Abstract
Most mobile phones today offer the possibility to play advanced games, and the boundary between personal computers and mobile phones are fading. However, few games make use of the fact that mobile phones are, indeed, mobile. Project Green Fox presents a set of prototypes for location based mobile games that force the player to interact with the real world. By combining the game world with reality the players get a sense of being inside the game, enhancing the experience.
# Contents

1 Introduction 8

2 Preliminaries 9

3 Game Concept Ideas 11
  3.1 General Ideas 11
  3.1.1 Death Game 11
  3.1.2 Divide a City 11
  3.1.3 Players as NPC 11
  3.2 Mini Games 12
    3.2.1 CamGame 12
    3.2.2 Tag 12
    3.2.3 Tetris 12
    3.2.4 On the Run 12
    3.2.5 Formation 13
    3.2.6 Formation Competition 13
  3.3 RPG 13
    3.3.1 Adventure RPG 13
    3.3.2 Live action RPG 13
    3.3.3 Board game RPG 14
    3.3.4 Miscellaneous features 14
  3.4 Urban Exploration 14
    3.4.1 Tourist Guide 14
    3.4.2 Short Quests 15
    3.4.3 On-going quests 16

4 Green Fox 17
  4.1 Vos 17
    4.1.1 Main Menu 17
    4.1.2 Venture 18
    4.1.3 Battle 19
    4.1.4 Shop 20
    4.1.5 Dialogue 21
    4.1.6 Quest 21
    4.1.7 Character sheet 22
  4.2 Minigames Web Interface 23
    4.2.1 Player Mappings 23
    4.2.2 AdminCommand 23
    4.2.3 Communication Log 23
    4.2.4 Minigames Core Log 23
    4.2.5 Minigames Message Log 23
    4.2.6 CamGame 24
  4.3 CamGame Web Interface 24
    4.3.1 Songs 24
    4.3.2 View High Scores 24
    4.3.3 User Mapping 24
    4.3.4 Games 24
    4.3.5 CamGame Core Log 25
4.3.6 CamGame Message Log 25
4.4 CamGame client 25
4.4.1 Start a New Game 25
4.4.2 Choosing Difficulty 26
4.4.3 Making Music 27
4.4.4 Finish a Game 28
4.5 Tourist Guide Client 29
4.5.1 Splash Screen 29
4.5.2 Tour Selection 29
4.5.3 Tour Details 30
4.5.4 Map 30
4.5.5 Place List 31
4.5.6 Adding Places 31
4.5.7 Place Details 32
4.6 Tourist Guide Web Interface 32
4.6.1 User Login 33
4.6.2 List Tours 34
4.6.3 View Tour 34
4.6.4 Add Tour 35
4.6.5 Edit Tour 36
4.6.6 List Places 36
4.6.7 View Place Details 37
4.6.8 Add Place 38
4.6.9 Edit Place 39
4.7 Agent SQ client 40
4.7.1 Splash Screen 40
4.7.2 Main Menu 40
4.7.3 Ask For a Mission 41
4.7.4 Mission Briefing 42
4.7.5 Distance Gadget 42
4.7.6 Clues 43
4.7.7 Player Statistics 43
4.8 Agent SQ Web Interface 44
4.8.1 List Quests 44
4.8.2 Add Quest 44
4.8.3 List Missions 45
4.8.4 Add Mission 46

5 System Description 47
5.1 IMS 47
5.2 Common Application Structure 47
5.2.1 Common Communication Framework 47
5.2.2 Client Structure 48
5.2.3 Servlet Structure 48
5.2.4 Database/Web Server Structure 49
5.3 Prototype-Specific Structure 49
5.3.1 The Tourist Guide 50
5.3.2 Agent SQ 52
5.3.3 Vos 54
5.3.4 CamGame 66
### 6 Evaluation and Testing

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Camgame</td>
<td>73</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Methods Used</td>
<td>73</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Result of the Evaluation Form</td>
<td>73</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Comments by Test-Subjects</td>
<td>75</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Our Comments / Conclusions</td>
<td>76</td>
</tr>
<tr>
<td>6.2</td>
<td>Tourist Guide</td>
<td>76</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Methods Used</td>
<td>76</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Comments by Test-Subjects</td>
<td>76</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Result of the Evaluation Form</td>
<td>76</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Our Comments / Conclusions</td>
<td>78</td>
</tr>
<tr>
<td>6.3</td>
<td>Agent SQ</td>
<td>78</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Methods used</td>
<td>78</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Result of the Evaluation Form</td>
<td>78</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Our Comments / Conclusions</td>
<td>80</td>
</tr>
<tr>
<td>6.4</td>
<td>Vos</td>
<td>80</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Methods Used</td>
<td>80</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Comments by Test-Subjects</td>
<td>81</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Result of the Evaluation Form</td>
<td>81</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Our Comments / Conclusions</td>
<td>82</td>
</tr>
</tbody>
</table>

### 7 Related Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Future work</td>
<td>85</td>
</tr>
<tr>
<td>8.1</td>
<td>Vos</td>
<td>85</td>
</tr>
<tr>
<td>8.2</td>
<td>CamGame</td>
<td>85</td>
</tr>
<tr>
<td>8.3</td>
<td>Tourist Guide</td>
<td>86</td>
</tr>
<tr>
<td>8.4</td>
<td>Agent SQ</td>
<td>87</td>
</tr>
</tbody>
</table>

### 9 Conclusions

### A Installation Instructions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Installing the MIDlets</td>
<td>88</td>
</tr>
<tr>
<td>A.1.1</td>
<td>Installing the MIDlets on a Phone</td>
<td>88</td>
</tr>
<tr>
<td>A.1.2</td>
<td>Installing the MIDlets on an Emulator</td>
<td>88</td>
</tr>
<tr>
<td>A.1.3</td>
<td>Setting up the Eclipse Projects</td>
<td>88</td>
</tr>
<tr>
<td>A.1.4</td>
<td>Setting up the Emulator in Eclipse</td>
<td>89</td>
</tr>
<tr>
<td>A.1.5</td>
<td>Changing Phone Number, SIP Address or Application ID</td>
<td>89</td>
</tr>
<tr>
<td>A.2</td>
<td>Installing the Servlets</td>
<td>90</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Deploying the Servlets</td>
<td>90</td>
</tr>
<tr>
<td>A.2.2</td>
<td>Setting up the Eclipse Projects</td>
<td>90</td>
</tr>
<tr>
<td>A.2.3</td>
<td>Restore Web Project</td>
<td>91</td>
</tr>
<tr>
<td>A.2.4</td>
<td>Changing the Context Root / Application ID</td>
<td>91</td>
</tr>
<tr>
<td>A.3</td>
<td>Creating the Databases</td>
<td>92</td>
</tr>
<tr>
<td>A.4</td>
<td>Installing the Urban Exploration Web Interface</td>
<td>93</td>
</tr>
<tr>
<td>A.4.1</td>
<td>Deploying the Servlet</td>
<td>93</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Setting up the Eclipse Project</td>
<td>93</td>
</tr>
</tbody>
</table>
# B Maintenance Instructions

B.1 Common Instructions .................................. 94  
  B.1.1 Servlet ........................................... 94  
  B.1.2 Client ........................................... 94  
B.2 CamGame ............................................ 94  
  B.2.1 Servlet ........................................... 94  
  B.2.2 Web Interface .................................. 95  
  B.2.3 Client ........................................... 96  
B.3 Tourist Guide ....................................... 97  
  B.3.1 Client ........................................... 97  
B.4 Agent SQ ........................................... 97  
  B.4.1 Client ........................................... 97  
B.5 Urban Exploration Web Interface ..................... 97

# C Suggested Future Work

C.1 Common Suggestions .................................. 99  
  C.1.1 Solve MySQL issue ................................ 99  
  C.1.2 IMS Self Authentication .......................... 99  
C.2 CamGame ............................................ 99  
  C.2.1 Abstraction Level ................................ 99  
  C.2.2 Communication .................................. 99  
  C.2.3 Resource Management ............................ 101  
  C.2.4 Game Logic ...................................... 101  
C.3 Vos .................................................. 101  
  C.3.1 Game server ...................................... 101  
  C.3.2 Map class ........................................ 102  
  C.3.3 Battles .......................................... 102  
  C.3.4 Optimisation ..................................... 102  
  C.3.5 Database ........................................ 103  
  C.3.6 Storing quests ................................... 103  
C.4 Tourist Guide ....................................... 103  
  C.4.1 Communication ................................... 103  
  C.4.2 Resource Management ............................ 103  
  C.4.3 Game Logic ...................................... 104  
  C.4.4 User Interface ................................... 104  
C.5 Agent SQ ........................................... 104  
  C.5.1 Abstraction Level ................................ 104  
  C.5.2 Communication ................................... 105  
  C.5.3 Location service ................................ 105  
  C.5.4 Resource Management ............................ 105  
  C.5.5 Game Logic ...................................... 105  
  C.5.6 User Interface ................................... 106

# D Communication Protocols

D.1 The Structure of Sent Data ............................. 107  
  D.1.1 Using MSRP ....................................... 107  
  D.1.2 Using Page Messages .............................. 108  
  D.1.3 Picture Sending ................................... 108  
D.2 The communication protocols .......................... 108  
  D.2.1 The Tourist Guide protocol ......................... 109
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.2.2</td>
<td>The Agent SQ protocol</td>
<td>111</td>
</tr>
<tr>
<td>D.2.3</td>
<td>The Vos protocol</td>
<td>113</td>
</tr>
<tr>
<td>D.2.4</td>
<td>The CamGame Protocol</td>
<td>115</td>
</tr>
<tr>
<td>E</td>
<td>Database Structure</td>
<td>117</td>
</tr>
<tr>
<td>E.1</td>
<td>The Tourist Guide</td>
<td>117</td>
</tr>
<tr>
<td>E.2</td>
<td>Agent SQ</td>
<td>117</td>
</tr>
<tr>
<td>E.3</td>
<td>Vos</td>
<td>118</td>
</tr>
<tr>
<td>E.4</td>
<td>CamGame</td>
<td>118</td>
</tr>
</tbody>
</table>
1 Introduction

The goal of Project Green Fox was to develop prototypes of location based games for mobile phones. A location based game is a game that you control by doing something in the real world, using, for example, GPS, the phone’s built-in camera, questions or some other method. Examples could be that you should race to a specified location on a map, take a photo of something green or answer a question which would be very hard to answer correctly if you’re not in the right place.

Our other goal was to implement those games using the IP Multimedia Subsystem (IMS), which is explained in detail later in this paper.

We didn’t have the ambition to create commercial products, our aim was to develop prototypes and see if they had the potential to become successful games or not. We decided that we would produce as many playable prototypes as possible, instead of developing fewer, more polished ones.

There are not that many released location based games on the market today, which makes our ideas very interesting. In a few years time, GPS receivers will probably be included in most mobile phones, and the market for location based mobile phone games will increase.

The project consisted of twelve master students in computer science from Uppsala University, that took the Project CS course during the autumn of 2008. Green Fox was a cooperation between Uppsala University, Green hat People and Ericsson Research. Ericsson Research helped us with tutorials, support and provided us with an implementation of IMS. Green hat People helped us with our concepts, shared their expertise about testing and have given us feedback on our games.
2 Preliminaries

Eclipse

Eclipse is an Integrated Development Environment (IDE) that helps developers work on, and structure, large programming projects.

GPS

The Global Positioning System (GPS) is a free-to-use navigation system that works worldwide. It uses a set of satellites to send signals that GPS receivers pick up to determine their exact location. Some mobile phones, like the Sony Ericsson C702, have built-in GPS receivers.

IMS

The IP Multimedia Subsystem (IMS) is a framework for easy implementation of applications using IP connections on mobile phones, and also for creating server applications quickly and easily. It abstracts away a lot of the underlying communication details from the developer, thus allowing the programmer to concentrate on program logic, rather than on connection issues.

Java EE

The Java Platform, Enterprise Edition (Java EE) is an extension of the Java Standard Edition libraries. Enterprise Edition has added libraries and functionalities for safer and easier server development.

Java ME

The Java Platform, Micro Edition (Java ME) is a subset of the Java Standard Edition, with libraries compressed to fit smaller devices, like mobile phones. It does not contain all additional classes that usually come with Java, since it would take up too much space on the memory constrained devices.

LWUIT

The Light-Weight User Interface Toolkit (LWUIT) is a library for creating graphical user interfaces for applications running on mobile devices, in an easier and more portable way than normal.

MIDLet

A MIDLet is an application intended to be run on a mobile phone. Usually it is compiled into a Java Archive (JAR) file that is then uploaded to the phone.

MJCF

The Mobile Java Communication Framework (MJCF) is an infrastructure and a set of APIs for using IMS. It is hosted by Ericsson as a beta service, is available for all mobile developers and is accessible from all of the major mobile operator networks in Sweden.
MSRP
Message Session Relay Protocol (MSRP) is a communication protocol for trans-
mitting a series of related instant messages in the context of a session, such as
the ones used in the Session Initiation Protocol. MSRP is used widely in the
IMS.

MySQL
MySQL is a free, open source SQL database application. We use MySQL for
storing all kinds of values and game data.

RPG
A Role-Playing Game (RPG) is a game where the player takes on the role as
a fictional character. It is usually played in a fantasy world with the help of
computers and/or imagination.

SailFin
SailFin is a Java application server. As the name implies, it is a host for several
server applications (so called servlets). It handles routing of messages to and
from the different servlets and the clients using them. The one used in this
project is hosted by Ericsson research at "imsinnovation.com".

SDS
The Service Development Studio (SDS) is a bundle of IMS application develop-
ment programs from Ericsson, for Microsoft Windows. It contains the Eclipse
IDE with plugin support for running local servers, creating and deploying mobile
applications onto phones and testing and debugging servlets within Eclipse.

SIP
The Session Initiation Protocol (SIP) is a protocol for creating, handling and
tearing down communication sessions between endpoints. IMS uses SIP exten-
sively.

SVN
Subversion (SVN) is a version control system. It is a tool for handling concurrent
development of a project by multiple people. It can be integrated into Eclipse
with either of the two plugins Subclipse and Subversive.

WTK
The Wireless ToolKit (WTK) is a toolbox for application development targeted
at mobile devices running Java ME. It includes a mobile phone emulator that
lets you test programs for mobile phones on a desktop computer, which is very
useful during application development.
3 Game Concept Ideas

After splitting up into three different groups each group came up with a set of ideas. Later the groups made prototypes of some of these ideas. Although they had to be changed somewhat to fit into what we could do (and later wanted to do) the general idea remained. Here we outline the main ideas for game prototypes that we had after the initial brainstorming.

3.1 General Ideas

3.1.1 Death Game

Death Game\textsuperscript{1} is a Swedish game played in the real world, where you should “kill” each other with vegetables and fruits. The idea is that you find an active player on the website, then you track your target down and “kill” him or her with some kind of (edible) weapon. For example, you could use a banana as a gun, find your target, then point your banana at him or her and say “Bang, you’re dead!”. After that you register it on the website. You can also team up and form player teams, and the individual or team who survive the longest win the game.

The problem with this idea is that both the predator and the prey have to be honest and report the kill on the website. The victim could, for example, lie and say that he didn’t get killed at all.

Our idea was to make a game based on the same concept, but make cheating harder by introducing GPS into the game. With this variant concept, other possibilities than the reducing of cheating can be introduced. For an example, you could place bombs or mines (think pumpkins) when you are near a target, and in that way kill the player when he or she gets too close, while everything is verified using the GPS system.

3.1.2 Divide a City

The idea was to split the players up into two or more large teams, and use some large area as the game area, for example a city center. That area is then divided into several smaller sectors that the teams should battle for control over. The exact mechanics for taking control of a sector is not really important to the concept, and could be pretty much anything.

A variation that sort of merges the formation idea below with this concept is that teams should try to surround the other team by moving around in the city. The mobile client can show the area currently controlled by each team, and once one team’s area contains the entire other team’s area, the game is over.

This game can be used as a container for all the other games as well. The idea is to save the progress of other played games and also let the result of other games affect the progress of capturing the city for each team.

3.1.3 Players as NPC

This idea was mainly meant as a sub-concept to the short quest and on-going quest game concepts, but can also be applied to RPG for example.

\textsuperscript{1}http://www.deathgame.se/
The idea is to use the players themselves as staff for the game, by making them interact with each other. This solves a big problem that Green hat People has at the moment, where many of their more exiting games require a lot of personnel from their side. An example of how this could be accomplished is that one player is told to go and sit at a café, waiting for a contact. When the contact arrives, they are to give him or her the secret phrase, and in return the player will receive a password that is needed for the next task in the game. Meanwhile, another player is instructed to go to said café and meet a contact with a secret password, and thus they exchange passwords and both thereby act as staff in the other person’s game. This can be varied in a lot of ways as well.

3.2 Mini Games

As the name implies, Mini Games should be easy to play and simple to understand, but, of course, be a lot of fun. These kinds of games should be playable at any time, anywhere, and it could be either a sub-game in some larger game or a small single game. Several ideas sprouted from these premises, and in the end it was the CamGame that came to be implemented, in part because it did not require GPS.

3.2.1 CamGame

This game is based on the cell phone’s camera and MIDI music. The player tries to use the camera to make melodies. The player is given a picture consisting only of a number of colors in a pattern, and is then supposed to find or arrange a similar setup around him or her, and take a picture of it. A note is then generated for the song that is being created, and the closer the picture was to the target, the closer the note is to the true value. The process is then repeated for each note in the melody. So essentially, this game allows the player to use a cell phone camera as a tool to match different colors to different musical notes.

3.2.2 Tag

This idea is based on the old children’s game where there is one hunter who chases everyone else. When he in some way catches someone else, that person too becomes a hunter. Last non-hunter wins. This could be expanded with the help of the mobile phone to handle tagging in different ways, and perhaps displaying the position of the players on a map in the client.

3.2.3 Tetris

Get your friends to form up and move in blocks of different shapes! Everyone has played or at least seen Tetris, but never with so many people or in such a physical manner.

3.2.4 On the Run

Inspired by the Swedish TV-show known as “På Rymmen”, some people are supposed to elude a number of chasers. Every now and then the Runners are supposed to do something, and their position is revealed. Survive for as long as possible.
3.2.5 Formation

A number of players have to cooperate, and physically move around, so that their positions together form the shape the game asks of you. Compete against your friends, time, or mere size. You may for example be asked to form the largest possible hexagon.

