Oleksiy Volovikov

Mobile Encounter Networks Application: A Gasoline Price Comparison System

Master's Thesis
in Information Technology
24th May 2006

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Jyväskylän yliopisto
Jyväskylä 2006

Abstract

Volovikov, Oleksiy
Mobile Encounter Networks Application: A Gasoline Price Comparison System / Oleksiy Volovikov
Jyväskylä: University of Jyväskylä, 2006
69 p.
Master's Thesis

This thesis describes a mobile encounter networks application, called Gasoline Price Comparison System (GPCS), which delivers the newest gasoline prices to mobile users using mobile encounter information diffusion. Other existing and potential applications of mobile encounter networks are described as well.

Mobile encounter networks emerge when mobile devices come across each other and form a temporary connection between them using a common short-range radio technology. Local information exchanges between mobile devices result in a broadcast diffusion of information to other users of the network with a delay.

Keywords: Mobile Applications, Mobile Encounter Networks, Information Diffusion, BlueCheese, Symbian OS, Bluetooth

Tiivistelmä

Volovikov, Oleksiy
Mobiilien kohtaamisverkkojen sovellus: Polttoaineen hintavertailujärjestelmä / Oleksiy Volovikov
Jyväskylä: Jyväskylän yliopisto, 2006
69 s.
pro gradu -tutkielma

Tässä tutkielmassa kuvataan mobiileja kohtaamisverkkoja varten suunniteltu sovellus nimeltä Gasoline Price Comparison System (GPCS, Polttoaineen hintavertailujärjestelmä), joka toimittaa uusimmat polttoaineiden hinnat mobiililaitteiden käyttäjille mobiilien kohtaamisten mahdollistaman informaation leviämisen avulla. Työssä esitellään myös muita jo toteutettuja ja mahdollisia sovelluksia mobiileille kohtaamisverkoille.

Mobiileja kohtaamisverkkoja syntyy, kun mobiililaitteet kohtaavat toisensa ja muodostavat väliaikaisen yhteyden käyttämällä lyhyen kantaman radiotekniikkaa. Mobiililaitteiden välisten paikallisten tiedonsiirtojen ansiosta tieto leviää yleislähetyksenä muille verkon käyttäjille viiveellä.

Avainsanat: Mobiilisovellukset, mobiilit kohtaamisverkot, informaation leviäminen, BlueCheese, Symbian OS, Bluetooth

Preface

This Master's thesis was done at the Department of Mathematical Information Technology, University of Jyväskylä, Finland. The thesis was inspired by professor Jarkko Vuori's idea about Gasoline Price Comparison System (GPCS). GPCS lets drivers' mobile devices automatically exchange information about gasoline while they are in immediate proximity to each other without going through a central server. These information exchanges are free of charge since there is no central server in the system and using a common short-range wireless technology the transmission does not cost to the users. Having possessed such information, the driver has the possibility to choose an appropriate place to refuel in the future.

I would like to thank Jarkko Vuori for the original idea, my supervisors Mikko Vapa and Matthieu Weber for their valuable advices and comments during the creation of this thesis. I express my gratitude to the head of the exchange program Vagan Terziyan and the international coordinator Helen Kaykova for their help and support during my stay in Finland. Finally I wish to thank the team of Bitcomp Oy and especially Jarmo Oittinen and Ville-Pekka Vahteala for the opportunity to learn and work with them for over a year and a half.

Glossary

Avkon	Series 60 extensions and modifications to Uikon and other parts of the Symbian OS Application Framework [8].
BlueCheese	A middleware for Symbian OS that facilitates communication between applications in mobile encounter networks.
Bluetooth	A global de facto standard for wireless connectivity. The technology is based on a low-cost, short-range radio link that operates in a globally available ISM band at 2.4 GHz, making Bluetooth usable worldwide [4].
DSA	(Data Sharing Applications) are data sharing mobile computing systems used to share their memory space and data to achieve some common benefits with the aid of data exchange between radio-equipped mobile devices.
GPCS	(Gasoline Price Comparison System) is a mobile encounter networks application for distributing gasoline prices between mobile devices running Symbian OS.
GPS	(Global Positioning System) is a satellite-based radio positioning system that provides positioning, velocity and time information to GPS device users.
GSM	(Global System for Mobile communications) the second generation digital cellular technology used for transmitting mobile voice and data services.
IDE	(Integrated Development Environment) is a type of computer software that assists computer programmers to develop software.
L2CAP	(Logical Link Control and Adaptation Protocol) is used within the Bluetooth protocol stack for segmentation, reassembling and protocol mixing.
Middleware	is a software that provides a programming model above the basic building blocks of processes and message passing [6].

MP2P	(Mobile Peer-to-Peer) extends P2P by allowing resource sharing in a mobile environment.
P2P	(Peer-to-Peer) refers to decentralized and self-organizing overlay architectures of equal and autonomous entities for sharing distributed resources.
RFCOMM	(Radio Frequency Communications Protocol) is a protocol located on the top of the L2CAP protocol. It emulates the RS232 serial port and in this way offers an API to software developers.
SDP	(Service Discovery Protocol) is a protocol located on the top of the L2CAP protocol. It handles the service discovery of the Bluetooth devices.
SPA	(Social Proximity Applications) are social mobile computing systems used to enhance existing social behaviors, practices, and experiments taking place in physical space ('human interaction') with the aid of data exchange between radio-equipped mobile devices ('digital interaction') [30].
Series 60	A feature-rich software platform for smartphones with advanced data capabilities that is optimized for the Symbian OS [33].
Symbian OS	The global industry standard operating system for smartphones, which is licensed to the world's leading handset manufacturers.
Uikon	The generic UI library and control framework common to Symbian Platforms [8].
UML	(Unified Modelling Language) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modelling and other systems.
WLAN	(Wireless Local-Area Network) is a type of local-area network that uses high-frequency radio waves rather than wires to communicate between nodes.

Contents

1 Introduction

This thesis was inspired by professor Jarkko Vuori's idea about Gasoline Price Comparison System (GPCS). GPCS lets drivers' mobile devices automatically exchange information about gasoline while they are in immediate proximity to each other. Having possessed such information, the driver has the possibility to choose an appropriate place to refuel in the future. These information exchanges are free of charge because there is no central server in the system and using a common short-range wireless technology the transmission does not cost to the users.

Since this kind of applications share their memory space and data to achieve some common benefits while running on radio-equipped mobile devices, they are called Data Sharing Applications (DSA) in this work. There are some potential applications in which such Data Sharing can be successfully used. In this thesis they are presented as well as some existing applications are briefly reviewed.

These mobile applications have appeared in the past few years forming a new class of mobile networks. These networks, called mobile encounter networks, allow local information exchanges between mobile devices using a common short-range wireless technology (such as Bluetooth [4] or 802.11 WLAN [16]) without going through a central server. Their main distinction from mobile ad hoc networks, which are not yet widely used on mobile devices market because of their complexity[1], is the absence of multihop routing and thus avoiding costs for creating/maintaining routes. Although multihop routing considerably extends reachability and decreases latency in mobile ad hoc networks, it is not applicable for mobile encounter networks since they are not intended for searching, but rather for spreading information to interested parties.

Another emerging direction in this field is social mobile computing which aims at supporting social encounters in physical space by providing information or services in connection with face-to-face encounters. According to Persson [30] this kind of applications are called Social Proximity Applications (SPA).

The thesis is roughly organized into two parts. The first part, Chapters 2-4, gives the general background for the research discussed in the thesis as well as the research itself: Chapter 2 considers existing applications of both DSA and SPA; Chapter 3 introduces mobile encounter networks, information diffusion in these networks, their benefits and shortcomings as well as some potential application areas where these net-

[1]Section 3.4 is dedicated to these issues.

works could be used [36]; Chapter 4 is dedicated to the BlueCheese middleware that facilitates communication between applications in mobile encounter networks providing higher-level functions for application-specific data exchanges. The second part, Chapter 5 and Appendix A, is the practical part of the thesis: Chapter 5 describes the UI application, called Gasoline Price Comparison System, which has been developed in the scope of this thesis as a prototype to illustrate the feasibility of mobile encounter networks applications and together with BlueCheese middleware, it might be used for studying the network characteristics of the system; Appendix A presents the functionality of GPCS engine's classes expressed in UML notation as well as the header files in which these classes are defined. Finally, the thesis is concluded in Chapter 6.

2 Existing Applications

2.1 Introduction

As mentioned earlier, there are two groups of applications currently on the market that suit to the mobile encounter network architecture:

- Data Sharing Applications (DSA) are data sharing mobile computing systems used to share their memory space and data to achieve some common benefits with the aid of data exchange between radio-equipped mobile devices;
- Social Proximity Applications (SPA) are social mobile computing systems used to enhance existing social behaviors, practices, and experiments taking place in physical space ('human interaction') with the aid of data exchange between radio-equipped mobile devices ('digital interaction') [30].

'Nokia Sensor' [26], 'Nokia Flier' [25], 'BEDD' [3], 'MobiLuck' [22], 'Proxidating' [31], 'Dreamlove' [10] and 'BuZZone' [5] belong to SPA whereas 'BEDD', 'Nokia Flier' and 'GPCS' belong to DSA. Of course, some applications, such as 'Nokia Flier' or 'BEDD', can belong to both groups.

2.2 Analysis and Comparison

As will be shown in the next section, all these applications utilize the same means to achieve rather different purposes. The applications, while running in the background of mobile devices, automatically exchange some specific pieces of information with others that come within the range of Bluetooth. The applications, which belong to SPA typically exchange personal information, such as personal pages or profiles or even business cards, in order to meet new friends or business partners. Many such applications allow the user to define search criteria used to alert him/her whenever interesting contacts have been found. The applications which belong to DSA typically exchange non-personal information collected from different sources, such as various events, jokes or even gasoline prices, in order to help in making decisions. Therefore DSA might also stand for Decision Support Applications. The applications BEDD and Nokia Flier belong to both groups. The BEDD application is multifunctional and has

features of both SPA and DSA whereas the Nokia Flier application is rather simple, but it might be used for both purposes.

Unlike DSA, today's SPA do not forward the information obtained from mobile devices in earlier encounters to mobile devices in new encounters. However, it might significantly increase the reach of new potential contacts. Whenever a match is discovered, people can then make direct contact with each other via contact information, such as phone number or e-mail.

2.3 Applications Overview

In this section the main features of the applications are briefly presented as well as some screenshots are provided. All these applications run on Series 60-based mobile devices. The only exception is the BuZZone application, which runs on PDA devices.

2.3.1 Nokia Sensor The Nokia Sensor application shown in Figure 2.1 is free of charge and is downloadable from [26], which describes it as follows:

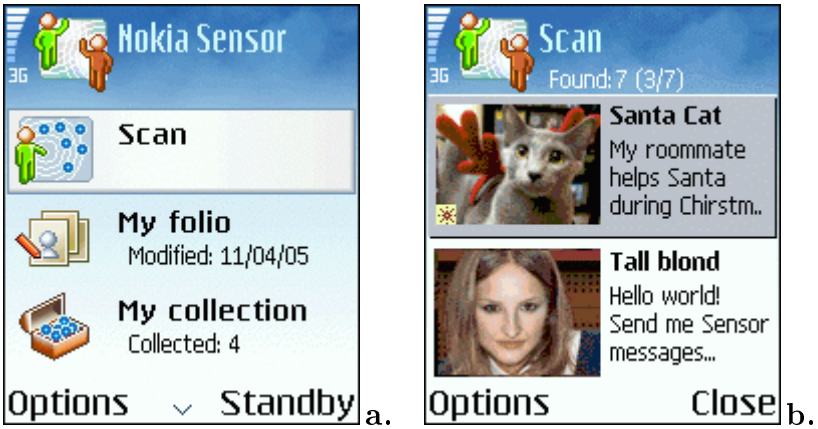


Figure 2.1: Sensor: main menu (a) and scan results (b) (adapted from [26])

> *With Nokia Sensor you can create your own personal pages - called a folio - on your phone. Then you can check out the folios of other Sensor users nearby, exchange messages, and share files. Nokia Sensor uses Bluetooth wireless technology, which means that it works within a 'circle' of up to 10 meters around your phone. When other Sensor phones come within this circle, your phone can 'sense' them and you can see their folios and send them messages. As soon as you step out of each other's circles, you are no longer able to communicate via Sensor. Because phones running the Sensor application communicate directly without going through an operator network service, communication is free of charge.*

2.3.2 Nokia Flier The Nokia Flier application is free of charge and is downloadable from [25], which describes it as follows:

> *Nokia Flier application allows you to create and locally distribute short messages containing text and a picture. When you have created your own flier you can publish it to other Nokia Flier users, who are close by (about 10 m) and have activated Nokia Flier application on the phone. Nokia Flier uses Bluetooth wireless technology for communicating with other phones. Nokia Flier application contains a screen saver that you can use to view received or saved fliers. For power saving reasons, Nokia Flier will be automatically turned off after 12 hours of use.*

2.3.3 BEDD The BEDD application shown in Figure 2.2 is free of charge and is downloadable from [3], which describes it as follows:

