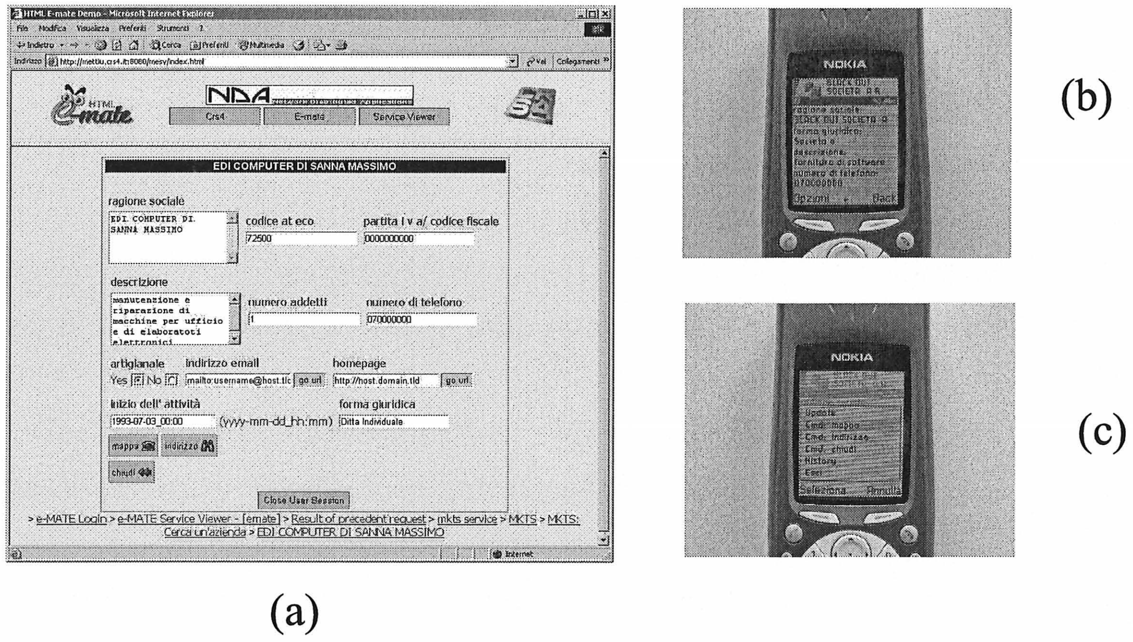
The decision support system is aimed to help investors in planning the installation of new industrial or commercial sites. Inferences are obtained using ontologies and bayesian networks.

The user interface leads the user through a sequence of question related to the nodes of the Bayesian network. Based on the answers provided by the user the system suggest a set of possible locations for the new site.

The ATECO classification allows user to browser categories and to consult a detailed view of any registered enterprise.

All the MKTS services are accessible by means of desktop computers, Web browsers, palm size computers and smart phones (Figure 49).

[2] ATECO is an enterprise classification system developed by Istat, the italian national institute of statistics



HTML E-mate Demo - Microsoft Internet Explorer

http://mettiu.crs4.it:8080/mesy/index.html

EDI COMPUTER DI SANNA MASSIMO

ragione sociale: EDI COMPUTER DI. SANNA MASSIMO	codice at eco: 72500	partita i v a/ codice fiscale: 0000000000
descrizione: manutenzione e riparazione di macchine per ufficio e di elaboratoti elettronici	numero addetti: 1	numero di telefono: 070000000
artigianale: Yes [✓] No [ ]	indirizzo email: mailto:username@host.tlc	homepage: http://host.domain.tld
inizio dell' attività: 1993-07-03_00:00 (yyyy-mm-dd_hh:mm)		forma giuridica: Ditta Individuale

> e-MATE Login > e-MATE Service Viewer - [emate] > Result of precedent request > mkts service > MKTS > MKTS: Cerca un'azienda > EDI COMPUTER DI SANNA MASSIMO

Figure 49: A *Enterprise* domain object presented to the user in a Web page (a). The same domain object accessed from a mobile phone in (b) and (c).

4.5.2 Lessons Learned

Some interesting issues arose during the development of this application. The first issue was about interactive maps. In fact, the application requires maps to be visible in all platforms. Thus, the initial set of classes provided with the MORE implementation needed to be extended with some geographical related types. The choice was to define a new Java interface called `Situated` (Figure 50).

Any object implementing such an interface must expose either lat-lon coordinates or a topological information like 1162 Park Avenue, NY, USA. Then a specific runtime editor for a `Situated[]` was implemented for Java Swing, Java AWT, HTML, J2ME platforms.

The rationale is that an array of situated objects can be rendered in a user interface by a map where the individual situated items are spots in the map. The user can point and click a spot, read some related information, use all the interactive maps related functions such as zoom in, zoom out, select a layer and so forth.

Therefore, such an editor is a client of a two tier architecture where the server is a GIS engine. The data between the GIS server and the editor are exchanged in the GML format. To improve performances in mobile networks, where the bandwidth is still an issue, a special compressed form of GML, called compact GML (cGML[20]), was used.

The second issue was about the interaction with the recommendation system. During the interaction, the system asks the user a question and then proceeds to the next question until a sufficient amount of information is gathered to provide a suggestion. This pattern of interaction cannot easily modeled with an object-driven design for the considerations already exposed in Section 2.1. This experience led the development toward the integration between MORE (the object-driven framework) with PLANES (the proof-of-concept task centered design environment).

Finally, one of the interaction requirements was to allow users to browse categories of enterprises with a total amount of thousands of possible objects to be loaded into memory and contained in a `java.util.Collection`. This fact led to an inefficient use of memory. Then, a special subclass of `java.util.Collection` was implemented in order to allow the lazy initialization of objects. Data was stored in a XML file and loaded as soon as requested by the user.

4.6 Composition of Services

The scenarios described in this chapter and implemented in the E-Mate project require the interaction and cooperation of different services. To meet this requirement the E-Mate project defines and implements a mechanism for the composition of services. The design goals of this composition mechanism can be summarized as follows:

- to build a new service starting from component services in a way that the resulting composite service has the following properties:

  - it can be packed and deployed in the service queue/application server architecture;
  - it can be published and discovered in the Jini architecture;
  - it can be loaded and delivered to users by means of the service viewer delivery architecture;
  - it can be used to compose other services.

- to build a new service without requiring that components have a mutual knowledge of interfaces. In other words, starting from two services s_1 and s_2 we can build a service s_3 even if s_1 makes no reference to the definition of s_2 and vice-versa. Under this condition s_1 and s_2 are decoupled.

- the service composition is defined a design-time but the discovery, lookup and linking of component services is performed at run-time.