3.2.6 Formation Competition

Team up and assassinate your opponents team members by cunning strategy and teamwork. You could for example be tasked to surround every member of the opposing team with your own, or create an arrow formation aimed at the opponent to defeat. Very much like the Divide A City concept, but on a smaller scale. Could be used as the game logic in that game.

3.3 RPG

The team came up with a number of ideas for RPG variations, which we will list here.

3.3.1 Adventure RPG

This concept idea was basically like how normal massive multiplayer online RPGs (MMORPGs) work, but using mobile phones with GPS. It would contain the normal RPG elements, such as players fighting monsters, interacting with NPCs and completing quests in order to get items and experience. Players could log in to the game and then see other people who are playing it and interact with them in different ways (fight them, trade items with them and so on). Because it is played in the real world, people could also interact physically. For example, instead of sending messages to each other when trading items, the players could simply talk to each other in the real world.

3.3.2 Live action RPG

The idea of this concept was that the mobile phones were to be used as tools to assist the playing of a normal live action role-playing game (LARP). In this kind of game, a group of people (often quite large) get together and physically role-play together, sometimes for several days. Players are assigned roles in which they should then act, and the arrangers of the event, called game masters, manage the game while it is being played. The mobile phones could be used for showing for example maps, or for getting information about what is happening in the game. Battles could also be aided by the phones, so that either they are carried out entirely on the phone, or perhaps so that the phone is used in place of other tools that are often used for this purpose, such as dice, pen and papers.

The game masters could use desktop clients for administrating the game. This would let them see where all the players are (using the GPS in the client phones), and add new content to the game while it is ongoing. For example, an advance in the story line could be announced to the players, or a monster could be added in a specific place.
3.3.3 Board game RPG

This idea was influenced by normal board game RPGs that you play with your friends at a table (such as Hero Quest). Instead of using a board and accessories, however, the game would be played outside using the mobile phones to handle the game mechanics. The difference in comparison to the adventure RPG concept was that this is smaller in scale. While the adventure RPG concept is continuous and played by many people at once, the board game idea was that each game would be an isolated experience that you play with people you know. Each game would be relatively short, and they could perhaps be downloaded from a server.

3.3.4 Miscellaneous features

We also came up with some features that can be added to the above concepts.

- Monsters are randomly generated in the area around the player.
- A virtual map of the geographical region the player is located in could be shown on the screen. It could show things such as other players, monsters, buildings, forests etc.
- Players could build structures in the game that could be seen by other players. A player could for example build a house where he could live, or a turret that fires at trespassers.
- A player could flee from a fight by physically running away from it.
- Monsters in the game are ghosts, which explains why they can’t be seen in the real world and we need a detecting tool (a phone) in the game. This make the game world and the real world similar.
- Players could own pets and play a Tamagotchi-like minigame in the game to take care of the pet.
- Quests could be statically made or dynamically constructed in the region where the player is.

3.4 Urban Exploration

3.4.1 Tourist Guide

The Tourist Guide is a mobile application where the phone is used as a tool in the real world, to replace real tourist guide persons or tourist guide books. The purpose is to let the user explore the area on its own.

*Below, “place” refers to a point of interest for a tourist.*

Planned Tour The planned tour could could be tailored to have a specific theme, like music or historic buildings. They should be made cyclic with multiple entry points from which the user can start using the tour. But also it should be possible to skip places if the user doesn’t want to visit them but instead continue to the place after the one skipped. If there are different tours available in the area it should be possible to change to another without to much problem.
To aid the user the application should contain a map which shows the locations of the different places. It would be good to have distance measurement and estimated time for walking etc.

**Information from Position** It should be possible to filter highlights on the map according to personal interest. A good feature would be to have user contributed content, then users can add new places they find. They can take a picture of the place and write a description. The coordinates (via GPS) should automatically be stored with the place. Later after reviewing the place can be added to a tour.

The type of information that can be presented together with a location can be texts, images, videos and speech. Speech might for an example be transmitted to the user while he or she walks between places, to give tourist information that isn’t tied to any specific place.

Another feature, that helps the user to travel between places, could be that the system retrieves public transport timetables and suggests to the user to e.g. take a bus.

### 3.4.2 Short Quests

This concept is a game that can be played in separate parts, where you as the user chooses the time when to play the game. It’s location based in that real life places play a role in the goal of the game. The game goal is that the player should get from his or her current position to a new position in as short time as possible with the help of some sort of description. The new position should preferable be a not-so-known location in the spirit of urban exploration.

A *quest* is an endeavour which is comprised of several *missions*. Missions are the smallest components of the game, that when completed pushes the game forward. Missions have a time limit, and should be completed before the timer has reached zero. The game is played in single player, but the high-score list should be shared among all users so they can compete against each other.

If a player thinks that a mission is too difficult he or she should be given the option to get a *hint* in exchange for some score reduction. In the mission briefing there might be a picture that is very zoomed in on some small detail, together with some text. Perhaps the user can’t guess anything about the place but then has the option to request a *hint*, which might provide be a better, more zoomed out picture. The text can also be alone, without a picture; the text might be a riddle and the answer can lead the player in the direction of the sought place, e.g. a house of a historic person. It can also be the other way around, at the place there is an answer to the question the player is given, and the task of the player is to provide this answer. If the player has good common knowledge, he or she can answer the question directly when is is given and gain a lot of points. E.g. in what year did king Gustav Vasa die? Then the player either can go to the Uppsala Cathedral and read it on his tomb, or answer right away if the answer is known.

The score gained from a mission is calculated from the time it took to do it, and how many hints that were used. These points are then added to the *quest score*, which in turn is added to the player’s score when the quest is completed.
One very interesting feature for casual gaming would be that the system arranges missions that are close to a path the user is planning to walk along. E.g. with a route going from east Uppsala to north Uppsala, the system will only add missions that are roughly on that path. This would allow for much more interesting walks than usual, without having to waste much more time than the walk would normally take. Taking a walk down town and have some extra time to kill? Then this would be perfect.

3.4.3 On-going quests

The on-going quests idea is the idea which brings the game closest to “reality”. It has big scenarios spanning for a long time and the game can try to poke the player to actually play. The game assumes that the player is always available to play, maybe he or she gets an SMS in the middle of the night which starts a mission or give a clue to something.

The game is contained in a game area which is typically a city or any other area with comparable size. But not everything happens in the physical world, the game will use a lot of ether-based game play elements. Ether-based could be SMS, e-mail, the web, instant messaging etc.

Quests can be intricate, requiring a lot of research to know what the next step is and how to solve it. Not all quests will be given to the player in obvious ways, if the player is observant, he or she can find them by stumbling upon them (e.g. on the web). The system will push information to the players, not just wait for users to initiate contact. The system will push information to the player, not just waiting for users to initiate contact like completing a given tasks. Also, the system should keep good track of each player, like what quests have been assigned to whom, what has been completed, points and so on. That will help the game creators to monitor the whole state of the game closely.
4 Green Fox

The Green Fox project resulted in four different prototypes, here we will describe them in detail from a user perspective.

4.1 Vos

In traditional role-playing games (RPGs), players assume characters in a virtual world. The player control the characters, interact with the world, engage in battles and complete quests. Vos is a mobile phone RPG game that make the user interact with reality. With the help of a relatively precise GPS the game defines a region as the game world where the game will take part. The game uses GPS to determine where the player is in the game, in other words: when the player moves in the real world the character moves in the game world. The player can engage monsters which are roaming in the region and battle them, visit shops and talk to non playable characters (NPCs).

4.1.1 Main Menu

![Vos Main Menu]

Figure 1: The main menu of Vos

When the game is started, the main menu will be shown. If the Play button is pressed the client will try to log on to the server and fetch the player’s game data. If the transmission of player data is successful, the game will start. If Exit is pressed the game will exit.
4.1.2 Venture

When the game is started you enter the venture mode. Here the game uses a satellite image of the game region as map. The red shining dot represents the player, and its position will be updated accordingly when the player is walking around in the region. The shield symbol shows where an NPC is located and the blue house is a shop. The eye icons show the positions of monsters. All these positions have areas that act as triggers. If the player is within such an area the corresponding event will trigger.

While in venture mode the player can access the quest log, inventory and attributes screens. It’s also possible to exit the game from the menu.
4.1.3 Battle

If the player runs into a monster, the game will enter the battle mode. On the top of the screen, the player can see the feedback information of the battle. The feedback could for example be Player took 100 in damage, The monster took 13 in damage or Player got level up. The monster is shown in the middle part of the screen. At the bottom the player can see attributes like health and mana. The commands for attacking, casting spells, using items and escaping are to the right.

To defeat your opponent you can use the ordinary physical attack using, for example, a sword. You can also open the spell book and use a magic spell. Spells are arcane powers that allow a person to manipulate energy or create something from nothing. Usually spells inflict more damage than the normal physical attack. However, each cast of a spell requires mana points. The image above shows the player’s spell book interface. The numbers to the right indicate the mana points required for the corresponding spell. The more powerful a spell is, the more mana points it demands. If the player doesn’t know what a spell
does he or she can press the Description button in the lower left corner. That will bring up a dialogue box with an explanation of the spell.

![Inventory](image)

Figure 6: The inventory

This is the player’s inventory. During the game it is possible to use items from the item list. For example, the player can use health or mana potions, or give an item to an NPC. All items the player collects during the course of the game will be listed here. For each item, from left to right, it displays the icon, the name and the number of that item the player has. The player can choose the Description command to view an explanatory text about a specific item. To use an item press the select button.

4.1.4 Shop

![Shop](image)

Figure 7: The shop

In a shop the player can view and buy the shopkeeper’s goods. To buy an item, simply select it in the list and press the select button.
4.1.5 Dialogue

When the player talks to an NPC, an interface similar to the shop will be shown. The player can choose what to say and which reply to make. Different replies will cause the NPC to react differently towards the player. The NPC can give quests to the player.

4.1.6 Quest

The quest log records all the quests that the player has accepted. In Vos, we divided quests into three categories: active, completed and failed. Active quests are those that the player is currently carrying out. Successfully finished quests will be placed in the completed log and if the player fails a quests it will be put in the failed log. A quest can have a time limit. If the player cannot achieve the goal within that limit he or she will fail the quest.

To view the details of a quest, simply scroll the selection bar to that quest and press the select button on the device. A new dialogue box will pop up
4.1.7 Character sheet

Figure 10: The character sheet

In the game the player has a number of attributes, which are listed in the character sheet. To open it, choose Character from the in-game menu. The attributes are described as follows:

**Level** The player’s current experience level. When a player get level up the player’s attributes will also increase, which make the player more able to withstand damage, perform more powerful melee attacks and cast more advanced spells.

**Health** This value indicates the current amount of health points of the player, as well as the maximum amount of health points at the player’s current level. When the player’s health reaches 0, the player dies and the game ends.

**Mana** Whenever the player casts a spell, it requires a certain amount of mana. This attribute shows the player’s current mana as well as the maximum mana at the player’s current level. Note that if the current amount of mana is lower than the amount of mana required by a spell, the player cannot cast that spell.

**Strength** This value measures the muscle and the physical power of the player. A stronger player can deal more physical damage when using ordinary attack.

**Wisdom** This attribute describes how well the player can learn and reason. Therefore, a player with higher wisdom can understand a spell better and thus make it deadlier.

**Experience** Every time the player defeats a monster, he or she gains experience. When enough experience has been accumulated, the player advances to the next experience level and thus becoming more powerful. The first
value reflects how many experience points the player currently has and the second value shows the amount of experience the player needs to advance to the next level.

4.2 Minigames Web Interface

![Minigames Web Interface](image.png)

Figure 11: The minigames web interface

The main web interface for the minigames handles all minigames. In our case, that means the CamGame, since it is the only one that is implemented.

4.2.1 Player Mappings

Here you can see what game every player is mapped to. This is generally for debugging the message routing system in the minigames core.

4.2.2 AdminCommand

Here you can send messages to the clients or the server pretending to be a client, mainly for debug purposes. You can also send commands directly to the minigames core. For more information about this, see Section B.2.2. You can also see all the different logs that have been registered in the minigames core.

4.2.3 Communication Log

View the communication class’ log.

4.2.4 Minigames Core Log

View the main log of the minigames core.

4.2.5 Minigames Message Log

View the message log of the minigames core. Every message that has been send from or received by the server is stored here.
4.2.6 CamGame
Enter the mini game CamGame’s section of the web interface. See Section 4.3.

4.3 CamGame Web Interface

![Camgame](image)

**Game list**

<table>
<thead>
<tr>
<th>Game</th>
<th>Players</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camgame</td>
<td>sip:<a href="mailto:Fershid.Hassan11@innovation.com">Fershid.Hassan11@innovation.com</a></td>
</tr>
</tbody>
</table>

- **Camgame version**: 1.3 codename also fish
- **Games**
  - Game
  - Highscore
  - Player/Users
  - Songs
  - Admin/Command

- **Logs**
  - Camgame core log
  - Camgame message log

![Figure 12: The CamGame web interface](image)

4.3.1 Songs
To add a new song to the database, click on the **Add song** button on the songs-page. When you do, some text-fields will pop up, and you’ll be able to fill them in with the appropriate information about the new song. The melody section of the song is supposed to be on this format: “tone,length,tone,length,...” where tone is the tone to play in midi value. 60 is a “C”, 61 is a “C#”, 62 is a “D” and so on. length is the length of the preceding tone, eg. 4 is twice as fast as 8.

To modify an already existing song, click on the **edit** link for that song, and then click on the desired value you’d like to edit and then press **update**.

4.3.2 View High Scores
View the high score table. On this page you can sort the different results as well as filter them according to some criteria.

4.3.3 User Mapping
View a list of which users are registered in which games. You can also view a list of every users that has played the game at some point in time.

4.3.4 Games
Here you can view and administrate games. You’ll also see a list of users that are playing that game at the moment.
**CamGame Instance**  Here you will see information about the progress of a certain game instance, and which players that are currently playing it. You can also view a log for this particular instance of the game.

There are two options for administrating the game that you can also access on this page:

- If the game is running on normal or hard difficult mode, you may add time to the time limit (You cannot do that while playing on easy mode since there is no time limit then).
- Delete the game.

4.3.5 CamGame Core Log
Displays the output log for the CamGame core, i.e. the container for all the instances of CamGames currently being played.

4.3.6 CamGame Message Log
The IMS communications log. It shows all the traffic communicated to and from the minigames server using the IMS platform.

4.4 CamGame client
Here we will introduce the Graphical User Interface of CamGame with the help of screenshot images.

4.4.1 Start a New Game

![Figure 13: The CamGame main menu](image)

This is the main CamGame GUI, the cross keys and the center confirm key are used for navigation.

The player can choose **Play** to start a new game, or **Exit** to exit the game.

After the player has started a new game, the difficulty level form will be shown. In this form, the player can choose at which difficulty level to play the game.
In the easy level, the player is requested to take a picture in a single color. For example, the player can be asked to take a picture in red.

In the normal level, the difficulty will be increased a little, the pictures can now be in up to two different colors, though single colored pictures are still possible.

In the hard level, the player is not only required to take a picture in up to three different colors but also has a time limitation. This means that the player have to finish each song within a set time limit, otherwise the server will terminate the game actively. At this difficulty the server will every few minutes (e.g. 5min intervals) send a message including the time left to play the game. The user will see the message in the phone in a dialog box.

4.4.2 Choosing Difficulty

![Difficulty selection](image)

Figure 14: Difficulty selection

After the player has selected the difficulty level for the game, the client will connect to the IMS game server. The next form shows a song list that contains all the songs which are available for play on the server.

Choose StartGame to start the game with the selected song.

Choose Menu to get more options. In this case, the player can select play songs to play the selected song.
4.4.3 Making Music

Figure 15: Song list

Figure 16: Song list menu

Figure 17: Making a song
Now the player will start creating the song!

In this form, the player will see the notes from the original song, which is what the player should aim at creating. Also, the picture at the bottom left shows what the player is supposed to take a photo of. The bottom right picture shows the picture that the player has taken. The more similar the taken picture is to the target on the left, the more accurate note you will create in the song. Press OpenCamera to open the camera screen in order to take a picture.

![Making a song](image18)

Figure 18: Making a song (menu)

The Menu button will provide more options, such as playing the original song or the song that the player has made so far.

4.4.4 Finish a Game

![The highscore](image19)

Figure 19: The highscore

Once the player chooses Done from the menu, the game will be finished and a high score will be shown.

This form contains some interesting statistics about the game that was played.

From the menu, the player can choose start a new game or exiting the game.
4.5 Tourist Guide Client

The main client interface consists of three screens (map, places, description) which the user can switch between by selecting a tab at the top of the screen.

4.5.1 Splash Screen

![Splash Screen](image)

Figure 20: The splash screen

When the client starts, it will display the Green Fox logo while it connects to the server and the GPS unit. The icons in the lower right corner will turn colored as the connections are established.

4.5.2 Tour Selection

![Tour Selection](image)

Figure 21: Tour selection

When the client is done loading it will display a screen with a list of tours where the user can select what tour to take. The user also has the option to view a more detailed description of the tour.
4.5.3 Tour Details

This screen displays a large picture representing the tour as well as a description of it.

Figure 22: Tour details

4.5.4 Map

The user can view a map of the current tour with all the places marked. A path is drawn where the user has walked on the map. Places that the user has visited are marked green, unvisited places are marked in red and the currently selected place is marked in light blue.

Figure 23: Tour map
4.5.5 Place List

![Place list figure]

Figure 24: Place list

This is a list of all the places in a tour, each place in the list also has a thumbnail picture and a short description. The user can select a place to view a detailed description of that place.

4.5.6 Adding Places

![Adding a place figure]

Figure 25: Adding a place

This feature allows the user to submit a new place to the server. This includes the position, which is taken from the phone’s GPS unit, name and description of the place. The user can also use the phone’s camera to take a picture of the place.
4.5.7 Place Details

When a user walks within the radius of a place on the map, the phone will vibrate and bring up a detailed description and a picture of that place. The user is also able to view the details of a place by selecting it in the place list.

4.6 Tourist Guide Web Interface

The Tourist Guide web interface is intended to be used by an administrator to manage and create tours and places available in the Tourist Guide.
4.6.1 User Login

When first entering web interface, the user is asked for his or her SIP id to be able to login to the system. The id is verified with the database and then the user should be able to perform tasks using the web interface.
4.6.2 List Tours

**Figure 28: Tour list**

The user can view a list of the tours which includes information such as name, description and a picture for each tour. This screen allows the user to view, edit or delete a tour.