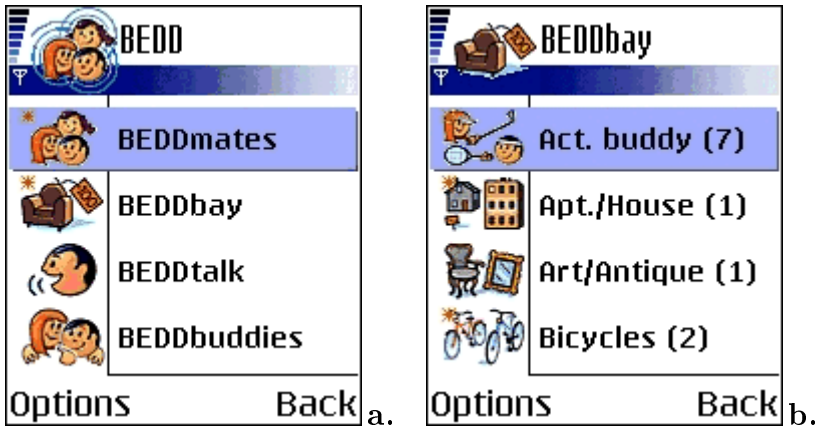


Figure 2.2: BEDD: main menu (a) and BEDDbay (b) (adapted from [3])

> *BEDD, while running in the background of your mobile phone, automatically exchanges your profile with other users that are in your proximity. BEDD also exchanges ads about things that you would like to buy or sell. The software then makes a correlation analysis and, if your criteria are met, alerts you to an exciting match! BEDD enables you to send free Bluetooth text messages or easily share video, image or sound files. You can also make regular mobile contact via SMS, MMS, phone call, IM or e-mail. BEDD is like social networking sites, on-line chat and newspaper classifieds, only all inside your mobile phone.*

2.3.4 Proxidating The Proxidating application suits well to the Dating service, which is described in Section 3.5. Another suitable application for the Dating service

is Dreamlove. The Proxidating application costs around 3 EUR and is downloadable from [31], which describes it as follows:

> *Proxidating is a totally new way for single people to meet up instantly. All you need to do is install Proxidating on your mobile phone, create your profile, enable Bluetooth and wait for your dream date to appear. Whenever you come within about 15 m of a person with a matching profile your phone will alert you. Only people with matching profiles will be linked via their phones. Proxidating automatically sends the text and image that you have defined to your potential date. In the same way, you will receive text and image from the matched partners' phone.*

2.3.5 MobiLuck The MobiLuck application shown in Figure 2.3 costs around 15 EUR and is downloadable from [22], which describes it as follows:

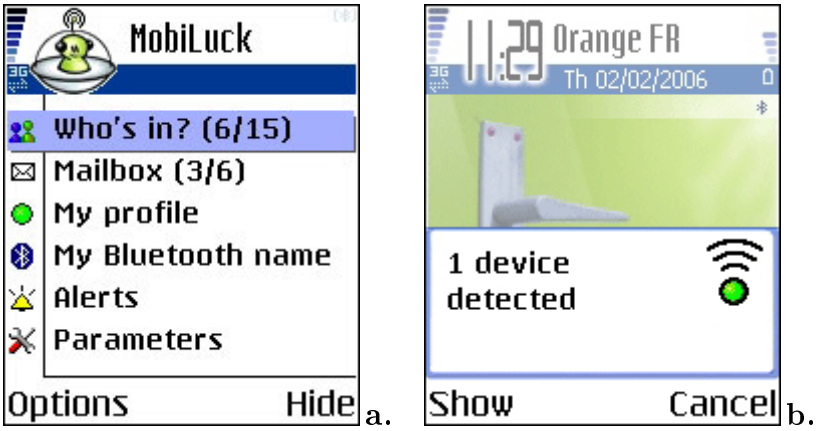


Figure 2.3: MobiLuck: main menu (a) and alert (b) (adapted from [22])

> *With MobiLuck you can detect all nearby Bluetooth devices (your cell phone rings or vibrates when it finds one), send messages and photos for free to friends or strangers with no need of their phone numbers, hear when you receive a Bluetooth message and reply to the sender, send your profile and receive profiles from other MobiLuckers including their photo, send MobiLuck to other people so you'll meet more and more MobiLuckers.*

2.3.6 BuZZone The BuZZone application shown in Figure 2.4 is free of charge and is downloadable from [5], which describes it as follows:

> *BuZZone application offers users the convenience of using Bluetooth-enabled laptops and PDAs to find new contacts, communicate over small distances, and share*

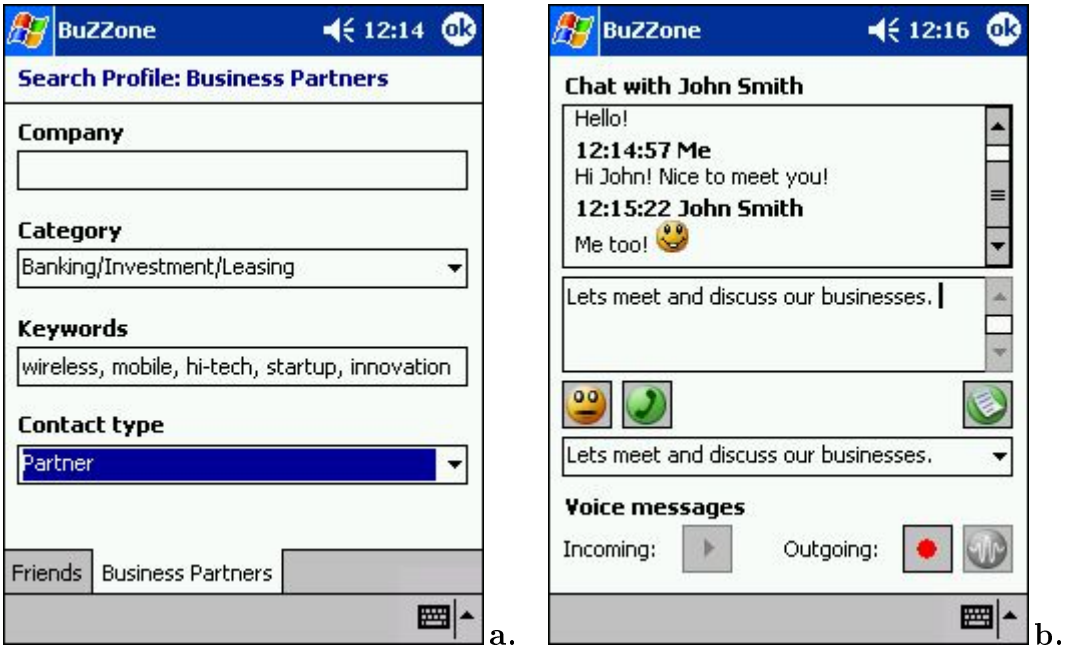


Figure 2.4: BuZZone: search profile (a) and chat (b) (adapted from [5])

information related to their business. BuZZone works as follows: your personal profile will tell other people about your interests, either personal or business, including photo and voice messages. In turn, you define your search criteria for people you want to make contact with. Wherever you are, at a trade show, in a subway or a coffee house, your BuZZone will be in continuous search for other BuZZone users located nearby. If it finds another user with a profile matching your search criteria, the program will invite both users to get acquainted and start talking. You can also join BuZZone-based wireless forums to discuss various topics with your virtual contacts.

2.4 Conclusion

Most SPA do not strictly suit to the mobile encounter network architecture because as will be shown in Section 3.3 the information being spread over such networks can be obtained not only from the mobile device which created it, but also from other mobile devices. Nevertheless, it is obvious that much faster diffusion of information can be achieved in this way. At the same time to prevent sharing as well as keeping in memory outdated information such applications might remove it automatically after the expiration time typical for a specific type of service. Of course, it is not always possible to communicate face-to-face or using short-range wireless technology with a

new contact in case the information was not obtained from origin. However, getting in touch with interesting contacts is still possible via SMS, MMS, phone call, IM or e-mail.

The Nokia Flier application is an interesting representative of DSA since the user can choose only one flier to share among a number of the obtained ones (or create a new one). Here we deal with simple collaborative filtering, since with this application only the best fliers are being spread over the network (see Joke Service in Section 3.5).

3 Mobile Encounter Networks

3.1 Introduction

Mobile peer-to-peer (MP2P) networks are designed for resource sharing in a mobile environment. These networks include infrastructureless ad hoc networks as well as infrastructure-based cellular networks with end terminals having capabilities to share their resources. There are various examples of peer-to-peer applications for ad hoc networks [18, 9, 38, 15], cellular networks [21, 2] or both [13, 17].

This chapter introduces a new class of mobile networks. These networks are called mobile encounter networks. Mobile encounter networks do not require an infrastructure and do not have problems of multihop communication requiring much lower density of mobile devices compared to ad hoc networks for operation. There are however certain limitations of applications operating in mobile encounter networks, but as will be shown, some applications are very feasible to be built using the mobile encounter network architecture.

3.2 Definition and Description

Mobile encounter networks are formed of two mobile devices coming across each other and having a connection between them using a short-range radio technology (such as Bluetooth [4] or 802.11 WLAN [16]). One encounter contains the discovery of devices, connection establishment between two devices and the exchange of data. One mobile device can form connections to multiple other devices in succession. In this way, the information from one device can be copied to other mobile devices. The duration of the encounter is usually short, because of the mobility of the devices, but it can also be long if the mobile devices are not moving. A mobile encounter network is the network resulting from all encounters. Mobile encounter networks are very dynamic and in contrast to ad hoc networks, they do not provide continuous multihop communication, but only a pair-wise communication between two mobile devices.

Peer-to-Peer refers to decentralized and self-organizing overlay architectures of equal and autonomous entities. Peer-to-Peer architectures are designed to support the finding and using of distributed resources and they do not usually have a central entity, which manages the network. Mobile Peer-to-Peer then extends Peer-to-Peer by allow-

ing resource sharing in a mobile environment. Mobile encounter networks are also used for resource sharing. They are also decentralized and consist of equal and autonomous entities, but do not require functionalities for self-organization.

Multihop resource discovery commonly found in peer-to-peer networks is missing from mobile encounter networks. Resource discovery in mobile encounter networks is done via pair-wise communication between two mobile devices inside an encounter and does not involve other devices outside the encounter. The way of obtaining data in mobile encounter networks is push-based rather than pull-based commonly found in other mobile peer-to-peer networks [28]. Mobile encounter networks are not intended for searching, but rather for spreading information to interested parties. Therefore, mobile encounter networks do not strictly belong to the areas of Peer-to-Peer or Mobile Peer-to-Peer.

3.3 Information Diffusion

Information diffusion in mobile encounter networks happens when a mobile device stores information obtained from another mobile device in an earlier encounter and later forwards the information to another mobile device in a new encounter. The diffusion of information in such a network is delayed and represents a way of replicating information lazily among the devices. The origin of the information diffused in mobile encounter networks can be any external source e.g., user or device.

The devices participating in the mobile information diffusion need to provide some resources for the diffusion process. For example, transmitting information always requires some amount of battery power and therefore some parts of the information diffused in the mobile encounter network need to be relevant for the user of the device. However, in general participating in such a network is inexpensive, because transmitting information using short-range radio technologies does not involve a network operator and consequently payments are not needed.

Some devices in the network might restrict the diffusion of information by not forwarding any information. Usually, this reduces the speed of information diffusion in mobile encounter networks, but in some cases it might be beneficial. For example in applications where any user can create content, a user could select which content is good and allow the further diffusion of that information to other mobile devices. Then, each mobile device when receiving the same content multiple times decides using for example a threshold how many times the same content needs to be obtained and if a given threshold is exceeded, the information is accepted. As a global effect, only content rated good enough would flow in the network. This is called collaborative

filtering [35].

Earlier studies on information diffusion in a delay-tolerant networks using simulations can be found from [19, 20, 28, 37].

3.4 Benefits and Shortcomings

Compared to infostations [14], mobile encounter networks provide faster diffusion of information, because mobile devices can obtain information not only from the infostation, but also from other devices.

In infrastructure-based information diffusion, for example in a GSM network, the network is used to transmit information from a mobile device to a centralized server and mobile devices use this centralized server to obtain data. Compared to infrastructure-based information diffusion, mobile encounter networks often provide a slower information diffusion and limited coverage. This is because the information is only available to those mobile devices which have encountered other mobile devices providing the information. However, there are certain advantages in information diffusion over mobile encounter networks. First, there is no need for infrastructure for transmission of data. Second, the information diffusion in mobile encounter networks is inexpensive, because no external service provider is needed for the transmission of data. Also, because all communication happens inside encounters between two mobile devices, there is no need for external server where information would be stored. Without an external server, which potentially could become a bottleneck in a large system, mobile encounter networks are also very scalable.

A mobile encounter network resembles an ad hoc network in the sense that it allows two mobile nodes that come within range of each other to establish a connection and exchange data. There are however many differences between mobile encounter networks and what is usually considered as ad hoc networking.

Perkins [29] shows that the main problem in ad hoc networks is to provide multihop routing of data (in a unicast, multicast or broadcast way) through the mobile nodes which are potentially moving and continuously changing the configuration of the network. A route in an ad hoc network can be repetitively broken due to a node in its path moving out of the reach of its neighbors and a significant research effort is put into designing algorithms for repairing broken routes without generating too much control traffic. Moreover, routing requires assigning global addresses to the nodes, since the data sent by a source node is targeted at a specific destination node (or possibly at a multicast group), which is not necessarily within the transmission range of the source.