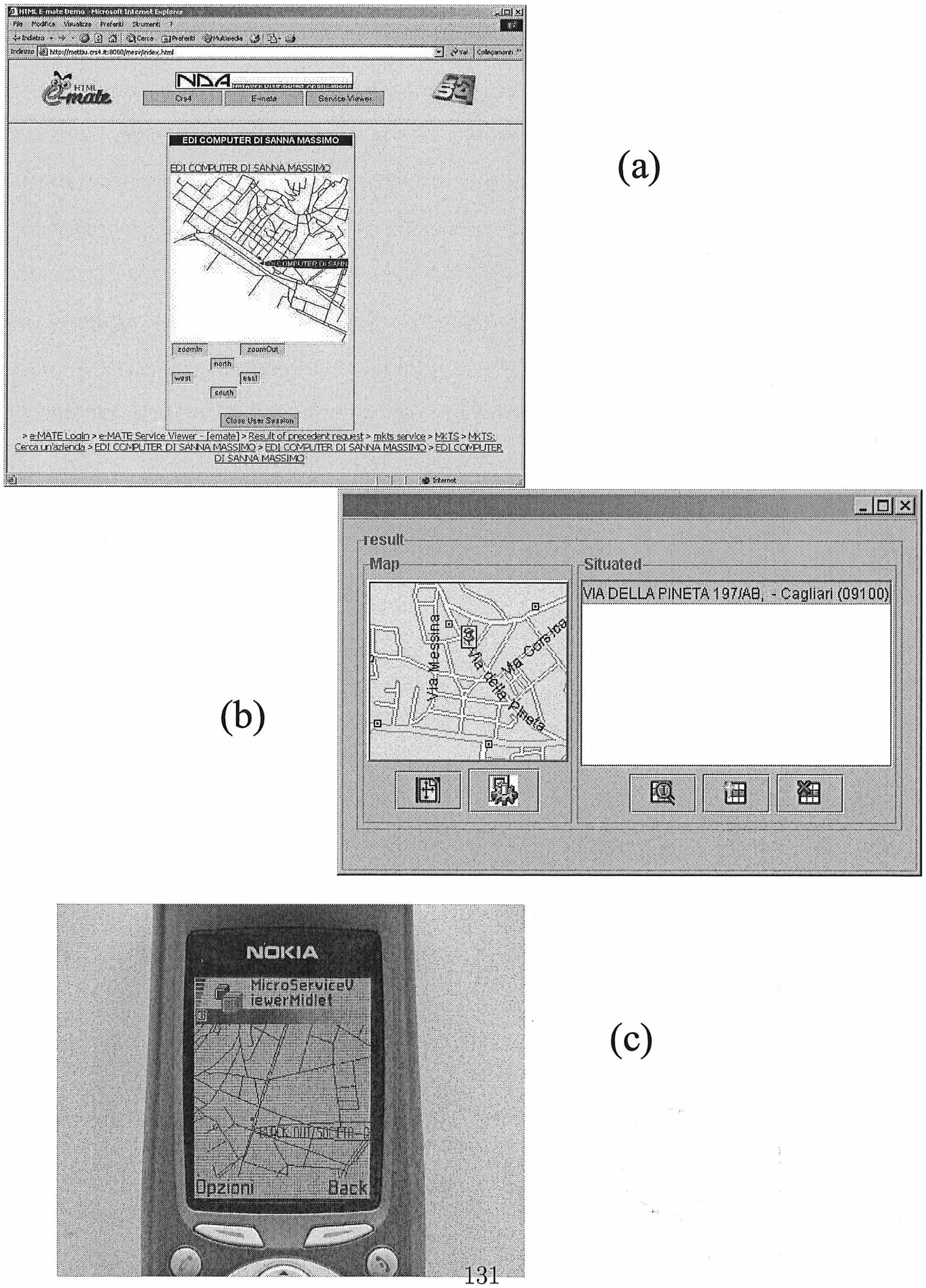
In this section we will not go into details of the E-Mate service composition because this is a large topic which would deserve an entire dissertation. We want only give an overview of the rationale and of the main issues.

As example of composition we can take two services: `AddressBook` and `Agenda`. They are decoupled, so no references of `AddressBook` are in the code of `Agenda` and vice-versa. Typically, the `Agenda` is used to fill a new `Appointment` in which a field of type `Person` is manually filled by the user. On the other hand, the use case of the composite service is: the user fills a new `Appointment` in the `Agenda` and can load the `Person` to meet selecting her/him form the `AddressBook`. Both services are interactive, so the user is able to launch the user interface of the `AddressBook` while interacting with the `Agenda`. It is worth noticing that two services can be composed even if they are decoupled but they must share a common knowledge of the data they exchange. In our example, both `AddressBook` and `Agenda` must refer to the data type `Person`.

The data exchange between different services is affected by some issues due to the classloading mechanism of the JVM. In fact, if the objects exchanged between two services are of a pre-installed[3] type (i.e type `String`) the data exchange is performed instantiating object whose classes are unambiguously loaded by the JVM. On the other hand, when we compose `AddressBook` and `Agenda` they both contain in their jar an identical definition for class `Person` and the JVM loads two classes: a class `Person` from the jar of the `AddressBook` and a class `Person` from the jar of `Agenda`. They are identical but distinct and an instance of `Person` cannot be exchanged between the two services.

To address this issue, the E-Mate implementation instantiates a new classloader for the composite services which is expected to load all the classes of component services. This way, the class `Person` is loaded once and its instances can be exchanged between the two services.

[3]In this case we mean pre-installed type any class already available in the standard library of the JVM



> e-MATE Login > e-MATE Service Viewer - [emate] > Result of precedent request > mkts service > MkTS > MkTS: Cerca un'azienda > EDI COMPUTER DI SANNA MASSIMO > EDI COMPUTER DI SANNA MASSIMO > EDI COMPUTER DI SANNA MASSIMO

Figure 50: Maps in three different modalities: (a) Web browser, (b) Java interface on desktop PC, (c) and cellular phone.

Chapter 5

RELATED WORKS

This chapter collects some of the main research efforts in the field of pervasive computing and interactive systems design. Section 5.1 presents a conceptual framework called Active Spacec. Section 5.3 and Section 5.2 present two works which are similar to MORE in their approach to provide direct interactions with domain objects. Section 5.4 introduces Model-driven and Task-driven design approaches. Finally Section 5.5 presents some standardization activities carried by W3C related to this thesis.

5.1 GAIA and Active Spaces

Active Spaces [54] are a new approach for modeling Ubiquitous Computing environments. Such an approach abstracts from the traditional computer model, and assumes that computers are simply a particular case of such an abstraction while other cases could be for instance Active City, or Active Office and so forth. In an Active Space the computation is affected by environmental variables such as noise, temperature, light, and environmental variables can be affected by computation. Traditional computing systems are composed by components belonging to the following categories: hardware, operating system, applications, and users. Assume two different implementations of the

Active Space model, the first one is a traditional computer, the second one is an Active Office. The similarities are:

- A computer has one or more processors, volatile and storage memory, and some peripherals connected via PCI, USB, and RS232 interfaces. An Active Space has as many processor as the number of devices, workstations, and servers located in the room; has volatile memory used by programs and storage services; the active office has some peripherals such as sensors, actuators, terminals, and so forth; peripherals are connected via network protocols such as IIOP, RMI, and DCOM.

- A computer has an operating system which manages computing resources and assign them to running programs. The communication between programs and the operating system is based on the "system calls". In an active office can work many users, using many terminals, and there may exist many processes interacting with sensors and actuators. The active space need an operating system to handle all concurrent activities and to avoid conflicts between users or applications while using the computing resources.

GAIA [23] is an operating system for active spaces. Regarding interactive applications, GAIA defines an extension of the Model-View-Control design pattern which adds the element Model Adapter. The Model Adapter translates at runtime the data type exported by a model in the format expected by the view.