4.6.3 View Tour

**Figure 29: View tour**

<table>
<thead>
<tr>
<th>Picture</th>
<th>Name</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="the_green_hat_tour.png" alt="Image" /></td>
<td><strong>The Green Hat Tour</strong></td>
<td>The surroundings of GHP.</td>
<td>View Edit Delete</td>
</tr>
<tr>
<td><img src="greenfox_tour.png" alt="Image" /></td>
<td><strong>Greenfox Tour</strong></td>
<td>See the Greenfox town!</td>
<td>View Edit Delete</td>
</tr>
<tr>
<td><img src="tou_of_itc.png" alt="Image" /></td>
<td><strong>Tour of ITC</strong></td>
<td>A shorter tour of only ITC.</td>
<td>View Edit Delete</td>
</tr>
</tbody>
</table>

The user can view a list of the tours which includes information such as name, description and a picture for each tour. This screen allows the user to view, edit or delete a tour.
The user can view details of a tour such as name, description and picture. This also allows the user to see a map of the tour and where the places that are in the tour are located.

### 4.6.4 Add Tour

#### Step 1: Add Tour

<table>
<thead>
<tr>
<th>Tour info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Abstract</td>
</tr>
<tr>
<td>Picture</td>
</tr>
<tr>
<td>Thumbnail</td>
</tr>
</tbody>
</table>

#### Map

![Map of tour locations](image)

**Figure 30: Add tour**

Adding a tour consists of two parts, first the user enters information such as name, description and picture, and selects a map area that should be used for this tour. In the second part, the user can select which places that belongs in the tour by picking them from a list of all places that are within the selected map area.
4.6.5 Edit Tour

Figure 31: Edit tour

The user can edit details of a tour such as name, description, picture and what places that should belong to a tour.

4.6.6 List Places

Figure 32: List places

The user can view a list of all places, which includes information such as name, description and a picture for each place. This screen allows the user to view, edit or delete a place.
### 4.6.7 View Place Details

#### Place Details

<table>
<thead>
<tr>
<th>Picture</th>
<th><img src="image" alt="Picture" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thumbnail</td>
<td><img src="image" alt="Thumbnail" /></td>
</tr>
<tr>
<td>Name</td>
<td>Gustaf Vasa kyrka</td>
</tr>
<tr>
<td>Description</td>
<td>The church Gustaf Vasa is situated at Odenplan, Stockholm. It was inaugurated in 1906, and is Stockholm largest church with 1,500 seats. The grandios altar piece was constructed by Scherhag Precht 1729-31.</td>
</tr>
<tr>
<td>Abstract</td>
<td>Stockholm's largest church.</td>
</tr>
<tr>
<td>Latitude</td>
<td>59.342494</td>
</tr>
<tr>
<td>Longitude</td>
<td>18.047636</td>
</tr>
<tr>
<td>Radius</td>
<td>50.0</td>
</tr>
<tr>
<td>Owner</td>
<td><a href="mailto:spps4am.Tavalkt28@innovation.com">spps4am.Tavalkt28@innovation.com</a></td>
</tr>
</tbody>
</table>

Figure 33: View place details

When the user chooses to view a place, this screen will be shown, where he or she can view all details of the place, such as name, description, picture, and the location of the place on the map.
4.6.8 Add Place

The user can add a place by entering its information in the form of name, description, picture etc. The user can choose the location of a place either by entering the coordinates or by marking the location on the map.
4.6.9 Edit Place

The user can edit the values of a place, such as name, picture, description and location.

Figure 35: Adding a place
4.7 Agent SQ client

4.7.1 Splash Screen

Figure 36: Agent SQ splash screen

When the client starts, it will display the Green Fox logo while it connects to the server and the GPS unit. The icons in the lower right corner will turn colored as the connections are established.

4.7.2 Main Menu

Figure 37: Main menu

When the client has connected, the player is presented with the main menu where he or she select one of these options; ask for a new mission, view the collected clues, view statistics, or exit the game.
4.7.3 Ask For a Mission

Figure 38: Asking for a mission

![Image showing a screen asking for a mission]

Figure 39: Accepting a mission

![Image showing a screen accepting a mission]

If the player asks the game for a new mission it will check with the server if there are any missions available close by. If there are any missions available, then the player is presented with a screen informing of the time limit to complete the mission and presenting the option to decline or accept the mission.
4.7.4 Mission Briefing

If the player has accepted a new mission the client will display a description and/or a picture to brief the user on the new mission. This screen also displays the number of points will be awarded if the mission is completed, and the amount of time left. The goal of each mission is to find a location, now it’s up to the player to find it as fast as possible!

4.7.5 Distance Gadget

To find the location and thus complete the mission, the player can use the Distance Gadget which indicates how far the player is from the location. The gadget will grow brighter the closer the player gets to the target. The distance in meters is also displayed in the middle of the gadget. This screen also displays the time left of the mission and how many points that will be awarded when the mission is completed.
4.7.6 Clues

When the player completes a mission he or she will get rewarded by getting a clue. A clue is a letter which must be used to form a word in order to complete a quest. The player can at any time guess what word the clues will add up to. Guessing wrong will generate a penalty, so be careful. When the correct guess is made, the quest is completed.

4.7.7 Player Statistics

The player can view his or her total score on this screen.
4.8 Agent SQ Web Interface

4.8.1 List Quests

![Figure 44: A list of all quests](image)

The user can view a list of all quests. A quest is basically just a word which the player can collect clues for and then guess.

<table>
<thead>
<tr>
<th>Quest</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>artemis</td>
<td>Delete</td>
</tr>
<tr>
<td>greenfox</td>
<td>Delete</td>
</tr>
<tr>
<td>pollax</td>
<td>Delete</td>
</tr>
<tr>
<td>uppsala</td>
<td>Delete</td>
</tr>
<tr>
<td>poseidon</td>
<td>Delete</td>
</tr>
<tr>
<td>j5l</td>
<td>Delete</td>
</tr>
<tr>
<td>j5lmust</td>
<td>Delete</td>
</tr>
<tr>
<td>j5lgran</td>
<td>Delete</td>
</tr>
</tbody>
</table>

4.8.2 Add Quest

![Figure 45: The add quest page](image)

The user can add quests using the web interface by typing in the word.
### 4.8.3 List Missions

<table>
<thead>
<tr>
<th>Picture</th>
<th>Place</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="House 1" /></td>
<td>House 1</td>
<td>Find the place where all computer scientists hang out on Polaskisäcken. But quick!</td>
<td>View Edit Delete</td>
</tr>
<tr>
<td><img src="image" alt="House 2" /></td>
<td>House 2</td>
<td>Find the place where wizards can help you with buying a new computer. In the old times the mathematicians dwelled here.</td>
<td>View Edit Delete</td>
</tr>
<tr>
<td><img src="image" alt="Laboratory" /></td>
<td><strong>The Ångström Laboratory</strong></td>
<td>Physicists dwell here. In the old time it was the place for the exercise field used by soldiers.</td>
<td>View Edit Delete</td>
</tr>
</tbody>
</table>

Figure 46: A list of all missions

The user can view a list of all missions, from this form he or she can delete or edit missions.
4.8.4 Add Mission

The user can add a mission by entering a description and/or adding a picture. The user also has to pick a place which should be associated with the mission (note that the places are shared between the Tourist Guide and Agent SQ). To complete a mission the player have to locate and go to this place.

![Add Mission Page]

---

**Mission info**

- **Briefing**
  - [ ]

- **Picture**
  - Choose...

**Mission place**

<table>
<thead>
<tr>
<th>Use</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑</td>
<td>Angstrom art</td>
</tr>
<tr>
<td>☑</td>
<td>Cottage Foundation</td>
</tr>
<tr>
<td>☑</td>
<td>Daws</td>
</tr>
<tr>
<td>☑</td>
<td>Fallen tree</td>
</tr>
<tr>
<td>☑</td>
<td>FLAG</td>
</tr>
<tr>
<td>☑</td>
<td>GHP lair</td>
</tr>
</tbody>
</table>

Figure 47: The add mission page
5 System Description

This section deals with the technical parts of our project. First is a general overview of the common system architecture of all our prototypes, followed by more technical details for each of the four prototypes.

5.1 IMS

All application prototypes make use of the IP Multimedia Subsystem (IMS) by using the infrastructure and APIs made available by the Mobile Java Communication Framework (MJCF). The MJCF offers APIs for both clients and servers using Java ME and Java EE respectively. The server software is deployed (as so called servlets) on the SailFin application servers hosted by the MJCF. The client software is deployed on SonyEricsson C702 mobile phones. The SonyEricsson C702 supports a Jp8 Java engine and the JSR 179 location API (with a built-in GPS receiver) and is therefore a sufficient client platform for all our prototypes.

5.2 Common Application Structure

All four implemented applications share a common structure to help with quick prototyping, as pictured in Figure 48. The structure of an entire application is split roughly into three parts: the client part running on the mobile phone, the servlet part running on an IMS application server and a part for database and web access running on an external server. The communication between the clients and the servlets go through the IMS network using MSRP sessions or IMS page messages, and the communication between the servlets and the external database server use regular Internet connectivity.

5.2.1 Common Communication Framework

We have written a common communication framework that all prototypes use. The communication framework provides a higher level of abstraction for communicating over the IMS network; this framework, in turn, is implemented using the MJCF APIs.

There are corresponding client and server side versions of this framework, that are designed to offer near identical APIs. The framework helps with establishing a connection between endpoints and then provides simple means of transmitting application-defined messages that are automatically formatted by the sending side and parsed by the receiving side. Each message has a specified label and zero or more message data parts. Each message data part is any collection of key-value pairs. There is also support for sending images using the communication framework. The images are transmitted as JPEG data, but are automatically converted into native LWUIT Image objects upon reception at a client, and to standard Java BufferedImage objects upon reception at a server. There is also limited support for IMS presence in the communication framework, that we regretfully have not had time to finish completely.

The framework is flexible enough to offer a simple way of implementing application-specific communication protocols. Indeed, each of our prototypes have defined their own communication protocols in this way.
Figure 48: The common framework structure.

See Appendix D for details on the message format.

5.2.2 Client Structure

The software running on the mobile phones is written in Java ME, and all four prototypes use the third-party library LWUIT to create their graphical user interfaces. The user interfaces are very application-specific, so all prototypes implement them separately.

Each prototype’s client uses the common communication framework described above.

5.2.3 Servlet Structure

The servlets are written in Java EE and run on the SailFin application servers provided by the MJCF.

We have written a common web framework for the servlets that let us, for an example, log any output from a servlet to a web page for debugging and maintenance reasons. All prototypes use both the common communication and web frameworks on the server side.
5.2.4 Database/Web Server Structure

All prototypes make use of a MySQL database that runs on an external server, and is accessed through the Internet by the servlet software using the Hibernate object-relational mapping library. The database store things like player information and high-scores, images and descriptions for game missions or tourist guide tours and melodies for the CamGame.

The Urban Exploration prototypes also make use of a web server on this machine to offer a web interface to their applications. The web interface runs on Apache Tomcat and uses the Stripes web presentation framework. The CamGame prototype instead has an extended web interface using the web framework on the servlet as described above.

5.3 Prototype-Specific Structure

Beside the common structure described above, each application prototype has a specific technical design and structure. An overview of these is given here.

The diagrams below show the Java packages of each software component, and a high-level hierarchy between them, as given by the uses relation. The uses relations are meant to give a general understanding of the software architecture, and do not correspond directly to any programmatic relationships. In some packages are also given a number of key classes or interfaces of that package, who merit specific mentioning.

All software components include one of the se.uu.greenfox.client or se.uu.greenfox.server packages, outlined in Figure 49. These are the packages for the common communication framework, described above. The key elements of these packages are the Communication class, which implements the actual communication using the MJCF APIs; the MessageProperties class, which is the internal representation of a message data part; and the GameCore interface, which is the callback interface implemented by the class using the Communication class. On the server side are also the Log class, which handles logging of events, and the WebInterface class, which provides a web interface to the servlet on the application server.

Figure 49: The structure of the common framework.

The communication protocols for each prototype are detailed in Appendix D, and the database structures for each prototype are detailed in Appendix E.
5.3.1 The Tourist Guide

The Tourist Guide stores no guide content on the mobile phones, instead data is requested from the server when it is needed. The content is stored on a MySQL database hosted on an external machine, and the server (servlet) software fetches data from there using Hibernate before sending it along to the client.

The places-of-interest are the core of the tourist guide content, each place can belong to any number of different tours. When the client is started and has connected to the server, it receives a list of all available tours to choose from. The client can start a tour or view detailed descriptions about each one. When a tour has been started, a list of all places-of-interest in that tour is sent to the client along with a map. The user can see himself or herself drawn out on the map, along with all the places. When the user has arrived within some specified radius of a place, or if the user requests it, some information about that place will be sent from the server and pop up on the client’s screen.

The phone’s GPS receiver is used to get periodic updates of the user’s location. It is set to update as often as possible, which on the SonyEricsson C702 means about once every other second. The user can also add new places using the phone, by simply taking a picture using the phone’s built-in camera, and optionally writing some description. The location of the added place is determined automatically using the GPS data. The data for the new place is then sent to the server which adds it to the database, where it can later be edited or added to some tour using the web interface.

Tourist Guide Client Component The client software is structured as shown in Figure 50, and it uses the MSRP version of the common communication framework.

![Figure 50: The structure of the Tourist Guide client.](image)

The **ui Package** The ui package contains, and abstracts away, all the logic for handling the user interface. The GUI class is the main class in this package,
which acts as a hub for all calls to and from the user interface code. The
three main interface modes that are switched between on the client are
the loading screen, the screen where the user selects or view information
about different tours and the screen where the user views places and the
map. These are handled by the IntroPane, TourPane and PlacePane
classes respectively. The class using the GUI class should implement the
UIUser callback interface that is used to signal interface events.

The connection Package The connection package contains all the logic for
communicating with the server. The most important class is the
ConnectionHandler, which implements the Tourist Guide communication
protocol, with the help of the common communication framework residing
in se.uu.greenfox.client. The methods available for the rest of the appli-
cation from ConnectionHandler are defined in the ConnectionSource
interface. Since the communication is asynchronous, the class using the
ConnectionSource should implement the callback interface
ConnectionUser.

The location Package The location package handles all positioning logic
using the GPS receiver. The main class that is responsible for reading data
from the GPS receiver is the LocationHandler. Once a LocationHandler
object has been created it starts to listen to the GPS receiver and passes
along any updates to its owner, which should implement the LocationUser
callback interface. The LocationHandler can also be given specific areas
to keep track of, and sends an alert to its owner once any such area has
been reached. The services available from the LocationHandler are spec-
ified in the LocationSource interface.

While data read from the GPS receiver is on the form of global, absolute
coordinates, the LocationHandler abstracts this away and passes along
positioning data in an internal format using a Cartesian coordinate system
measured in metres. The Cartesian coordinates are calculated using the
very first position read from the GPS receiver as the origin. This system
simplifies the logic in the rest of the code handling positions. In particular,
the work of the Map class, which handles drawing maps with places and
users on them, is much simplified.

The util Package The util package contains helper classes that don’t really
fit in anywhere else.

Tourist Guide Server Component The Tourist Guide server is structured
as shown in Figure 51. It is basically an interface to the MySQL database,
residing on an external server, which the clients use to fetch data. All game
related operations are triggered only by receiving messages from a client, and
the server will respond appropriately.

The server uses the common se.uu.greenfox.server package in the MSRP
version since it handles large amounts of data due to picture sending. The
logging and web interface parts of the common package are used to provide a
simple log interface for the servlet.

The se.uu.greenfox.ue.touristguide Package The most significant class
in this package is the TouristCore which extends the GameCore in the
shared server package. This class contains a method for each message type that it can receive from the clients, and those will fetch the appropriate data from the database and send back a response. Every operation in the class is triggered by client messages.

5.3.2 Agent SQ

The Agent SQ game stores the game content on the server and uses the client for instance data, e.g. measuring time and calculating the score of each completed mission. The database contains a set of quests that are comprised of one or several missions. Each mission is associated with a place. Also, information about each player is stored in the database, e.g. the total score and which quests and missions that have been accomplished.

The game uses the phone’s GPS receiver, since the game is location based, so that the player is only offered missions that are in the player’s vicinity without having to manually type in location information. The GPS receiver is also used to verify that the player has actually reached the mission’s goal, and to give constant clues by reporting the distance to the goal in real-time.

The communication between the server and the client is event driven, where the server responds to client requests. The client pulls new missions, stored scores and new clues (when a mission is completed) from the server. The client push information about finished missions along with the elapsed time to the server.

On a separate server (not a server provided by MJCF), a web interface is hosted that allows administrators to add new missions (linking them to existing or new places) and add new quests. The web interface currently doesn’t support any functionality to monitor players progress as they are playing.

That same server also hosts the MySQL database.

Agent SQ Client Component  The client software of the Agent SQ game is structured as shown in Figure 52. It uses the MSRP version of the common communication framework.

The game package holds classes related to the game logic, which is used in turn by the ui package and of course by the main (<<default>>) package which contains the very important ShortQuest class (the MIDlet class). The util package has a facility for timing events that is used both by the game logic (for the countdown of a mission) and ui (for non-event triggered updates on the screen). The connection package handles everything that has to do with the connection to the server, which in turn uses the client common package
se.uu.greenfox.client. The last package is the one that takes care of GPS location positioning, and is naturally called location.

The **ui** Package The **ui** package contains classes for visually presenting the game. It has one "root" class called **GUI**, which is used to route all events and which transfers information to and (partly) from the interface.

Information is routed back to the rest of the program via the **UIUser** interface, which is implemented by the main **ShortQuest** class.

**DistanceGadget** is an abstract class that only contains static methods for creating an image of the distance gadget with the help of GPS distance information.

All the different “panes” (views/windows on the screen) all extend the **CommonPane** class as it contains some very general stuff common to nearly everyone of these.

**ActiveMission** is the class that holds the instance data that concerns the **GUI** about the mission that is currently being played.

The **game** Package **GameState** contains instance data for a mission that does not concern the graphical parts (e.g. mission timer and score). The **GameUser** interface lets this package signal game instructions (e.g. time’s up!) back to the **ShortQuest** class. The rest of the classes are data containers for missions and quests etc.

The **util** Package The important **TimerFacility** class, which has one global instance, takes care of Java timers, like storing them with an ID so that
they can be easily removed when not needed anymore. Timers are used for the mission timeout events, and also for periodic updates of the GUI.

The connection Package The ConnectionHandler class is the class used by the ShortQuest class to handle all communication with the server. It is defined by the interface ConnectionSource. The ShortQuest callbacks that the ConnectionHandler needs are defined in the interface ConnectionUser.