Compared to ad hoc networks, mobile encounter networks differ in that they do not

provide any routing facility, since the goal is to spread information to as many nodes as possible rather than to target-specific destinations: a source node running one given application over a mobile encounter network sends data to any other node running the same application coming within its transmission range. The other node will cache the data, and later send it further when it comes within range of still other nodes. There is no need for mechanisms preventing data to loop back to its original source as in many ad hoc networks protocols. Moreover, since data is sent only to neighbors which are within transmission range, global addressing is not necessary (the underlying communication medium takes care of assigning addresses to the nodes which are within range of each other since actual data transmission requires distinguishing different neighbors). In particular, mobile encounter networks do not require the functionalities commonly found in the network layer of protocol stacks whereas the scope of ad hoc networks is mainly in the network layer. Mobile encounter networks operate in the application layer and require from the protocol stack only unreliable link layer transmissions using some wireless radio technology and a reliable data transport functionality of transport layer (such as TCP).

3.5 Feasible Application Areas

In this section five application areas where mobile encounter networks could be used are briefly presented whereas the next chapter is completely dedicated to one particular application of mobile encounter networks, called Gasoline Price Comparison System (GPCS).

3.5.1 Grocery Store Price Service Having bought some goods at a grocery store, a user of the Grocery Store Price Service gets the possibility to share their prices, time and location of the store with other users encountered at the streets or other public places. Being aware of prices of goods taken from different grocery stores, the user can choose the cheapest place for shopping next time. The idea of the Grocery Store Price Service is very similar to GPCS that is considered in Chapter 5.

3.5.2 Dating Service Every user of the Dating Service creates his/her profile of personal characteristics as well as a filter describing what characteristics are preferred for matching new friends. After that the user's mobile device is ready to share the profile with other devices it encounters as the users come across. Having received a profile from a paired device, both applications match their filters with the user profile instantly. In case of a match, the applications from both sides inform their users about

it. The rest depends on them.

3.5.3 Joke Service The users of the Joke Service can create new jokes and exchange jokes with other users encountered. Such application can provide to the users the possibility to rate incoming jokes or even prevent their further propagation, making it possible to propagate only a subset of available jokes that the majority of users like most whereas bad jokes will be quickly eliminated. A similar application is the tourist attraction service, where users rate different tourist attractions and via collaborative filtering the application can provide rated information about different tourist attractions.

3.5.4 Event Service In the Event Service, the initial content for example tourist information is obtained from an infostation, because such content is usually provided by commercial or state organizations. However, the utilization of mobile encounter networks makes information diffusion much faster due to their ability to retransmit the content to other mobile devices. The same kind of mechanism could also be used for example to deliver Really Simple Syndication (RSS) Feed Services [32] to users subscribed to certain RSS feeds.

3.5.5 Newspaper Service Newspapers could also be distributed using mobile encounter networks. Short-range radio technologies are usually faster compared to infrastructure-based networks, for example GSM/GPRS networks, and therefore large data files are more efficient to be delivered using mobile encounter networks. Also for the user, the delivery of newspaper would be cheap. To avoid piracy, the contents of the newspapers would be transferred using mobile encounter networks, but the key for accessing an encrypted content could be obtained via centralized server. With this kind of a mechanism the newspaper provider would be able to charge for the content.

3.6 Conclusion

In this chapter the following aspects of mobile encounter networks have been considered: definition and description as well as benefits and shortcomings of these networks, information diffusion in these networks as well as five application areas where these networks could be used.

Mobile encounter networks are emerging as a new area of mobile communication, because of wide-spread use of short-range radio technologies in today's mobile devices. Some applications are well suited for mobile encounter networks, which are restricted

to only one-hop communication. Compared to ad hoc networks, simpler algorithms can be used, and compared to cellular network based MP2P applications, no infrastructure is needed.

4 BlueCheese Middleware

4.1 Introduction

Since applications in mobile encounter networks interact with each other through mobile communication technologies, such as Bluetooth [4] and 802.11 WLAN [16], it is reasonable to create a middleware module that hides the details of each of them providing application developers with technology independent higher-level functions for communication in these networks.

BlueCheese middleware was developed in a student software project during Autumn 2003 in the Department of Mathematical Information Technology at the University of Jyväskylä. The project's goal was to build a middleware for studying the feasibility of applications in mobile encounter networks. This middleware runs on mobile devices with Symbian OS and uses a common short-range radio technology as a transmission method. In the current implementation of the middleware Bluetooth radio technology was chosen for this purpose because of its availability in the majority of Symbian OS mobile devices, although 802.11 WLAN also might be used in future releases.

BlueCheese middleware was released under a public license. The license used is the Academic Free License 2.0 [1].

In this work the middleware is briefly considered from the point of view of the application developer. The full project documentation is available on the Web and can be found at the project homepage [23].

4.2 Application and Middleware

Applications have a user interface through which users interact with the software performing some specific task. The application might want to communicate with other applications, but does not want to know how this communication is handled in details: it relies on higher-level functions to communicate with other applications. Applications also handle data specific to one task (the task for which the application is designed) and *understands* the meaning of the data.

Middleware, on the other hand, does not have a user interface running without user intervention. It handles tasks that are generic and/or common to several applications. It provides the application with higher-level functions for communication, and takes

care of the details of the procedure. It does not know anything about the data that it is given by the application. Only one instance of the middleware is loaded into the memory while serving numerous applications.

4.3 Purpose and Functionality

The main purpose of BlueCheese middleware is to facilitate communication between applications in mobile encounter networks providing higher-level functions for application specific data exchange. A GSM-based location service is an extra feature of BlueCheese that gets current location from GSM base stations and compares locations to the current one when needed as described in detail in Section 4.4. However, GSM network is not needed for BlueCheese operation, but the location information can be used whenever it is available.

BlueCheese middleware provides its services to multiple applications simultaneously, however, it allows only pair-wise communication between two mobile devices. BlueCheese finds new mobile devices automatically and establishes a connection to the first one seen in the range unless it has already been seen recently. Other found mobile devices are queued and served as soon as the existing connection is closed. BlueCheese also handles queuing of the packets in both the sending and receiving sides of the connection.

There are two different data transferring modes in BlueCheese. The first one is the packet mode and the second one is the stream mode. The packet mode is meant for sending and receiving packets with limited size and it provides a connectionless service. The stream mode, on the other hand, is meant for large amounts of data and it is connection-oriented.

Here are the functionalities required from the middleware by applications:

- create a session with the middleware and inform it of the kind of data the application is interested in;
- release the session with the middleware;
- whenever a mobile device is in range, the application is informed that a communication can take place with the device;
- send a small amount of data to the remote device (data packet);
- receive a small amount of data from the remote device (data packet);
- send a large amount of data to the remote device (data stream);

- get a large amount of data from the remote device (data stream).

Before the application can use the services provided by the middleware, it must create a session with the middleware and inform it of the kind of data the application is interested in. The middleware performs automatic scanning of mobile devices at background and informs the application whenever a new suitable device is found. Then the application is able to send/receive a small amount of data in the packet mode or a large amount of data in the stream mode. When the application does not need the services anymore, it has to release the session with the middleware.

4.4 GSM-based Location Service

The most efficient geographical location is done using GPS (Global Positioning System), a satellite-based infrastructure that allows to know one's precise location (using latitude and longitude) on the surface of the Earth. It is very precise, but requires a dedicated receiver. Most mobile devices do not integrate a GPS receiver.

The GSM-based location service is meant to be a reasonable alternative to GPS, providing its solution based on knowledge of the GSM network's design. In the GSM network the covered area is divided into zones called local areas. Each local area is divided into cells. Each cell has a unique identification number within a local area, and each local area has a unique identification number within a network. Most of the Symbian OS mobile devices have access to the GSM network and due to that they are able to get the identification numbers of the current cell and local area. Using this data, the GSM-based location service was implemented, which is not as precise as GPS, but still gives an estimation of the distance between the device's position and an entity located in a given cell. The estimation procedure is presented below:

- if the device and the entity are in the same cell, they are very close to each other;
- if they are in different cells, but within the same local area, they are quite close to each other;
- if they are in different local areas, they are far away from each other.

As the mobile device is likely to move within a GSM network, it can remember in what zone (a set of cells and/or a set of local areas) it is often located, and consider that this zone is *home* and everything located in this zone is *close*. The device can then accept location-dependent data whose origin is within the zone, and reject data coming from the outside of that zone.

One can also imagine to build a map of the cells which are encountered while moving, exchange these maps with the mobile devices that one communicates with, and thus slowly build a bigger map which would also allow to estimate straight-line distances between two cells. The distance unit could be the number of cells one needs to cross to go from one of these cells to the other one.

Nevertheless, one possible inconvenience of the use of this idea is that different mobile operators have different GSM networks and therefore different identification numbers of cells and local areas for the same zone. That is why, in order to build the map shared by community, mobile devices have to remember the identification numbers of numerous GSM networks that eventually leads to an increased amount of bytes to be stored and/or transmit.

4.5 BlueCheese Protocol Stack

As mentioned earlier, the Bluetooth radio technology was chosen as a transmission method in the current implementation of the middleware. Thus, the BlueCheese protocol stack includes the Bluetooth protocol stack. The Bluetooth transmission protocols

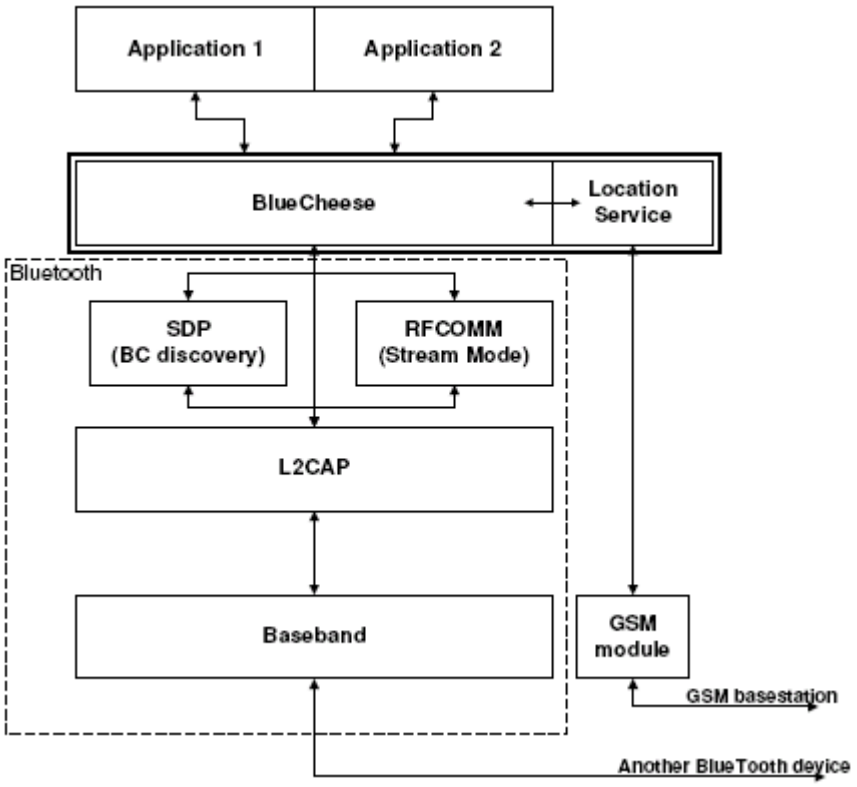


Figure 4.1: BlueCheese middleware (adapted from [24])

L2CAP and RFCOMM as well as the SDP protocol are involved in the communication process between Bluetooth-enabled mobile devices:

- L2CAP (Logical Link Control and Adaptation Protocol) is used for the packet mode transfer;
- RFCOMM (Radio Frequency Communications Protocol) is used for the stream mode transfer;
- SDP (Service Discovery Protocol) is used for discovering other mobile devices.

Figure 4.1 illustrates a number of different protocols and modules involved in BlueCheese middleware and their relations. Refer to [4] for more information about the Bluetooth radio technology.

4.6 Conclusion

BlueCheese middleware has been briefly considered in this chapter from the point of view of the application developer. The middleware was released under public license and was built for studying the feasibility of applications in mobile encounter networks. The main purpose of BlueCheese middleware is to facilitate communication between applications providing higher-level functions for application-specific data exchange. The Bluetooth short-range radio technology is used as a transmission method. The location service is an extra feature of BlueCheese middleware that locates the mobile device position in the GSM network based on the identification numbers of the current cell and local area.

5 Gasoline Price Comparison System

5.1 Introduction

Gasoline Price Comparison System (GPCS) is a mobile application, executed in mobile devices with Symbian OS (Series 60 Platform) [11, 34, 33]. Its purpose is to help making decisions of where to refuel. To achieve that, a mobile device collects the following attributes of every gas station where the user's car is refuelled and diffuses them to other mobile devices:

- Brand and location of gas station;
- Price and type of gasoline;
- Time of buying gasoline.

A middleware, called BlueCheese, that uses Bluetooth [4] as a transmission method, can be integrated with GPCS to make the information exchange possible. BlueCheese middleware was developed in a student software project [23] during autumn 2003 in the University of Jyväskylä (see Chapter 4 for details). However, BlueCheese has not been integrated with GPCS yet. Both the application and middleware were implemented using the C++ programming language, Series 60 SDK for Symbian OS and Microsoft Visual C++ 6.0 IDE.

5.2 Application Scenario

The following scenario illustrates the use of the application. A driver, equipped with GPCS, buys gasoline at his/her favorite gas station. The attributes, described above, are sent into the application with a bill for the gasoline, bought by using the driver's mobile device. It is expected that mobile device holders will have such a possibility very soon. Having received these attributes at the gas station, the mobile device starts diffusing them using a short-range radio technology such as Bluetooth to other mobile devices it encounters as the car moves around. After a certain period of time the application removes them automatically to prevent sharing and keeping in memory outdated information. The same way other drivers, equipped with GPCS, share the information about other gas stations. By making such exchanges, all participants

receive the information about further gas stations on their way, making it possible to choose the best place where to refuel next time, saving their money and time. This also boosts market-based economy by giving customers equal information about the market situation.