5.2 Coupling Application Design and User Interface

The approach of coupling together the application objects and the user interface is not a novel one. One of the first attempts is due to de Baar, Foley and Mullet [18]. In their work they split the system design in two main parts: data objects design and interaction objects design. Data objects define attributes and actions, interaction objects activate actions, and manipulate data object's attributes. The design process takes as input a data model and send it to an inference engine aimed to map data objects and interaction object. Such an inference engine is parameterized by a set of selection rules and layout rules. Both types of rules are implemented by a if-then-else list. The design process relies on two third party tools: D2M2 Edit, and a ad-hoc modified version of DevGuide by Sun Microsystems. D2M2 Edit is aimed at the visual definition of data models. The data model produces by D2M2 is then dragged on DevGuide which connects to the inference engine and produce a formal specification of the user interface in a proprietary format called "User Interface Description File". Such a specification can be translated into a compilable source file for a specific graphical user interface toolkit and programming language.

The main difference between this approach and MORE is that mapping and user interface generation occur at design-time in the de Baar et al. system and the result is a specification file. On the other hand, MORE uses a set of selection rules which takes into account the actual device and the actual delivery context, thus, the user interface is produced and displayed at runtime. In our approach the emphasis is more on the device diversity and on the unpredictability of actual device type.

5.3 Naked Objects

Naked Objects [47] authors envision a new philosophical approach to requirements analysis, system design, and implementation of enterprise software. The rationale behind is close to the conceptual level of design in MORE, in which objects are at the same time building blocks of software and an expressive language to communicate functionalities. Such a software development vision is well explained in [48] where the author expres their skepticism about some current practices in software development which represent a step backward with respect to the potential of true object oriented design. The practices accused are the following: Business process orientation, Task-oriented user interfaces, Use-case driven methodologies, the Model-View-Controller pattern, and Component-based software development. In Naked Objects, the emphasis is on the behavioral-completeness of objects. Objects must contains both data and relevant operations and must be exposed to users as are, with Naked Objects mediators. Such an approach targets the agility of the development process, freeing the design from any other aspect not relevant to the pure comprehension of the problem. Naked Object provides an implementation of a Java framework. New Naked Objects can be created by extending base classes; the user interface aimed to interact with such objects is automatically provided by the runtime environment which inspects objects properties.

MORE and Naked Objects share the same vision of objects as expressive building blocks of software, nevertheless there are some important differences: Naked Objects has been developed to address the need for a new agile approach to the development of enterprise software, MORE focuses much more on the management of heterogeneity of delivery contexts by the direct manipulation of Java instances. With respect to multi-platform access to services, while Naked Objects authors consider multi-platform access to services just a possible side-effect of their architecture, MORE has focused an important part of the whole development to the implementation of multiple viewing mechanisms: Java Swing, PersonalJava, Interactive TV, Wireless Application Protocol(WAP), HTML with Cascade Style Sheets (CSS) and Javascript, simple HTML, and VoiceXML. On the other hand, Naked Objects reached objectives that have been partially disregarded or not fully addressed in MORE, such as the implementation of objects persistence mechanism integrated in the framework, and a strong support to agile practice such as unit and acceptance testing.

5.4 Task-Driven User Interface Design

The user interface is the "communication" channel between the user and the system and it can be considered the physical implementation of the human-machine interaction. The main problem related to the design of graphical user interfaces (GUI) is the portability of the code, in fact, even if we use a multi-platform language like Java, an application based upon a component library like Swing cannot be executed in a device with small footprint and minor hardware capabilities. Moreover there are devices that cannot run a Java Virtual Machine (JVM) at all or devices with mono-dimensional input like a voice interface. The discipline of model-based user interface design advocates the explicit denotation of the conceptual abstractions that underlie a user interface. This formal description can then be used to drive the user interface at run time, and it can also serve additional purposes. For instance, the tool TERESA [41] aims to the generation of several user interfaces, for many different devices, from a single task model. The approach is to define a task model using a tool called CTTE [46] and enabling each task for one or more end user devices. So a task could be enabled for the PC but not for a PDA. The final result of the process is a set of user interfaces tailored for a particular device.

Like our work, TERESA addresses the issue of the automatic generation of user interfaces in the context of multi-channel access to services. This leads to a similar result: task models may be a useful abstraction that can be adapted to the final delivery context. Unlike our work, TERESA is strongly focused on task models and data is simply data without the behaviour completeness of domain objects in MORE. Last difference, TERESA does not address any issue related to the deployment, discovery, and delivery of services. It seems to assume that user interfaces are simply document in HTML, WML or other markup languages to be delivered by a Web service.

5.5 W3C Standards and Device Independence Activity

The *Device Independece Activity* aims to provide: *Access to a Unified Web from Any Device in Any Context by Anyone [67]*

Among the topics subject to standardization there are: user preferences and device capabilities (Subsection 5.5.1), and flexible forms able to work seamlessly with different user interfaces (Subsection 5.5.2).

These efforts prove that the issues addressed in this thesis are considered critical by an organizations like W3C and that the emerging solutions will shape the future of computer applications.

5.5.1 CC/PP

The issue to adapt Web contents to the final delivery context is not new and various solutions have been put in place. Most of them are based on the automatic detection of the *UserAgent*, i.e. the short description of the device accessing a content. Below we present some of the most used techniques based on the user agent analysis.

- On-the-fly content selection and presentation based on user agent detection, using scripting languages,
- HTML object and link elements have mechanisms defining alternate behaviours,
- SMIL, the multimedia language for audio/visual content, has a switch element defining alternate elements to choose from, and can be used, for example, to choose some content based on available bandwidth,
- CSS also has such a mechanism called Media Queries for selecting appropriate style sheets.

The shortcoming of these approaches is that they are *static* and while the number of possible user agent is increasing the existing adapted contents are no longer sufficient. Moreover, the user agent does not provide any detailed information such as the screen resolution or the number of colors thus the adaptated content may be not really adapted whenever these parameters differs from their default values. Finally, to get a good adaptation, the content should be re-authored for a large variety of user agents making life hard to content authors.

The CC/PP [53] framework aims to solve these problems. CC/PP stands for Composite Capabilities/Preferences Profile, and is a system for expressing device capabilities and user preferences. With CC/PP it is possible to specify, for instance, that a device has a 256x172px screen resolution or that even if the color depth is 32bit the user is able only to distinguish few colors. CC/PP is based on RDF, and it is not limited to a fixed set of capabilities. but instead new

5.5.2 XForms

The massive adoption of Web technologies have raised some issues related to one of the most important elements of the HTML language: the `form` element. To better integrate forms with newcoming XML standards a new form specification called *XForms* [51] has been produced. The new XForms have the following advantages:

- they are written in XML, present XML data, and submit XML;
- they combines existing XML technologies;
- they make easier to author complicated forms;
- they are internationalized;
- they are accessible;

- they are device independent.

The last feature is strongly related to our work and is a key factor for the delivery of Web services over multiple platforms.

Chapter 6

Conclusion and Future Work

This chapter closes this work with a summary of the issues addressed, solutions proposed, and results concretely achieved. Furthermore, in Section 6 are introduced and described some promising future research directions.

Summary

This thesis faces the realization of software applications for computing-pervaded environments where applications are required to automatically adapt to the actual delivery context, to the actual device, and to allow user to seamlessly change device during a session of use.

Design objectives of this work are:

- A methodology for the design and implementation of platform-independent applications that allows programmers to implement once the application and to deliver it in any device even those not known at design time.
- A technique to refine applications in order to better adapt them to the actual delivery context and letting the user to focus on tasks rather than on objects.