The location Package The LocationHandler class and the two interfaces LocationSource and LocationUser are structured in the same way as their counterparts in the connection package. This package takes care of proximity alerts and the retrieving of GPS positioning data. As opposed to the location package of the Tourist Guide software, the lack of a map in this game makes it unnecessary to translate positions to an internal Cartesian system.

Agent SQ Server Component The Agent SQ server is structured as shown in Figure 53. It uses Hibernate to fetch and store data in a MySQL database hosted on an external machine. This game builds its missions by using places already stored for the Tourist Guide in the same database, so it uses the se.uu.greenfox.ue.touristguide package for those database accesses. The se.uu.greenfox.server package contains the shared server components such as the communication framework and logging facility.

![Figure 53: The structure of the Agent SQ server.](image)

The se.uu.greenfox.ue.shortquest Package The most important class in this package is the QuestCore which extends the GameCore in the shared server package. It’s the brain of the game in the server, handling rules and requests by the clients etc. Apart from that, the package also contains classes for dealing with the database mappings.

5.3.3 Vos

UI description

BattleUI (BattleUI.java): The main LWUIT components we use in the battle UI are Label and Button. Images can be loaded onto both components. So if an UI element doesn’t require player input, we use Label. Otherwise, Button is the choice.
When the UI is initialized, the main form is created together with all the other components and the images are loaded onto their corresponding components. We use `CoordinateLayout` as the layout of the main form so that we gain more control over where we want to place a component. The `IndicateBar` is a customized `Component` class. It overrides the `paint` method so that it can draw a colored rectangle to represent either a health bar or a mana bar. Meanwhile, it provides a method called `update`, which can update the current value of the bar so that the `paint` method can color a right portion of a bar.

Another thing worth notifying is that when it is a monster’s turn, we don’t want the player to have any input. Therefore, we disable the four buttons when the player turn is over and enable them again when the monster’s turn is over. But if we simply disable the four buttons, Java will throw an `ArrayIndexOutOfBoundsException` exception because there is no focussable component in the main form when it tries to update the focus. To fix this problem, we added a dummy button and placed it out of the screen. Whenever the four buttons are disabled, it is enabled and vice versa. At last, the player’s input events to those four buttons are captured by the `PlayerController` class in the `rpg.battle` package. The `actionPerformed` method of that class then handles each event accordingly.

**MapUI (MapUI.java):** Since we want to place a picture of a region in the real world as a map, so that the player can use it to guide him or her during the adventure, we set the picture of the place as the background image of the form. In order to let this background show, we set the contentpane of the form transparent.

The menu items are `Command` objects in LWUIT and when we add them to the form, LWUIT will automatically organize them into a popup menu. The player can access the inventory, the quest log and the character sheet from the menu. The inventory is an object of the ListForm class and the quest log is an object of the QuestLog class. We will describe them in their own sections later. As for the character sheet, we make use of the `GlassPane` concept. As the name suggests, it is like a transparent overlay the developer can draw things on it without destroying the contents below. The whole character sheet is drawn by the `paint` method of the `AttributeGlassPane` class.

The player is represented by a shining red dot in the map. Monsters, quest items, the NPC and the shop are represented by their own icons respectively. Except the player mark, all the symbols are labels loaded with images. In order to get that shining effect, the player mark uses an animation to change the label’s background colour periodically. The colours we use are red with different gray levels. Finally, the map symbols’ positions are converted from their respective GPS positions, so we are sure that they reflect the positions in the real world.

**Inventory and Spell Book UI (ListForm.java):** This UI contains only a form with a list in it. It is used in the inventory and the spell book. Like the map UI, an image is set as the background of the main form. The working mechanism of the `List` component is a little intricate. It
takes advantage of Swing’s style of MVC design patterns. For the Model part, we create a class called ListItem. It contains the name, icon and description of an item. We just need to pass objects of this class to a list and it will take care of them. The difference between the List component and other components is if you simply add a List object into a form, it won’t be shown. To show a list, we need a class that implements the ListCellRenderer interface. When we implement the class, we define how a cell (item) of the list should be drawn and then set an object of it as the cell renderer of the list. This process is like defining the View part of the MVC model. With these two parts, a list should work properly.

However, the List component in LWUIT has a defect. When we navigate the selection to the ends of the list, if we continue navigating it in that direction (that is, the selection goes out of range), the list will lose focus. If there are other components in the same form, this won’t cause any inconvenience. But in our case, the whole form only has a list. Then when the list loses the focus, the user will find that the controls become irresponsible. To cope with this, we introduce a dummy button and then place it out of the screen so the player won’t see it. This dummy button will receive the focus when the list loses it, which eliminates the defect of the List component.

Quest Log UI (QuestLog.java): Because of the needs for this game, we need a UI that can organize and display three categories of quests, namely, active, failed and completed. The TabbedPane class naturally becomes our choice because quests of different categories can be organized into corresponding tabs. All quests of the same type are put in a List object. Quests are therefore just cells in the corresponding list. We use three Container objects to enclose the lists. These Container objects are in turn enclosed by a TabbedPane object.

Main Menu UI (MainMenu.java): The main menu of our game is very simple, there are just two choices with a background image. Both choices, play the game or exit it, are actually buttons loaded with images. In order to have the highlighting effect when a choice is selected, we prepare two pairs of images for both buttons, namely, selected images and normal (un-selected) images. Based on the current status of each button, the proper image is loaded. When the player starts the game for the first time, a login dialogue box is shown displaying the login progress in this menu. After login, if the player presses the play button, a label will be added to the main menu telling the player to wait. After a short period for preparation, the map will be shown and the game starts.

Figure 54 shows a brief structure of the Vos client with main packages and their important classes and interfaces.

Figure 55 shows a brief structure of the Vos server with main packages and their important classes and interfaces.

Battle system The main class of the battle package is BattleState. This class represents the state we change to when a battle is started. It takes care of all the high-level functions of a battle, and controls the flow of it. Every battle
is fought between a number of creatures (currently, only two creatures can fight, the player and an enemy, although it could be extended quite easily to allow an arbitrary amount of enemies), and each creature is controlled by a controller. The controller tells the creature what it should do. Such a controller can be either the computer (like for the enemies), or the player.

**Creatures** The *Creature* class represents some sort of “creature”. This creature could be an animal, a human being, a robot or some other kind of entity. These creatures are used when fighting in a battle and when walking around the map. A creature has a number of attributes. These include health, mana, strength, wisdom, experience and level. It can also hold items, spells and weapons. *Creature* provides methods for altering the state and attributes, but does not contain logic for fighting in a battle. This is instead implemented by *BattleCreature*.

*BattleCreature* represents a creature that fights in a battle. It extends *Creature* because it is a creature, with additional behaviour that is needed in battles. The main addition is that battle creatures are tickable (implement the *Tickable* interface), so that they can “think” and perform actions. The actions a battle creature can perform are attacking, casting...
spells and using items. These are performed using the `performAction` method, which takes the action to perform as an argument.

The method `handleAction` is the counterpart of `performAction`. It is called when another creature performs an action on this creature, and takes that action as argument. This creature is then supposed to react to the performed action and handle the repercussions of it. For example, if the other creature attacks or casts a spell on this creature, the damage taken will be applied in the `handleAction` method.

When a creature is performing an action, it may take some time before that action is finished being performed. This is to make it more realistic, and also for allowing such things as animating the action being performed (although this is not currently done).

Flow of Battles The `BattleState` class keeps track of the controller for each creature that fights the battle. Since the battles are turn-based, only one controller is active at any time. When a controller is active, it gets to execute and give commands to the creature it controls. When it has finished, the next controller becomes active and the old one is stopped.

The `tick` method of `BattleState` handles the execution of the controllers and the creatures. This method begins by ticking all the creatures in the battle. It then ticks the controller that is currently active, but no other controller. Because of the controller abstraction, this controller could be either the computer or the player, but the `BattleState` class does not need to know which one it is.

Note that all the creatures are ticked in every tick of `BattleState`. This is because the creatures are considered to be independent entities, that can think of their own. The controllers tell the creatures what to do, but it’s the creature’s responsibility to act on their commands.

Creature Events The battle state listens to the events of the creatures fighting in the battle (by implementing the `CreatureListener` interface). This
lets it find out when for example a creature takes damage or dies, and react to it appropriately. This is the way the messages on the top of the battle screen are triggered; whenever a creature is hurt, a message indicating the damage taken for the creature is shown.

It also lets the battle state find out when a creature has finished performing an action. When the creature of a controller has performed the action given to it by the controller, that controller is stopped and the next controller in line gets to execute.

When a creature has died, the battle is ended (since we only have implemented battles between two creatures). Depending on if the creature that died was the player’s or the enemy’s, the battle is either won or lost.

Controllers The Controller class represents a controller that controls a creature in a battle. There are two different subclasses of Controller, namely PlayerController and ComputerController. These respectively represent the player’s and the computer’s controllers. Each controller has an associated creature, that it tells what actions to perform in the battle. The controller is supposed to assign an action to its creature when it is ticked, in whichever way it wants.

The player’s controller chooses this action by listening to the graphical user interface. It adds the commands for the different actions possible to the BattleUI instance of the battle state it is a part of, and then reacts whenever commands are executed in the user interface.

The computer’s controller currently just tells its creature to attack its opponent, although this could be extended with some kind of intelligence to make it more interesting.

Actions Actions performed by battle creatures are represented by the Action class. This abstract superclass just contains information about the source and target of an action (who performs the action and on whom it is applied, respectively). Note that the source and target can be the same creature, for example when a creature uses an item on itself. Specific actions are subclasses of the Action class. The currently available subclasses are AttackAction, SpellAction and ItemAction. AttackAction represents the action of attacking another creature with a weapon, SpellAction represents casting a spell on another creature, and ItemAction represents using an item on a creature. These subclasses contain extra information that depends on the kind of action it is, like the spell cast or item used.

Items, Spells and Weapons As mentioned above, creatures can hold items, spells and weapons. These are represented by the Item, Spell and Weapon classes, respectively. These classes are general superclasses that are extended by more specific classes. For example, items used for curing health or mana are represented by the CureItem class.

For handling sets of items and spells, we have implemented the ItemSet and SpellSet classes. These hold collections of items and spells, respectively.
Map  The Map class is used in two ways: it represents the state we are in when walking around in the world, and it also represents the game world and all the objects inside it. It contains all the objects that can be seen while walking around in the world (so called “map objects”). These objects include the player itself, enemies walking around, shops and non-player characters.

Game World All objects that can be seen on the map are instances of the MapObject class. The Map class keeps track of all of the map objects in the world, including the position they are currently in. That is, map objects do not themselves keep track of their positions—this is done by the map instead. If any object should be added, removed or moved, it has to be done through the corresponding method in Map, namely addObject, removeObject or moveObject, respectively. Letting the map handle this makes it much easier to keep the world consistent and to handle any side effects that occur when objects are modified.

The map keeps track of when map objects start overlapping each other, and then notifies the overlapping objects. This makes it easy to code the map objects, since they don’t have to handle collision detection themselves. It also notifies when two objects that were previously overlapping are no longer overlapping each other.

These checks are done whenever a map object is added, removed or moved, by calling the checkForOverlaps method. This method simply checks for new overlaps between objects, or whether any objects that did overlap no longer do so. It then calls either the startedOverlapping or stoppedOverlapping methods of the MapObject class to inform the objects of the respective event.

Position Handling The map does not use GPS coordinates directly to represent the positions of objects in the map. Instead, the map—as well as most other parts of the game—only uses an internal coordinate system, which we call “game coordinates”. Game coordinates are just normal cartesian coordinates with x and y components. Because they do not depend on the GPS, we can play the game in an emulator using only the keys to move around in the game.

Of course, we want to use the GPS in the real game. Therefore, we must be able to convert GPS coordinates into game coordinates. The retrieving and conversion of GPS coordinates into game coordinates is taken care of by the GPSPositionRetriever class. The relation between GPS coordinates and game coordinates is as follows: The origin of the game coordinate system (the coordinate (0, 0)) corresponds to some GPS position, which currently depends on where the player started the game. A game coordinate (x, y), where x and y are positive, is then interpreted to be the point in the real world lying x metres to the east and y metres to the north of this origin point. If they are negative, they will be to the west and south, respectively.

Since the origin differs between each run of the game, the game coordinate corresponding to a certain GPS position will also change each time the game is run. Therefore, the game can only use these coordinates in a relative manner, and not absolute. The reason for having the origin be
different each time the game is run is because of how the relative coordinates are calculated. If we had a fixed origin, we could get large numerical errors if the origin was set a long way from where the player is playing the game.

**Position Retrievers** The *PositionRetriever* interface represents a retriever of positions. It retrieves positions for the player’s location using some source and then reports whenever a new position for the player is discovered. The reported position is given in game coordinates. Using the observer pattern, other classes can listen to these position updates through the *positionUpdated* method.

There are two different implementations of *PositionRetriever*, namely *GPSPositionRetriever* and *FakePositionRetriever*. As the name suggests, *GPSPositionRetriever* retrieves positions using GPS. On the other hand, *FakePositionRetriever* fakes positions by keeping its own internal position and then reporting this as the player’s position from time to time. It allows this internal position to be changed by pressing the navigation keys on the mobile phone. In this way, we can move around in the game without having to physically move. (Note that we could also use the emulator’s built-in GPS functionality to fake positions, but then we can’t use the keys to move easily.)

*GPSPositionRetriever* also employs some simple methods to guard from the jitter in the positions retrieved by the GPS device. It uses an internal class *PlayerPositionFilter* that can accept or reject new positions, and calculate a corrected position given the player’s current position and the newly received GPS position.

**Map Objects** Objects on the map are represented by the *MapObject* class. This is an abstract superclass that is extended by all the actual map objects.

Each map object has a radius, which gives the size of the bounding circle of the object. This circle is used for calculating when two map objects overlap. Whenever two map objects’ circles start overlapping, the *startedOverlapping* method is called, and when they stop overlapping the *stoppedOverlapping* method is called.

Map objects are ticked in each tick by the *Map* class. This allows them to perform actions, for example moving around.

A map object can have a context sensitive command associated with it. Whenever the player overlaps the map object, the command is then shown to the player on the screen. If the player chooses this command, the *commandExecuted* method of the map object is called. A command is set by specifying the string that will be shown on the screen to the *setContextSensitiveCommand* method of *Map*.

The following are the subclasses of *MapObject* and what they represent:

**CreatureObject**
- Represents a map object that has an associated creature. It only provides methods to set and get the creature, and also automatically removes the map object from the map if the creature has died.
PlayerObject: Extends CreatureObject and represents the player on the map. It does not contain any special functionality, since the moving of the player is done by the Map class using a position retriever, and other actions initiated by the player are also handled in the Map class. The creature of the player object is the player’s creature.

EnemyObject: Also extends CreatureObject and represents an enemy on the map. The enemies move around in a random manner on the map, and this movement is handled in the tick method. The movement is very crude at the moment, and more intelligence for the enemies would be preferable.

ShopObject: Represents a shop on the map. The only special functionality of it is that it calls enterShop of the Map class when the player executes the context sensitive command of the shop.

ItemObject: Represents an item on the map. It has an associated item (an instance of the Item class), and allows the player to pick it up by executing the context sensitive command of the item map object.

Quests The rpg.quest package contains the framework used for constructing quests in Vos. Quests are built up in a hierarchical manner from tasks. On a high level, a task is something the player should do. Let us use the following small quest as an example:

“Runar, the local craftsman, has lost one of his magic stones. Go and pick it up, and then return to him within 5 minutes.”

This quest could be seen as a task to carry out. A task can also have subtasks which build up the bigger task. In the example quest, picking up an item could be one subtask and returning to Runar within 5 minutes could be the other subtask. A task can thus be seen as a tree, where each node is a task and the children of a node is its subtasks.

The Task class is used to represent tasks. It is supposed to be extended by the specific tasks that are to be implemented. Every task has a list of subtasks, which it can start or stop as it sees fit, and which it listens to in order to find out whether any of them complete or fail. When that happens, it can decide what to do next. Tasks can be in one of several states during their lifetimes. When a quest is started, the tasks of the quest are initially inactive. When a certain task is currently the one the player is carrying out, the task is in its active state. A task finishes by becoming in the completed or failed state, depending on if the player completed it successfully or not.

Sequential and Parallel Tasks The SequentialTask and the ParallelTask classes implement two useful tasks that can be used for composing new tasks: sequential and parallel tasks. Both of the classes extend the Task class, and thus have a number of subtasks. They differ in the way they start the subtasks, and when they complete.

A sequential task starts its subtasks one at a time, and when the currently active subtask has finished successfully, it starts the next subtask in line. When the last subtask has finished, the sequential task completes successfully. Should any subtask fail, the sequential task fails immediately.
A parallel task starts all of its subtasks at once, and then waits for all of them to finish. If all finish successfully, the parallel task also completes successfully, but if any subtask fails, the parallel task fails immediately.

Using these tasks, it is easy to create new kinds of tasks. Many tasks in a quest are sequential at the top level. As mentioned above, the example quest could be seen as composed of two subtasks. These tasks are sequential, since one is carried out after the other. So the supertask in this quest is a sequential task.

In the same example, the second subtask (returning to Runar within 5 minutes) could be seen as a supertask itself. In this case, it would be a parallel task where the subtasks are returning to Runar, and not letting 5 minutes pass. These would be started at the same time, and when both of them complete the supertask would complete. In case of the timer task, it could be seen as being in the complete status until more than 5 minutes have passed, after which it would fail.

**Helper Tasks** Aside from `SequentialTask` and `ParallelTask`, several simple helper tasks are available in the quest package to use for composing bigger tasks. For example, `GoToObjectTask` is a task that completes when the player goes to a specific object. These tasks listen to the events of the map using the `MapListener` interface, and use this to decide when the task completes. These simple tasks all extend the class `SequentialTask`. This is because they can be seen as sequential tasks with no subtasks. (They could also extend `ParallelTask` in the same way.)

**Managing Quests** The `Quest` class of the `rpg.common.battle` package represents the “status” of a quest. It stores the text description of the tasks in the quest (to be shown to the user), as well as the states of each subtask. However, it does not contain the actual tasks of the quest. These tasks are instead created when needed by the `QuestManager` class. This class takes quests and starts the needed tasks for them, informing the quest about the progress of the subtasks. This lets the `Quest` class keep track of the states of each subtask.