5.3 User Interface

At the current stage of development there are four *views* in the application: gas stations view, gasoline attributes view, profiles view and profile settings view. In this work the term *view* is used in consideration of the View Architecture that is one of the different approaches available in Avkon for writing an application UI [7]. Each view is described in the following sections.

5.3.1 Gas Stations View The purpose of the Gas Stations View is to display brief information about currently known gas stations. Each gas station, represented here by its brand logo and location, occupies a separate item in a list as illustrated in Figure 5.1 (a). The items in the list are sorted by gasoline price as a primary key and time as

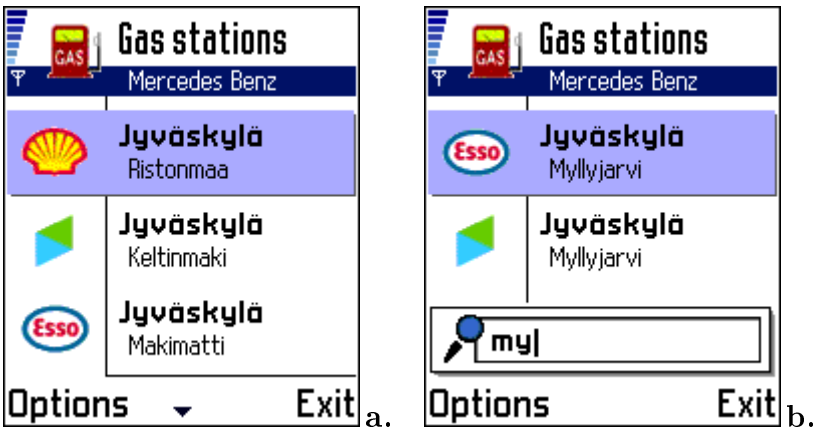


Figure 5.1: Gas stations filtered by the profile in use (a) plus location (b)

a secondary key. They are also filtered by the profile currently in use, whose name can be seen at the application navi pane, e. g. *Mercedes Benz* (for more information about the profiles see the description of the Profiles View presented below). In case the driver is interested in a particular location or region rather than the lowest price, there is a popup toolbar for filtering items on the fly, as depicted in Figure 5.1 (b). The toolbar can be easily activated just by starting typing on the mobile device keypad. Having chosen a desired item from the list, the driver can press either the scroll key or *Options* and *Open* to study the corresponding gasoline attributes, residing on the other view.

Finally, let us consider the events, which lead to modifications in the list:

- A new item addition or an existing item update. It arises after buying gasoline or after an encounter with another mobile device;
- An existing item removal. It arises after a certain period of time when the information about gasoline is considered outdated.

5.3.2 Gasoline Attributes View The purpose of the Gasoline Attributes View is to display the attributes of a particular gasoline as well as the size of the list. As described earlier, these attributes include *location*, *brand* and *type* as well as *time* and *price*. The combination of the first three attributes defines the so called *gasoline id*. The last two attributes are the variables associated to it; their values are being constantly updated. Since gas stations offer different gasoline types, such as 95E, 98E and Diesel (and the profiles support displaying different gasoline types simultaneously), several items in the list in the Gas Stations View might be identical. As shown in Figure 5.2

Figure 5.2: Gasoline attributes: 1st (a) and 7th (b) items of the list

the application status pane duplicates the information of the appropriate list item in the Gas Stations View, whereas the other attributes are presented in a convenient list-form in the main pane: the attribute names are on the left side, the attribute values are on the right side. According to Figure 5.2, 95E gasoline price was 1.219 EUR per a liter at the Shell gas station in Ristonmaa, Jyväskylä 3 minutes ago (a), whereas the same gasoline was slightly more expensive at the Esso gas station in Myllyjarvi, Jyväskylä 2 minutes ago (b).

The driver can go to the next or previous item of the list by pressing the *Left* or *Right* arrow on the scroll key.

5.3.3 Profiles View The purpose of the Profiles View is to manage user defined profiles as illustrated in Figure 5.3. It is possible to create a new profile, remove an existing one, activate or personalize it. Nevertheless, the profile currently in use cannot

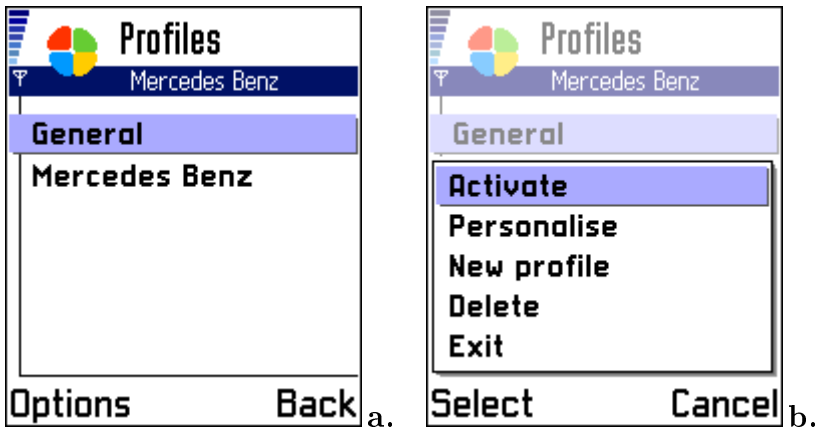


Figure 5.3: User-defined profiles (a) and menu (b)

be removed. Only the available options are displayed in the menu, e. g. while the active profile is selected in the list, the options *Activate* and *Delete* are not displayed. In spite of the fact that the application is able to keep many profiles, it uses only one profile at any one time. While launching the application for the first time, the default profile named *General* is created. This profile does nothing by default, however, it can be personalized. It can also be removed, once another profile has been created and activated.

Using profiles, the driver can specify gasoline preferences for a particular car to filter out the irrelevant information. These profiles become very useful in case the driver has several cars and these cars use different gasoline. The profile currently in use is shown at the application navi pane.

5.3.4 Profile Settings View The purpose of the Profile Settings View is to modify settings of a particular profile as depicted in Figure 5.4. The driver can specify the following gasoline attributes to be taken into account while filtering:

- A name of the profile, which is an arbitrary text used for profile identification;
- A set of gasoline brands, which are predefined alternatives defined in the application resource file;
- A set of gasoline types, which are predefined alternatives defined in the application resource file.

According to Figure 5.4, the profile named *Mercedes Benz* is intended for a vehicle that uses both 95E and 98E gasoline provided by all gasoline suppliers, such as Shell, Esso and Neste.

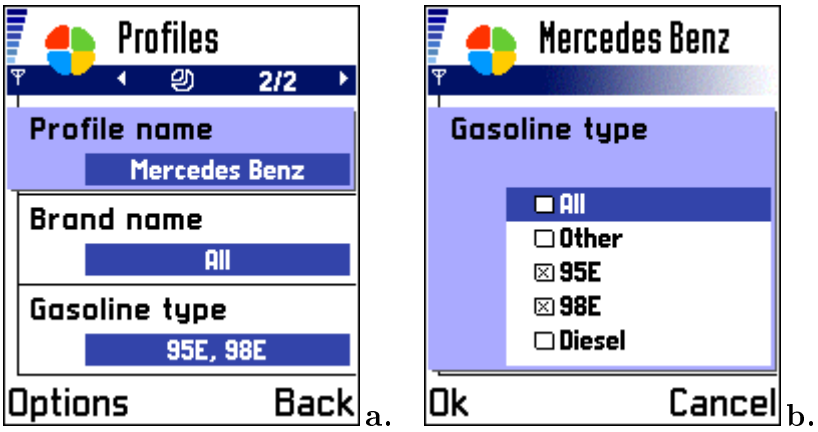


Figure 5.4: Profile settings (a) and gasoline type alternatives (b)

The driver can go to the next or previous profile by pressing the *Left* or *Right* arrow on the scroll key.

5.4 Design and Implementation

In this section the following aspects of the application are considered: localization, advantages of splitting the application into two parts, the UI and the Engine, their architectures, a scenario for communication between two mobile devices with a practical example and interactions between the UI and the Engine parts.

5.4.1 Language Localization Localization needs should be considered from the beginning of a project [8] since in order to meet localization requirements applications must distinguish language-specific data from common data, e.g. programmers develop the code whereas translators provide language-specific data. The localization support in Symbian OS is provided by compiled resource files, which are not embedded into the application file and thus new resource files can be added on-the-fly. While launching the application, Symbian OS loads the appropriate resource file base on the current system language. Currently GPCS supports two languages: English and Finnish.

5.4.2 Splitting the UI and the Engine Applications are normally split into two parts, the Engine and the UI, to aid maintainability and flexibility. The application engine, also known as the application model, deals with the algorithms and data struc-

tures needed to represent the application data. The application UI, sometimes called the app, deals with the on screen presentation of the application data and the overall behavior of the application [7].

GPCS was designed in consideration of this statement. The application consists of two main components: `Gpcs` and `GpcsEngine`. The `Gpcs` component is the standard set of UI classes that provides the user interface framework and exists as a standard Series 60 Application. The `GpcsEngine` component provides the model and data for use by the `Gpcs` component and exists as a Shared Library DLL providing a fixed API that can be used by more than one program. Being designed this way the application has several advantages:

- Changes to the user interface are less likely to affect the model;
- When porting to another Symbian OS mobile device, typically the model remains untouched and all that needs to change is the UI.

The following component diagram illustrates the split of classes over the `Gpcs` and `GpcsEngine` components, and their interrelationships:

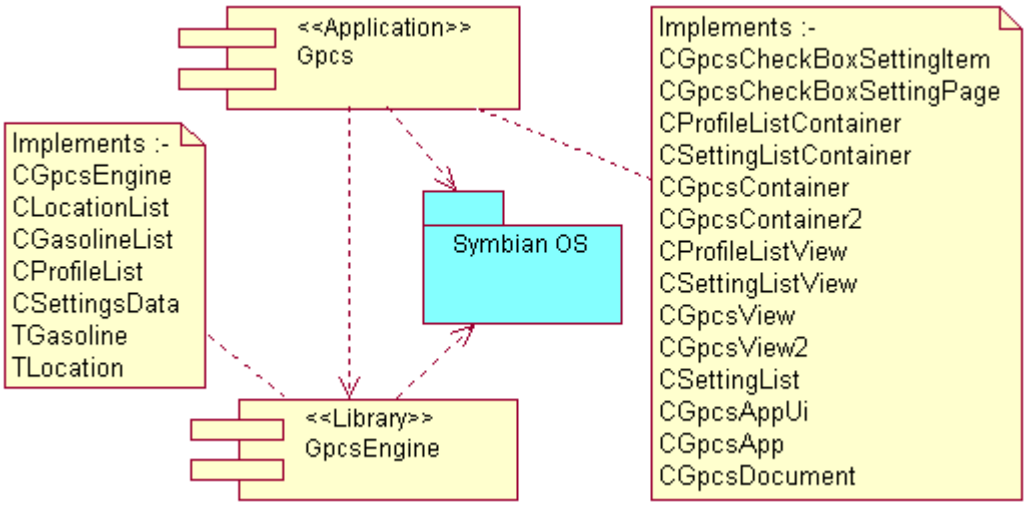


Figure 5.5: Splitting the UI and the Engine

5.4.3 GPCS UI Series 60 adds a User Interface Layer (Avkon) onto the underlying Uikon from Symbian OS v6.1. Avkon provides a set of UI components and an application framework designed specifically for Series 60 devices [8]. Application UIs can be simple with only one main screen, e. g. a calculator, or complex with many screens, e. g. a messaging application. Three architectural approaches have been identified for writing an application UI in Avkon [7]:

- Traditional Symbian OS Control Architecture;
- Dialog Based Architecture;
- View Architecture.

The choice of application architecture will depend on the application complexity, the view navigation and communication requirements and the screen layout requirements. As mentioned previously, View Architecture was chosen for the GPCS UI. This approach allows applications to register *views*, with one being active in each running application at any one time. It does not dictate what a view is, but it provides support for a view being a display page on the screen [7]. Applications designed using View Architecture consist of four main application framework classes as shown in Figure 5.6.

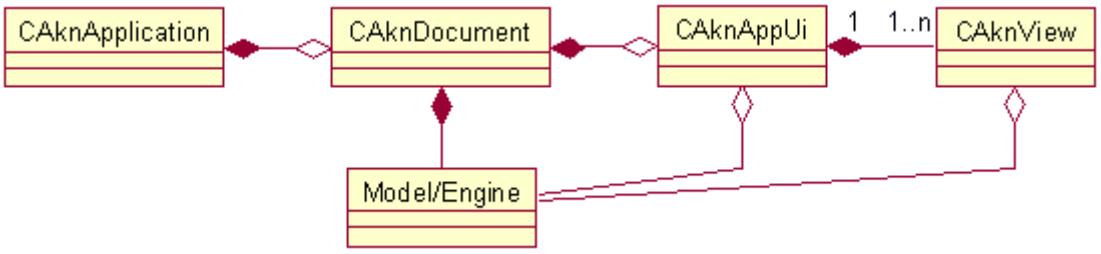


Figure 5.6: View Architecture

The *Application* class operates as a startup object for the Series 60 application framework and defines the application's properties. It also creates the document. The base class for the application class is `CAknApplication`.

