# Chapter – 8
# Designing, Deploying & Modeling MoBlog over IMS

# Chapter – 8
# Designing, Deploying & Modeling MoBlog over IMS

IMS Service deployment and delivery support always encourage user to create more and more service. Although IMS 2.0 able to work with Web 2.0 over IMS Network. During research survey phase, new innovative service needed to deploy for observe IMS open service provision nature, with the conception there are lot more application observed including Location based services, Games, Network Games as well. But finally Mobile based blogging selected for considering IMS Service provision and rendering over User equipment phase.

The MoBlog does not automatically support either self-presentation or intercommunication of the participants. Instead suggest that functions of the MoBlog alter situational while participants engage themselves in different ways and levels to the participative actions and processes of the MoBlog. The participants simultaneously manage multiple ways of being present and display multiple levels of presence within practices of distributing pictures, seeing them or interacting by writing of them. In what follows suggest that MoBlog may serve as a tool for storing, publishing, sharing or communication or all of those together depending on the situational varying activities in which participants actively engage themselves.

With a simple aim to make prototype application for IMS MoBlog service aim to create with J2ME and supportive tools and planning to deploy over own IMS light size Client through provision of this MoBlog through Open IMS Core Test bed setup.

## 8.1 Introduction to Blogging

Web logs, or blogs, are a rather young, popular phenomenon on the Internet. They turn into places where people can express their ideas, opinions and emotions. It is an open forum for everyone who wishes to participate. People writing about the same subjects can be grouped together, so it is easier to read about the subjects you are interested in. Expressing what is on your mind is nice, but receiving feedback makes it a whole lot more [81]. Communities are created this way; friends can reply to each other, people can develop new views on things, and all that by writing and receiving feedback.

### 8.1.1 Blogs – a space for participation and communication

One basic curiosity of the blog and Blogging is that it is highly self-referential medium. Large amount of the blogs with their respective topics, are practical and theoretical issues of the Blogging culture. In addition, blog researchers tend to have a blog of their own: theory is connected to practice. In [185] assess the way that the scientists who do not have experiences of Blogging often treat blogs as a homogenous mass and do not recognize variations between them. Neither of the writers of this paper has experiences of writing a blog, though gained a good experience in viewing them.

In order to avoid giving too simplistic view of the phenomenon under inspection approach the nature of the MoBlogging from three different angles combining discussion and results of the research on MoBlogs, on weblogs and on mobile technologies, as part of this research aimed at the early stage to propose MoBlog over Next Generation Network enable IP Multimedia Subsystem.

When Justin Hall [186] first introduced the idea of a MoBlog he contrasted it with weblog. He considered a weblog to be a record of travels on the Web, whereas a MoBlog for him was a record of travels in the world. Interestingly, Julian Gallo [187] brief that after sending pictures to his MoBlog he experienced that he is making neither a photo album nor a web log, but a visual map with the data of where he has been and what he has seen.

Every form of the web communication has characteristics of its own. While blog communication on the whole obscures the ideas of private and public, individual and group and ideas of fact and fiction. This proposed work “The MoBlog”, in turn, enlarges the idea of the shared instant experience. The very characteristic of the MoBlogging is instantaneous, since it provides a place and possibility to send personal views and flashes of one’s instant moments in a world around him and share these experiences by communicating with other people.

### 8.1.2 Basics about MoBlog

In principal, MoBlog works as a medium for personal publishing or for communication and creation of social relations and ties. Like homepages, the MoBlog serves as a channel of self-performance providing media consumers with the possibility of becoming media producers themselves. The MoBlog’s technological possibilities lean on its affordances to save and distribute author’s life story as pictures (and as text). It not only affords possibility for self presentation and self identification, while displaying author’s mundane life, his/her instant experiences and everyday items of the immediate environment, but it also provides channel for communication with others. Although the interconnectedness and interpersonal communication within the web community does not always emerge unaided. The simple “seen-snapped-posted” –publishing structure is not enough in order to catch the audience’s whole attention, but the MoBlog site may need less aggressive promotion in order to be noticed. [185]

The MoBlogging requires not only access to the Internet for photo sharing purposes but also the device with the help of which the personal views of the instant environment and moments can be saved. Going further to examine the camera phone use and multimedia messaging some interesting observations have been made in the research areas of sharing digital images. In their experiment [188] found out that multimedia messaging (MMS) between friends is not working as independent sequence of interaction, but is likely to be related to the previous interaction of them. In this study the posted messages had various different contents, such as postcard postings, rumors, stories, jokes, teasing, failure snaps and requests to have others' pictures. In [189] research results echoes with this since they reported that image-contained MMS messages were tend to be used as a tool for creating a story or a joke, for expressing emotions or even for making art around them.

What comes to the typical patterns of using MMS messages and practices around camera phone image sharing at least following observations have been made. In the research of Kindberg [190] the interest was to analyze what people photograph with mobile phones and how they use the images. The images were used both for sharing and for personal use, and for affective reasons and functional use. Based on users' intentions behind the captures the researchers identified six subcategories of the picture use. The affective functions contained enriching a shared experience, communicating with an absent friend or family or personal reflection or reminiscing. Functional intentions behind the image use included supporting a mutual task with people co-present, supporting a task with remote people or supporting a personal, practical task. In [190] also concluded that the capture and send culture of the camera phone pictures has collided with practical and technological barriers and people are more likely to use mobile devices for capture and show purposes.

Similarly Daisuke [191] noted in his ethnographic study of camera phone usage in Tokyo that users do not prefer to email images to one another but they are rather likely to share them with others showing them right from the handset screen. In the same research Okabe also came to conclusion that camera phone actually has various different uses including personal picture archiving, intimate picture sharing with other people, peer-to-peer news reporting and online picture sharing.

It seems that while talking about the image capturing and sharing them with the help of mobile devices the patterns of use tend to vary a lot. In [191] seems to agree as he points in his study that the use of the camera phone is still emergent practice since the patterns of use have not yet totally stabilized. People are still working out the social protocols and norms for appropriate visual information sharing. Moreover, if look back to the culture of MoBlogging it seems that dealing with a rather inchoate phenomenon, which by no means has made any breakthrough in Internet users' daily practices. With the research in [185] estimate that in a context of the whole BlogoSphere MoBlogs are just “a niche within the niche” and it may be assumed that in the long run only minority of the Internet and mobile users ever start a MoBlog of they own. It may be that e-mail attachment and MMS mobile phone messages are still the most popular forms of interpersonal visual communication, although online photo albums, mobile blogs and photoblogs may increase further interest of those users who actively search applications for digital photo sharing.

## 8.2 MoBlog Project

For this research, examine closely the practical functions of MoBlogs as a media of self-presentation and intercommunication of the participants over IP Multimedia Subsystem, deployed Open IMS Core with own IMS Client. A detailed analysis of the structure and the content of MoBlog contributions have not been conducted earlier.

### 8.2.1 Functions of the MoBlog

Within this study interested in the way people choose, adapt and manage different participation and communication practices in the context of one virtual and visual medium, MoBlog. While considered particular constraints and affordances of this one communication channel came up with two analytical concepts: attractiveness and responsiveness.

These concepts are imposed here to examine differences in MoBlog uses and to show how the functions of the MoBlog alter in terms of the different kind of actions of the author and the possible visitors. During the study attractiveness of the MoBlog was measured in terms of the statistics of the viewed pictures and responsiveness in terms of the statistics of the added comments. While paying attention to authors' and visitors' actions in the situated contexts of MoBlog practices came up with the following categories presented in table 8.1:

Table 8.1. Functions of MoBlogs

| | **Non-responsive** | **Responsive** |
|---|---|---|
| Non-attractive | Store | Share |
| Characteristic of use | No views, no comments | Few views, some comments |
| Attractive | Publish | Communicate |
| Characteristic of use | Lot of views, no comments | Lot of views and comments |

In the first category of MoBlogs both attractiveness and the responsiveness of the blog were recorded to be minimal or total null i.e. the MoBlog did not gain any viewers or commentators. Contrasting to the weak reception of the images by the part of the web community the MoBlogging was contributed to be a type of Store. Where some views and comments were to be recorded the MoBlog function turned to Share kind of a Blogging. In this category a rather small community of people communicated around the published pictures. When MoBlog and its picture gallery seemingly attracted a mass of audience to view pictures, MoBlogging worked rather as a forum of publishing. Though, in this Publish -category pictures didn't seem to launch any interaction between the participants. In the last category, communicate MoBlogging, both attractiveness and responsiveness were measured to be high on the grounds of the viewed and commented picture entries.