The `QuestTask` class is used as the topmost task of all quests. This means that all the tasks of the quest are subtasks of this task. Since this class extends `SequentialTask`, all quests are seen as sequential at the top level. It is the states of these subtasks of the quest task that are stored in the `Quest` class, so it only keeps track of the states of the top level tasks.

When a quest task is created from a quest, the states of the top level tasks are initialised according to the `Quest` instance. This means that we can store the state of a quest and later restore it in the same state, with the caveat that only the states of the top level tasks are stored. Any lower level tasks’ states will not be restored. Thus, each top level subtask must be able to be started anew separately from any other task, and not rely on that the results of an earlier completed subtask will still be present in the current running of the quest.

**Network** The `rpg.common.network` package contains common classes for network functionality used both by the client and the server. Both also each have
their own network packages (`rpg.network` and `rpg.server.network`, respectively), that add more specific functionality to the common network package.

The server and client can send Java objects between each other by converting them to datagrams (described below). This allows for example the player to send the creature information to the server by sending a `Creature` object, that the server then can recreate.

**Network Layers** Our network code is divided roughly into three layers, in order to ease the writing of the code that uses the network. The layers are as follows:

1. Datagram layer. The lowest-level layer. It sends datagrams between two SIP users.
2. Connection layer. This layer adds the concept of connections between users. It also adds a class used for the different types of game specific messages that can be sent.
3. Network layer. This is the highest-level layer. It provides high-level methods that are used by the code that wants to communicate over the network.

The layers are described in more detail below.

**Datagram Layer** The datagram layer is a simple layer that is used for sending datagrams between users. The `Datagram` class is used for holding the contents of an object, which can then be sent over the network (see the "Serialisation of objects" section). The datagram is converted to an object of the `MessageProperties` class, and then this is sent using the `Communication` class. On the receiving side, the `MessageProperties` instance is converted back to a datagram.

The `Datagram` class is actually the same as the `MessageProperties` class of the `se.uu.greenfox.client` package, except that it’s used in both the server and client code. The reason for doing this is so that the server and client can use the same code, thereby avoiding duplication of code. (However, having the `MessageProperties` be common between the server and client from the beginning would be preferable, since we would not need the `Datagram` class then.)

**Connection Layer** The connection layer builds on the datagram layer and adds to it the concept of a connection between users. Connections are represented by the `Connection` class. This class abstracts away the way that users are connected, and how messages are sent between them (for example, if a persistent connection like MSRP is used, or if decoupled page messages are sent over IMS). Connections can be connected and disconnected, and listened to using the `ConnectionListener` interface.

This layer also adds the `Message` class, which represents the different kinds of messages that can be sent in the game between the server and client. For example, a login request containing the player that wants to login is one type of message.

Connections are managed on both the server and the client with the `ConnectionManager` class. This class is used for storing connections and
reporting the events of them. Other classes can listen to the manager (using the same ConnectionListener interface for listening directly to a connection), and whenever an event happens for a connection, the manager will alert the listeners of the event. The server uses a connection manager to store all the connections to the connected clients, and the client uses it to store the connection to the server.

**Network Layer** The network layer is the top layer and provides high-level network functionality. The idea of this layer was that it should provide a simple interface for doing things like logging in, and similar network functions. It would then take care of the details of performing that function and sending the necessary messages, relieving the other parts of the program of this.

The main class of this layer is the Network class. It provides high-level network functions like logging in and out, and can also be listened to using the NetworkListener interface. It will then report on events like when a login is successful (or has failed) and when the network is ready to be used.

**Serialisation of Objects** Java’s Serializable interface in J2SE allows easy serialisation of objects, that is, converting an object into a byte array that can then be sent and used to restore the same object somewhere else. Unfortunately, J2ME does not support serialisation, so we have to do this conversion manually if we want to send objects over a network. To do this, we made an interface Sendable that is implemented by all objects that we want to send over the network. This interface has a method convertToDatagram that returns a datagram of the object that can easily be sent over the network. The classes implementing this interface also supply a constructor that takes such a datagram as argument and creates an object corresponding to the data in it. This is nowhere as smooth as using automatic serialisation, but it’s the best we can do.

**Flow of Messages** On both the server and the client, the way the messages flow through the network layers are essentially the same. When an IMS page message is received on the client, for example, the flow is like this:

1. The Communication class receives the page message and converts it to a MessageProperties object.
2. ClientCore class of the rpg.network package receives the message, and converts it to a datagram (rpg.common.network.Datagram).
3. The client core uses the connection manager to get the connection of the user who sent the datagram, and passes the datagram to the corresponding connection.
4. The connection converts the datagram into a message (rpg.common.network.Message), and alerts its listener of the incoming message.
5. The Network class listens to the connections and receives the message. It then alerts its listeners of the event that occurred, according to the message that was received.
When sending, the flow goes in the opposite direction. The sequence is identical on the server, but using the server classes instead.

**Client-Side Network** The `ClientCore` class implements the `GameCore` interface on the client side, and is the initial receiver of messages from the server. It then converts these into datagrams, and uses a connection manager to handle the message to the connection of the user who sent the message. `ClientConnection`, a subclass to `Connection`, is the client-side implementation of connections.

The `Network` class of the client provides the high-level functions used on the client, like requesting login and logout, and registering the position of the player. The `NetworkListener` interface of the client can then be used to listen to the network, which then notifies of events such as successful login or logout.

The `LoginManager` class is used for logging in and out of the game. It provides the methods `login` and `logout`, which talk to the network and also show a dialog on the display that reports the progress of logging in and out.

**Server-Side Network** The classes on the server are much the same as on the client. `ServerCore` and `ServerConnection` are the counterparts of `ClientCore` and `ClientConnection`.

`Network` and `NetworkListener` work in the same way as on the client, except that they provide server network functionality, like accepting login requests and responding to logout requests.

**Dialogue** The `Dialogue` package is made to display lists in trees and have customizable actions for selected items. The dialogue consists of a set of “entities” and each of them contain a list of choosable items. When an item is pressed, the optional action is performed and the link is followed to either another entity or to the exit state.

`Dialogue` is made to be used in a big variety of situations. In the RPG project it’s used for conversations and for choosing and buying items in the shops.

**Shop** The shop module is essentially an interface to `Dialogue` with some automated features that make the construction of shops easier.

**Sound** The sound module is made for playing MIDI (Musical Instrument Digital Interface) and WAV (WAVeform audio format) files in the game. It is possible to play several music files at the same time. All sounds can be started, stoped and paused at any time.

**5.3.4 CamGame**

CamGame is a game where musical notes are generated based on the pictures taken by the camera on the mobile phone. This is done by storing a number of songs in a database on a server, each song consisting of a sequence of MIDI tones. When the client connects to the server, it sends the desired difficulty level
of the game, and based on that the server will generate some random pictures that are supposed to be taken by the mobile phone in order to generate the same song. The higher the similarity between the photos taken by the phone camera and the generated pictures, the higher the score will be, and the closer to the original notes the received notes will be. If a taken photo is a bit off from what it is supposed to look like, the received note will also be a little bit off, and so on. The client will always sample the taken picture and send some hex colour values to the server where they are compared to the actual target picture’s colours. When the game is finished the server will calculate the total score based on the difference between the generated notes and actual notes. This game uses the page message flavour of the common communication framework.

**CamGame client component** Figure 56 roughly shows the connection between all classes and packages for the CamGame client.

![Figure 56: The structure of the CamGame client.](image)

The **musicmaking Package**  This package acts as the application logic for the client. It includes only one class called **MusicMaking**, which handles all the menu events by implementing the LWUIT **ActionListener** interface and sending an instance of itself to the **GUI** class in the **ui** package. This class also handles all the IMS communication by implementing **GameCore**, which is the callback interface for the **Communication** class in the **se.uu.greenfox.client** package. Some methods in this class are available to **ui** classes via the **UIUser** interface. Pre-parsing of the received messages from the server is also done in this class, and the resulting vector data is passed along to the **GUI** class for post-processing.

The **ui Package** This package handles and implements all the interactions in the user interface. It consists of the following classes:
GUI: This class manages the graphical user interface by connecting the other classes in the `ui` package. It registers the events for menus by taking an instance of the `MusicMaking` class as an `ActionListener`. The main interactions with the camera, including starting the camera and taking snapshots, are managed here.

Camera: This class uses `videoControl` and `mediaPlayer` to create and start the camera. The camera object will be stored in a container so that it can be easily retrieved and used by other interface classes.

ButtonList: Implements a list of buttons by extending the `Form` class from LWUIT. The regular `List` object in LWUIT has many bugs and is thus not very usable. This class provides two different types of lists. The first type is a simple list that shows the entries one by one in a row and allows the user to select them. The second type is a list that can store four elements in each row, where each element can have an image and an index; they can also store more data like note volume and note duration.

Note: The `Note` class represents a single note, this includes the note duration, volume, index and the image associated with the note. It has several getter and setter methods to allow access to the properties of the note.

SingleNoteLabel: This class extends the `Label` class from LWUIT. In order to represent the notes on the main form we use this class which stores five different notes, and also handles the drawing and positioning of the notes on the `Label`. Many instances of this `Label` type will be used to hold the whole song. There will be a transition between each instances that allows switching between them in a nice looking way.

NoteForm: The main container for all `SingleNoteLabels` is `NoteForm`, it is extending the `Form` class from LWUIT and provides all the operations for navigating through the notes. This class overrides the `keyPressed` method in order to provide modified navigation on the `SingleNoteLabels`. The images and resources will be loaded and initialized in this class.

UIUser: The callback interface implemented by the `MusicMaking` class to handle events in the `ui` classes.

Client Sequence Diagram  The sequence diagram in Figure 57 and Figure 58 tries to explain in more details how the whole system is working, and it also represents how the classes collaborate with each other. On the top of this diagram, the named panes indicate actual Java classes in the system. The arrows show the communication between them, and the short numbered descriptions above these arrow lines describe the major system behaviour. The short descriptions without numbers are for the human player’s behaviour.
Figure 57: An overview of the CamGame client operation.
Figure 58: An overview of the CamGame client operation continued.
This diagram only represents the most important behaviours, and it does not show which method will be used for each event. For such details, please see the documentation in the source code. The diagram does not contain the package `se.uu.greenfox.client`, which is the package for the common communication framework. It is a common package, which could be used by any system, and it is only used by the `MusicMaking` class for communication with the server.

**CamGame Server component** Figure 59 shows the general structure of the CamGame server software.

![Figure 59: The structure of the CamGame server.](image)

**The `se.uu.greenfox.minigames` Package** This package contains the parts of the Minigames servlet that do not depend on any specific mini game. Its primary task is to be the communication bridge between the mini games and the IMS network. The only implemented mini game (as of when this document is written) is the CamGame.

**MiniGames:** The main class in this package is the `MiniGames` class. It contains all web interface methods, like `doGet` and `doPost`, to interact with the web browser of an administrator. All HTML post requests that are received are redirected to the `MiniGamesCore`, which in turn handles each request. The `MiniGames`' web interface allows the user to initialize the servlet, send administrative commands to the running server to query it for its status or to command it to do some other things. There are more information on the administrative commands in Section B.2.2. Note that any command sent to the `MiniGames` class is redirected to the `MiniGamesCore`. Also, in the web interface the user may emulate messages from either clients or servers which can be very useful for debugging purposes.

**MiniGamesCore:** Messages are received from the IMS network by the common `se.uu.greenfox.server` package, which acts as the interface
towards IMS. When the MiniGamesCore class gets a message from there, it checks what player is the recipient of that message, and then looks in the player mappings table and redirects the message to the core of the player’s current game.

Every player is at all times mapped to at most one game. This, of course, means that a player can play only one mini game at a time. When an admin command is sent to the MiniGamesCore class, it is redirected to the appropriate mini game’s core, or MiniGamesCore applies the command directly if the command is directed to the MiniGamesCore itself. If an admin command is redirected to the CamGameCore, it correspondingly applies it itself if the message is a CamGameCore command, or alternatively redirects it to the appropriate CamGameInstance, if that is specified in the command.

In order to keep a state on the web interface, hidden textboxes with information are used.

The se.uu.greenfox.minigames.camgame Package Here we find every class that have anything to do with the database can be found. It uses Hibernate to load objects from the database hosted on the external server.

Song: The Song class represents a single melody that can be played in a CamGame instance.

User: The User class represents a player that can play the CamGame.

Highscore: The Highscore class stores the bestregistered score by a user on a certain difficulty and song.

The se.uu.greenfox.minigames.camgame.server Package

CamGameCore: The CamGameCore has the responsibility of keeping track of all running CamGameInstances. When it gets a message from the MiniGamesCore, it redirects the message to the appropriate game instance in which the recipient player is currently playing. When a CamGameInstance wants to send a message, this is also done via the CamGameCore (So that the CamGameCore can log it). CamGameCore just redirects the message to the MiniGamesCore.

CamGameWebinterface: In the CamGameWebinterface you can find a list of all running CamGames and the high-score list of each. This list can be sorted by many parameters like “by song”, “by best results” and so on, or by combinations of them. In the CamGameWebinterface you can also find a list of all players that have ever played the CamGame, and what CamGameInstance they’re currently playing, if they’re playing at the moment. You can also find links to the admin command page described above.

CamGameInstance: The responsibility of the instances of this class is to keep the state of a running game, that is, the results of the picture, the current song, what player/players are playing this game, at what difficulty the game is played at and how long time that is remaining until the game ends (if running at higher difficulties).
6 Evaluation and Testing

To see that our applications worked the way we expected (or to find out that they didn’t), we did some testing. From Green hat People, we received a standard form that they use to give to people playing their games to fill in. So we used this form on our testers to find out their opinions on the games.

The form that was given to all participants contained the following questions:

1. How did you like this experience?
2. How much did you like the experience on a scale from 1 to 10? (1 = Not good at all, 10 = AWESOME)
3. What parts, if any, were extra good?
4. What was good?
5. What could have been better?
6. How much (on a scale from 1 to 10) would you like to do something similar to this again?
7. How much would you recommend this to a friend (on a scale from 1 to 10)?
8. What do you think this experience was worth in SEK?

We also had another variant of the form which did not ask for a scale on question 6 and 7.

6.1 Camgame

6.1.1 Methods Used
We asked some of our friends to play the game and give us feedback by saying what they spontaneously was thinking while playing.

6.1.2 Result of the Evaluation Form
1. How did you like this experience?
   - “The game was slow. Nothing happened”
   - “Slow and bad user feedback”
   - “Feels innovative”
   - “Fun idea which I think can be used by a young audience and for example in educational situations. Can also be used as a component in another game. Just like Agent SQ, this technique can be used in other games to augment the experience.”

2. How much did you like the experience on a scale from 1 to 10? (1=Not good at all - 10=AWESOME)
   - 4
3. What parts, if any, were extra good?
   - “Innovative gameplay”
   - “The simplicity in the graphics, that you understand immediately what it is about, was cleverly done.”

4. What was good?
   - “Fun way to pass the time, a bit challenging”
   - “Interesting idea”

5. What could have been better?
   - “The game should have used shapes instead of colors”
   - “Better and more intuitive interface, instructions and descriptions”
   - “The packaging - the feeling that this is a completed product”

6. How much (On a scale from 1 to 10 or Yes/No) would you like to do something similar to this again?
   - 5
   - 7
   - “Yes”
   - “No”

7. Would you recommend this to a friend (On a scale from 1 to 10 or Yes/No)?
   - 5
   - 1
   - “If it was free, yes”
   - “Yes”

8. What do you think this experience was worth in SEK?
   - -100 SEK
   - 5 SEK
   - “Zero. An entertaining experience if it is free”
   - “Privately - Preferably financed by advertisement”
6.1.3 Comments by Test-Subjects

- The program should be tolerant against users that’s pressing multiple times. (has been resolved in the latest build)
- Inform when user is supposed to wait for communication and when user is supposed to act. (has been resolved in the latest build)
- The colors on the client are horrible. (has been resolved in the latest build)
- Annoying that you have to press “yes” to allow the program to do things all the time.
- The interface is not intuitive, it’s difficult to understand what to do when no one tells you.
- It should show the color that’s supposed to be taken while taking the picture (Don’t switch to the camera-only menu).
- There’s an annoying delay on the camera.
- There seems to be a bug which generates the wrong length of tones sometimes. (has been resolved in the latest build)
- Same pictures appearing multiple over and over (comment while playing easy mode)
- If the marker is moved after a picture is taken but before the servers reply is received, the picture is registered on the wrong note (has been resolved in the latest build)
- BUG: Register the picture on another note that was selected (stepped forward and took a picture and it was registered on another one) (has been resolved in the latest build)
- The addresses should be unique the games interfere with each other.
- The game seems to be more unstable after aborting the game and restart a new one (without restarting the midlet)
- The first letter in difficulties should be capitalized.
- When the program is waiting for a response from the server or when it is performing some heavy calculation, the user should be informed about this, so that he/she does not think that the pressing of the button has been ignored or similar.
- There’s no back-button after a picture has been taken. (has been resolved in the latest build)
6.1.4 Our Comments / Conclusions
When we tested on some of these people, the server was very slow. This gave
the game a worse result since people got bored of the waiting times. there are
more suggested work in this case in future works part of the report.

We had not thought anything about the HCI (Human Computer Interaction), which obviously generated a lot of confusion. The user interface is already
as simple as possible but to guide the player throughout the game we may need
more dynamic help feature where it can suggest a next move for user or describe
the procedures.

If this game would have a future, it would be as a subgame of another game.

6.2 Tourist Guide

6.2.1 Methods Used
We used some friends but also people that we normally don’t interact with.
During the test a person from the development team assisted them.

The testers got one phone per person but choosing the same location to walk
to so that the developer could be with them all the time.

6.2.2 Comments by Test-Subjects

• When pressing Start tour some loading indicator should be displayed. The
  user wasn’t sure that the game was trying to fetch the tour from the server.

• Seemed like there was a bug where the user started the tour within a
tourist place, then the trigger for “flip to info tab” triggered many times.

6.2.3 Result of the Evaluation Form

1. How did you like this experience?
   • “Interesting. Exiting. Something new. I wanna try it again to ex-
     plore more. It’s fun.”
   • “I like the informations with the pics and the map is nice. But
     cellphone ’shut down’ sometimes.”
   • “The game would be good for users who are new to a place.”
   • “Very fun - surprisingly fun.”
   • “Very good. An improvement of something that has been done before
     but with other means (e.g. personal guides or signs in the area).
     The technology makes it cheaper and enables e.g. tourist to discover
     places in their on pace.”

2. How much did you like the experience on a scale from 1 to 10? (1=Not
good at all - 10=AWESOME)
   • “9”
   • [no answer given]
   • “8”
   • “9”
3. What parts, if any, were extra good?