The *Document* class is used to store the application persistent state. An application must have an instance of the Document class, although it may only be required to launch the AppUi. The base class for the documents is `CAknDocument`.

The *AppUi* class is responsible for handling application-wide events such as Options menu commands, opening/closing files and the application losing focus. It typically has no screen presence. Instead it delegates drawing and screen-based interaction to the Views it owns. The base class for AppUis is `CAknViewAppUi`.

The *View* class is responsible for displaying data on the screen that the user can interact with. Typically, Views are notified of updates in the model's state by an observer mechanism. They also pass user commands back to the AppUi. The base class for Views is `CAknView`.

Note that all visible controls in Symbian OS must be derived from `CCoeControl`. However, the `CAknView` class is derived directly from `CBase`, but a `CAknView`-derived class typically has a `CCoeControl`-derived container.

The application framework is responsible for the creation of the Application. The Application constructs the Document. The Document constructs the Engine and the AppUi, which in turn creates the Views.

5.4.4 GPCS Engine The engine is represented in the UI by a `CGpcsEngine` class that shares responsibility between the three classes, `CProfileList`, `CLocationList` and `CGasolineList`, as illustrated in Figure 5.7. All three classes are derived from a standard Symbian OS template class, `CArrayPtrFlat`, inheriting the behavior of flat arrays. However, different classes, `CSettingsData`, `TLocation` and `TGasoline`, are specified as a template argument.

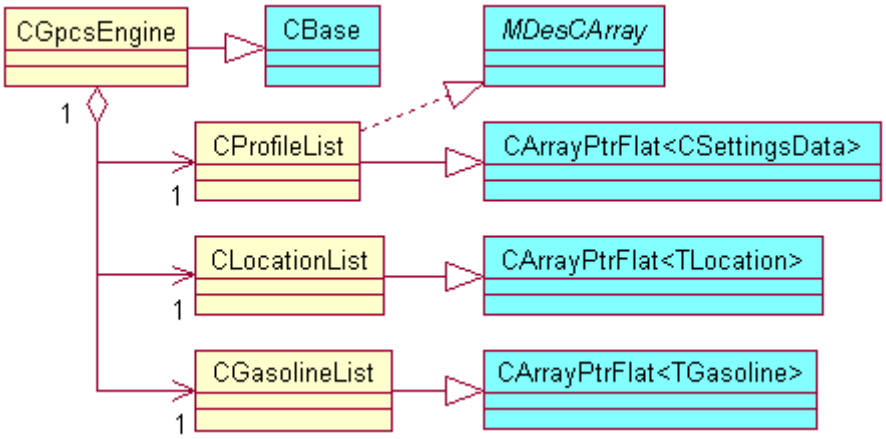


Figure 5.7: GPCS Engine Class Diagram

The first class, `CProfileList`, is responsible for manipulating data of the user defined profiles. It implements a `MDesCArray` interface to have the behavior of descriptor arrays. Thus, an instance of the class can be treated as a descriptor array of the profile names. The template argument, `CSettingsData`, was designed for keeping settings of a particular profile.

The second class, `CLocationList`, is responsible for manipulating data of the gas station locations. Each location is split into major and minor parts, e. g. *Jyväskylä* city and *Myllyjärvi* city section, that are stored in memory as singletons. For this purpose two instances of a standard Symbian OS class for descriptor arrays, `CDesCArray`, are used as private member variables (not shown in the class diagram for simplicity). An index number of the given major part, an index number of the given minor part and the hash value obtained by hashing the major and minor parts (called *location id*) represent an instance of the `TLocation` class or an item of the `CLocationList` class.

The third class, `CGasolineList`, is responsible for manipulating data of the gaso-

line attributes. As mentioned earlier, these attributes include *brand* and *type* with predefined alternatives and therefore they have numerical equivalents as well as *time* and *price* that are numbers as such. Together with *location id* of the given location, they represent an instance of the `TGasoline` class or an item of the `CGasolineList` class. There are two private member variables of the `CDesCArray` class (not shown in the class diagram for simplicity) for keeping *brands* and *types*.

It is obvious that instances of the `TLocation` and `TGasoline` classes are linked to each other through *location id*. This is fine for network communication during an encounter, but practically inconvenient for manipulating data inside the application. Indeed, each time a gas station location, where a particular gasoline has been bought, is to be displayed on the screen, it is necessary to look for *location id* of the gasoline among numerous items of `CLocationList`. In order to speed up the link between these objects the following trick is performed: instead of *location id* each object of `TGasoline` class contains the index number of an appropriate item of `CLocationList` while the objects are inside a device.

The following method of the `CLocationList` class illustrates the way of obtaining *location id*:

```
01:     TUint CLocationList::MakeId(const TDesC& aMajor,
02:           const TDesC& aMinor) const
03:     {
04:        TUint id = 0;
05:        TInt i;
06:        for (i = aMinor.Length() - 1; i >= 0; i --)
07:        {
08:           id = (id << 1) + aMinor[i];
09:        }
10:        for (i = aMajor.Length() - 1; i >= 0; i --)
11:        {
12:           id = (id << 1) + aMajor[i];
13:        }
14:        id %= 65536;    // id ∈ [0, 65536)
15:        return id;
16:     }
```

As mentioned previously, C++ programming language was chosen for the GPCS implementation. The method takes two input arguments of type `TDesC`, which is an

abstract base Symbian OS class for descriptors, – the major and minor parts of a given location and returns an integer – *location id* specific to the input arguments. The method signature tells us that neither the input arguments nor the class internal state will be changed after the method invocation. There are two cycles in the method used for accessing individual characters of the major and minor parts. A code of the current character is added to double sum of codes of previous characters as shown in the lines 8 and 12. Since *location id* occupies 2 bytes in the `TGasoline` class, it equals residue of division of the total sum by 65536, the line 14. See Appendix A for more information about the functionality of the `GpcsEngine` component classes.

Splitting the diffusion data into two classes, `CLocationList` and `CGasolineList`, in conjunction with an appropriate communication protocol has several advantages:

- The memory required for the data is kept minimal;
- The duration of the communication is kept minimal;
- Network traffic is kept minimal.

5.4.5 Communication Scenario Let us consider a possible scenario for communication between two mobile devices. As soon as a connection is established the devices start to send each other their gasoline attributes collected so far (or, in other words, they send each other objects of the `TGasoline` class). Having received them the device skips over their *location id*s. If *unknown* ones exist, it makes a request for their meanings (or, in other words, a request for the major and minor parts that are accessible through particular objects of the `TLocation` class).

Reducing the duration of the communication is a very important task because an encounter is usually short. In this scenario the devices send each other all minimal pieces of information that might be relevant avoiding the negotiation at the application level. Since gasoline attributes are rather small, in the current implementation 8 bytes per instance of the `TGasoline` class, 128 such objects occupy only 1 Kb whereas a Bluetooth symmetric link allows data rates of 432.6 Kbps. The decisions of what to keep and what to leave out are made off-line based on the *time* attribute of the given object with particular *brand*, *type* and *location id*s. Now the benefits of sending gasoline attributes and gas station locations separately become evident. Each *unknown* location is received only once by a particular mobile device whereas gasoline prices are constantly changing. Moreover, gas station locations usually occupy more space because of the textual content.

A simple and practical example of such communication between nodes `A` and `B` is

illustrated in Figure 5.8. The following three lists of objects of the `TGasoline` class are used during the example explanation provided below:

- `MAIN`: accessible to the user through the UI as depicted in Figure 5.2;
- `TODO`: not accessible to the user since *location ids* are *unknown*;
- `TEMP`: exists temporarily only at the time of an encounter.

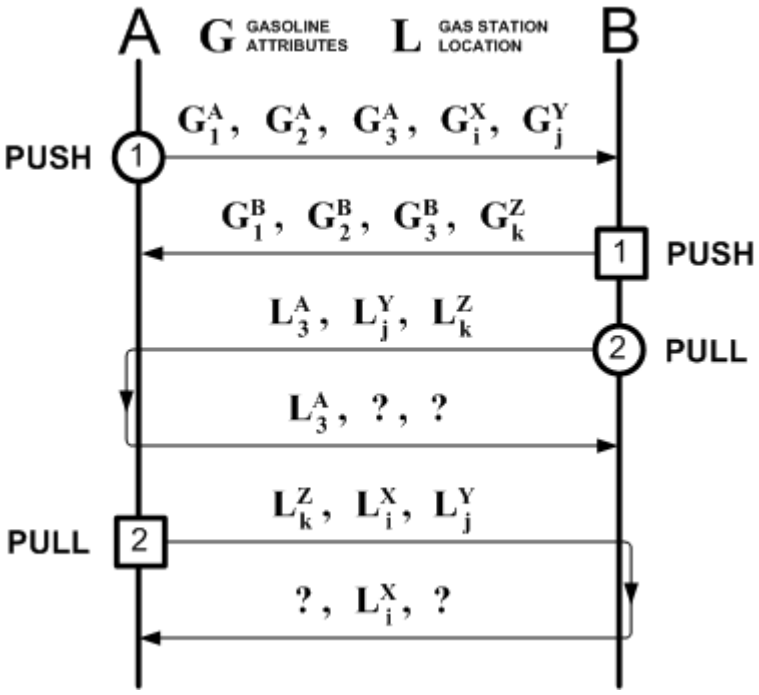


Figure 5.8: Communication example

Where G_i^N is the i^{th} item of `MAIN` at node `N`. First the nodes, `A` and `B`, push the gasoline attributes to each other (1). Then each node pulls the *unknown* locations, if any (2):

1. Initial state. Let us assume that `A` has two objects with *unknown* locations, which have been received from nodes `X` and `Y`, whereas `B` has only one such object, which has been received from node `Z`:

	A	B
MAIN	G_1^A, G_2^A, G_3^A	G_1^B, G_2^B, G_3^B
TODO	G_i^X, G_j^Y	G_k^Z
TEMP	-	-

2. Having established the connection, `MAIN+TODO` are pushed to another node where `TEMP` is filled:

	A	B
TEMP	$\mathbf{G_1^B}$, $\mathbf{G_2^B}$, $\mathbf{G_3^B}$, $\mathbf{G_k^Z}$	$\mathbf{G_1^A}$, $\mathbf{G_2^A}$, $\mathbf{G_3^A}$, $\mathbf{G_i^X}$, $\mathbf{G_j^Y}$

3. Items' *location id*s of `TEMP` are skipped over. Let us assume that `A` finds only one *unknown id* in the last item, whereas `B` finds two *unknown id*s in the third and in the last ones. These items are moved into `TODO`:

	A	B
TODO	$\mathbf{G_k^Z}$, G_i^X, G_j^Y	$\mathbf{G_3^A}$, $\mathbf{G_j^Y}$, G_k^Z
TEMP	G_1^B, G_2^B, G_3^B	G_1^A, G_2^A, G_i^X

4. Items' locations of `TODO` are pulled from another node. In case `TODO` is empty, an appropriate notification must be sent, e.g. *<All-Found>*.

5. After pulling or after *<All-Found>* from both sides the connection is closed. The objects with received locations are moved from `TODO` to `TEMP`:

	A	B
TODO	G_k^Z, G_j^Y	G_j^Y, G_k^Z
TEMP	$\mathbf{G_i^X}$, G_1^B, G_2^B, G_3^B	$\mathbf{G_3^A}$, G_1^A, G_2^A, G_i^X

6. In Off-line mode objects of `TEMP` are analyzed by *time* with appropriate (identical *location*, *brand* and *type*) objects of `MAIN` and the newest ones are moved to `MAIN`. `TEMP` is cleaned:

	A	B
MAIN	$\mathbf{G_4^A}$, $\mathbf{G_5^A}$, $\mathbf{G_6^A}$, $\mathbf{G_7^A}$, G_1^A, G_2^A, G_3^A	$\mathbf{G_4^B}$, $\mathbf{G_5^B}$, $\mathbf{G_6^B}$, $\mathbf{G_7^B}$, G_1^B, G_2^B, G_3^B
TEMP	-	-

Having exchanged the data `A` got three new objects, G_1^B, G_2^B and G_3^B, and a new location, L_i^X, whereas `B` got four new objects, G_1^A, G_2^A, G_3^A and G_i^X, with a new location, L_3^A; each node still has two objects with *unknown* locations, G_j^Y and G_k^Z.

5.4.6 Updating Views As mentioned in Section 5.3, the list of collected gasoline attributes is being updated constantly due to the following events: new items are coming after either buying gasoline or an encounter with another mobile device and because existing items are leaving the list after certain period of time. The application is designed so that new items having come to the list are immediately displayed on either the Gas Stations View or the Gasoline Attributes View whereas outdated items are dropped during invoking the application. Let us consider these two types of events in detail.