The results of this part of the study suggest that MoBlog's functions alter situational while authors and visitors engage themselves in modifying their participating levels, whether by viewing or commenting or by viewing and commenting. It is worth to notice that one single MoBlog does not represent a one single category as pure, but the functions of a MoBlog may vary in the course of the time depending on how the web audience welcomes the MoBlog and how they take part to the participative processes of it. At one time a MoBlog may have viewers as well as commentators, but in the next moment it may not attract even viewers. To better understand the variations between different kinds of practices within MoBlogging now show more closely some examples of the each category.

### 8.2.2 MoBlog hierarchy

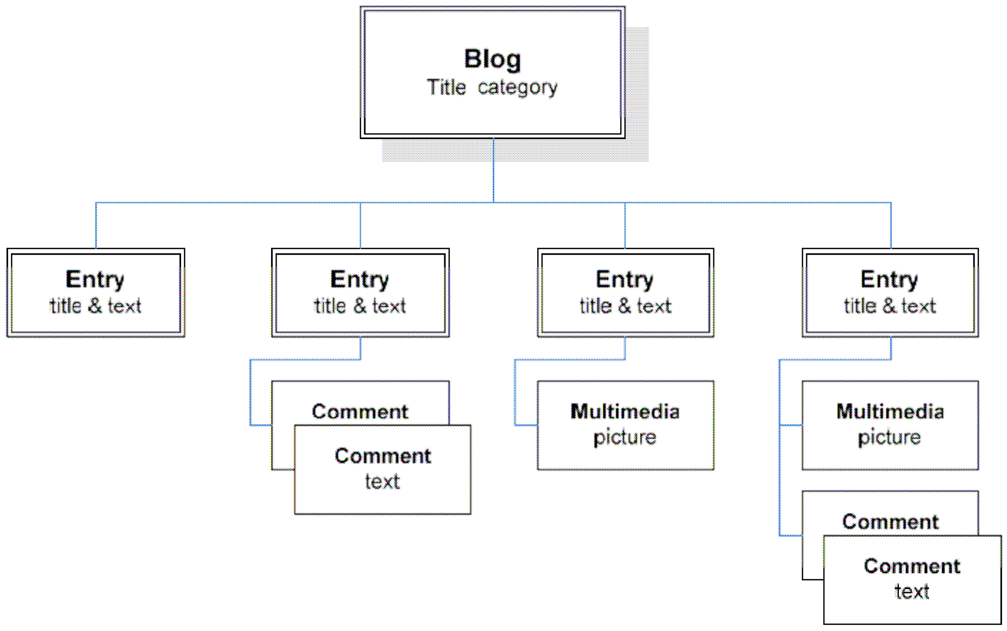


Figure 8.1: Diagram with MoBlog hierarchy

There is a strict hierarchy in this blogs, which is depicted above. The figure shows all the possible combinations that can be made with entries, multimedia and comments. Each block represents an element, with the name of the element on top and the most important properties of the element below. Appendix I demonstrates Global Diagram of MoBlog Service.

#### 8.2.2.1 Blog

A user is entitled to a single blog, which has a title and a category to describe it. Every user can specify the title of their blog and the category at the time of creation. The title can be anything, and is often a short description of what the blog is about. The categories are fixed, and a choice can be made from Sport, IT, Leisure and Other.

#### 8.2.2.2 Entry

Entries are the heart of the blog and are described by a topic and text. Entries are what the user wants to tell to the world. The topic can be a short description of what the entry is about. Only the owner of the blog can create entries on his blog, edit or delete them. Other users can only read these entries.

#### 8.2.2.3 Multimedia

Owners can attach multimedia content to their entries. Every entry is limited to a single piece of multimedia; currently MoBlog only supports a picture. It is possible to expand this to audio recordings and even video recordings but these are not implemented at this time.

#### 8.2.2.4 Comment

The topic of a comment is automatically generated from the text. Every user has the ability to post comments on the entries of other blogs, and every user can read all the comments posted. These comments are the reactions of the general public to what is written in an entry.

## 8.3 Study of various MoBlog stage

### 8.3.1 Capture and store

In the first phase of MoBlog, this works to Store -MoBlogging. In this category of MoBlogs the content is based on a kind of random snapping and random picture gallery exposition. The photos do not seem to have obvious relation to one another, but they all represent kind of momentary flashes of author’s everyday life and mundane instant environment. The meaning and the purpose of the pictures do not open very clearly to viewers. There is no plot, logic structure or visual narrative that distinctly relates these snap-shots as a “family of images”. The only continuity between the picture entries may be found in the time span of the photographing as the photos have been dated to the sequential days.

### 8.3.2 Capture and share

In the second phase of MoBlogs, share, the visual content of the blog produces some views and some comments among the MoBlog visitors. It is likely that the mobile picture gallery is collected around a specific topic or images are in other ways related to each other. The content may be constructed for example with family photos or photos of pets and therefore the blog is more likely to spur acquaintances, family or friends or small circle of the Web readers to view and comment the entries.

### 8.3.3 Capture and publish

In the third category of MoBlogs, publish, the blog and the participative processes around it allow the author him/herself to become a publisher. Among the web community publish MoBlogging attracts people’s interest in viewing with glossy advertising style pictures, pictorial news reports or images that in other ways draw people’s attention. Thus, publish – type of MoBlog has a character of personal soapbox or professional journalistic gallery. Online moblog publishing offers certain opportunities for individual publishers but also involves some risks. Mielo states [193] that MoBlog has actually become a medium of choice to the journalists in reporting about wars, riots and other newsworthy crises around the world because of the medium’s particular characteristics: it is portable, uncomplicated and instantaneous.

In the publish phase of MoBlogging images may be found rather exhibitionistic. The author of the MoBlog is engaged in personal impression management by creating a persona of celebrity and publicity with qualifications of good appearance, outfit and faultless condition. She is not only exposing her own body in her personal blog but brings forward her boyfriend with images of his trained body. The content of the pictures is emphasized by author’s positive assessment about his boyfriend’s looks. The MoBlogger’s main interest and concern seems to be, how to appeal to the web audience.

Without going any further in cultural analysis of how and why some MoBloggers, women as well as men, are willing to present their bodies and sexualities openly in Web, the Blogging culture in overall contains the possibility of managing and controlling one’s self-presentation and personal impression. Reed [194] states in his study that research subjects noted repeatedly that weblogging gave them pleasure of exposing them and their life in public and moreover to totally strangers. On the other hand, along with writing personal blog people came to realize that exposing oneself may be harmful, since the Web records and saves the data in accumulative way and also because blog contents are always subject to the readers’ misinterpretations. The publish type of MoBlogging may also cause other types of negative results and responses within web community.

### 8.3.4 Capture and communicate

In contrast to the previously described types of MoBlogging the last category communicate is not only featured by the attractiveness but also by the responsiveness of the content as it enhances communication among the MoBlog community. Such elements of MoBlog that may reinforce the interpersonal communication.

## 8.4 Displaying a Graphical User Interface with J2ME

Creating a Graphical User Interface (GUI) is a difficult thing to do with J2SE. This is because the screen itself is rather large, and there are a lot of options in the available API. Under J2ME creating a GUI is much easier because of device limitations and the smaller API. There are two approaches to handle UI: the MIDP high-level API and the MIDP low-level API. The MIDP high-level API has a lower granularity that is provided to the programmer, but makes abstraction of low-level design issues.

When using the low-level API, one must control every behavior of every component that is displayed at all time. Because this is not easy to do, chosen to use the high-level API, especially the Form and the List class.

### 8.4.1 Using a Form

A Form is type of screen that can hold different types of components. It is can be used for input and output at the same time.