- A middleware and a network architecture for the management of the whole life cycle of applications comprising the support for design, deployment, discovery, lookup and delivery of software in pervasive computing environments.

To address the points above we have conceived and experienced the solutions summarized as follow:

- A design methodology and a framework for the implementation of pervasive applications which allows the automatic generation of user interfaces in any device. The methodology enforces the human-to-object direct interaction and it is based on the reflective capabilities of object-oriented programming languages. We have developed a framework called MORE (Multimodal Object REnderer) which delivers applications in multiple platform without the need to re-write the user interface.

- To handle the complexity of interaction with thousands of possible domain objects deployed in a pervasive computing environment, we have developed a simple task-centered approach together with the tools to define and execute task models. The task approach is fully integrated with the MORE engine and thus it reuses the capability to generate user interfaces for different platforms.

- To fully support the design and delivery requirements of the points above, we have designed and developed an architecture for the whole life cycle of software applications in pervasive computing environment based on the Jini connection technology. The mentioned architecture meets the requirements for deployment, discovery, lookup and delivery of services.

Future Work

This section describes promising research directions still open to further improvements.

Dynamic Partitioning of Objects

The architecture proposed in this thesis is based on the dynamic partitioning of services between user device tier, proxies and remote tiers. According to the actual delivery context objects can behave as follows:

- they move to the end user device which renders an appropriate user interface;
- they move to an appropriate HTTP proxy, are translated into XML documents by the remote tier of the object renderer on the proxy and finally rendered as user interfaces by the local tier of the object renderer in the user device;
- they are translated into the appropriate mark-up language understood by the end user device. The object renderer is exclusively implemented in the remote tier.

This solution allows three different architectures and is based on the assumption that domain objects cannot be split into parts dynamically at run-time.

If we could overcome such a limitation, we would have the possibility to build more performant and dynamically adapted distributed systems. A promising research direction is in the design and implementation of a system able to perform dynamically and according to the delivery context the appropriate partitioning of objects into sub-objects with shared variables.

Taking as example an object of type `Foo`

```
public class Foo{
    int x,y;
    Collection c;

    public int doSum(){
        return x+y;
    }

    public Collection doSorting(){
        [... snip ...]
        c.add(new Integer(x));
        c.add(new Integer(y));
        [... snip ...]
        return c;
    }
}
```

In a J2ME device the class `Foo` cannot be loaded because `Collection` is not part of the Java library. Therefore, the object must be instantiated in a J2SE virtual machine and accessed by means of a network protocol. If instead, we could split the `Foo` instance into two parts `foo1` and `foo2` as follows:

```
public class Foo1{
    int x,y;
    public int doSum(){
        return x+y;
    }
}


public class Foo2{
    Collection c;
    int x,y;
    public Collection doSorting(){
        [... snip ...]
        c.add(new Integer(x));
        c.add(new Integer(y));
        [... snip ...]
        return c;
    }
}
```

we could instantiate `Foo1` in the local tier while the `Foo2` would be still instantiated in the remote tier. Invoking `doSum()` on `Foo1` does not affect the computation on the remote `Foo2` and it can be effectively performed in the end user device. If the value of either `x` or `y` are changed in one of the two sub-objects then the other must be updated before it performs a computation based on the value of `x`, `y`. In other words, `x` and `y` are shared variables and each sub-object must manage its local copy: whenever a shared variable is written in Foo1 the local copy of `Foo2` must be invalidated and vice versa. To invalidate a local copy it is necessary that a message is sent through the net from the sub-object performing the writing to the other. An invalidated variable cannot be used in the computation, its up-to-date value must be requested before to proceed. Thus if `Foo1.doSum()` is far more used than `doSorting()` and an efficient mechanism manages the coordination of `x` and `y` between `Foo1` and `Foo2` minimizing the number of messages exchanged, then the overall configuration would be far more efficient.

The benefit in pervasive computing is evident: programmers can develop domain objects focusing on core functionalities and later objects are automatically partitioned into sub-objects and deployed in different tiers according to the actual delivery context. In such a way, the final architecture of any object is unpredicatable at design time because it is computed at run time according to the pervasive computing environment.

Beyond Java

Another limitation of the architecture proposed in this thesis is in the exclusive use of Java as programming language. The whole architecture could be replicated in other languages than Java as far as they allow remote classloading and reflective operations. Nonetheless, the architecture would not allow the interoperability between services written in two different languages. If we can relax somehow the constraint of remote class loading and code mobility, we could try to design a pervasive computing system based not on the Jini connection technology and not based on Java. A suitable solution can be in the use of a protocol-centered architecture where elements in the network can be implemented in any language and exchange data but not code. Protocol-centered architectures such as peer-to-peer systems could be effectively designed to implement discovery, searching and publishing of resources and services in a completely decentralized manner.