- “I can see my trial on the map. It’s so fun when I see myself moving like a dot.”
- “The GPS map is extra good.”
- “The time I saw my track on the map.”
- “When reaching the circle - the feeling of having arrived.”
- “What is especially good is that one can visit a tourist place in one’s own pace.”

4. What was good?

- [no answer given]
- “You can add places with pics by yourself.”
- “The introduction of a place is triggered when I am near it.”
- “Simple, clear.”
- [no answer given]

5. What could have been better?

- “Maybe adding more challenges and tasks during the game.”
- “User friendly interface and the ‘shut down’ problems.”
- “Triggering once is enough, no need to trigger from time to time.”
- “Download time.”
- “What unfortunately isn’t so good with this is that it is a consumer product which requires one to have GPS in one’s cellphone. Hence it will take some time before it can be implemented in the market. Also the design/graphics can be improved.”

6. How much (On a scale from 1 to 10 or Yes/No) would you like to do something similar to this again?

- “10”
- “If it is for free”
- “7”
- “Yes!”
- “Yes, absolutely”

7. Would you recommend this to a friend (On a scale from 1 to 10 or Yes/No)?

- “10”
- “Sure!”
- “7”
- “Yes!”
- “Yes, absolutely”
8. What do you think this experience was worth in SEK?

- “10 for each time”
- “I won’t buy it but I am thinking of hiring the people who made it.”
- “100 forever. 20 SEK / time.”
- “100 SEK / time.”

6.2.4 Our Comments / Conclusions

The participants overall liked the Tourist Guide application. The big minus was GPS problems (due to phone crashing or losing connection to the GPS network). Also, the server could be very slow and unstable (IMS innovation’s fault). We also see the value of having an online-form where you can’t answer in wrong syntax, e.g. on a scale from 1 to 10 the test-subject answered with text instead.

6.3 Agent SQ

6.3.1 Methods used

As with the Tourist Guide, we used both friends and people that we normally don’t interact with as testers, and a developer walked with them at all times.

The Agent SQ prototype was tested with several people sharing the same phone, in order for them to get the same mission and go to the same place. After the test was completed they filled in this form.

6.3.2 Result of the Evaluation Form

1. How did you like this experience?

- “It is a nice live game. I enjoyed it and I think I will play the commercial one.”
- “Fun, some exercise which is more fun than a normal walk.”
- “Fun, a little different.”
- “Interesting game.”
- “The game is funny!”
- “Funny! Challenging!”
- “Entertaining & simple concept which can be used in many ways. To collect letters and construct a word was a good & interesting way to make a competition of this but I see even more cooler ways of using the technology.”

2. How much did you like the experience on a scale from 1 to 10? (1=Not good at all - 10=AWESOME)

- “8”
- “8”
- “7”
3. What parts, if any, were extra good?

- “The idea of the game”
- “To receive a picture of the place which one should go to.”
- “To get the missions.”
- “When the number of the distance meter decreased.”
- “The idea of the game!”
- “The meter measurement, the clocks which ticks down.”
- “The simplicity in finding a thing, that the number of meters are counting down and that the clock ticks gave full score! But I would like to use it to other things than collecting letters. Example: The thing that one should find is several kilometers away and one must use buss/tube/taxi etc. to find it. The thing that is found is woven into a story and gives the player a key which helps them make progress in the game. Really good treasure hunts / score hunts can be constructed with this.”

4. What was good?

- [no answer given]
- “That it says how many meters that are left. That one can see the mission briefing again.”
- “That one sometimes saw pictures of the place one should find.”
- “The hint of the distance to the target.”
- “The design part.”
- “Conceptual structure, sub challenges :) End goal.”
- “See above.”

5. What could have been better?

- “The keyword should be easier. The GPS system can be improved.”
- “The GPS connection was lost, so we didn’t get any letter. Once it said “mission failed” but it shouldn’t have.”
- “That we couldn’t find the place albeit we where at the right place.”
- “The performance of the GPS”
- “The ‘no responding’ problem.”
- “The packaging could have been more ‘tight’.”
- “The design (that which is seen in the display) can be enhanced a little bit.”
6. How much (On a scale from 1 to 10 or Yes/No) would you like to do something similar to this again?
- “7”
- “7”
- “6”
- “9”
- “Yes, I will. In the summer.”
- “Absolutely!”
- “Yes.”

7. Would you recommend this to a friend (On a scale from 1 to 10 or Yes/No)?
- “9”
- “7”
- “8”
- “9”
- “Yes, I will.”
- “Yes!”
- “It depends on who has constructed the game. I wouldn’t recommend whichever game without that I know that the person who has constructed it has a grip of it.”

8. What do you think this experience was worth in SEK?
- “100 for playing one week”
- “80 (using whenever one wants to)”
- “One time: 10. Several times: 50 SEK”
- “40 each time”
- “500 unlimited for ever.”
- “Depends on context. Company: 500 SEK. Individual: 30-50 SEK.”
- “Worth money but is of the cheaper kind.”

6.3.3 Our Comments / Conclusions
The participants overall liked the Agent SQ game. As with the Tourist Guide, the big problems were GPS problems and server performance. The suggestion about making the Agent SQ prototype more intriguing was already thought of in the concept idea but we did not have the time to implement it.

6.4 Vos
6.4.1 Methods Used
[Same as above, maybe we should merge “Methods used” because they don’t vary that much from the different prototypes.]
6.4.2 Comments by Test-Subjects

- Really cool concept
- Missing content
- Would be better with real-time battles
- Missing multiplayer mode

6.4.3 Result of the Evaluation Form

1. How did you like this experience?
   - “Nice, but kind of cold (outside)”
   - “Good, the game need some polishing but otherwise it had a nice feel to it.”

2. How much did you like the experience on a scale from 1 to 10? (1=Not good at all - 10=AWESOME)
   - 8
   - 8

3. What parts, if any, were extra good?
   - “GPS-based, cool!”
   - “Good placement of some elements and integration with the game”

4. What was good?
   - “The hunt for monsters gave a real adrenaline rush”

5. What could have been better?
   - “Faster battles”
   - “Better placement of items and monsters. More intuitive UI”

6. How much (On a scale from 1 to 10) would you like to do something similar to this again?
   - 7
   - 8

7. Would you recommend this to a friend (On a scale from 1 to 10)?
   - 6
   - 7

8. What do you think this experience was worth in SEK?
   - 10-25 SEK
   - 7.95 SEK/day
6.4.4 Our Comments / Conclusions

People like the concept very much, though most of our test subjects point out that it would be better if we had more and better content. Sometimes the GPS was a bit off which made the test subjects a bit confused. Overall the comments were positive and some of the subjects wanted to play the game even more.
7 Related Work

Location-based games using mobile phones have existed since around the year 2000, with BotFighters\(^2\) generally being considered as the first. Many of the earlier games used rather primitive techniques to locate the users, usually based on finding the "cell" that the user is currently roaming around in the mobile network. One of the first games to use GPS to accurately locate users was Geocaching\(^3\). In this game, the player is supposed to seek out and find a small container that has been hidden by another user. Players are only given the GPS coordinate of the container, but no other game mechanics are used to play the game (in particular, no actual mobile phone client is needed for playing it).

It is not until recently that GPS functionality has become common in ordinary mobile phones, and this has made it possible to create more interesting applications that use precise positioning to involve the user.

Naturally, since this project was made in collaboration with Green hat People, some of our prototypes are in some ways similar to earlier games made by them. The Tourist guide prototype is similar to GHP’s *Stadsvandring med mobilen*\(^4\). Both of these games let the users walk in a virtual tour around a predefined site and receive information through the mobile phone about the locations they visit. The biggest difference is that GHP’s game is done entirely without using positioning hardware to handle the game mechanics, instead relying on having staff walk along the player and inform the system when certain places are reached. The Tourist Guide, in contrast, uses GPS to pinpoint the players and works completely autonomously. *Stadsvandring med mobilen* also uses only SMS to handle the communication to the users and doesn’t use any advanced client software on the phones. The Tourist Guide on the other hand communicates over the Internet using IMS, and runs on a client developed in Java ME.

Agent SQ is similar to *Kreativa uppdraget*\(^5\), another game by GHP. Both games are fast-paced and require the player to perform different missions in order to gain points. As with the Tourist Guide, Agent SQ is more technically advanced and doesn’t rely on having administrators directing the course of the game. *Kreativa uppdraget*, on the other hand, offers many more types of missions than Agent SQ, which only offers two kinds of missions (getting to a place and guessing a word).

Other games in the same spirit (not made by Green hat People) are Tourality\(^6\) and GPS Mission\(^7\). Tourality also involves the player racing to get to certain places, but also has multi-player support, so that you can compete with other players in getting first to a place. It does not have the same concept of collecting letters in order to complete a word, however, as in Agent SQ. GPS

---

\(^1\)http://en.wikipedia.org/wiki/BotFighters (there is no official web page)
\(^2\)http://www.geocaching.com
\(^3\)http://www.greenhatpeople.com/teambuildingspel/teambuilding-vara-spel/stadsvandring-med-mobilen
\(^4\)http://www.greenhatpeople.com/teambuildingspel/teambuilding-vara-spel/det-kreativa-uppdraget
\(^5\)http://www.tourality.com
\(^6\)http://www.tourality.com
\(^7\)http://www.gpsmission.com
Mission allows anyone to create missions using a tool, and then to publish them for others to download and play. This was an idea we also had in mind for our games, and it is available through the web-interface, but for Agent SQ there is no implementation for adding content from the mobile client.

Since role-playing games are a very popular game genre, several location-based RPGs have been developed or are in development. Two of these are Parallel Kingdom\(^8\) and Virtualpunk\(^9\). These both offer many of the same features found in Vos (indeed, they are common features in any RPG), such as a map showing the location of the player and other game objects, battles with enemies, items that can be used and non-player characters that can be interacted with.

One of the main differences between Vos and these two games is the way battles are handled. In Vos, battles take place in a different setting than when walking around the map, and the battles are turn-based. The battles in Parallel Kingdom and Virtualpunk take place directly in the map, and seem to use a more real-time based approach.

Whereas Vos and Virtualpunk are single-player games, Parallel Kingdom offers full multi-player functionality where several players can play together at the same time. This was one of the features we would have wanted to include in Vos, given more time to develop it. Parallel Kingdom does not seem to have any quest functionality, however, which has been a main focus for Vos.

\(^8\)http://www.parallelkingdom.com
\(^9\)http://www.virtualpunk.com
8 Future work

8.1 Vos

- Multiplayer
  One of the features we wanted to implement into Vos but did not have time to do is multiplayer support. This would mean that several players could play the same game at the same time, possibly together or against each other. Players could see other nearby players and interact with them, for example trading items with them. Players could also venture on quests and fight battles together.

- More content
  Most of the development time of Vos was spent making frameworks for the different game features, such as battles, quests and the map. This left little time for adding actual content that take use of these features, so adding this would be of high priority in order to make the game more interesting. Types of content to add is for example more quests, enemy types, items, weapons, spells, places and NPCs.

- Continuous map
  Currently, we have only made a single map of a region here at Polacksbacken where Vos can be played. The map doesn’t scroll when you move outside this region; it is completely static. Making the map continuous so that you can move anywhere and having the map automatically update to show the current region the player is in would make the game much more mobile and fun to play.

- Real-time battles
  One possible improvement is making the battles be real-time instead of turn-based, as they are now. This would make the battles more action-paced and realistic. The battles could also take place in the map setting instead of in its own screen, to make the game more seamless. This would also open for more possibilities of using the location of the user in the battles. For example, the user could run away from a battle by simply running away from the enemy in real life.

- Enemy intelligence
  The enemies in the game are currently not very intelligent at all. When battling they don’t use any strategies to try to defeat you, which makes the battles quite monotonous and unvaried. Also, when walking around the map, the enemies move in a random manner. Making the enemies walk more intelligently, perhaps attacking or shying away from the player, or even forming groups with other creatures, would further involve the player and make the location-based aspect of the game more interesting.

8.2 CamGame

- Multiplayer
One idea would be to add support for many people working on the same song. Cooperating with your friends is usually more fun than playing all by your self.

- More intricate pictures
  The current implementation on picture recognition is very specific to this game. It would be fun to have shape and contrast recognitions as well, which would require a lot more advanced picture recognition functionalities.

- More user friendly web interface
  Meaning prettier overall looks, easier navigation between pages and more options for modifying games.

- More intuitive GUI
  The GUI for making music part will be improved to ensure easier interaction, so the players will spend less time learning the game and more time enjoying it. This may also include a tutorial or short guide of the game.

### 8.3 Tourist Guide

- Combining planned tour with information from position concepts
  To let the tourist explore an area on its own but keep a planned tour in mind where those places needs to be visited in a specific order. The places besides those in the planned tour should share the same topic as the planned ones. For example, if there is a tour showing the life of an historic person the information from position places could for instance describe interesting anecdotes and the planned tour could tell about events which led to the person being famous.

- Being able to categorize/tag tours
  Given some categories or tags of interest the servlet should present tours which match the request from the tourist. For example if a tourist is interested in vikings the system could filter the places and only present those that are relevant.

- Time estimation
  Currently there is no way for a user to know how long it will take to walk a complete tour, a solution to this would be to present time estimations of tours when choosing one, as well as presenting approximation of how long time it’s left of the tours.

- Suggest travel routes
  A feature that could come in handy is if the tourist is presented different ways of transportation from where he stands at, for example if in Stockholm City the subway stations nearest should be presented as well as what time the next train is leaving to the station closest to the destination.
8.4 Agent SQ

- Player competition with scores
  The prototype doesn’t support any interaction between players. The easiest and most needed is a common scoreboard so the points the player collects actually means something.

- Binding missions and quests together with a question
  When arriving to a place the game can give a question that has an answer in relation or at the found location. This would make it more worth the effort. E.g. “You have arrived to the old ruins, but what is hidden under the biggest rock?”. Then the answer is a clue to solve the quest.

- Making the reward of missions more fun
  Getting a few letters and trying to combine them to a word isn’t so fun really. Suggestion: replace the simple clues with something more elaborate, e.g. a rebus with some words and pictures.

- Multiplayer
  More interactive multiplayer isn’t really in the scope for this game as it should be fast to run a session whenever one wants. An intriguing multiplayer part is best left for an implementation of the on-going quests game described under game concept ideas chapter.

9 Conclusions

Throughout the whole development process of both the game ideas and the prototypes we felt that location based gaming is the new way of playing games. The gaming world is moving from the traditional way of how users interact with the games, where instead of using a game pad with twelve different buttons and joysticks the games and the interfaces are being more formed to fit real life tools and instruments.

Our game ideas with the concrete prototypes have taken this evolution further and extended the playground to the real world. Our belief in that we are evolving how games can be played was established by the test persons’ reactions during the evaluation and testing; summarized feedback was that the gaming ideas as well as the prototypes was something new, exciting, interesting, different and so on.

Time is always a crucial matter; in our case it limited us to implementing the basic form of our different game ideas. One common feature which we want to see in the prototypes is support for co-op and multiplayer, which with the new technology that we have used and evaluated we know that it’s not far away from being realized.
A Installation Instructions

A.1 Installing the MIDlets

A.1.1 Installing the MIDlets on a Phone
Simply copy the compiled .jar file to the Applications or Games folder on your phone. On the phone, go to the appropriate folder and launch the application. This will trigger a one-time installation process, after that the game will start.

Note that in the current implementation of IMS, the SIP address of the user, as well as the phone number to the phone, needs to be hard coded into the client. See below for instructions how to set up and compile a new version if you need to change these.

Also note that all prototypes are developed for the SonyEricsson C702 phone model, and are not guaranteed to work on any other model.

A.1.2 Installing the MIDlets on an Emulator
If you use the Eclipse IDE, simply click the run button and choose the corresponding MIDlet from the list. For instructions how to set up the project, see below.

If you want to run a standalone emulator you will need either the Sun WTK or the SonyEricsson WTK installed.

• Open the KToolbar application for the WTK of your choice, then select Create project from JAD/JAR file... from the File menu.
• Find the .jad file for MIDlet you want to use and open it.
• Click the Run button and choose the game in the list on the emulator (there should only be one choice).

A.1.3 Setting up the Eclipse Projects
You need to have the IMS development environment installed, see the tutorials on the MJCF portal for instructions for that.

• Open Eclipse (SDS), then choose Import... from the File menu.
• Select General > Existing Projects into Workspace and press next.
• Click the Browse button of the Select root directory field and find the project directory. Make sure the project is selected in the Projects list. Check the Copy projects into workspace box if you want to copy it to your local workspace, then press Finish.
• Right click the project name in the Project Explorer in Eclipse, and choose Properties.
• Under the J2ME category, set the Group to your installed WTK. The Device should be set to DefaultColorPhone if you use the Sun WTK or SonyEricsson_JP8_240x320_Emu if you use the SonyEricsson WTK. Press Apply.
• Under the Java Compiler category, set Compiler compliance settings to “1.3”. The Enable project specific settings box will probably need to be checked in order to change this. Press OK.

• The project should now be built without errors.

• To compile a JAR and JAD for installation on a real phone or stand alone emulator, right click the project in the Project Explorer and choose J2ME > Create Package. The files will be placed in the Deployed directory inside the main project directory.

A.1.4 Setting up the Emulator in Eclipse

• To setup the game to run on an emulator inside Eclipse, select Open Run Dialog... from the Run menu.

• Click the New launch configuration icon at the top left of the list. Enter an appropriate name for the configuration, and make sure that the correct project is listed in the Project field.

• Tick the Midlet box, then click Browse... and find the midlet class for your game. These are:
  - “TouristGuide” for the Tourist Guide.
  - “ShortQuest” for Agent SQ.
  - “musicmaking.MusicMaking” for CamGame.
  - “rpg.main.RPGGame” for the Vos RPG.

• Go to the Emulation tab and in the Security Domain dropdown box, select manufacturer if you want to skip all confirmation dialogs. To get the same behaviour as on a real phone, select identified.third.party.

• Finally click the Run button. The next time you want to start the emulator, you can find your run configuration in the dropdown list next to the Run icon in the toolbar, or by going back to the Run dialog and selecting it in the list.

A.1.5 Changing Phone Number, SIP Address or Application ID

These values are hard coded into the clients because of the way the current IMS implementation works, and will therefore need to be changed every time you want to deploy the game to a new phone. If you’re running the game on an emulator the phone number does not matter. If you change the SIP address, remember to change the password as well.