In the first case the views are notified of updates in the Engine state by an observer mechanism [12]. The following figure illustrates the relations between the Engine and the Gas Station View (or, in other words, the relations between `CGpcsEngine` and `CGpcsView` with `CGpcsContainer` classes):

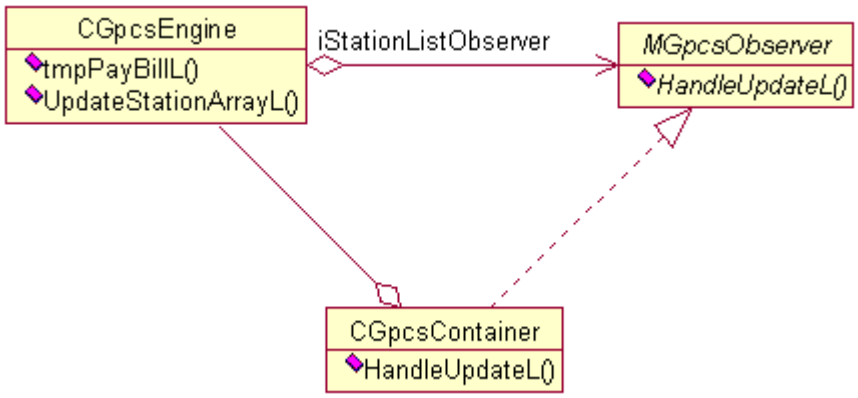


Figure 5.9: Simple observer mechanism

As shown in Figure 5.9, the `CGpcsContainer` class implements the `MGpcsObserver` interface that has only one pure abstract method `HandleUpdateL()`. This method is called by the `CGpcsEngine` class through a pointer, `iStationListObserver`, whenever it is necessary to update the view. In the current implementation of the software there is only one way to get new gasoline attributes. For this purpose the `CGpcsEngine` class has a public method `tmpPayBillL()` that is used to emulate paying bills as described in Section 5.2. Its another public method, `UpdateStationArrayL()`, is typically called by the `CGpcsContainer` in the scope of the `HandleUpdateL()` method to update the view based on the new Engine state.

The same observer mechanism is used between the Engine and the Gasoline Attributes View, but different players are involved:

- A `CGpcsContainer2` class instead of `CGpcsContainer`;
- A `iDetailsListObserver` pointer instead of `iStationListObserver`;
- A `UpdateDetailsArrayL()` method instead of `UpdateStationArrayL()`.

Let us summarize what was said above by the following diagram that shows the sequence involved when the user presses the *Call* key on his/her mobile device while the application is running and focused:

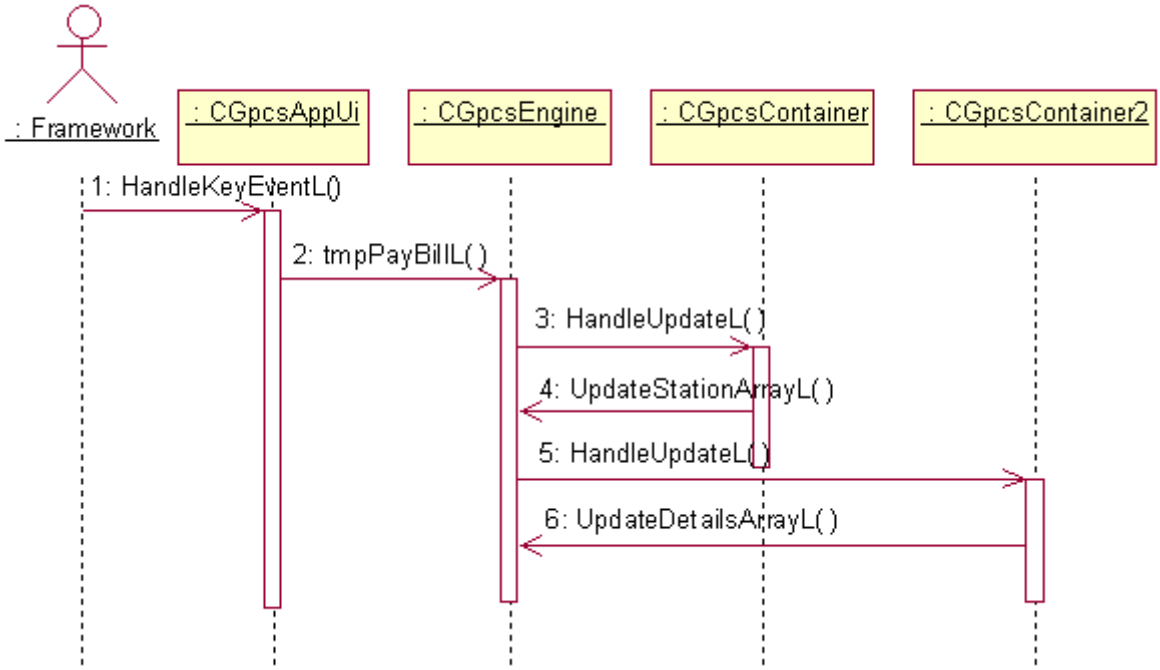


Figure 5.10: Updating views

Message Description

1 The framework notifies the application that an event has occurred through the `CGpcsAppUi::HandleKeyEventL()` method. The event type is checked before continuing with this sequence.

2 The `CGpcsEngine::tmpPayBillL()` method is called in turn. This method emulates paying bills and delivers new gasoline attributes with them to the Engine state.

3 – 4 If the `iStationListObserver` pointer is not equal to `NULL` then the Gas Station View is active and has to be updated. If so, the Engine calls `CGpcsContainer::HandleUpdateL()`. This, in turn, calls `CGpcsEngine::UpdateStationArrayL()`.

5 – 6 These two messages are similar to the previous ones, but they deal with the Gasoline Attributes View.

Having been asked to exit, the application first saves all collected data to a file associated with it. The data is loading back into the application each time the user launches it. While restoring the data, outdated gasoline attributes are dropped and thus only the fresh information is provided to the user at the start time. While running, however, the application does not care about their expiration time. This simplifies the implementation without essential costs because in our case there is no exact definition

of the threshold when the data can be considered outdated. It is clear that the value of the threshold depends on the frequency of price changes at gas stations. In order to determine the approximate value, further research is required. Anyway it should remain the same for all participants in order to speed up the communication between mobile devices.

5.5 Conclusion

In this chapter Gasoline Price Comparison System, its purpose, application scenario, user interface, as well as design and implementation have been considered. GPCS supports two languages, English and Finnish. It is split into two parts, the Engine and the UI, to aid maintainability and flexibility. A scenario for communication between two mobile devices with a practical example and interactions between the Engine and the UI parts have been presented as well.

Gasoline Price Comparison System has been developed as a prototype to illustrate the feasibility of mobile encounter networks applications. In GPCS, data originates at multiple sources and when these individual data pieces are collected, a complete list of gasoline prices will be obtained.

6 Conclusions and Future Work

Different existing mobile applications, which belong to either SPA or DSA, or both, have been briefly reviewed in Chapter 2. These applications have appeared in the past few years because of the wide-spread use of short-range radio technologies in today's mobile devices. DSA form a new class of mobile networks, called mobile encounter networks, whereas SPA, being a more restricted group of mobile applications, do not strictly suit to the mobile encounter network architecture.

Chapter 3 introduces mobile encounter networks, which emerge when two mobile devices come across each other and establish a temporary connection between them using a common short-range radio technology. Local information exchanges between mobile devices result in a broadcast diffusion of information to other users of the network with a delay. Compared to ad hoc networks, simpler algorithms can be used, and compared to cellular network-based MP2P applications, no infrastructure is needed. Section 3.5 presents five application areas where mobile encounter networks could be used.

Since applications in mobile encounter networks interact with each other through mobile communication technologies, such as Bluetooth [4] and 802.11 WLAN [16], it is reasonable to create a middleware module that hides the details of each of them providing application developers with technology-independent higher-level functions for communication in these networks. Chapter 4 considers such middleware, called BlueCheese, which has been developed in a student software project at the University of Jyväskylä. In the current implementation of the middleware, Bluetooth radio technology was chosen as a transmission method because of its availability in the majority of Symbian OS mobile devices, although 802.11 WLAN also might be used in future releases.

Chapter 5 describes the UI application, called Gasoline Price Comparison System, which has been developed in the scope of this thesis as a prototype to illustrate the feasibility of mobile encounter networks applications. In GPCS, data originates at multiple sources and when these individual data pieces are collected, a complete list of gas prices will be obtained. GPCS runs on mobile devices with Symbian OS and together with BlueCheese middleware, it might be used for studying the network characteristics of the system. Section 5.4 presents a scenario for communication between two mobile devices with a practical example. Finally Appendix A reports the functionality of

GPCS engine's classes expressed in UML notation as well as the header files in which these classes are defined.

The future work concentrates on modelling and simulation of mobile information diffusion in mobile encounter networks and field testing of GPCS and BlueCheese using Bluetooth-enabled mobile devices.

7 Bibliography

[1] Academic Free License 2.0. http://www.opensource.org/licenses/afl-2.0.php.

[2] Andersen F.-U., de Meer H., Dedinski I., Kappler C., Mäder A., Oberender J. and Tutschku K., An Architecture Concept for Mobile P2P File Sharing Services, Workshop Proceedings of Informatik 2004 - Algorithms and Protocols for Efficient Peer-to-Peer Applications, pp. 229–233, 2004.

[3] BEDD Community. http://www.bedd.com/.

[4] Bluetooth Consortium. http://www.bluetooth.org/.

[5] BuZZone Application. http://www.buzzone.net/.

[6] Coulouris G. Distributed Systems: Concepts and Design. Addision-Wesley, 2001.

[7] Developer Platform 2.0 for Series 60: Application Framework Handbook. Available at http://www.forum.nokia.com/.

[8] Developer Platform 2.0 for Series 60: Designing C++ Applications. Available at http://www.forum.nokia.com/.

[9] Ding G. and Bhargava B., Peer-to-peer File-sharing over Mobile Ad hoc Networks, Proceedings of IEEE Annual Conference on Pervasive Computing and Communications Workshops, pp. 104–109, March 2004.

[10] Dreamlove Application. http://www.dreamlove.it/.

[11] Edwards L., Barker R., and the Staff of EMCC Software Ltd. Developing Series 60 Application: A Guide for Symbian OS C++ Developers. Addison-Wesley, 2004.

[12] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[13] Garg N., Shao Y., Ziskind E., Sobti S., Zheng F., Lai J., Krishnamurthy A. and Wand R., A Peer-to-Peer Mobile Storage System, Technical Report, TR-664-02, Computer Science Department, Princeton University, October 2002.

[14] Goodman D., Borrás J., Mandayam N. and Yates R., INFOSTATIONS: A New System Model for Data and Messaging Services, Proceedings of the IEEE Vehicular Technology Conference '97, vol. 2, pp. 969–973, Phoenix, AZ, May 1997.

[15] Helal S., Desai N., Verma V. and Lee C., Konark. A Service Discovery and Delivery Protocol for Ad-Hoc Networks, Proceedings of the IEEE Wireless Communication and Networking Conference (WCNC 2002), New Orleans, LA, March 2003.

[16] IEEE 802.11 Wireless Local Area Networks - The Working Group for WLAN Standards. http://grouper.ieee.org/groups/802/11/.

[17] Kato T., Ishikawa N., Sumino H., Hjelm J., Yu Y. and Murakami S., A Platform and Applications for Mobile Peer-to-Peer Communications, http://www.research.att.com/~rjana/Takeshi_Kato.pdf.

[18] Klemm A., Lindemann C. and Waldhorst O., A Special-Purpose Peer-to-Peer File Sharing System for Mobile Ad Hoc Networks, Proceedings of IEEE Semiannual Vehicular Technology Conference (VTC2003-Fall), Orlando, FL, October 2003.

[19] Kurhinen J. and Vuori J., Information Diffusion in a Single-Hop Mobile Peer-to-Peer Network, Proceedings of the 10th IEEE Symposium on Computers and Communications, ISCC 2005, Cartagena, Spain, 2005.

[20] Kurhinen J. and Vuori J., MP2P Network as an Information Diffusion Channel, Proceedings of the 62nd IEEE Vehicular Technology Conference, VTC Fall 05, Dallas, USA, 2005.

[21] Marossy K., Csucs G., Bakos B., Farkas L. and Nurminen J. Peer-to-peer content sharing in wireless networks. The 15th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 2004). Volume 1, pp. 109–114. 5–8 Sept. 2004.

[22] MobiLuck Application. http://mobiluck.com/.

[23] MoPeDi Project Homepage. http://kotka.it.jyu.fi/mopedi/.

[24] MoPeDi Project – Software Design. http://kotka.it.jyu.fi/mopedi/docs/documents/SoftwareDesign.pdf.

[25] Nokia Flier Application. Available at http://europe.nokia.com/nokia/0,,58683,00.html.

[26] Nokia Sensor Application. http://www.nokia.com/sensor.

[27] Object Management Group. UML Notation Guide, OMG Unified Modeling Language Specification v. 1.5, pp. 3-1–3-176, 2003.

[28] Papadopouli M. and Schulzrinne H., Effects of power conservation, wireless coverage and cooperation on data dissemination among mobile devices, Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing (MobiHoc 2001), Long Beach, California, October 4–5, 2001.

[29] Charles E. Perkins, editor. Ad Hoc Networking. Addison-Wesley, 2001.

[30] Persson, P. and Jung, Y. Nokia Sensor: From Research to Product, Conference on Designing for User eXperience (DUX), San Francisco, CA, Nov 3-5, 2005.

[31] Proxidating Application. `http://www.proxidating.com/`.

[32] RSS 2.0 Specification. `http://blogs.law.harvard.edu/tech/rss/`.

[33] Series 60 Platform. `http://www.series60.com/`.

[34] Tasker M., Dixon J., Shackman M., Richardson T. and Forrest J. Professional Symbian Programming: Mobile Solutions on the EPOC Platform. Wrox Press, 2000.

[35] Terziyan V., Collaborative Filtering, Lecture Notes, `http://www.cs.jyu.fi/ai/vagan/Collaborative_Filtering.ppt`.