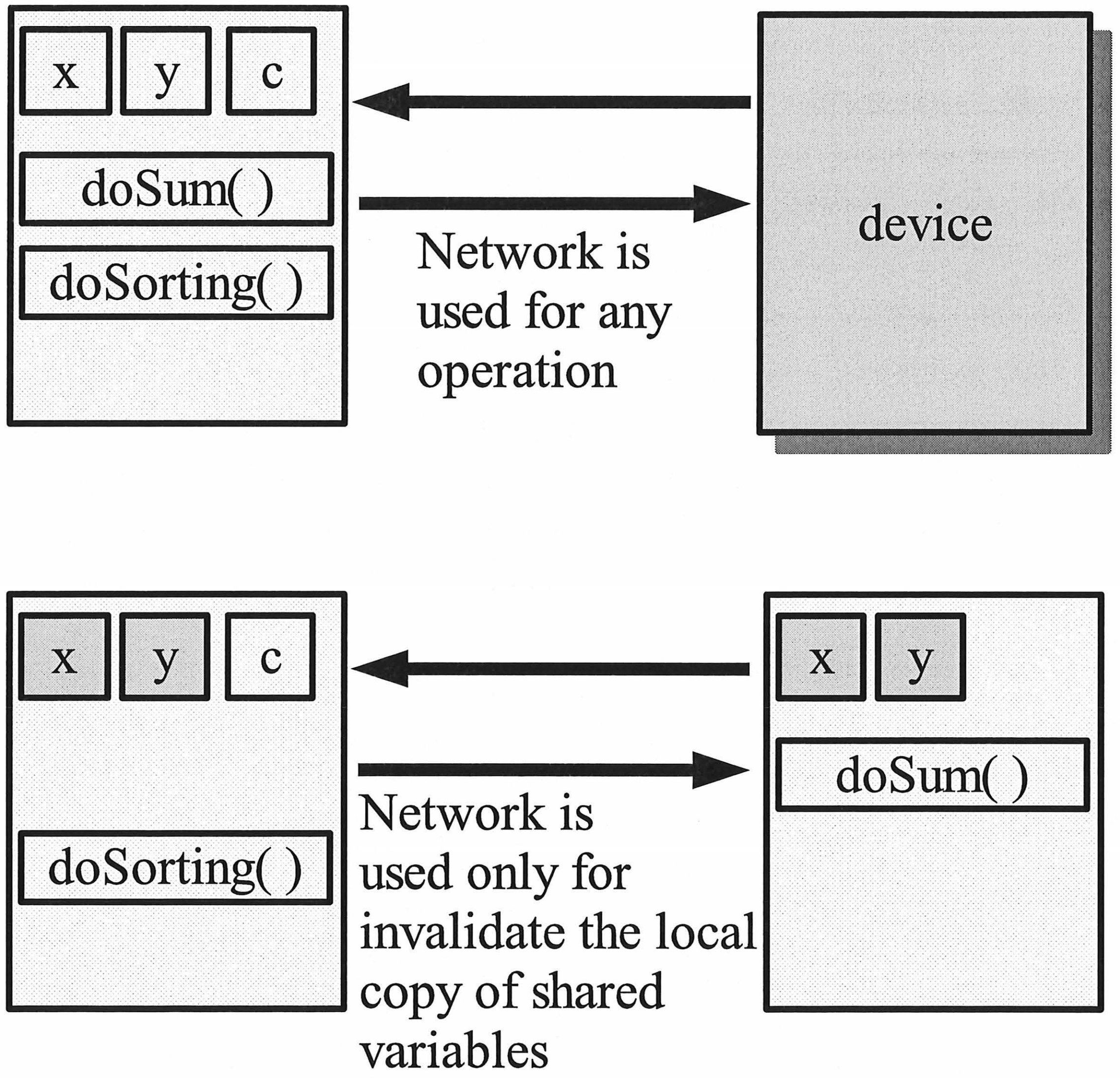

x
y
c
doSum( )
doSorting( )
Network is used for any operation
device
x
y
c
doSorting( )
Network is used only for invalidate the local copy of shared variables
x
y
doSum( )

Figure 51: If a device must communicate with a remote object (top) then the network is used for any message from the user to the object. In a dynamically partitioned architecture (bottom) objects are split into parts. Some part runs on the remote server and some run on the user device minimizing the number of messages sent from client-side to server-side and vice versa.

Disappearing User Interfaces

In this thesis we tackled some of the issues related to the design and delivery of services over a multitude of heterogeneus devices. However, we did not take into account one of the aspects that, according to some relevant research articles and in line with an IST call of the European Union [72], is a step ahead the traditional interaction paradigm: how to make the computing disappear and blur in the background. The *disappearing computer* seems to be a perfect integration of extremely complex technologies in everyday life, where computation is controlled by users "unconsciously" with the minimum amount of explicit HCI.

From a psycological point of view, it should be assessed whether the technology disappears because it becomes more user-friendly and more adaptive to our habits or instead because we do adapt to technology changing our habits. For instance, some programmers consider `vi` an "invisible" tool because they are so accustomed to use it that their fingers can activate commands even if they have difficult to say which key activates which function. This an example of how the human being can adapt to a very difficult tool to learn. The question is: does the tool become invisible because it is well designed or because it is reliable? Human beings are likely more adaptive than computers and a technology becomes invisible when perhaps the user is strongly motivated (or he cannot avoid using the tool), he survived the learning curve, and the tool is well designed and very reliable (we always realize the complexity of machines at the very moment they break).

From a technology point of view, design methods, input/output technologies, and sensing distribution schemas are key factors for the invisible computer. These factors must take into account the context. For instance, how the situation, the user profile, the social relationship between user in the same environment can have an impact in the choice of input/output devices and sensors.

From a social point of view, there are issues related to privacy of users when in a disappearing and ubiquitous computer thousands of sensors register our actions and sensible data are available to anybody who controls the pervasive computing environment. Another problem is how to draw the attention of one user without disturbing his neighbors. Other relevant questions are: who owns the radio channel? Which policy about bandwidth and processing power allocation? Rich people will be advantaged compared to poors? Which impact can have in our life a failure or a bug in a ubiquitous and invisible computer system?

Final Remarks

Although the ubiquitous computer is still to come, today there are less and less human activities that can be performed without the computer. So everyday life is pervaded by computers but yet this pervasiveness lacks of organization, structure, and design methods. People use a PC to work, a play station to play games, a PDA to store appointments and read mail, but all those elements are like separate islands in the computing archipelago. The development of the Internet has leveraged more organization and has acquired a critical mass for the development of new applications based on the cooperation of distributed services.

What the future will bring to us is difficult to say. Perhaps, all everyday objects, including some parts of the human body, will be connected in a huge wireless network where processors, sensors, and actuators will cooperate to relieve us from tasks and to meet needs that today we are not able to fully appreciate.

This scenario is impressive, and at the same time, frightening. Then, I'd like to close this thesis citing Alan Kay's[1] thoughts: *The best way to predict the future is to invent it.*

[1]Alan Kay is one of the inventors of the Smalltalk programming language and one of the fathers of the idea of Object Oriented Programming. He is the conceiver of the laptop computer and the architect of the modern windowing GUI.

Appendix A

Pocket Calculator Code

```
public class Calc {
    private String display = "";
    private String accumulator = "";
    private String operation;

    private static final
    Calc _instance=new Calc();

    boolean appendingDigits = true;

    private Calc() {

    }

    public static Calc getInstance(){
        return _instance;
    }
```

```
void append(String digit){
    if(appendingDigits){
        setDisplay(getDisplay()+digit);
    }
    else{
        setDisplay(digit);
        appendingDigits = true;
    }

}



public void _0_() {
    append("0");
}

public void _1_() {
    append("1");
}

public void _2_() {
    append("2");
}

public void _3_() {
    append("3");
}

public void _4_() {
    append("4");
}
```

```
public void _5_() {
    append("5");
}

public void _6_() {
    append("6");
}

public void _7_() {
    append("7");
}

public void _8_() {
    append("8");
}

public void _9_() {
    append("9");
}

public void division() {
    accumulator = getDisplay();
    setDisplay("");
    operation = "division";
}

public String getDisplay() {
    return display;
}
```

```
public void multiplication() {
    accumulator = getDisplay();
    setDisplay("");
    operation = "multiplication";
}

public void result() {

    Double o1 = new Double(getDisplay());
    Double o2 = new Double(accumulator);
    Double r = new Double(0.0);
    if (operation.equals("sum"))
        r = new Double(o1.doubleValue()
        + o2.doubleValue());

    if (operation.equals("subtraction"))
        r = new Double(o2.doubleValue()
        - o1.doubleValue());

    if (operation.equals("division"))
        r = new Double(o2.doubleValue()
        / o1.doubleValue());

    if (operation.equals("multiplication"))
        r = new Double(o1.doubleValue()
        * o2.doubleValue());

    setDisplay("" + r.doubleValue());

    accumulator="";
    appendingDigits=false;
    }

    public void setDisplay(String newDisplay) {
        Object old = display;
        display = newDisplay;
        //fireModelChange(old,display);
    }

    public void sqrt() {
        setDisplay("" + Math.sqrt(
        new Double(getDisplay()).doubleValue()));
        appendingDigits=false;
    }

    public void subtraction() {
        accumulator = getDisplay();
        setDisplay("");
        operation = "subtraction";
    }

    public void sum() {
        accumulator = getDisplay();
        setDisplay("");
        operation = "sum";
    }

    public String toString() {
        return "a more elaborate Calc...";
    }
}
```