The creation of the Form object is done the same way as every object creation in Java, with the constructor. Once created one can easily put components on the Form using the append method.

```
Form screen = new Form("title");
screen.append("Appending a String");
TextField text = new TextField("TextField title", "textfield
     contents", 50,
TextField.SENSITIVE); screen.append(text) ;
```

*Code 8.1: Creating and using a Form*

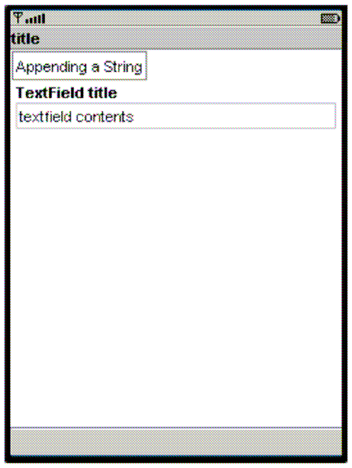


*Figure 8.2: The result of Code 8.1*

### 8.4.2 Using a List

When one wants to display a list, the Form class can be used but it is not recommended. Instead, one can use the List class. It is designed to display a list. The addition of elements to the list is a very easy process to do. It is just the calling of the append method.

```
List screen = new List("title", List.IMPLICIT);

screen.append("first item", null); screen.append("second
     item", null);
```

*Code 8.2 Creating and Using List*

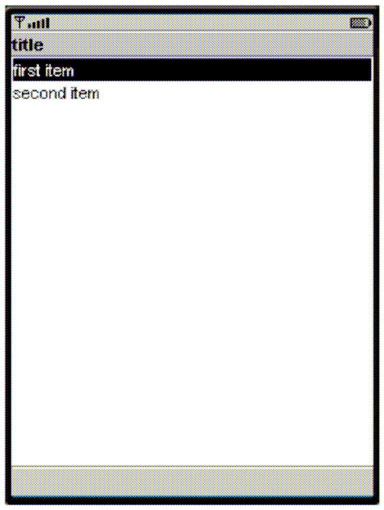


*Figure 8.3: The result of Code 8.2*

### 8.4.3 Adding Buttons

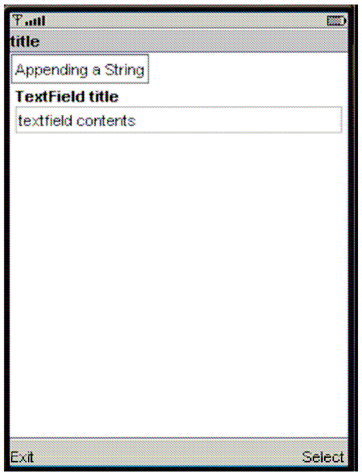
Because the lack of a mouse, one has to place buttons on the screen. Under J2ME the Buttons are called Commands. Multiple buttons can be placed on a particular screen. Each button can have a different function. This way one can interact with the device. The placement of the buttons is manufacturer dependant. If the device has its exit buttons on the left, the device will force an exit button on the left side. This cannot be overridden. It is this mechanism that makes the buttons place in a non wanted order. Sun is aware of this problem, but doesn't give an answer to it.

The code that is displayed uses the Form code used in Code 8.3: The use of Commands.

```
Command exit = new Command("Exit", Command.EXIT, 1); Command
     select = new Command("Select", Command.OK,i);


screen.addCommand(exit); screen.addCommand(select);
```

*Code 8.3: The use of Commands*


title

Appending a String

TextField title

textfield contents

*Figure 8.4: The result of Code 8.3*

### 8.4.4 Starting Screen for Moblog

```
public Form getForm() {
        if (form == null) {
            // write pre-init user code here
            form = new Form("Moblog", new Item[] {
getStringItem() });
            form.addCommand(getExitCommand());
            form.addCommand(getOkCommand());
            form.setCommandListener(this);
            // write post-init user code here
        }
        return form;
    }
```

Code 8.5 Welcome Screen

Codes gives us Result like below (with Mobile device dependency)

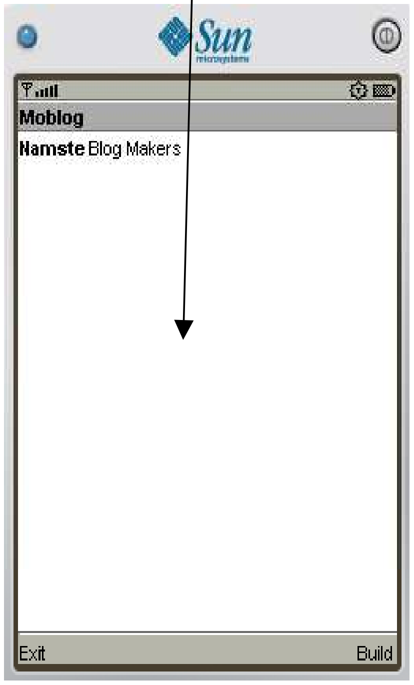


*Figure 8.5: The result of Code 8.5*

### 8.4.5 File Browser windows for Moblog

```
public FileBrowser getFileBrowser() {
        if (fileBrowser == null) {
            // write pre-init user code here
            fileBrowser = new FileBrowser(getDisplay());
            fileBrowser.setTitle("fileBrowser");
            fileBrowser.setCommandListener(this);

fileBrowser.addCommand(FileBrowser.SELECT_FILE_COMMAND);
            fileBrowser.addCommand(getBackCommand());
            fileBrowser.addCommand(getOkCommand1());
            // write post-init user code here
        }
        return fileBrowser;
```

Code 8.6 File Browser

Codes below gives us Result like below (with Mobile device dependency)

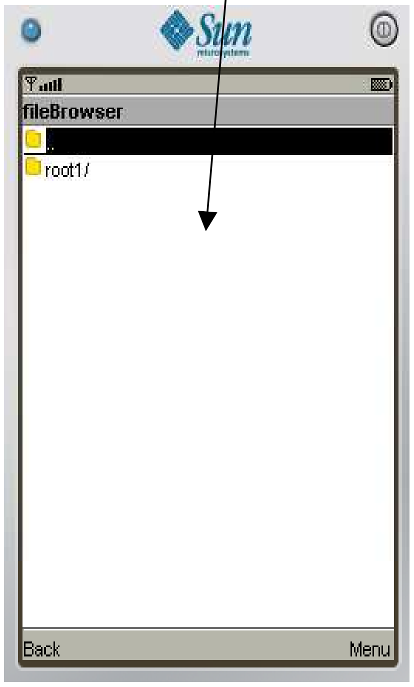


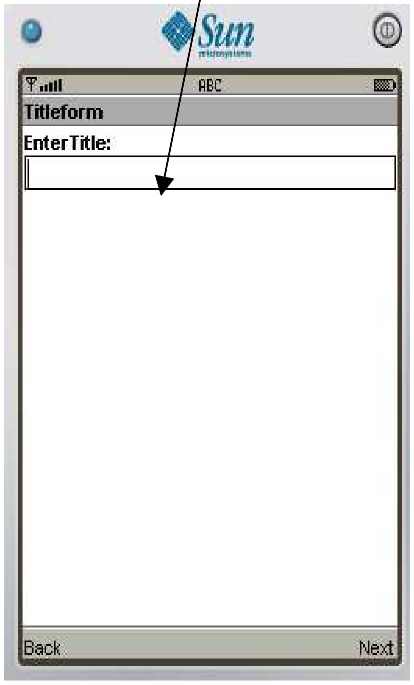
*Figure 8.6: The result of Code 8.6*

### 8.4.6 MoBlog Title window

```
  public Form getForm1() {
        if (form1 == null) {
            // write pre-init user code here
            form1 = new Form("Titleform", new Item[] {
getTextField() });
            form1.addCommand(getBackCommand1());
            form1.addCommand(getOkCommand2());
            form1.setCommandListener(this);
            // write post-init user code here
        }
        return form1;
    }
```

Code 8.7 for Blog Title Window

Code for next form concern for Title for MoBlog.



Titleform

EnterTitle:

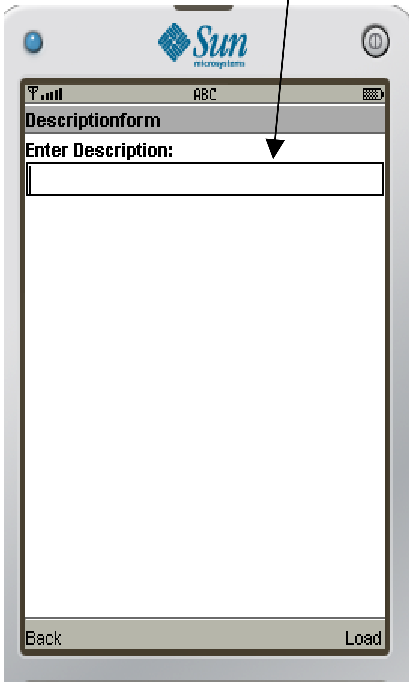
Figure 8.7: The result of Code 8.7

### 8.4.7 MoBlog Description windows

```
    public Form getForm2() {
        if (form2 == null) {
            // write pre-init user code here
            form2 = new Form("Descriptionform", new
Item[] { getTextField1() });
            form2.addCommand(getBackCommand2());
            form2.addCommand(getOkCommand3());
            form2.setCommandListener(this);
            // write post-init user code here
        }
        return form2;
    }
```