If you change the application ID (context root) of the server application, you will also need to change the appID value in the clients. This is found at the same place as the SIP address and the phone number.

• For the Tourist Guide and Agent SQ, these values are found as constants at the top of the connection.ConnectionHandler class.

• For the CamGame, they are found as arguments to the Communication constructor call in the musicmaking.MusicMaking.connectIms method.
For the Vos RPG, these values are found as constants at the top of the `rpg.network.ClientCore` class.

### A.2 Installing the Servlets

#### A.2.1 Deploying the Servlets

- Go to the MJCF Portal (http://www.imsinnovation.com/) and log in.
- Press the Upload Sailfin Application button to the left, then browse for the compiled .war file for the game you want to deploy.
- Click Upload. It will take some time before the servlet is deployed, you can check the log by clicking the View deployment log button to the left.
- Once the log shows that your deployment was executed successfully, go to http://as.imsinnovation.com/context_root/servlet in order to visit your servlet, where `context_root` is the context root for your game. These are by default as follows:
  - “gf_ue_tourist2” for the Tourist Guide.
  - “gf_ue_quest” for Agent SQ.
  - “minigames” for CamGame.
  - “gf_rpg2” for the Vos RPG.

Note that you need to visit the servlet page at least once every time the servlet is restarted, either by you re-deploying it or by the sailfin server being restarted, in order to initialize it. The client will show a dialog saying that the server has not been initialized if this has not been done when you try to connect.

#### A.2.2 Setting up the Eclipse Projects

You need to have the IMS development environment installed, see the tutorials on the MJCF portal for instructions for that.

- Open Eclipse (SDS), then choose Import... from the File menu.
- Select General > Existing Projects into Workspace, and press next.
- Click the Browse button of the Select root directory field and find the project directory. Make sure the project is selected in the Projects list. Check the Copy projects into workspace box if you want to copy it to your local workspace, then press Finish.
- Right click the project name in the Project Explorer in Eclipse, and choose Properties.
- Under the Java Compiler category, set the Compiler compliance level to “5.0” or higher.
- Under the Project Facets category, make sure that the list contains the facets “Dynamic Web Module” 2.4 or higher and “Java” with the same version as you selected in the previous step. If they are missing, check the instructions in section A.2.3.
• Under the **Targeted Runtimes** category, make sure that **J2EE Preview** is selected. If you do not have it in the list, click the **New...** button and select **Basic > J2EE Preview**. Deselect the **Also create new local server** box, and click **Finish**.

• Press **OK** and the project should build. You may get an error starting with “CHKJ3000E: WAR Validation Failed”, you may safely ignore this error.

• To compile a WAR file for deployment, right click the project name in the Project Explorer, then choose **Export > WAR file**. Click the **Browse** button to select where the WAR file should be stored, then press **Finish**. Note that you should always use the exact same name for the WAR files you export within the same project, in order to work around a memory leak in the Sailfin application server.

### A.2.3 Restore Web Project

If the **Dynamic Web Module** project facet is not present when you import the project, you need to follow these instructions.

• Right click the project name in the Project Explorer in Eclipse, and choose **Properties**.

• Under the **Java Build Path** category, select the **Libraries** tab, then select the **Web App Libraries** entry in the list. Press the **Remove** button.

• Go to the **Project Facets** category. If you get a dialog asking you to save modifications to the **Java Build path** property page, click **Apply**.

• Click the **Modify Project...** button. Check the boxes for **Dynamic Web Module** and **Java**, and make sure their versions are 2.4 or later and 5.0 or later, respectively. Click **Finish**, then **OK**.

### A.2.4 Changing the Context Root / Application ID

In order to change to context root or application ID of the servlet, you need to make changes in three of the XML files, as well as changing the appID of the clients that are to connect to the game.

• In the **WebContent/WEB-INF/ims.xml** file, change the **service-id** tag to `<service-id id="your_id_here">`.

• In the **WebContent/WEB-INF/sip.xml** file, change the **display-name** tag to `<display-name>your_id_here</display-name>`, and the **app-name** tag to `<app-name>com.imsinnovation.com.your_id_here</app-name>`.

• In the **WebContent/WEB-INF/sun-web.xml** file, change the **context-root** tag to `<context-root>/your_id_here</context-root>`. Note the leading “/” in this case.

For instructions on how to make the client modifications, see section A.1.5.
A.3 Creating the Databases

You will need MySQL installed to set up the databases for the games. We used MySQL Ver 14.12 Distrib 5.0.51a.

- Open a terminal on the server
- Change the directory to where you put the provided SQL-files. The files are named by project name then by either Structure or All, if you use the structure files only the structure of the database will be created, but if you use the All files both the structure and test data will be created. As an example, the files for the Tourist Guide are: TouristGuideStructure.sql and TouristGuideAll.sql.
- These commands requires you to have root access to the database. If you don’t have it you will need to ask the system administrator to either supply you with a root account or ask him to generate the account and database for you.
- In the terminal, write “mysql -uroot -p” and enter the root password when mysqladmin requests it.
- The servlets (also including the Urban Exploration Web Interface) each have a hibernate configuration file. They are located in the src/hibernate directory in each project directory and are called hibernate.cfg.xml. It is vital that the configuration files are changed so that the applications knows where to find the database and use the right account and password.
- You need to change the URL where the database is located, the password and the username. The values used does not matter, as long as the same is used in hibernate.cfg.xml as when setting up the database.

```
<property name="hibernate.connection.password">
  your_password
</property>

<property name="hibernate.connection.url">
  jdbc:mysql://server_URL/db_name?autoReconnect=true
</property>

<property name="hibernate.connection.username">
  your_username
</property>
```
- In MySQL admin write “create database db_name;”.
- Write “use db_name;”.
- Write “create user ’your_username’@’%’ identified by ’your_password’;”.
- Write “grant all privileges on ’db_name’ . * to ’your_username’@’%’;”.
- Exit mysqladmin by writing ”exit”.

92
• Lastly, write “mysql -uroot -p your_password < file”, where 
  file is the appropriate SQL-file as mentioned above.

• The database is now initialized and ready for use.

A.4 Installing the Urban Exploration Web Interface

To be able to install the Web interface you need an Application Server to run 
the web interface servlet. We recommend the Tomcat Application Server.

We used Apache Tomcat/6.0.18 on a Linux system, using Java 1.6. These in-
structions will assume that you are running a similar system with the mentioned 
applications installed.

A.4.1 Deploying the Servlet

• Surf in to the Tomcat Manager, in our case it’s located at 

• Under WAR file to deploy, choose the exported WAR file and press 
  Deploy.

• The context root of the servlet will be the filename without the extension.
  So for example if the exported WAR file is named TouristWeb.war the 
  context root will be TouristWeb, and the complete URL in this case will 

• The servlet is now deployed and ready to be used.

A.4.2 Setting up the Eclipse Project

See section A.2.2 for instructions, but instead of Compiler compliance level 
“5.0” it should be “6.0” and under Project Facets, the Dynamic Web Module 
should be “2.5” and Java should be “6.0”.

93
B Maintenance Instructions

B.1 Common Instructions

There are some general maintenance issues which needs to be addressed due to MJCF application server and as well as problems regarding lost connections. Bear in mind that these applications are prototypes and they are deployed on an Application Server which is meant for development and not commercial purposes.

If there are no details for either client or servlet under each prototype then the common problems as well as their instructions apply.

B.1.1 Servlet

The servlets won’t automatically run after deployment, they also need to be initialized to start serving clients. The initialization is done by visiting the specific servlet address. That is, the MJCF application server address following the context root path with the servlet name at the end. Example: http://as.imsinnovation.com/gf_ue_tourist2/servlet. In this case, the “gf_ue_tourist2” part is the context root and “servlet” is the servlet name.

This initiation is required every six hours because of the available application server on the MJCF portal is currently being restarted at that interval.

B.1.2 Client

The client is dependent on that the server is running and is initialized. In some specific scenarios there is information loss due to connection failure between the client and servlet. The general work-around is to restart the client if the GPS or GPRS connection is lost.

It will be explained in more detail for each prototype it concerns.

B.2 CamGame

B.2.1 Servlet

Since SIP messages are used for communication, losing connection causes no problems for the servlet; the game keeps on waiting until the client ends the game, the servlet is restarted or if the game is removed manually from the web interface.

If the servlet is restarted while playing the game, all games are aborted and cannot be resumed. All information about the running games, such as score and progress is destroyed.

Most needed functionality for maintaining the servlet is available through the web interface. That includes monitoring games being played, editing the database and handling the game core itself.

http://developer.imsinnovation.com/tikiwiki/tiki-view_forum_thread.php?comments_parentId=438&forumId=6&highlight=restart
B.2.2 Web Interface

Using the web interface for the servlet you can administer the currently running servlet.

Other than the regular web interface, there are a few administrative commands you can enter under the admin command section.

The commands available for MiniGamesCore

Arguments are enclosed in tags (<>), example <minigame>
Optional arguments are enclosed in brackets ([]), example [<minigame>]

help [<minigame>]
   If no argument is given, a help message is displayed.
   If <minigame> is camgame, it shows the help for camgame.

clear [<log>]
   Clears all the log messages in log <log>
   There are at the moment two logs, a message log and a core log. These are represented by 'mes' and 'core'
   Example: 'clear core' or 'clear mes'

echo <string>
   Echoes the string <string> to the message log.

version [<minigame>]
   Shows the version of MiniGamesCore if no argument is given.
   Shows the version of the CamGame if camgame is passed as <minigame>.

The commands available for CamGameCore

Arguments are enclosed in tags (<>), example <minigame>.
Optional arguments are enclosed in brackets ([]), example [<minigame>],

version
   Shows the version number of MiniGamesCore.

camgame end all
   Ends all running games.
camgame end <number>
   Ends the game with id <number>.
camgame g <number> <command>
   Sends command <command> to the game instance with id <number>.
camgame list
   Lists all games running.

version
   Shows the version number of MiniGamesCore.

song list
   Lists all songs available.
song modify song <id> <melody>
   <id> is the id of the song to modify.
<melody> is the new song. It should be of the form:
<tone>,<len>,<tone>,<len>,...,<tone>,<len>, For example:
song modify 1 67,10,65,10,
It sets the melody of song with id <id> to <melody>.
song modify addedby <id> <user>
  <id> is the id of the song to modify.
  <user> is the new user that added the song.
  Set the user that added the song with id <id> to <user>.
song modify title <id> <title>
  <id> is the id of the song to modify.
  <title> is the new title.
  Set the title of song with id <id> to <title>.
song add <title> [melody] [addedby] <melody>
  <title> the title of the song.
  <melody> is the melody of the song itself.
  <addedby> is the id of the user that added this song.
  the form of the melody should be:
  <tone>,<len>,<tone>,<len>,...,<tone>,<len>,
  Adds a new song with title <title>, melody <melody> and
  it’s added by the user <addedby>, if specified.
song delete <id>
  Deletes the song that has id <id>.
user list
user add <sip> [nick]
  Adds the user with sip address <sip> to the database.
  If <nick> is given, the nickname of user will be <nick>.
user delete <id>
  Deletes the user with id <id>.
user modify nick <id> <attrib> <newvalue>
  <id> the users id
  <nick> the users nickname
  <newnick> the new nickname the user will be referred as.
highscore get <userid> <songid> <difficulty>
  Shows the result of the user with id <userid> regarding song
  with title <song> at difficulty <difficulty>.
highscore list
  lists all high scores.
player <id>
  Show all players that’s playing the game.

The commands available for CamGameInstance
addtime <time>
  Adds <time> minutes to the game.

B.2.3 Client
The Client may break in some cases with an out of memory error. Often this
occurs because of other running application at the time or even applications
that have been running, before starting the CamGame. Even after closing some applications they still occupy some memory in the phone and it may cause CamGame to run out of memory while playing. To workaround these problem, restart the phone which will in turn clear out the memory.

If for some reason during the game IMS network or the servlet application restarts, then the game state on the servlet will be lost and this will cause the client to fail. The only solution is to restart the game.

It is possible that the phone can lose connection while playing the game. If a message delivery failure occurs, the client will keep trying to send the message until it eventually succeeds. There are still situations where the delivery failure will not be known to the client (this is due to lack of callbacks from the MJCF libraries), resulting in losing the message, and thereby losing information. The solution in this case is to try to redo the last action.

Note that the client relies heavily on the game servlet, so make sure the game servlet, and in extension the IMS network, is stable before you try to run the client.

B.3 Tourist Guide

B.3.1 Client

When the client is running and communicating with the server, it can occur that the client loses the connection. The client does not get notified that the connection has been lost, but nothing more will be presented in the game. What can be confusing here is that the GPS updates the user’s current position on the map which gives the impression that the server works, but when trying to fetch information about a specific place it will not work. The only solution for this is to restart the client, which unfortunately resets the state of the user in the tour.

B.4 Agent SQ

B.4.1 Client

If the client loses connection to either GPS or IMS network during a game, then it’s necessary to restart the phone. Though the information you lose is the current mission you were playing and not the whole quest, what you managed to achieve as well as what missions you have completed will not be lost, so continuing on the quest with a new mission is possible. The side effect from the disconnection is that the servlet will count the error as failure and deduce points from the player, because the game starts to decrease points from the start of the mission until the time limit is met.

B.5 Urban Exploration Web Interface

As mentioned in the system description, the Tourist Guide and Agent SQ prototypes have a web interface of their own to control game specifics. The application server which the web interface is running on needs to be restarted every 8 hours to workaround an issue with MySQL server. The problem consists of

---

11See Section 4.6 and Section 4.8 for more information on how the web interface works.
MySQL server shutting down the connection due to inactivity (wait_timeout) for the web interface without notifying it. The connection pool used in the web interface fails to check if the connection is alive or not (even though it has been specified that it should do a test query to check if the connection is alive or not).
C  Suggested Future Work

C.1  Common Suggestions

C.1.1 Solve MySQL issue
As mentioned in Section B.5, there is a problem with MySQL and the web interface, or rather with the Hibernate framework which lets us map our Java objects to relational database tables and vice versa.

This time-out problem is not an issue while we are using the MJCF application servers, because they are restarted every 6 hours anyway, and the MySQL time-out limit is 8 hours. It would be of interest to solve this issue though, when moving the application to a commercial application server with better up-time and stability.

This type of problem has been addressed on different development blogs and tutorial sites\(^\text{12}\), and the solution consists of using a connection pool which checks if the connection is alive or not.

The implementation of the connection pool should be on each prototype as well as the web interface for Tourist Guide and Agent SQ.

C.1.2 IMS Self Authentication
As IMS should be self-authenticated (by phone or SIM card), the game shouldn’t need any settings for user name and password, but the current implementation of IMS doesn’t use this. So the client right know is hard-coded to use a specific SIP address (user name), and password. One way to solve this is to provide a download service which ask for credentials and packages a tailored jar file for the user. The user then has only to give a SIP address and password once, then the installed game will always use those for authentication. As far as it is possible, having to manually type user name and password upon start-up should be avoided.

C.2 CamGame

C.2.1 Abstraction Level

- Currently the ButtonList class is specialized only for the songsList view, while it can be generalized to handle any arbitrary list.

- The Communication class covers many methods that are not employed by the code, it is possible to make it a library and then use obfuscation to achieve a smaller jar package for deployment and therefore get a faster loading time and less transferring time over Bluetooth.

C.2.2 Communication

- The main protocol used in the CamGame communication is generally quite inflexible (see Appendix D for more details). For an example, in our protocol the keys and values used for specifying “regions” in images are:

\(^{12}\text{http://michaelstudman.com/fullfathomfive/articles/2004/06/07/mysql-dropping-connections-and-hibernate}\)
This form of representing the colours for image regions cannot be generalized to any arbitrary regions, for instance we can’t specify more than 4 classified regions, and also the position of those regions is not defined in this protocol. A good implementation can cover these restrictions and provide any classification of pixels in the captured image. The same situation applies to the server responses with the label \texttt{songandpics}.

- The server implements, at the moment, a very simple and specialized picture recognition method. It could be extended to handle shape recognition, take contrasts into consideration and some other things to make the game more interesting and maybe more tolerant. This requires a change in communication however, most likely sending the entire pictures between client and server. For a game that is supposed to be fast and quick to play, the delay may be too high for this to be realistic.

- Using page messages is much slower than using MSRP, depending on the server load it sometimes takes up to 2 seconds to send and receive a message. The CamGame client is constantly in communication with the server, so it is possible to define a new protocol based on MSRP and experiment with it in different situations. It is, however, expected that if the MSRP connection drops there may be difficulty in re-initiating it, and therefore we risk losing the connection to the server.

- A centralized user authentication system can be defined to manage all the users and the process of logging in and out. It can include the user names and passwords, and maybe a configuration database to store all the users’ profiles and configurations. On the client, the authentication should include an input field for user name and password, and on the server it should save the user credentials and the profile information in a database.

- If the IMS server restarts or crashes for some reason, then the next time it initializes the servlet we lose the state of any running game which leads to inconsistency in the running clients. Hence implementing a mechanism to store the state of all running games on the server can be expected as a future work.

- In the case of the client losing network coverage, and thus Internet connectivity, the messages sent will naturally fail to reach the destination. The page message version of the common communication framework tries to solve this issue by resending the messages until they are successfully delivered, but there are situations where the IMS framework reports a success while the message has actually not reached the destination. In these situations we need to either have the IMS system upgraded to resolve these issues, or we have to manually implement a protocol that expects a response (echo) for every message sent, so that it can retry or report the failure if an echo is not received at the sender side after some time.
The client and the server handle the messages in an asynchronized fashion, meaning that they can send and receive messages at any time without considering the actual order of messages. It is also possible to implement synchronized communication, so that the client or server must wait for the appropriate response before sending the next message, this may also help to resolve the issues with failed messages.

C.2.3 Resource Management

Careful handling of the resources used by every mobile application is quite vital, since the memory can be filled with images and other resources being used by the application. Especially in a game like CamGame there is a high possibility that we run out of memory by capturing more and more images.

The most resource consuming objects in CamGame are the images received from the server and the images being captured by the camera while playing. CamGame handle these images in the following ways:

- The images from the server will be redrawn dynamically based on the server response, so at each time there is only one image resident in memory and this way it saves memory and will not consume much memory resources, even though it may require more processing instead.
- The images being captured by the camera must be saved in order to view them later. This can require a lot of memory if we keep all the images in memory. CamGame handles this by resizing the images to only cover the visible area on the screen. So we still keep all the images in memory, but they are smaller in size. This way of handling images can still lead to problems in longer songs where we have more notes and thus more images are being taken. In order to solve this we need to either encode images in JPEG or GIF format to reduce the size even further, or save the images directly on the phone memory card and load them every time they are needed. This again has the side effect of more processing on the phone.