Appendix B

XML Schemas

B.1 Schema for Response in Thin and HTTP-only Client Architectures

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com)
by Andrea Piras (CRS4 ICT/NDA) -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xsd:complexType name="slot_type">
        <xsd:annotation>
            <xsd:documentation>it is like java field</xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:choice>
                <xsd:sequence>
                    <xsd:choice>
                        <xsd:sequence>
                            <xsd:element name="value" type="xsd:string">
                                <xsd:annotation>
                                <xsd:documentation>
                    the value of the slot,
                    recovered by the field value of java o
                    bject</xsd:documentation>
                             </xsd:annotation>
                          </xsd:element>
                        </xsd:sequence>
                        <xsd:element name="element"
               type="collection_type"
               minOccurs="0"
               maxOccurs="unbounded"/>
                    </xsd:choice>
                 </xsd:sequence>
                 <xsd:sequence>
                    <xsd:element name="reference" type="xsd:string">
                        <xsd:annotation>
                           <xsd:documentation>
                 it indicates the reference to a not
                 primitive stored object contained
                 by main object</xsd:documentation>
                        </xsd:annotation>
                    </xsd:element>
                 </xsd:sequence>
               </xsd:choice>
               <xsd:element name="parameter"
            type="parameter_type"
            minOccurs="0"
            maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name"
        type="xsd:string" use="required"/>
```

```
    <xsd:attribute name="type"
type="xsd:string" use="required"/>
    <xsd:attribute name="alias"
type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="parameter_type">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="name"
    type="xsd:string" use="required"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
<xsd:complexType name="collection_type">
    <xsd:annotation>
        <xsd:documentation>
    it is like java collection field
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="name"
    type="xsd:string" minOccurs="0">
            <xsd:annotation>
                <xsd:documentation>
        Name of an element in
        the collection
        </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="type"
    type="xsd:string" minOccurs="0"/>
```

```
        <xsd:element name="value"
    type="xsd:string" maxOccurs="unbounded">
            <xsd:annotation>
                <xsd:documentation>
        the value of a
        collection element.</xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="reference" type="xsd:string">
            <xsd:annotation>
                <xsd:documentation>
        it indicates the reference to
        a not primitive stored object
        contained by the collection.
        </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="checked"
    type="xsd:boolean" minOccurs="0">
            <xsd:annotation>
                <xsd:documentation>
        indicates if it's item is selected
        and its presence indicates
        that it's a 'selection'
        </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="operation_type">
    <xsd:annotation>
```

```
        <xsd:documentation>
    it is like java method
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="return_type"
    type="xsd:string" minOccurs="0">
            <xsd:annotation>
                <xsd:documentation>
        the type of returned value
        by the operation.
        </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="parameter"
    type="parameter_type" minOccurs="0"
    maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name"
type="xsd:string" use="required"/>
    <xsd:attribute name="alias"
type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:complexType name="object_type">
    <xsd:annotation>
        <xsd:documentation>It's the type definition
    of "object".
    At first use, it is made only by the class
    name on the object browser must work.
    Working on the different states of the some object,
    it is made by slot items (like java fields)
    and operation items (like java methods)
    </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="slot"
    type="slot_type" minOccurs="0"
    maxOccurs="unbounded"/>
        <xsd:element name="operation" type="operation_type"
    minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name"
type="xsd:string" use="required"/>
    <xsd:attribute name="reference"
type="xsd:string" use="required"/>
    <xsd:attribute name="alias"
type="xsd:string" use="required"/>
</xsd:complexType>
<xsd:element name="hmi" type="hmi_type"/>
<xsd:complexType name="hmi_type">
    <xsd:sequence>
        <xsd:element name="session" type="xsd:string">
            <xsd:annotation>
                <xsd:documentation>"session" is the session id
        which the object browser work
        </xsd:documentation>
            </xsd:annotation>
        </xsd:element>
        <xsd:element name="processor" type="xsd:string">
            <xsd:annotation>
                <xsd:documentation>the processor file name
        for working on the object
    </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="stylesheet"
type="xsd:string"
default="toHTMLObject.xsl"
minOccurs="0">
        <xsd:annotation>
            <xsd:documentation>
    if written, it indicates which xsl document
    must to use for this slot
    instead of to use default one.
    </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="xmlobject" maxOccurs="unbounded">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="object_type"/>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="history">
        <xsd:annotation>
            <xsd:documentation>
    the references lists of the
    previous browsed object.
    </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
            <xsd:sequence>
```

```
                        <xsd:element name="element"
            type="collection_type"
            minOccurs="0"
            maxOccurs="unbounded"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Appendix C

PLANES: Text Notation

C.1 Introduction

In this appendix we describe the grammar used to express task models in textual notation. The elements of the grammar are expressed as follows:

```
expr:
expr1 expr2
```

It means: `expr` is composed by `expr1` followed by `expr2`

```
expr:
expr1
expr2
```

It means: `expr` is either `expr1` or `expr2`

```
expr:
expr1+
```

It means: `expr` is `expr1` one or more times

```
expr:
expr1*
```

It means: `expr` is `expr1` zero or more times

- Characters literals are expressed with `'a'`,`'b'`,`'c'` etc.
- String literals are delimited by double quotes.
- Expressions can be grouped inside parenthesis.
- Some expressions may rely on primitive expression types such as STRING and INTEGER.

C.2 PLANES Grammar

```
task_model:
task_definition+

task_definition:
task_name ":=" task_expression



task_expression:
task_name
task_definition

task_name:
STRING

task_definition:
composite_task
leaf_task
repeat_task
```

```
composite_task:
composite_task_type task_set

task_set:
'(' task_expression (',' task_expression)+ ')'

composite_task_type:
"SEQUENCE"
"CHOICE"
"INTERLEAVING"

leaf_task:
leaf_task_type ('(' param (',' param)+ ')')*

param:
STRING

leaf_task_type:
"SHIELD"
"INVOKE"
"INSTANTIATE"
"INPUT"
"SELECTION"

repeat_task:
"REPEAT" '(' task_expression ',' max_iterations ')'

max_iterations:
INTEGER
"FOREVER"
```