Code 8.8 Blog Description window

Codes below used for getting Description from MoBlog Result like below (with Mobile device dependency)



Descriptionform

Enter Description:

Figure 8.8 The result of Code 8.8

### 8.4.8 MoBlog Loading Windows

```
 public SplashScreen getSplashScreen() {
        if (splashScreen == null) {
            // write pre-init user code here
            splashScreen = new
SplashScreen(getDisplay());
            splashScreen.setTitle("Loading");
            splashScreen.setCommandListener(this);
            splashScreen.setFullScreenMode(true);
            splashScreen.setImage(getImage1());
            splashScreen.setText("Loading Blog");
            splashScreen.setTimeout(5000);
          }
        return splashScreen;
    }
```

Code 8.9 Blog Loading window

)

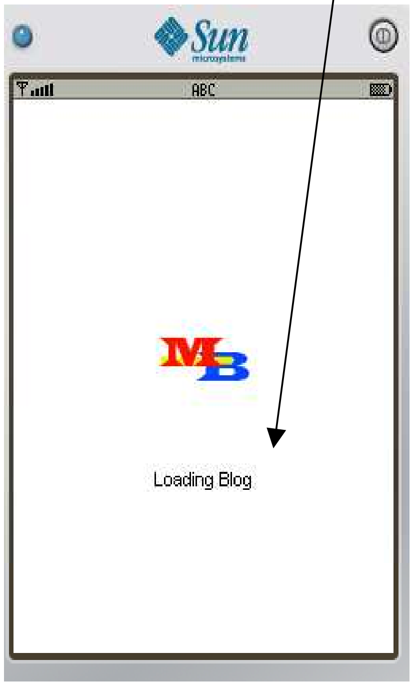
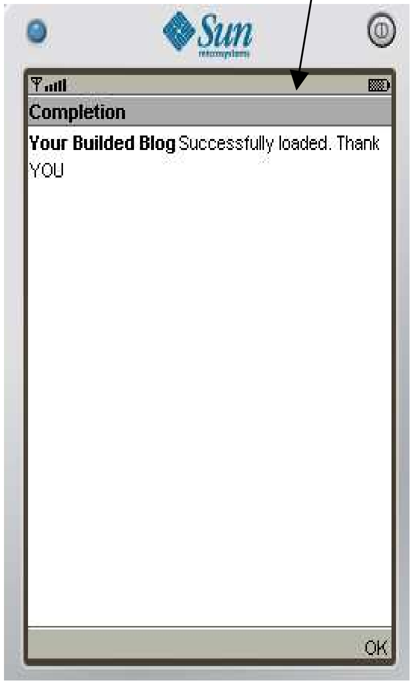


Figure 8.9 The result of Code 8.9

### 8.4.9 MoBlog Completion Windows

```
    public Form getForm3() {
        if (form3 == null) {
            // write pre-init user code here
            form3 = new Form("Completion", new Item[] {
getStringItem1() });
            form3.addCommand(getOkCommand4());
            form3.setCommandListener(this);
            // write post-init user code here
        }
        return form3;
    }
```

Code 8.10 MoBlog Completion window



**Your Builded Blog** Successfully loaded. Thank YOU

Figure 8.10 The result of Code 8.10

### 8.4.10 Advantages of MoBlog

As shown, the creation of a GUI is very easy, mostly because of the device limitations.

- The small screen limits us in displaying components.
- This gives the programmer the advantage to create a to the point interface.
- Because of these limitations the programmer doesn't have the urge, and the tools, to build a very large and therefore complicated UI.
- Because there is no urge to create a large UI, the programmer has more time to create the business logic.

## 8.5 Integrated Development Environment

There are many Integrated Development Environments (IDE) available in the market. This project design chosen to use the free IDE Eclipse [195] to develop the Java source code. Eclipse provides us with an easy to use environment. There are many plug-ins available, which makes it a very flexible IDE.

Eclipse is a Java based application that runs on J2SE. By default the compilation is done on the same Java edition as where the IDE runs on. In order to compile this source code to use the J2ME compiler. This can be done manually using the command line, which is rather complex, and therefore Ant [196] was introduced. Ant is a building tool much like make [197] for C but it is XML based. It compiles the source according to a build file. This build file can contain much more than just the lines to compile the source. It can contain commands to modify the file system. This way the source code can be compiled into a specific directory. The build file can also contain commands to jar the compiled files together. In present work, this to make the MIDlet suite.

Ant [196] is commonly used in many Java projects as build system. This is another advantage for using Ant. To do the Unified Modeling Language (UML) design, chosen to use ArgoUML [192]. It provides the user with an easy to use interface. One of the advantages of ArgoUML is that it automatically creates source code.

## 8.6 Reading pictures from the picture directory

Devices equipped with a camera also include software to take pictures. These pictures are stored on the device in a particular directory. Adding one of those pictures to an entry seemed to be a good idea. One could take several pictures and write the entries afterwards, adding the pictures from the device.

J2ME does not provide a framework for file access. This complicated the problem quite a bit for this framework. After some research, only one possible solution was found which would discuss in more detail. The four steps used to read files from the device were:

1. Open a socket to the local file system
2. Point the socket to the correct directory
3. Read an array of bytes specified by the filename from the socket
4. Close the socket.

This approach worked quite well for a fixed file name, which removed the option of keeping multiple pictures for later use and made the user responsible of storing a picture with the correct filename. To remove this constraint, a small directory browser needed to be created. This would allow a user to scroll through the directories and files, and load the desired picture. A Java written directory browser exists and was tested, but required proprietary libraries. These were hard to find or include in the project, so the tested browser turned out to be unusable.

The major problem with adding this feature was not a browser, but displaying the pictures. During tests, pictures were successfully read from the local file system and transmitted to the server. However, it was not possible to display the picture locally after reading it from the file system. The system was able to recognize the size of the picture, width and height, but did not display the picture itself.

Retrieving these pictures from the server was not possible either. The mobile phone comes with an application to take pictures and these pictures were a lot bigger in data size. Problems with the transport protocol between server and client prevented pictures from arriving at the client side.

Until now the goal was to fit this support in the current application. The screens added to display pictures, and the framework used to exchange pictures with the server had to be used. If not use the existing features; create a specific set of classes to handle pictures from the file system. This would lead to having several versions of a single feature. At that point the decision was made to consider this feature, retrieving pictures from the file system, as not possible and research on it was stopped. However the application and the server support pictures. The only requirement is that these pictures are taken from within the application. This is briefly explained in 4.9 Taking Pictures.

## 8.7 J2ME record management store

Sometimes it is desired to store information between MIDlet invocations or for the MIDlet itself between user sessions. This requires a persistent storage of data on the system. J2SE/J2EE users have a wide range of possible storage systems such as various database engines or a set of files. J2ME has a system which works like a simple, record oriented database and it is called Record Management System (RMS). This system is used in the application and therefore a short introduction to RMS is given. A record oriented database keeps all information stored in records. A unique identifier, a record identifier, identifies these records. The user can store information in a specified record by supplying the record identifier when writing information to the database. If a record already exists, it will be updated with the new information.

Records can be retrieved using the same record identifier when performing a read operation. These two basic principles are implemented in the Record Management System.

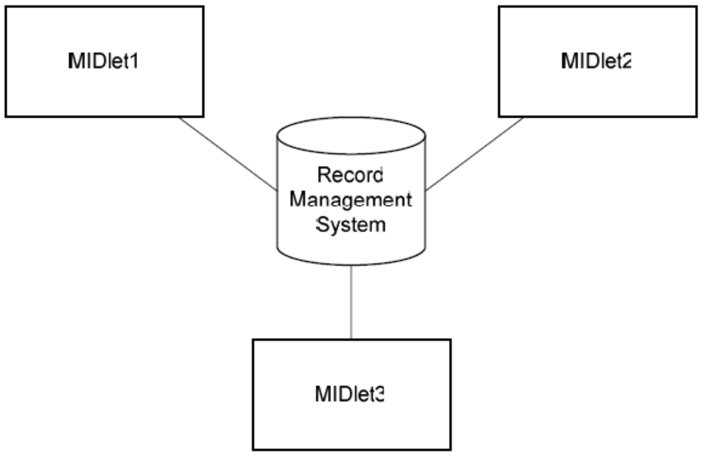


Figure 8.11: RMS with MIDlets

Each MIDlet can store a set of records in the system by using the RMS interface classes. The device in its turn will ensure that the data is stored persistently and remains intact between MIDlet invocations (e.g. the data has to remain present after a battery change). The device also keeps track of the pieces of information stored by each MIDlet. This way the data is protection from other MIDlets, and cannot be changed between invocations of the MIDlet that stored the information.