[36] Volovikov O., Vapa M., Weber M., Kotilainen N. and Vuori J. Mobile Encounter Networks and Their Applications. The 17th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 2006), In submission, Sept. 2006.

[37] Wolfson O., Xu B. and Sistla P., An Economic Model for Resource Exchange in Mobile Peer to Peer Networks, The 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), 21–23 June 2004, Santorini Island, Greece.

[38] Xue G.-T., Li M.-L., Deng Q.-N. and You J.-Y., Stable Group Model in Mobile Peer-to-Peer Media Streaming System, The 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems, October 24–27, 2004, Fort Lauderdale, Florida, USA.

A Classes Functionality

As mentioned previously in Section 5.4, the application is split into two parts, the UI and the Engine. The UI part implements the standard set of Symbian OS application classes. Their details are omitted in this work due to space limit. Instead, attention is given to the functionality of the Engine part classes. The following table shows the header files in which the classes are defined:

Class	*Definition*	*Description*
`MGpcsObserver`	GpcsEngine.h	defined interface to observe engine state
`CGpcsEngine`	GpcsEngine.h	defined interface for engine
`CLocationList`	EngineUtil.h	defined interface for gas station location list
`TLocation`	EngineUtil.h	defined interface for gas station location
`CGasolineList`	EngineUtil.h	defined interface for gasoline attributes list
`TGasoline`	EngineUtil.h	defined interface for gasoline attributes
`MProfileObserver`	SettingsData.h	defined interface to observe profile state
`CProfileList`	SettingsData.h	defined interface for profile list
`CSettingsData`	SettingsData.h	defined interface for profile settings

Table A.1: GPCS Engine Classes

The classes as well as their attributes and methods are presented in this appendix in detail. The parameters and the return values of the methods are explained. The methods inside the boxes are written as they were defined in the header files of the software. The classes are expressed in UML notation [27]. Their names use the initial letter (`M`, `C`, `R` or `T`) that comes from the Symbian OS coding conventions [11, 34].

A.1 Class MGpcsObserver

The `MGpcsObserver` class defines an interface with one pure virtual function to implement an observer mechanism as depicted in Figure 5.9.

MGpcsObserver
+HandleUpdateL()

The class consists of nothing but the following method:

```
virtual void HandleUpdateL(TUint aIndex) = 0
```

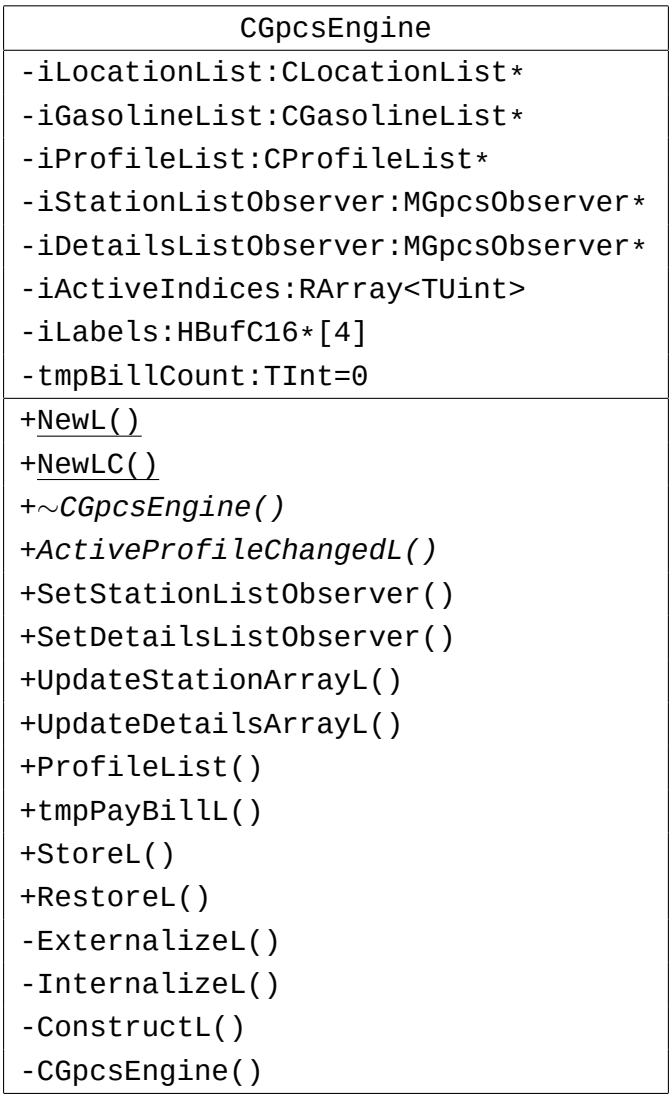
Notifies about modifications in the Engine state.

Parameters:

`aIndex` [in] the index number of the modified item.

A.2 Class CGpcsEngine

The `CGpcsEngine` class represents the Engine in the UI. It is derived from a standard Symbian OS class, `CBase`, a base class for all classes to be instantiated on the heap. It implements a `MProfileObserver` interface to observe the state of the active profile.



The class consists of the following attributes and methods:

- `iLocationList` is a pointer to the gas station location list;
- `iGasolineList` is a pointer to the gasoline attributes list;
- `iProfileList` is a pointer to the profile list;
- `iStationListObserver` is a pointer used to notify the Gas Stations View about modifications in the Engine state;
- `iDetailsListObserver` is a pointer used to notify the Gasoline Attributes View about modifications in the Engine state;
- `iActiveIndices` stores indices of the items to be shown to the user based on the active profile settings;
- `iLabels` stores the language-dependent labels used in the application;
- `tmpBillCount` is a counter used to emulate paying bills.

```
static CGpcsEngine* NewL()
```

Operates as a factory function – a static function that acts as a constructor.

Returns a pointer to the new object.

```
static CGpcsEngine* NewLC()
```

Operates like `NewL()` but does not pop the object from the cleanup stack before returning.

Returns a pointer to the new object.

```
virtual ~CGpcsEngine()
```

Destructor. Destroys the allocated memory.

```
virtual void ActiveProfileChangedL()
```

Updates `iActiveIndices` based on the active profile. This function implements the interface `MProfileObserver::ActiveProfileChangedL()`.

```
void SetStationListObserver(MGpcsObserver* aObserver)
```

Sets a new value for `iStationListObserver`.

Parameters:

`aObserver` [in] a new value of `iStationListObserver`.

```
void SetDetailsListObserver(MGpcsObserver* aObserver)
```

Sets a new value for `iDetailsListObserver`.

Parameters:

`aObserver` [in] a new value of `iDetailsListObserver`.

```
void UpdateStationArrayL(CDesCArray* aItemArray) const
```

Updates the Gas Stations View based on the new Engine state. It is typically called by the `CGpcsContainer` in the scope of the `HandleUpdateL()` method.

Parameters:

`aItemArray` [in, out] the array to be modified.

```
TInt UpdateDetailsArrayL(CDesCArray* aItemArray, TDes& aTitle,
   TUint& aTotal, TUint aIndex) const
```

Updates the Gasoline Attributes View based on the new Engine state. It is typically called by the `CGpcsContainer2` in the scope of the `HandleUpdateL()` method.

Parameters:

`aItemArray` [in, out] the array to be modified;

`aTitle` [out] the gas station location of the selected item used as the application title;

`aTotal` [out] the amount of the filtered items in the list;

`aIndex` [in] the index number of the selected item.

Returns the index number of *brand* of the selected item.

```
inline CProfileList& ProfileList()
```

Returns a reference to the member variable `iProfileList`.

```
void tmpPayBillL()
```

Emulates paying bills and delivers new gasoline attributes to the Engine state.

```
TStreamId StoreL(CStreamStore& aStore) const
```

Creates a stream within the supplied file. Calls the `ExternalizeL()` method of the class.

Parameters:

`aStore` [in, out] the stream store to which data is saved.

Returns the stream identifier.

```
void RestoreL(const CStreamStore& aStore, const TStreamId&
  aStreamId)
```

Creates and opens a stream using the supplied file and stream identifier. Calls the `InternalizeL()` method of the class.

Parameters:

`aStore` [in] the stream store containing data;

`aStreamId` [in] the stream identifier.

```
void ExternalizeL(RWriteStream& aStream) const
```

Writes the class state to the stream.

Parameters:

`aStream` [in, out] the stream to which the object is written.

```
void InternalizeL(RReadStream& aStream)
```

Reads the class state from the stream.

Parameters:

`aStream` [in, out] the stream from which the object is read.

```
void ConstructL()
```

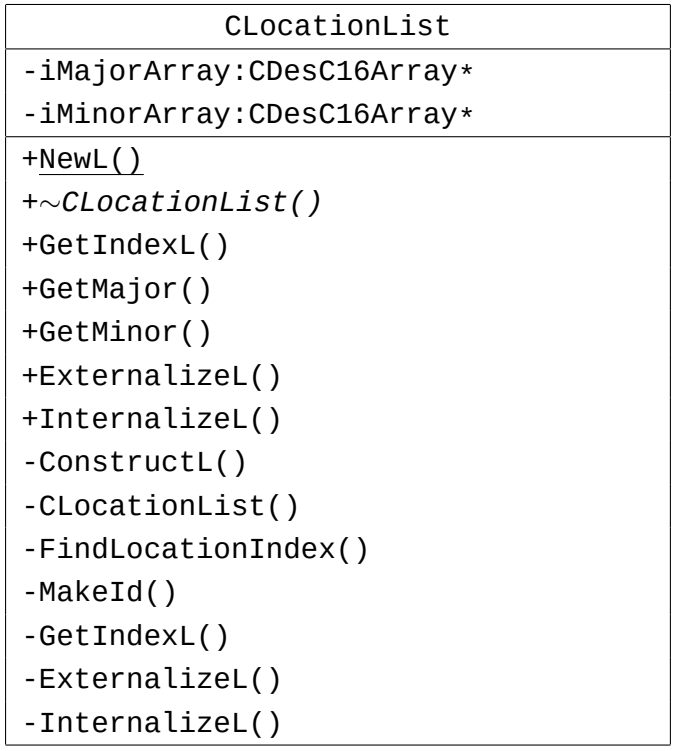
Second-phase constructor. Allocates memory required for the object.

```
CGpcsEngine()
```

Constructor. Creates the object and initiates member variables with default values.

A.3 Class CLocationList

The `CLocationList` class is responsible for manipulating data of the gas station locations. It is derived from the standard Symbian OS template class, `CArrayPtrFlat`, inheriting the behavior of flat arrays where `TLocation` is specified as a template argument.



The class consists of the following attributes and methods:

- `iMajorArray` stores the major parts of location;
- `iMajorArray` stores the minor parts of location.

```
static CLocationList* NewL(CCoeEnv& aCoeEnv)
```

Operates as a factory function – a static function that acts as a constructor.

Parameters:

`aCoeEnv` [in] the control environment.

Returns a pointer to the new object.

```
virtual ~CLocationList()
```

Destructor. Destroys the allocated memory.

```
TUint GetIndexL(const TDesC& aMajor, const TDesC& aMinor)
```

Gets an index number of the gas station location in the array with particular major and minor parts. If the item is not found, a new one is created.

Parameters:

`aMajor` [in] the major part of the gas station location;

`aMinor` [in] the minor part of the gas station location.

Returns an index number of the gas station location in the array.

```
const TPtrC GetMajor(TInt aIndex) const
```

Parameters:

`aIndex` [in] an index number of the gas station location in the array.

Returns a non-modifiable pointer descriptor to the major part of the gas station location.

```
const TPtrC GetMinor(TInt aIndex) const
```

Parameters:

`aIndex` [in] an index number of the gas station location in the array.

Returns a non-modifiable pointer descriptor to the minor part of the gas station location.

```
void ExternalizeL(RWriteStream& aStream) const
```

Writes the class state to the stream.

Parameters:

`aStream` [in, out] the stream to which the object is written.

```
void InternalizeL(RReadStream& aStream)
```

Reads the class state from the stream.

Parameters:

`aStream` [in, out] the stream from which the object is read.

```
void ConstructL(CCoeEnv& aCoeEnv)
```

Second-phase constructor. Allocates memory required for the object.

Parameters:

`aCoeEnv` [in] the control environment.

```
CLocationList()
```

Constructor. Creates the object and initiates member variables with default values.

```
TInt FindLocationIndex(TUint aId) const
```

Finds an index number of the gas station location in the array by location id. If the item is not found, returns -1.

Parameters:

`aId` [in] the gas station location id.

Returns an index number of the gas station location in the array.

```
TUint MakeId(const TDesC& aMajor, const TDesC& aMinor) const
```

Parameters:

`aMajor` [in] the major part of the gas station location;

`aMinor` [in] the minor part of the gas station location.

Returns the gas station location id.

```
TUint GetIndexL(CDesC16Array& aArray, const TDesC& aItem) const
```

Utility function. Used to find the particular item in the specified array. If the item is not found and not empty, it is appended to the end of the array.

Parameters:

`aArray` [in, out] the major part of the gas station location;

`aItem` [in] the minor part of the gas station location.

Returns an index number of the particular item in the specified array.

```
void ExternalizeL(const CDesC16Array& aArray, RWriteStream&
  aStream, TInt aFrom) const
```

Utility function. Writes the array state into the stream.

Parameters:

`aArray` [in] the array to be written into the stream;

`aStream` [in, out] the stream to which the array is written;

`aFrom` [in] the start position.

```
void InternalizeL(CDesC16Array& aArray, RReadStream& aStream,
   TInt aFrom)
```

Utility function. Reads the array state from the stream.