Appendix D

PLANES:XML Output for the Agenda Task Model

```
<taskmodel>
    <task classname="crs4.planes.core.TaskSequence"
     reference="22514347" name="Root">
        <type>SEQUENCE</type>
        <subtask>
            5313146</subtask>
        <subtask>
            9708927</subtask>
        <subtask>
            13623369</subtask>
        <subtask>
            26089635</subtask>
        <subtask>
            32739270</subtask>
        <subtask>
            23342038</subtask>
    </task>
```

```
<task classname="crs4.planes.core.RepeatTask"
reference="5313146" name="agenda">
    <type>REPEAT</type>
    <subtask>
        2737550</subtask>
    <repeat>0</repeat>
</task>
<task classname="crs4.planes.core.TaskChoice"
reference="2737550" name="TaskChoice">
    <type>CHOICE</type>
    <subtask>
        6888942</subtask>
    <subtask>
        19658898</subtask>
</task>
<task classname="crs4.planes.core.ReferenceTask"
reference="6888942" name="ReferenceTask">
    <type>REFERENCE</type>
    <referenced>addAppointment</referenced>
</task>
<task classname="crs4.planes.core.ReferenceTask"
reference="19658898" name="ReferenceTask">
    <type>REFERENCE</type>
    <referenced>modifyAppointment</referenced>
</task>
<task classname="crs4.planes.core.TaskSequence"
reference="9708927" name="addAppointment">
    <type>SEQUENCE</type>
    <subtask>
        30167145</subtask>
    <subtask>
        11742932</subtask>
    <subtask>
        29857804</subtask>
    <subtask>
        13594894</subtask>
</task>
<task classname="crs4.planes.core.InstantiateTask"
reference="30167145" name="instantiateAppointment">
    <type>INSTANTIATE</type>
    <objectType>Appointment</objectType>
</task>
<task classname="crs4.planes.core.InputTask"
reference="11742932" name="fillAppointmentForm">
    <type>INPUT</type>
    <variable>when,why,where,whom</variable>
</task>
<task classname="crs4.planes.core.ShieldTask"
reference="29857804" name="SHIELD">
    <type>SHIELD</type>
</task>
<task classname="crs4.planes.core.InvokeFunctionTask"
reference="13594894" name="invokeWriteOnDisc">
    <type>INVOKE</type>
    <function>save</function>
</task>
<task classname="crs4.planes.core.TaskSequence"
reference="13623369" name="modifyAppointment">
    <type>SEQUENCE</type>
    <subtask>
        24769387</subtask>
    <subtask>
        22805949</subtask>
</task>
```

```
<task classname="crs4.planes.core.ReferenceTask"
reference="24769387" name="ReferenceTask">
    <type>REFERENCE</type>
    <referenced>selectAppointment</referenced>
</task>
<task classname="crs4.planes.core.TaskChoice"
reference="22805949" name="TaskChoice">
    <type>CHOICE</type>
    <subtask>
        7310123</subtask>
    <subtask>
        18474371</subtask>
</task>
<task classname="crs4.planes.core.ReferenceTask"
reference="7310123" name="ReferenceTask">
    <type>REFERENCE</type>
    <referenced>editAppointment</referenced>
</task>
<task classname="crs4.planes.core.ReferenceTask"
reference="18474371" name="ReferenceTask">
    <type>REFERENCE</type>
    <referenced>removeAppointment</referenced>
</task>
<task classname="crs4.planes.core.TaskSequence"
reference="26089635" name="editAppointment">
    <type>SEQUENCE</type>
    <subtask>
        19601861</subtask>
    <subtask>
        11068806</subtask>
    <subtask>
        7654146</subtask>
```

```
</task>
<task classname="crs4.planes.core.ReferenceTask"
reference="19601861" name="Reference">
    <type>REFERENCE</type>
    <referenced>fillAppointmentForm</referenced>
</task>
<task classname="crs4.planes.core.ShieldTask"
reference="11068806" name="SHIELD--2">
    <type>SHIELD</type>
</task>
<task classname="crs4.planes.core.ReferenceTask"
reference="7654146" name="ReferenceTask">
    <type>REFERENCE</type>
    <referenced>invokeWriteOnDisk</referenced>
</task>
<task classname="crs4.planes.core.TaskSequence"
reference="32739270" name="selectAppointment">
    <type>SEQUENCE</type>
    <subtask>
        29106105</subtask>
    <subtask>
        29812760</subtask>
</task>
<task classname="crs4.planes.core.InstantiateTask"
reference="29106105" name="instantiateList">
    <type>INSTANTIATE</type>
    <objectType>Appointment[]</objectType>
</task>
<task classname="crs4.planes.core.ObjectChoiceTask"
reference="29812760" name="SELECTION">
    <type>SELECTION</type>
</task>
```

```
    <task classname="crs4.planes.core.TaskSequence"
    reference="23342038" name="removeAppointment">
        <type>SEQUENCE</type>
        <subtask>
            28291271</subtask>
        <subtask>
            25106497</subtask>
    </task>
    <task classname="crs4.planes.core.ShieldTask"
    reference="28291271" name="SHIELD--3">
        <type>SHIELD</type>
    </task>
    <task classname="crs4.planes.core.InvokeFunctionTask"
    reference="25106497" name="INVOKE(remove)">
        <type>INVOKE</type>
        <function>remove</function>
    </task>
</taskmodel>
```

Appendix E

Layout

```
package crs4.more;
import java.util.Iterator;
/**
 * A Layout contains information about how members of a model are flushed in
 * a AbstractView. It makes sense only for visual views.
 */
public class Layout extends MoreObject{
    private java.util.Collection sequence =
new java.util.ArrayList();
    public final static java.lang.Object NEWLINE =
"newline";
    public Layout() {
    }
    public void addMember(String memberName){
        sequence.add(memberName);
    }
    public void addNewLine(){
        sequence.add(Layout.NEWLINE);
    }
    public Iterator layoutIterator() {
        log("Layout layoutIterator():" + sequence);
        return sequence.iterator();
    }
}
```

Bibliography

[1] Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification.* Addison-Wesley, Reading, MA, USA, 1999.

[2] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman, and Deborra Zukowski. Challenges: an application model for pervasive computing. In *Proceedings of the sixth annual international conference on Mobile computing and networking*, pages 266–274. ACM Press, 2000.

[3] T. Berners-Lee, R. T. Rielding, and H. Frystyk Nielsen. Hypertext transfer protocol - HTTP/1.0. http://www.w3.org/hypertext/WWW/Protocols/HTTP1.0/draft-ietf-http-spec.html, 1995.

[4] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[5] Alan Borning. Programming language aspect of ThingLab: A constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[6] Paul De Bra, Geert-Jan Houben, and Hongjing Wu. AHAM: A dexter-based reference model for adaptive hypermedia. In *UK Conference on Hypertext*, pages 147–156, 1999.

[7] D. Carboni, S. Sanna, S. Giroux, and G. Paddeu. Interactions model and code generation for J2ME applications. *Lecture Notes in Computer Science*, 2411:286–290, 2002.

[8] Davide Carboni, Sylvain Giroux, Gavino Paddeu, Andrea Piras, and Stefano Sanna. Delivery of services on any device: From Java code to user interface. HCI International 2003, Lawrence Erlbaum Associates, Inc., Publishers, June 2003.

[9] Davide Carboni, Sylvain Giroux, Andrea Piras, and Stefano Sanna. The Web around the corner: Augmenting the browser with GPS. In *Proceeding of the WWW2004 International Conference.* ACM Press, May 2004.

[10] Davide Carboni, Sylvain Giroux, Eloisa Vargiu, Claude Moulin, Stefano Sanna, Alessandro Soro, and Gavino Paddeu. E-mate: An open architecture to support mobility of users. In Hele-Mai Haav and Ahto Kalja, editors, *Databases and Information Systems II*, page 15. Kluwer Academic Publishers, 2002.

[11] J. Carnes. Using the utm map coordinate system, http://www.maptools.com/usingutm/, 2002.

[12] Nicholas Carriero and David Gelernter. Applications Experience with Linda. In *Proceedings of the ACM Symposium on Parallel Programming*, pages 173–187, New Haven, July 1988. ACM Press.

[13] Leonardo Chiariglione. Standards: MPEG: a technological basis for multimedia applications. *IEEE MultiMedia*, 2(1):85–89, Spring 1995.

[14] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. *ACM SIGPLAN Notices*, 22(12):156–162, December 1987.

[15] Larry L. Constantine. The emperor has no clothes: Naked objects meet the interface. *Constantine Lockwood, Ltd – http://www.foruse.com/articles/.*

[16] Ole-Johan Dahl and Kristen Nygaard. SIMULA, an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

[17] DAVIC. http://www.davic.org.

[18] Dennis J. M. J. de Baar, James D. Foley, and Kevin E. Mullet. Coupling application design and user interface design. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, Beyond Widgets: Tools for Semantically Driven UI Design, pages 259–266, 1992.