In order to use a RecordStore, it has to be opened first. Methods are provided to open an existing record store, create a new one, or open a RecordStore and if it does not exist create it. After opening, the MIDlet can read records using their record number, or write records. Deleting a record does not move all the records with a higher record number up one place, so references to a certain record number will always work as long as the record is not deleted. Writing to an existing record number will cause an update of that record. After all operations, the RecordStore has to be closed.

The developer is responsible for handling the data read from a record store correctly. Casting is required as record stores only keep a sequence of bytes and cannot tell which data type was read.

## 8.8 MoBlog client

Now that the environment is explained, it is time to explain the application itself. This chapter will try to show how the application is constructed, which functionality was implemented and how some issues were solved.

First of all a description of the pattern used in this application is given. It explains why this pattern is used and provides information about why the packages and classes are organized in this way. Next each package is highlighted along with the most important classes and features.

### 8.8.1 Model View Controller design pattern

When creating a project, it is useful to have guidelines, a roadmap that can be used. In the software development world, there are many design patterns that can be followed. For this Model View Controller (MVC) design pattern chosen. This pattern defines a way of communication between input, output and the business logic. Figure 8.12 shows how the MVC pattern works. The model represents the business core of the application. This model has one or more views that display output. The controller does the input to this model.

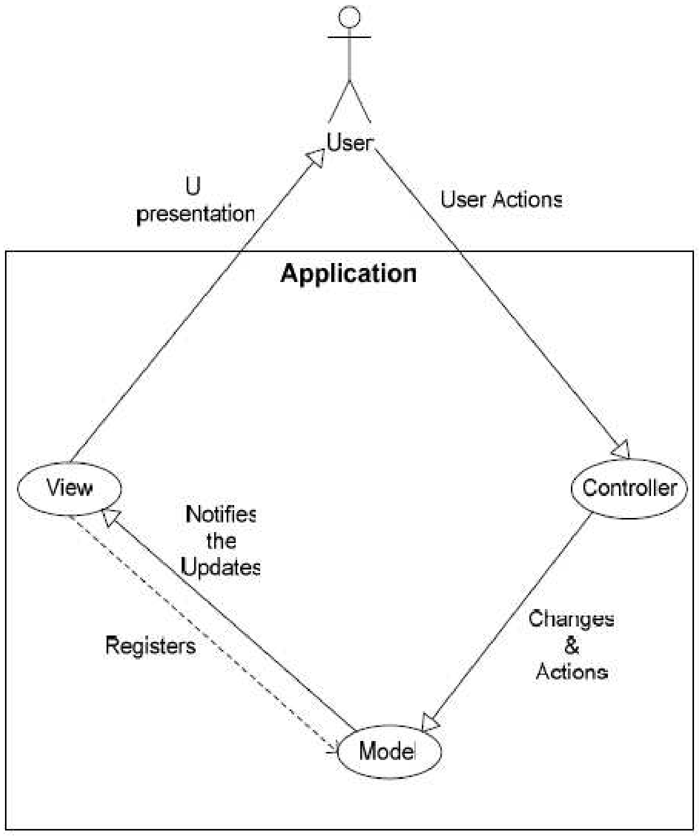


Figure 8.12: The MVC pattern

The model notifies the view of its internal state. When this internal state changes, a notification is send. This notification holds information concerning the new state of the model. Changes to this internal state can be made by certain inputs from the user.

In order to receive the updates, the view must register to the model. A reference to this view is than held within the model. When using J2SE, this MVC pattern can simply be implemented by using some dependencies to available classes within the API. The classes must serve as parent to the actual model, view and controller.

In J2ME these classes are not available due to a smaller API. Therefore could not use their dependencies. Therefore, make these MVC depending methods within these classes.

In the MoBlog case, the MVC pattern can be used twice. Once to display the information to the user, and once to make the core aware of the network layer state. Table 8.2 shows the classes needed to implement the MVC pattern that controls the display to the user.

*Table 8.2: MVC classes*

| MVC | Class | Explanation |
|---|---|---|
| **Model** | BlogEngine.java | Holds the business logic. |
| **View** | UserInterface.java | Displays the state of the model. |
| **Controller** | UserInterface.java | Gives input to the model. |

### 8.8.2 Packages

It is not mandatory to divide the source code into different packages, but for readability reasons it is preferred. This Code is divided into three different packages. Every package represents a well-outlined part of the complete project. A short overview of each class is given, to create an understanding of how the application works. Short code snippets are added to clarify things where possible or needed. The division for the packages also is based upon the MVC design pattern used.

#### 8.8.2.1 Package com.siemens.mobile.core

This is the heart of the application. Most of the functionality is implemented in the classes found in this package. This package implements most of the Model part of the MVC pattern.

**BlogEngine**

This class creates every other object. One of the most important tasks of this class is formatting messages before they are sent. Several methods transform arguments into XML, which in turn is passed to the SipRelay. Sending requests relies on the methods built into the Stack class that is discussed below.

The second task of the BlogEngine is parsing any XML data that is received from the SipRelay class. When the update method is called, the data passed to it will be parsed with a specific handler for that type of data. For example, if the data contains information about a list of entries, the EntryListHandler is used. After parsing the XML data, the BlogEngine will inform the user interface of the new information, which could display it immediately, or when the user asks for the information.

The BlogEngine implements the Singleton pattern, hi this pattern, only a single instance of a class is allowed. To achieve this effect, the constructor is hidden and can only be called by using a different method. This method will check a reference to the BlogEngine object, and if it does not exist, it will create the object. This piece of code shows the implementation of the singleton pattern.

```
static public BlogEngine getlnstance()
{
if (instance == null)
instance = new BlogEngine ();
return instance;
}
```

*Code 8.11: Singleton pattern implementation*

**SipRefresher**

This is one of the smaller classes in this project. A session in the IMS network starts with a REGISTER request, this request has a limited lifetime, noted in the expiry field of the request. To keep a session active, a new REGISTER request has to be sent before the original expires. This class handles the sending of a new REGISTER request and keeps the session alive. It transmits a request after 90% of the original expiry time. The functionality of this class is used as a fix for the exceptions thrown by the Jain SIP stack [198]. Whenever the connection to the proxy was closed, exceptions were thrown by the stack. By reducing the expiry time, the refreshing of the session keeps the connection to the proxy alive, and the application running.

**SipRelay**

There are two main functions in this class. The first function is acting as a listening point for incoming requests and responses. The methods involved in this process are defined in the SipListener interface, which is implemented by this class.

Incoming responses are first checked for a known subject in the reason phrase. This is done using a Hashtable mapping the subject to a certain Message Type. This Hashtable is created and filled when the application starts. Afterwards only a quick lookup using the subject found in the message as key is required to determine further action. Once the type of response is known, the SipRelay updates the BlogEngine with the information found in the response. If the response contains authorization information, an update is invoked on the Stack class, and a new REGISTER request is sent. When the subject found in the reason phrase is unknown to the SipRelay or no reason phrase is present, the response code is checked.

The second function of this class is setting the correct subject before sending requests using the Stack class. Sending a request is actually a three step process involving the BlogEngine, the SipRelay and the Stack classes. First the BlogEngine generates the correct body for the request, and passes other required parameters (i.e. an identifier of the blog) along with the body to the SipRelay. The SipRelay will add the correct subject, and invoke a method on the Stack that effectively sends the message.

**Stack**

The Stack class was created as an interface between the application and the SIP stack implemented below the application. First it creates the objects required to communicate over SIP, it initiates the SIP stack. Next it creates headers that remain constant throughout the entire invocation of the application, and creates listening points for incoming requests or responses. It contains methods to send SIP requests such as REGISTER, MESSAGE or SUBSCRIBE, and will handle the authentication required in the REGISTER requests. Finally it has a method to create a response for a request. NOTIFY requests are sent from the server to the client, and according to the SIP protocol, the client has to send a response. This method generates these responses from the received request. This means header fields are copied automatically or switched where necessary.