Parameters:

`aArray` [in, out] the array to be read from the stream;

`aStream` [in, out] the stream from which the array is read;

`aFrom` [in] the start position.

A.4 Class TLocation

The `TLocation` class stores data of a particular gas station location. `CLocationList` is declared as a friend class by `TLocation` so that `CLocationList` methods could have access to the private attributes of `TLocation`.

TLocation
-iLocationId:TUint -iMajorIndex:TUint -iMinorIndex:TUint
-ExternalizeL() -InternalizeL()

The class consists of the following attributes and methods:

- `iLocationId` the hash value obtained by hashing the major and minor parts of the location (called *location id*);
- `iMajorIndex` an index number of the location major part;
- `iMinorIndex` an index number of the location minor part.

```
void ExternalizeL(RWriteStream& aStream) const
```

Writes the class state to the stream.

Parameters:

`aStream` [in, out] the stream to which the object is written.

```
void InternalizeL(RReadStream& aStream)
```

Reads the class state from the stream.

Parameters:

`aStream` [in, out] the stream from which the object is read.

A.5 Class CGasolineList

The `CGasolineList` class is responsible for manipulating data of the gasoline attributes. It is derived from the standard Symbian OS template class, `CArrayPtrFlat`, inheriting the behavior of flat arrays where `TGasoline` is specified as a template argument.

The class consists of the following attributes and methods:

- `iTypeArray` stores the gasoline type alternatives;
- `iBrandArray` stores the gasoline brand alternatives;
- `iNextTime` stores the current base time;
- `iRangeId` stores the current range id.

```
static CGasolineList* NewL(CCoeEnv& aCoeEnv, const TTime&
   aCurTime)
```

Operates as a factory function – a static function that acts as a constructor.

Parameters:

`aCoeEnv` [in] the control environment;

`aCurTime` [in] the current time.

Returns a pointer to the new object.

```
virtual ~CGasolineList()
```

Destructor. Destroys the allocated memory.

```
TUint AddL(TUint aIndex, const TDesC& aType, const TDesC& aBrand,
   const TDesC& aPrice)
```

Adds new gasoline attributes. If the item already exists, it will be overwritten by a new price (and time).

Parameters:

`aIndex` [in] an index number of the gas station location in the member variable `CGpcsEngine::iLocationList`;

`aType` [in] a non-modifiable descriptor to the gasoline type;

`aBrand` [in] a non-modifiable descriptor to the gasoline brand;

`aPrice` [in] a non-modifiable descriptor to the gasoline price.

Returns an index number of the gasoline attributes in the array.

```
void GetTime(const TGasoline& aGasoline, TDes& aTime) const
```

Renders a time value as text.

Parameters:

`aGasoline` [in] an object that contains the time value as number;

`aTime` [out] on return, contains the time value as text.

```
void GetPrice(const TGasoline& aGasoline, TDes& aPrice) const
```

Renders a price value as text.

Parameters:

`aGasoline` [in] an object that contains the price value as number;

`aPrice` [out] on return, contains the price value as text.

```
const TPtrC GetType(const TGasoline& aGasoline) const
```

Gets a non-modifiable pointer descriptor to the gasoline type.

Parameters:

`aGasoline` [in] an object that contains the index number of the gasoline type.

Returns a non-modifiable descriptor to the gasoline type.

```
const TPtrC GetBrand(const TGasoline& aGasoline) const
```

Gets a non-modifiable pointer descriptor to the gasoline brand.

Parameters:

`aGasoline` [in] an object that contains the index number of the gasoline brand.

Returns a non-modifiable descriptor to the gasoline brand.

```
void ExternalizeL(RWriteStream& aStream) const
```

Writes the class state to the stream.

Parameters:

`aStream` [in, out] the stream to which the object is written.

```
void InternalizeL(RReadStream& aStream)
```

Reads the class state from the stream.

Parameters:

`aStream` [in, out] the stream from which the object is read.

```
void ConstructL(CCoeEnv& aCoeEnv)
```

Second-phase constructor. Allocates memory required for the object.

Parameters:

`aCoeEnv` [in] the control environment.

```
CGasolineList(const TTime& aCurTime)
```

Constructor. Creates the object and initiates member variables with default values.

Parameters:

`aCurTime` [in] the current time.

```
TUint FindIndex(CDesC16Array* aArray, const TDesC& aItem) const
```

Utility function. Finds an index number of the item in the array. If the item is not found, it returns zero, which means the item is unknown.

Parameters:

`aArray` [in] an array where to find;

`aItem` [in] an item to be found.

Returns an index number of the item in the array.

```
TUint GetPrice(const TDesC& aPrice) const
```

Converts a price value presented as text into number.

Parameters:

`aPrice` [in] a descriptor to a price value.

Returns a price value as number.

```
TInt Now() const
```

Returns the number of minutes passed since the base time.

A.6 Class TGasoline

The `TGasoline` class stores data of particular gasoline attributes. `CGasolineList` and `CGpcsEngine` are declared as friend classes by `TGasoline` so that their methods could have access to the private attributes of `TGasoline`.

TGasoline
-iGasolineId:TUint32 -iGasolineVal:TUint32
-ExternalizeL() -InternalizeL()

The class consists of the following attributes and methods:

- `iGasolineId` gasoline id that consists of *location*, *brand* and *type* due to a binary structure;
- `iGasolineVal` gasoline value that consists of *time* and *price* due to a binary structure.

```
void ExternalizeL(RWriteStream& aStream) const
```

Writes the class state to the stream.

Parameters:

`aStream` [in, out] the stream to which the object is written.

```
void InternalizeL(RReadStream& aStream)
```

Reads the class state from the stream.

Parameters:

`aStream` [in, out] the stream from which the object is read.

A.7 Class MProfileObserver

The `MProfileObserver` class defines an interface with one pure virtual function to implement an observer mechanism.

MProfileObserver
+ActiveProfileChangedL()

The class consists of nothing but the following method:

```
virtual void ActiveProfileChangedL() = 0
```

Notifies about modifications in the active profile.

A.8 Class CProfileList

The `CProfileList` class is responsible for manipulating data of the user-defined profiles. It is derived from a standard Symbian OS template class, `CArrayPtrFlat`, inheriting the behavior of flat arrays where `CSettingsData` is specified as a template argument.

CProfileList
-iObserver:MProfileObserver& -iDefault:HBufC* -iCurrent:TInt -iActive:TInt -iCount:TInt -iActiveChanged:TBool
+NewL() +~*CProfileList()* +ExternalizeL() +InternalizeL() +ChangedNotifyL() +AppendNewL() +Delete() +CurrentProfileChanged() +Current() +Active() +SetCurrent() +SetCurrentNext() +SetCurrentPrev() +SetActive() +Count() +GetCurrent() +GetActive() *-MdcaCount()* *-MdcaPoint()* -ConstructL() -CProfileList()

The class consists of the following attributes and methods:

- `iObserver` is a reference used to notify about modifications in the active profile. This is set by the constructor;

- `iDefault` stores the language-dependent default profile name;
- `iCurrent` stores the index number of the selected profile;
- `iActive` stores the index number of the activated profile;
- `iCount` stores the number of user-defined profiles;
- `iActiveChanged` is a flag used to check if the active profile has been changed.

```
static CProfileList* NewL(CCoeEnv& aCoeEnv, MProfileObserver&
   aObserver)
```

Operates as a factory function – a static function that acts as a constructor.

Parameters:

`aCoeEnv` [in] the control environment;

`aObserver` [in] the value of `iObserver`.

Returns a pointer to the new object.

```
virtual ~CProfileList()
```

Destructor. Destroys the allocated memory.

```
void ExternalizeL(RWriteStream& aStream) const
```

Writes the class state to the stream.

Parameters:

`aStream` [in, out] the stream to which the object is written.

```
void InternalizeL(RReadStream& aStream)
```

Reads the class state from the stream.

Parameters:

`aStream` [in, out] the stream from which the object is read.

```
void ChangedNotifyL()
```

If `iActiveChanged` equals to `ETrue` then unsets it and notifies the observer that the active profile has been changed.

```
void AppendNewL()
```

Appends a new profile onto the end of the array.

```
TBool Delete(TInt aIndex)
```

Deletes profiles by position.

Parameters:

`aIndex` [in] the position within the array.

Returns `ETrue` if the specified profile is not active, otherwise EFalse.

```
inline void CurrentProfileChanged()
```

Sets `iActiveChanged` to `ETrue` if `iCurrent` equals `iActive`.

```
inline CSettingsData& Current()
```

Returns the current profile.

```
inline CSettingsData& Active()
```

Returns the active profile.

```
inline void SetCurrent(TInt aIndex)
```

Sets the current profile by position.

Parameters:

`aIndex` [in] the position within the array.

```
inline void SetCurrentNext()
```

Sets the current profile to the next.

```
inline void SetCurrentPrev()
```

Sets the current profile to the previous.

```
inline void SetActive(TInt aIndex)
```

Activates profiles by position and sets `iActiveChanged` to `ETrue`.

Parameters:

`aIndex` [in] the position within the array.

```
inline TInt Count() const
```

Returns the value of `iCount`.

```
inline TInt GetCurrent() const
```

Returns the value of `iCurrent`.

```
inline TInt GetActive() const
```

Returns the value of `iActive`.

```
virtual TInt MdcaCount() const
```

Returns the number of user-defined profiles in the array. The function implements the interface `MDesCArray::MdcaCount()`.

Returns the number of user defined profiles.

```
virtual TPtrC MdcaPoint(TInt aIndex) const
```

Indexes into the array of user-defined profiles. The function implements the interface `MDesCArray::MdcaPoint()`.

Parameters:

`aIndex` [in] The position of the profile within the array. The position is relative to zero.

Returns a non-modifiable pointer descriptor representing the descriptor element located at position `aIndex` within the array.

```
void ConstructL(CCoeEnv& aCoeEnv)
```

Second-phase constructor. Allocates memory required for the object.

Parameters:

`aCoeEnv` [in] the control environment.

```
CProfileList(TInt aGranularity, MProfileObserver &aObserver)
```

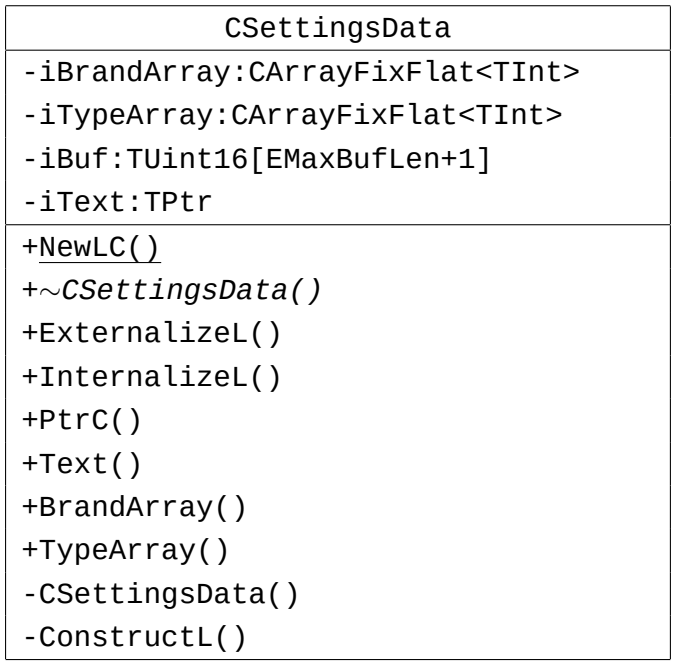
Constructor. Creates the object and initiates member variables with default values.

Parameters:

`aGranularity`	[in] the granularity of the array;
`aObserver`	[in] the value of `iObserver`.

A.9 Class CSettingsData

The `CSettingsData` class stores settings of a particular profile. It is derived directly from the `CBase` class.



The class consists of the following attributes and methods:

- `iBrandArray` stores gasoline brand alternatives of the profile;
- `iTypeArray` stores gasoline type alternatives of the profile;
- `iBuf` stores the profile name with a column separator;
- `iText` is a modifiable pointer descriptor to the profile name.

```
static CSettingsData* NewLC(const TDesC& aDefault = KNullDesC)
```

Operates as a factory function – a static function that acts as a constructor.

Parameters:

`aDefault` [in] the default profile name.

Returns a pointer to the new object.

```
virtual ~CSettingsData()
```

Destructor. Destroys the allocated memory.

```
void ExternalizeL(RWriteStream& aStream) const
```

Writes the class state to the stream.

Parameters:

`aStream` [in, out] the stream to which the object is written.

```
void InternalizeL(RReadStream& aStream)
```

Reads the class state from the stream.

Parameters:

`aStream` [in, out] the stream from which the object is read.

```
inline TPtrC PtrC() const
```

Returns a non-modifiable pointer descriptor to the buffer which contains a profile name with the column separator.

```
inline TPtr& Text()
```

Returns a modifiable pointer descriptor to the buffer which contains a profile name.

```
inline CArrayFix<TInt>& BrandArray()
```

Returns gasoline brand alternatives of the profile.

```
inline CArrayFix<TInt>& TypeArray()
```

Returns gasoline type alternatives of the profile.

```
CSettingsData()
```

Constructor. Creates the object and initiates member variables with default values.

```
void ConstructL(const TDesC& aDefault)
```

Second-phase constructor. Allocates memory required for the object.

Parameters:

`aDefault` [in] the default profile name.