[19] Roberto Demontis, Eva Lorrai, Vladimiro Marras, and Claude Moulin. A multimodal approach to emergency situations: Modelling with multi-agent system. In Alessandra Raffaetá and Chiara Renso, editors, *Proceedings of the CRGD (Complex Reasoning on Geographical Data) Workshop, ICLP'01 Conference*, pages 85–93. CRGD (Complex Reasoning on Geographical Data)Workshop, S. T. A. R., November 2001.

[20] Emanuela Devita, Andrea Piras, and Stefano Sanna. Using compact GML to deploy interactive maps on mobile devices. In *Proceedings of WWW2003 - Poster session*, 2003.

[21] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, 1986.

[22] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley, Reading, MA, USA, 1999.

[23] GAIA. Web site: http://choices.cs.uiuc.edu/gaia/.

[24] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 2000.

[25] Sylvain Giroux, Claude Moulin, Raffaella Sanna, and Antonio Pintus. Mobile lessons: Lessons based on geo-referenced information. In Margaret Driscoll and Thomas C. Reeves, editors, *Proceedings of the E-Learn 2002 conference, World Conference on E-Learning in Corporate, Government, Healtcare and Higher Education*, pages 331–338. Association for the Advancement of Computing in Education (AACE), AACE, 2002.

[26] A. Goldberg and D. Robson. *Smalltalk 80 - the Language and its Implementation.* Addison-Wesley, Reading, 1989.

[27] GPSWeb. Web site: http://www.crs4.it:8000/gpsweb.

[28] Graham Hamilton and David Geary. *JavaBeans Specifications & Tutorial.* Addison-Wesley, Reading, MA, USA, 19xx.

[29] H. R. Hartson, A. C. Siochi, and D. Hix. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Trans. on Inf. Sys.*, 8(3):181, 1990.

[30] Thomas A. Herring. The Global Positioning System. *Scientific American*, 274(2):44–50 (Intl. ed. 32–38), February 1996.

[31] C. A. R. Hoare. *Communicating sequential processes.* computer science. Prentice-Hall International, Englewood Cliffs, N.J, 1985.

[32] Susanne Hupfer. Make room for javaspaces, part 6-build and use distributed data structures in your javaspaces programs. http://www.javaworld.com/javaworld/jw-10-2000/jw-1002-jiniology.html, 2000.

[33] Itautec. http://www.itautec.com.br/ebusiness/ihc_english.htm.

[34] JavaSoft. Java Native Interface specification, November 1996. Release 1.1.

[35] Alan C. Kay. The early history of smalltalk. In *History of Programming Languages*, pages 511–579. ACM Press/Addison-Wesley, 1996.

[36] Michael Kay. *XSLT, Programmer's Reference.* Wrox Press Ltd., 2000.

[37] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, August/September 1988.

[38] Rodger Lea. *HAVi: example by example: Java programming for home entertainment devices.* Prentice Hall PTR example by example series. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 2002.

[39] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–214, November 1986.

[40] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155, 1987.

[41] Giulio Mori, Fabio Paterno';, and Carmen Santoro. Tool support for designing nomadic applications. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 141–148. ACM Press, 2003.

[42] Ajith K. Narayanan. Realms and states: a framework for location aware mobile computing. In *Proceedings of the 1st international workshop on Mobile commerce*, pages 48–54. ACM Press, 2001.

[43] Networldmap. A geographical map of the internet. http://www.networldmap.com/.

[44] NMEA. Nmea 0183 standard: http://www.nmea.org/pub/0183/index.html.

[45] Scott Oaks, Li Gong, and Bernard Traversat. *JXTA in a Nutshell.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 2002.

[46] Fabio Paterno. *Model-Based Design and Evaluation of Interactive Applications.* Applied Computing. Springer-Verlag, 1999.

[47] Richard Pawson and Robert Matthews. Naked objects: a technique for designing more expressive systems. *ACM SIGPLAN Notices*, 36(12):61–67, December 2001.

[48] Richard Pawson and Robert Matthews. *Naked Objects.* John Wiley and Sons Ltd, 2002.

[49] H. Pigot, B. Lefebvre, J.G. Meunier, B. Kerherv, A. Mayers, and S. Giroux. The role of intelligent habitats in upholding elders in residence. In *Proc. of 5th international conference on Simulations in Biomedicine*, April 2003.

[50] Angel R. Puerta, Henrik Eriksson, John H. Gennari, and Mark A. Musen. Beyond data models for automated user interface generation. In *Proceedings of the HCI'94 Conference on People and Computers IX*, Notations and Tools for Design, pages 353–366, 1994.

[51] T. V. Raman. *XForms: XML Powered Web Forms.* Addison-Wesley, 2003.

[52] Bharat Rao and Louis Minakakis. Evolution of mobile location-based services. *Commun. ACM*, 46(12):61–65, 2003.

[53] F. Reynolds. Composite Capability/ Preferences Profiles: A user side framework for content negotiation. http://www.w3.org/TR/NOTE-CCPP/, 1999.

[54] Manuel Roman and Roy H. Campbell. Gaia: enabling active spaces. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 229–234. ACM Press, 2000.

[55] Salutation. Web site: http://www.salutation.org/.

[56] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, pages 10–17, August 2001.

[57] Upendra Shardanand and Patti Maes. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1, pages 210–217, 1995.

[58] Brian C. Smith and Carl E. Hewitt. A Plasma primer. Report, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, September 1975.

[59] Sun. Java remote method invocation specification. Technical report, Sun Microsystems, 1997. http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.

[60] Sun Microsystems. www.sun.com.

[61] Pedro A. Szekely, Piyawadee Noi Sukaviriya, Pablo Castells, Jeyakumar Muthukumarasamy, and Ewald Salcher. Declarative interface models for user interface construction tools: the MASTERMIND approach. In *EHCI*, pages 120–150, 1995.

[62] Andrew S. Tanenbaum. *Modern operating systems.* Prentice-Hall, Inc., 1992.

[63] Thomas C. Valesky. *Enterprise JavaBeans: developing component-based distributed applications.* Addison-Wesley, Reading, MA, USA, 1999.

[64] G. van Rossum. A tour of the Python language. In R. Ege, M. Singh, and B. Meyer, editors, *Proceedings. Technology of Object-Oriented Languages and Systems, TOOLS-23*, pages 370–??, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1998. IEEE Computer Society Press.

[65] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, January-March 1999.

[66] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 35(2):46–55, February 1997.

[67] W3C. Device Independence Activity. http://www.w3.org/2001/di/.

[68] W3C. Http extension framework. http://www.w3.org/Protocols/HTTP/ietf-http-ext/.

[69] Mark R. Walker, Jim Edwards, Michael Jeronimo, John G. Ritchie, and Ylian Saint-Hilaire. Remote I/O: Freeing the experience from the platform with UPnP* architecture. *Intel Technology Journal*, 6(4):30–36, November 2002.

[70] M. Weiser. The world is not a desktop. *Interactions*, 1(1):7–8, 1994.

[71] Marc Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, September 1991.

[72] J. Wejchert. The disappearing computer. IST call for proposals. Available online at: http://www.cordis.lu/ ist/fetdc.htm, February 2000.

[73] M. Welie. Patterns in interaction design - http://www.welie.com/.