Some headers require the IP address of the device. J2ME [199] does not support a way to easily discover your own IP address; there is no system parameter which can be used. To solve this problem, used a small bit of code:

```
SocketConnection sc = (SocketConnection)
Connector.open("socket://" + PROXY_IP + ":" +
PROXY_PORT);
IPAddress = sc.getLocalAddress(); if (sc != null)
sc.close();
```

*Code 8.12: Discovering IP address.*

The idea is simple; open a new connection to the proxy server, and look for the IP address associated with this connection on this side. Once the IP address is known, use it in headers and the connection becomes useless so it is closed.

### 8.8.3 MoBlog

This is more of a dummy class. It contains the main method for the application and is called when the MIDlet is started. It creates the BlogEngine object, which handles the creation of all other objects. It is also the last object to exist when the application is terminated. It invokes the destroy method on the

BlogEngine, which in turn destroys all other objects. After this the MIDlet ends.

#### 8.8.3.1 Package com.siemens.mobile.util

This package is a miscellaneous package. It contains smaller classes that provide a specific functionality used in the application and it is not a part of the MVC structure.

**Storage**

Only a few settings are stored on the device, most data is retrieved from the server. To store these settings, this class is created as interface to the RMS system. It is responsible for keeping track of the records used, updating these records and retrieving information from the record store. The Storage class sets default values for the settings when an application is used for the first time. Settings can be changed from the menu, and using certain methods in this class, the information is stored in a record store. Finally methods exist to retrieve every piece of data stored in a record store. These are called in various places of the application, i.e. when forming certain SIP requests, sending a picture, or writing logs.

**Logger**

Early development was done using Siemens S65 Emulator software. This emulator has support for standard printing methods. This class formats output to the standard output and adds a category in front of the message. The output window is easier to read this way, and you can locate a problem much quicker. All output from the program has to be sent using a method call on this class. It has support to turn log in levels, *OFF, DEBUG, INFO* and *ERROR.* These levels are defined in the code when making a call to an output method.

**MessageTypes**

Implementation of the MVC pattern uses numbered events. The numbers are passed as an argument to the method calls. This class defines a textual representation of the numbers.

```
public    static final   int  RESPONSE_CODE = 0;
public    static final   int  NO_ENTRY_DATA = 1;
public    static final   int  NEW_BLOG = 2;
public    static final   int  NEW_ENTRY = 3;
public    static final   int  NEW_ENTRY_DATA = 4;
public    static final   int  NEW_COMMENT = 5;
```

*Code 8.13: Snippet from MessageType class.*

This code snippet shows examples from the definitions found in the Message Types class. Their definition makes them immutable and available from every object in the application. They allow us to map names to the numbered events that improve readability of the code.

#### MD5Calc

An implementation of the MD5 hashing algorithm. It can be used to generate an MD5 hash from a given piece of data. The authorization system uses MD5 and therefore relies heavily on this class. The authorization is done in the Stack class, which uses this MD5 calculator.

#### FakeSipRelay

A class used in the early development of the application. This class simulates a SIP connection, and responds with preset answers to requests. In the early stage of development, this was used mainly to test XML parsers and the updating between classes.

### 8.8.4 Package com.Siemens.mobile.util.xml

The bodies of this SIP messages are defined in Extensible Markup Language (XML). In order to process these bodies, an XML parser is required. J2SE and J2EE come with a parser that follows the Document Object Model (a DOM Parser) and one that follow the Simple Access Parser API (a SAX Parser).

A DOM parser handles an XML document as a set of objects. The document is parsed once and stored in memory in a tree structure. This makes it useful when several reads or writes are performed on the document. However, it requires for the entire document to be in memory that can become a problem on systems with limited resources such as a mobile phone.

The SAX parser parses a XML document from start to end, and going back to a previous section is not possible. This makes it a slower and less efficient system when various sections are consulted repeatedly. It does however reduce the memory usage that is an advantage for a limited environment.

This application only requires a single reading of the XML document. All the data is collected this way and stored into a temporary object. Another class that is responsible for displaying the contents will then read the object. As both the XML document, and the object containing the data are only read once and sequentially, it is not necessary to store the data in a tree structure. Therefore a SAX parser will offer all the required functionality.

Presence of a SAX Parser is not guaranteed in J2ME, since it is implemented in Java Specification Request 5 (JSR 5). Not every system has the JSR 5 installed and therefore create own basic XML parser. The SAX Parser in J2SE consists of several classes. One class is the actual parser; it reads the XML data and acts depending on the input. Another class is the default handler that is passed to the parser.

When reading the XML, the parser will call methods on the default handler supplied to it. The developer to achieve a certain result can then implement these methods. Limited functionality was required from a parser, due to the basic XML layout (see Appendix J: XML Message Structure for an overview of the XML structure). Implementation of an XML parser is similar to the SAX Parser and consists of two major types of classes.

**XML Parser**

```
public XMLParser(String source, XMLHandler handler)
```

*Code 8.14: XMLParser.Java – Constructor*

The constructor of the XML Parser class takes two arguments, an input of the type String and an event handler which implements the XMLHandler interface. The XML Parser is the core of the system; it reads the XML data and calls events on the event handler. The parser reads character after character from the source and checks this character for a possible event. Events can be an opening tag or a closing tag, plain text between two tags or the start or end of an XML document.

**XML Handler**

```
public     void startDocument(String header);
public     void startElement(String tag);
public     void endElement(String tag);
public     void startData(String data);
Public     void endDocument();
```

*Code 8.15: XML Handler methods*

These methods are defined in the XML Handler interface. Every handler must implement this interface in order to be compatible with the XML Parser. The implementation of these handlers is done with a finite state machine. A state is used for each different tag found in a particular XML source and in each state the data between the tags is read into memory.

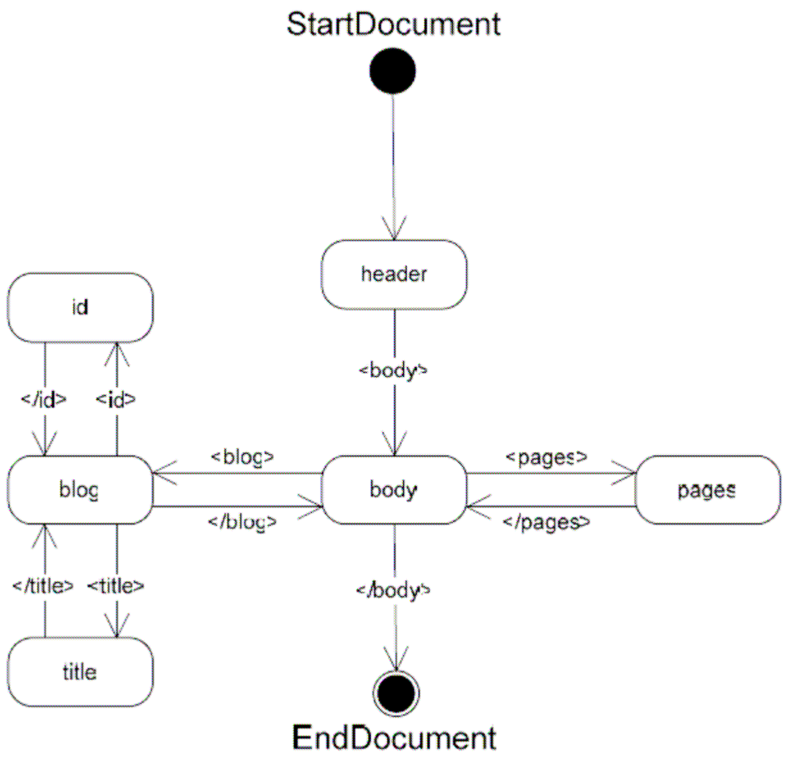


*Figure 8.13: State Diagram of BlogListHandler*

Figure 8.13 shows a state diagram for the BlogListHandler. This handler is used when a list of blogs is received in XML format and needs to be parsed. The tag causing the state transition is noted next to each transition. After reading the data between two tags, the state machine returns to the previous state (i.e. after reading the ID, it returns to the blog state).

This makes the system order independent. Putting the ID before or after the title does not matter; both ways appear identical to the state machine and the handler. Several types of XML sources exist, and handler for each type. All these handlers require a single parameter in their constructor.

This reference is used as a container to store all the data found in the XML source. The class used to display the data will parse the container according to rules set during development. These rules specify where a certain element is located in the data container.

### XML Escaper

Not all characters are valid in XML data. For example an ampersand ('&') is not a valid character. It is used in escape sequences for these special characters. This class handles the conversion from special character to escape sequence and the reverse using the methods shown below.

These methods are static so every handler can call them to convert characters. Each method is built with a small lookup table containing the characters and the escaped alternatives, or escaped characters and their normal representation.

```
public static String escapeXMLChars(String input) public
static String unescapeXMLChars(String input)
```

*Code 8.16: XML Escaper*

Table 8.3 shows an overview of the characters that require escaping. These escape sequences are the same as the ones used in Hypertext Markup Language (HTML).

*Table 8.3: Escaped characters in XML*

| **Character** | **Escape sequence** |
|---|---|
| & | &amp;amp; |
| ‘ | &amp;apos; |
| “ | &amp;quot; |
| < | &amp;lt; |
| > | &amp;gt; |

### 8.8.5 Package com.siemens.mobile.views

All the classes that control the display and the keyboard are collected in this package. One could separate the display from the keyboard, but in some cases these two components work very close together. When the keyboard is used for data input, the text is added to the display immediately. Make a view for every kind of screen that must be displayed. For instance, there is a class that provides the main menu, a class for a new entry, etc. Discussion of different kinds of screens and the classes those are responsible for them. This package implements the View and Controller part of the MVC pattern. The View part is where items are put on the display. When the keyboard is used, the input is added to the screen and the object that is displayed. This object can then pass the input to another object, which plays the role of Controller.

During practical implementation, this MoBlog over Nokia Mobile Phone, real time implementation and analyze service rendering.

## I. Main Menu

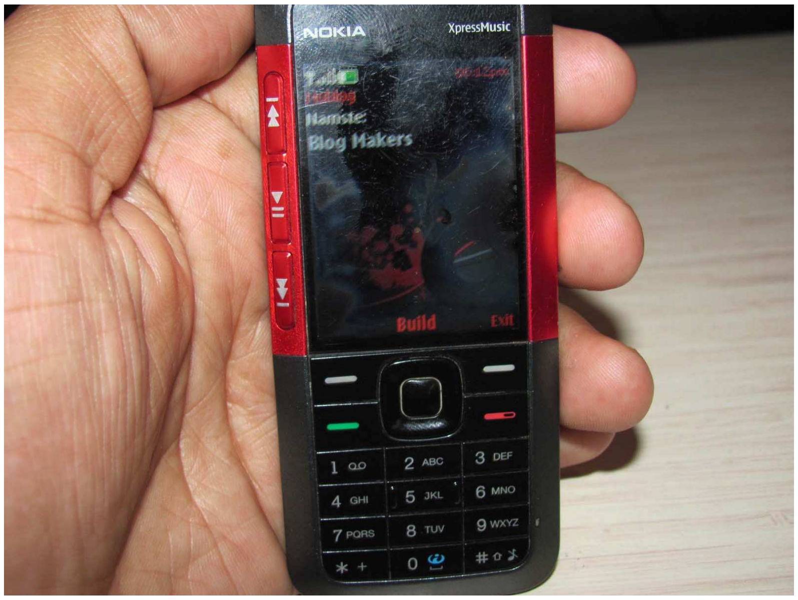


*Figure 8.14: Main menu*

This figure shows the result of the main menu code. Here chosen to use a List. There is only one option that can be selected at a time. Therefore a List is the perfect chose. On this List, only two Buttons reside: one to activate a selected item and another to exit the program.

## II. My Blog Screen

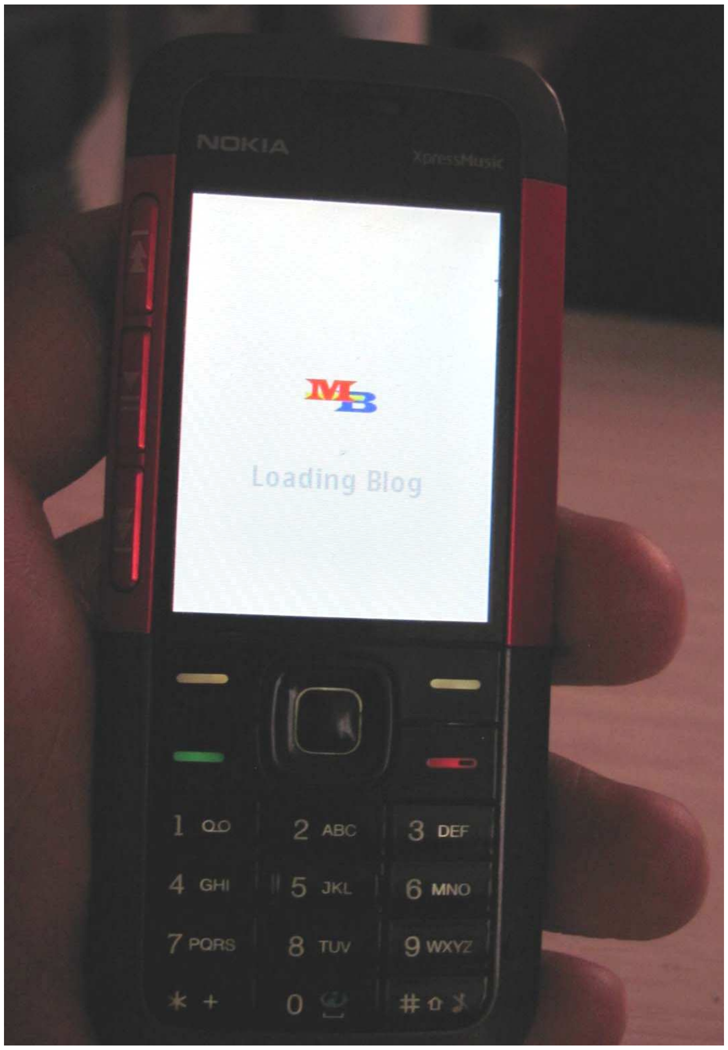


*Figure 8.15: My Blog Loading Screen*

As shown in Appendix J: MoBlog Menu Structure, the My Blog option from the Main Menu holds two screens. When a user is new to the system and doesn't have a blog, the new blog screen is shown. If the user already has a blog, the entry list of his own blog shown in figure 8.15.

## III. Entry Screen

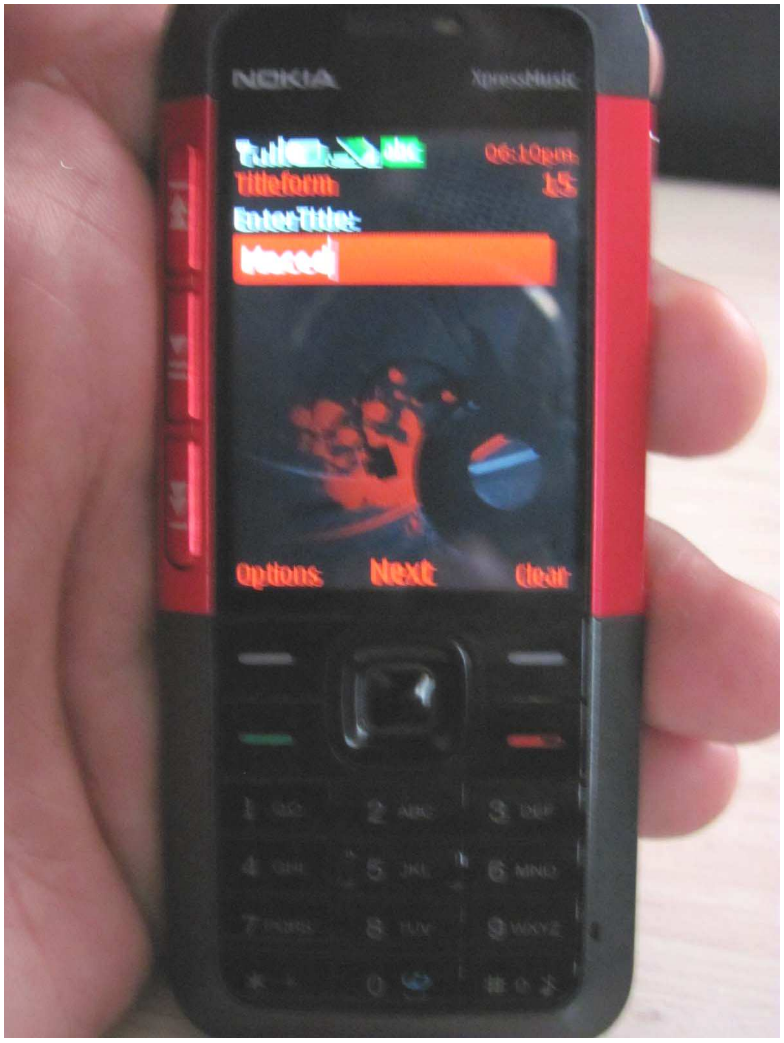
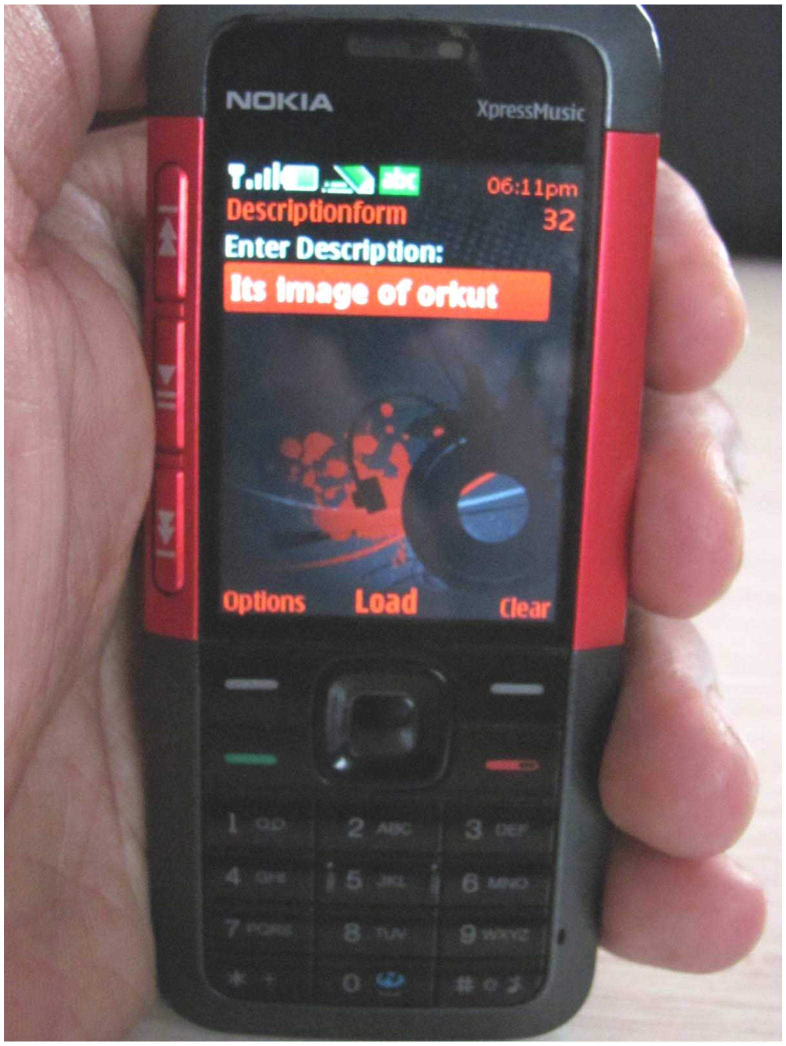


*Figure 8.15: An EntryScreen*

The entry screen is based on a Form instead of the List. On a Form one can easily display a String. This can be done without any reformatting of that String. Beside the Strings there are a few buttons placed on the Form. Here one can see that when multiple buttons need to be shown, the device automatically combines them into an Options button.

## IV. Description of MoBlog



Descriptionform

Enter Description:

Its image of orkut

*Figure 8.16: A Comment List*

Besides an entry, a blog can also contain comments. All the comments for one particular entry are collected in a CommentListScreen. Again, as with the EntryList, use the List object instead of a Form.

## 8.9 MoBlog Server

### 8.9.1 Session Initiation Protocol - Application Server

The Session Initiation Protocol - Application Server (SIP-AS) is a system designed for easy application development. It provides an easy to use Java interface to the SIP protocol stack. This keeps the focus on the application development and abstracts the connection and protocol issues of SIP. In this thesis, the server side is built on the Ubiquity Software implementation of a SIP-AS. The SIP-AS has several functions [200]:

- Service director
- Service Host
- Logging Extraction
- Management Server

The access to the IMS network is built in two pieces, a ServiceDirector and a ServiceHost. The ServiceDirector is the access point to the SIP-AS. It handles basic TCP and UDP communication and forwards incoming packets to the correct ServiceHost. The ServiceHost handles all SIP traffic, generating responses to the requests and delivering the requests to the servlet that handles them. These servlets are an extension to the classic Java servlets and work in a very similar way.

Some form of administration is needed, which can be done using the Managementserver component. This provides an easy way to develop new services or to upgrade current running services, and keeps track of deployment or general runtime errors. This SIP-AS can be used for easy log extraction with the LoggingExtraction. The information is collected from incoming and outgoing requests and can be used to generate statistics or billing for the use of a particular service.

### 8.9.2 Implementation of the MoBlog Server

This section explains a little about the server side of the application. Figure 8.17 gives an overview of the server architecture. The server is written in Java using the Java 2 Enterprise Edition framework and runs JBoss for the web interface.

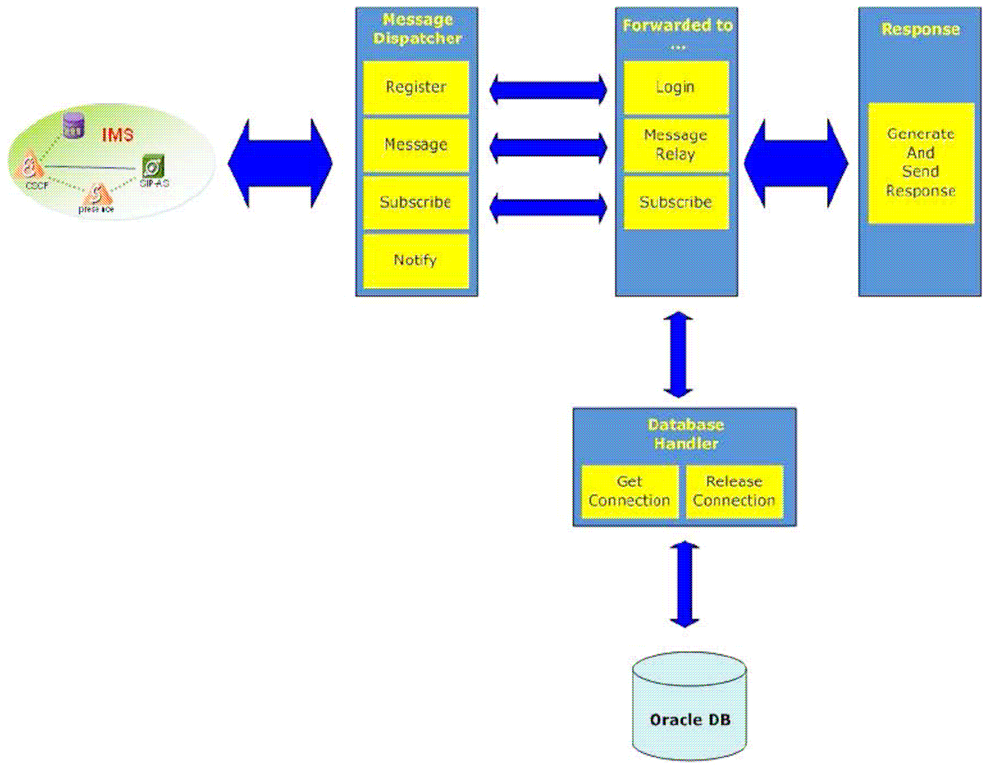


*Figure 8.17: MoBlog Server Architecture*

The main task of the server is data storage and retrieval. An Oracle database is used for this storage and contains all the information. User data, blog entries, pictures, ... Users have to register themselves to the server in order to get access to the content. The server receives the REGISTER request which is sent from the client to the IMS network. The server will store information about the user until a REGISTER is received to end the session. After the registration, a regular message is sent to the server by the client to discover information about the clients own blog. This makes it easier to retrieve the list for the My Blog screen.

Once the IMS network and the server know a user, they can start using the Picture Blog. The server will handle all their requests using the MessageDispatcher. This class will inspect every request and forward it to the class responsible for dealing with that type of request. The server will automatically generate a response to the request. While storing data about every blog, the server also keeps track of changes made to blogs. Upon these changes it can use the notify system to inform users about these changes. If a user wants to receive these notifications they have to subscribe to a blog. The server keeps a list of which user is subscribed to which blog, and which notifies are pending. A pending notification is only removed from the list once it has successfully been sent to the client. The server also provides a web based user interface. This way you can access the content without using a mobile device with the MoBlog application installed.