C.2.4 Game Logic

CamGame supports three different game difficulties:

- **Easy**: The pictures the user should take are all single-colour.
- **Normal**: The pictures have up to two different colours.
- **Hard**: The pictures have up to three different colours.

In each difficulty the image is divided into four equal regions, and based on the difficulty we will map a different number of colours to each region. The logic could be generalized to consider irregular shapes and any number of colours.

C.3 Vos

C.3.1 Game server

As it is now, the server for Vos does not handle any game logic at all. Its sole function is letting players log in and out of the game. Perhaps the most
important function to implement would be a game server that could handle parts of the game logic and extend others. In particular, the server could keep track of the positions of players playing the game. This could then be used in order to implement multi-player functionality.

Another big improvement would be to dynamically send the map of the region the player is currently in, instead of just having a static map hard-coded into the client application as it is now. This would allow the player to move around to any location, and the map would automatically adjust to show the current region. To do this, some sort of map service could be used to fetch an image of the player’s current location. The Map class on the client would then have to be adjusted to dynamically fetch this map, and not use hard-coded GPS positions for the game region.

Even more functionality could be put on the server. For example, the locations of shops and other game objects could be stored on the server and sent to the client as needed. This would facilitate changing or adding new places without the need for changing the client software. Enemy creatures could also be stored on the server, in order for them to be seen by several players at once. In order to implement this extended server, the server would probably need something similar to the Map class, in order to keep its own view of the game world.

C.3.2 Map class

The Map class currently represents two things: the state when we are walking around in the world, and the game world containing all the objects in the world. This dual nature is mostly due to historical reasons and because it is a simple way to do it, but it would probably be better to keep these separate concepts in different classes. The game world could be in its own class, and the map state could then include an instance of this class and operate on it. This would reduce the size of the Map class and make it much more manageable.

C.3.3 Battles

The battles could be made real-time quite easily. The BattleState class would have to be modified so that it doesn’t give only a single controller execution time in each tick. Instead, all the controllers could be ticked in each tick of the state.

Battles could also be extended to allowing the player to fight more than a single enemy at once. Some parts of the battle package do not assume that we fight only a single monster, but others do. The PlayerController class, for example, always chooses the first of the enemy creatures as the action target of the player’s creature, no matter how many there are. This would have to be extended so that the player can choose which enemy creature to attack. Also, the BattleUI class would have to be able to draw more than one enemy on the screen at once.

C.3.4 Optimisation

The code has not been optimised much at all, since we instead focused on writing as correct code as possible. For example, checking for overlaps in Map is done
between all objects in the map, with no sort of data structure to speed it up. It also doesn’t take into account whether the object was added, removed or moved when doing the check, even though this could be used to make it faster.

Another possible improvement is that when we display the inventory or spell list, we create and add new list items to the list each time we press the button. This could be stored and updated as necessary instead.

C.3.5 Database

The database is very simple in its current implementation. As mentioned in Appendix E, it does not map the values of the classes we store to cells in the tables – it’s all binary. This means we can’t perform any database queries on the data, which might be necessary if the game gets more advanced. Changing this would entail making tables for each of the classes we would want to store, and making Hibernate mappings for them.

C.3.6 Storing quests

As described in the System Description section of this report, quests can be restored from a previous state. We can also convert quests into Datagrams to be able to store them on the server. Everything is therefore in place for us to be able to store quests and later receive and continue them when the player logs in. The only thing that is not implemented is the loading of quests in the Player class. They should be loaded when the player is created from a Datagram. Another possibility is to make the QuestSet class handle the loading and saving of quests, making it implement the Sendable interface.

C.4 Tourist Guide

C.4.1 Communication

- The SIP-addresses and passwords that are used for authentication in the IMS network are hard-coded in the client. In the meantime, before this authentication issue becomes integrated into the phone/SIM card (as it should be once the IMS system becomes more mature), an easy-to-use login system needs to be created. It should preferably store the credentials in the mobile phone’s record store so that it can automatically connect after the information has been entered the first time.

- A functionality that puts a timer for waiting for responses on messages like “get this description” should be added, so that the application can try resending the message and/or inform the user, if for some reason a message or its response fails to be delivered.

C.4.2 Resource Management

Currently all images fetched from the server are stored in the mobile phone memory in order to avoid costly retransmits. There is no bound on the amount of stored images though, which means that if we start many tours and fetch many images without restarting the application, we might run out of memory on the mobile phone. A check to make sure that we do not run out of memory should be added.
C.4.3 Game Logic

- If the program is closed, everything about tour selection, place selection and the walked path is lost. If a user by accident closes the application it should per default return to the same state when started if the user is close to the last location (when he or she closed the application).

- If a new tour is started when one is already being taken, any places shared between the tours that were visited on the old tour should be marked as visited also on the new tour.

- It can be very frustrating for the user if loss of GPS or network connection occurs. The client could try to pre-fetch and cache the textual descriptions of places nearby, so a user can at least read about the places (needs to be manually selected if it was GPS that was lost). Pre-fetching images could also be possible, but considerations have to be done about the total amount of data transfers that occur (who cost the user money).

C.4.4 User Interface

- The pane in the client application for adding a new place is quite ugly and doesn’t show any preview of the picture taken. It should be refined.

- A visual, more prominent warning about loss of GPS and/or network connection is preferable, so that the user knows what’s wrong if no updates of position occurs or no description of a place is displayed.

- When starting a tour, the interface should switch to the “in-tour-mode” immediately, and not wait for the list of places to be received from the server.

- The web interface could be improved with statistics of the users, like what tours and places are requested the most etc.

- Currently the number of tours and places in the Tourist Guide are few, but if this number is increased the current design and layout of the web interface will cause issues in how user-friendly the interface is.

- Implementing categories to separate tours as well as adding support of tagging places will generate more accurate listings in the web interface.

- In the web interface, presenting the search results on several pages with an option to sort the items instead of having a long unsorted list is a desirable feature.

C.5 Agent SQ

C.5.1 Abstraction Level

The GUI code for the client is already very abstracted with easy functionality to create new panes, but there could be work done to abstract the game logic code. It feels very spread out how it works now, and more decisions and the like should be moved into the game package.
C.5.2 Communication

- We always trust that a message received is well formed, which is not good enough for a production environment. It needs more error handling, e.g. if a message which should contain a certain bundled variable and value according to the protocol is received, but is missing those values, we should handle it in some clever way. Right now it is not handled in any way, which means that it is quite likely that the application will crash if it receives malformed messages.

- It seems hard to detect if the connection to the IMS network is lost. As the client and server don’t have any contact with each other in the time between the player gets a mission and the player reaches the mission goal, it will be very irritating for the player if he or she doesn’t get their points when they have spent some time and walked far to get it. Perhaps the client should store accomplishments in the phone for later synchronization with the server.

- Communication is performed asynchronized on the lower level of the communication framework. There isn’t any good check for handling situations where the messages arrive in a completely different order and at unexpected times, which would be preferable to have.

C.5.3 Location service

Currently there is no indication if the GPS stops receiving new positions, except that the values displayed don’t change; for the player that means that the exact distance shown on the distance gadget gets stuck. A timer should be used to warn the user that GPS connection is probably lost if it hasn’t received a position update within a specified time frame.

C.5.4 Resource Management

The phone’s battery seems to be depleted fast when using the game. We suspect the GPS receiver is the reason for this as it is constantly used. Investigating if it’s the GPS receiver, the display or something else that is mostly responsible for the battery drain could be an improvement.

C.5.5 Game Logic

- Instead of sufficing just to arrive to a place, it can be more fun if the player is given a question upon arrival and the answer is somewhere at that place. The answer is another clue on the path of solving the quest. This connects the missions and the quests together and make them feel more exciting.

- The direction feature of the distance gadget never got completed, the code is half-way done and not tested (in the util.Direction class). Finishing this could make it more fun in the start of a mission if you can (only vaguely) can make out the direction to the mission goal.
• Remove the display of exact distance in the distance gadget and add the planned hint system, in which we can have it as one of the hints that you can buy for some score points.

C.5.6 User Interface

• Fix the very cool feature that prints text one character at a time (co-denamed the Hacker Text Area). Note that the current (disabled) implementation have a strange bug which give runtime exceptions. This is probably partly caused by race-conditions of threads and perhaps bugs in LWUIT. When it is enabled it looks very cool and more agent-style.

• One urgent fix is to show the state of the connection to the server and when it is waiting for server responses. This will help avoid much confusion for the player.

• The web interface needs to display more information from the database. Now it only handles static data (like locations, quests, missions etc.) but no dynamic data (like player quest and mission status).
D Communication Protocols

D.1 The Structure of Sent Data

The common communication framework deals with messages in the high-level terms of labels, message data parts and key-value pairs. The underlying messages sent using the MJCF APIs are, however, simply byte strings. This section describes how the higher-level message abstraction is represented using these byte strings. There are actually two different flavours of the communication framework, one using MSRP and one using IMS page messages. Their APIs are almost identical, but they have some differences in how messages are actually sent. Two application prototypes use the MSRP version, and two use the page message version.

D.1.1 Using MSRP

MSRP works by keeping a persistent session between the endpoints, and sending data in this session. It has a much higher throughput than the page messages, and is as such more suitable for sending any larger quantity of data, like pictures. Indeed, the only reason we don’t use MSRP for all of our prototypes is that the connection might get silently broken after a while, without any callbacks to the applications. This is most likely due to bugs in the MJCF.

The higher-level messages has a label (any string of characters), and zero or more data parts. Each data part is a collection of key-value pairs, where the value part is a list of one or more atomic string values. The key-value pair collections are represented as MessageProperties objects, and there are methods to serialize or de-serialize these objects to or from character strings. The character strings for the label and the data parts are then converted to byte strings using the UTF-8 character encoding. These byte strings are finally formatted into a single byte string message in a way that allows each component byte string to be extracted and parsed separately at the receiver. In order to achieve this, we prepend to each component byte string a four-byte size field that, when parsed as an integer value in network byte order, gives the size of the next component string in bytes.

As an example, consider the message that has the label “Hello” and one data part consisting of the key-value pairs “lat” with value “0” and “long” with value “0”. The data part will be serialized into a character string like this: “lat=0;long=0”. Both character strings are then converted into byte strings using the UTF-8 encoding, size fields are prepended. The resulting message is shown in Figure 60.

<table>
<thead>
<tr>
<th>Data</th>
<th>00 00 00 05 48 65 6C 6C 6F 00 00 00 06 6C 65 74 30 3D 30 7C 30 3D 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Size (5)</td>
</tr>
</tbody>
</table>

Figure 60: A sample message in its byte string form.

At the receiver the process is reversed, and the framework invokes a callback to a listener while providing the message label and a number of MessageProperties objects (one in this example).
D.1.2 Using Page Messages

Page messages are small messages sent outside of any session. They are in nature quite like regular SMS. The size of the data that can be sent in page messages is quite small (only about 400 bytes is officially supported), and the messaging is rather slow. The page messages have built-in support for multiple message parts though, which means that we don’t have to use the size fields as with the MSRP version of the framework. Other than that, the messages are structured in the same way.

The page message version of the framework also has support for automatically resending messages that failed to be delivered, something that the MSRP version lacks, but the underlying protocol in that version ensures that messages are re-sent until they arrive, if possible.

D.1.3 Picture Sending

The common communication framework also provides a way to send pictures between client and server. Since picture sending does not fit as naturally into the key-value pair format as other messages, we provide a special case for convenient picture sending (although an application could handle its own picture sending using normal messages if it wanted to).

A picture message is always formatted in the following way: The label is “Picture”, and there are two data parts. The first data part is a normal key-value part with the single key “name”, and the name of the picture as the corresponding value. The second data part is the raw picture data in JPEG format.

When sending pictures in this way, the framework will handle them for you and provide already parsed picture objects for the listener callbacks. On the client side it will provide LWUIT Image objects, and on the server side it will provide standard Java BufferedImage objects.

Both versions of the communication framework support picture sending, although MSRP is a lot faster and more stable. Due to sizes exceeding 400 bytes, sending pictures using page messages may cause the receiving client to be silently disconnected from the IMS network due to bugs in the MJCF and the IMS system architecture.

D.2 The communication protocols

The communication protocols for each prototype are listed here. The format of this section is as follows: A listing of all defined message types for each protocol is given first. Each message type’s different parts are indicated by square brackets “[ ]”, and the label of a message is listed as the first part. The direction of the message type (client-to-server, server-to-client or both) is indicated before the message. The values given in the key-value pairs are chosen as examples only. Whenever a message is not deemed self-explanatory, a comment is given in *italic* outside the brackets.

After the listing comes a schema of the message passing between client and server during a normal execution of the application. Each message is here referred to by its label.
D.2.1 The Tourist Guide protocol

This is a listing of all the message types used in the Tourist Guide communication protocol. A schema of a normal execution of this protocol is given in Figure 61.

![Diagram of the Tourist Guide protocol execution](image)

Figure 61: A normal execution of the Tourist Guide communication protocol.

(Client -> Server)

[Hello]

[lat=59.84; long=17.65]
(Server -> Client)
[Tours]
[name=Tour1; abs=A short tour.; thumb=t1thumb.jpg; map=t1map.jpg; lat1=5.0; long1=1.0; lat2=2.2; long2=3.1]
[name=Tour2; abs=A long tour.; thumb=t2thumb.jpg; map=t2map.jpg; lat1=5.1; long1=1.2; lat2=3.2; long2=8.9]
...

(Client -> Server)
[GetTourInfo]
[name=Tour1]

(Server -> Client)
[TourInfo]
[name=Tour1; desc=This is a very nice tour.; pic=t1pic.jpg]

(Client -> Server)
[GetPicture]
[name=t1map.jpg]

(Server -> Client)
[GetPicture]
[name=home.jpg; newname=newplace157.jpg]

The newname key is used by the server if it wants the picture sent using another name than what the user originally chose.

(Server -> Client or Client -> Server)
[Picture]
[name=pic.jpg]
[<JPEG data>]

(Client -> Server)
[StartTour]
[name=Tour2]
D.2.2 The Agent SQ protocol

This is a listing of all the message types used in the Agent SQ communication protocol. A schema of a normal execution of this protocol is given in Figure 62.

(Server -> Client)
[Hello]
[lat=0.0; long=0.0]

(Server -> Client)
[QuestInfo]
[nrclues=3; letters=<G, H>; scores=<79, 58>;
 times=<84, 168>; questscore=137; maxcluescore=100]
Figure 62: A normal execution of the Agent SQ communication protocol.

(Client -> Server)
[GetMission]
[lat=59.84; long=17.65]

(Server -> Client)
[MissionNotAvailableHere]

(Server -> Client)
[Mission]
[long=59.845; long=17.655; rad=10.0; text=Secret place, find it!; pic=secret\_place.jpg; timer=400]

(Client -> Server)
[AcceptMission]

(Client -> Server)
[Cancel]
[timeout=false]

(Client -> Server)
[MissionFailure]
[score=137]
D.2.3 The Vos protocol

This is a listing of all the message types used in the Vos communication protocol. A schema of a normal execution of this protocol is given in Figure 63. The names which are surrounded by single quotes are actually the value of the variable within the single quotes (in string format). They are written like this for readability.

(Client -> Server)
    [Arrived]
    [time=111; score=72]

(Server -> Client)
    [MissionSuccess]
    [clue=P]

(Client -> Server)
    [Guess]
    [word=GHP]

(Server -> Client)
    [QuestSuccess]
    [score=209; time=363]

(Server -> Client)
    [WrongGuess]
    [score=179]

(Client -> Server)
    [GetPlayerStats]

(Server -> Client)
    [PlayerStats]
    [totalscore=4209]

(Client -> Server)
    ['LOGIN_REQUEST']
    [Player name=Gunnar]
Figure 63: A normal execution of the Vos communication protocol.

(Server -> Client)

[’LOGIN_RESPONSE’]

[Quests=Test quest; Test quest=<1, 1>; Type=Human;  
  Class=rpg.common.battleCreature;  
  Position=<3.0, 3.0>;  
  12, ‘MAX_MANA’, 12, ‘STRENGTH’, 5,  
  ‘WISDOM’, 5, ‘LEVEL’, 1, ‘EXPERIENCE’, 0,  
  ‘EXPERIENCE_NEXT_LEVEL’, 200>;  
  Spells=’Fire blast’, ‘Lightning bolt’, ‘Earth quake’,  
  ‘Ice storm’;  
  Items=’Stone’, 85, ‘Coin’, 20, ‘Mana potion’, 10,  
  ‘Health potion’, 22>;  
  Name=Gunnar; Weapon=Sword;  
  User=sip:gunnar@example.com]

(Client -> Server)

[’LOGOUT_REQUEST’]

[Quests=Test quest; Test quest=<1, 1>; Type=Human;  
  Class=rpg.common.battleCreature;  
  Position=<3.0, 3.0>;  
  10, ‘MAX_MANA’, 14, ‘STRENGTH’, 7,  
  ‘WISDOM’, 7, ‘LEVEL’, 2, ‘EXPERIENCE’, 234,  
  ‘EXPERIENCE_NEXT_LEVEL’, 400>;  
  Spells=’Fire blast’, ‘Lightning bolt’, ‘Earth quake’,  
  ‘Ice storm’;  
  Items=’Stone’, 83, ‘Coin’, 20, ‘Mana potion’, 10,  
  ‘Health potion’, 20>;  
  Name=Gunnar; Weapon=Sword;  
  User=sip:gunnar@example.com]

(Server -> Client)

[’LOGOUT_RESPONSE’]

[Logout succeeded=true]
D.2.4 The CamGame Protocol

This is a listing of all the message types used in the CamGame communication protocol. A schema of a normal execution of this protocol is given in Figure 64.

Figure 64: A normal execution of the CamGame communication protocol.

(Client -> Server)
[start]
[game=camgame; difficulty=normal]

(Server -> Client)
[songlist]
[songs=<song1, 1, song2, 2, ...>]
name, id ...

(Client -> Server)
[getsong]
[song=2]

(Server -> Client)
[song]
[sequence=<65, 10, 62, 15, ...>]
midivalue, length (ms) ...
(Client -> Server)
[play]
[song=2]

(Server -> Client)
[songandpics]
[sequence=<65, 10, FF014B, FF014B, FF014B, FF014B,
   62, 15, ...>]

midivalue, length (ms), upper left color, upper right color, lower left
color, lower right color ...

(Client -> Server)
[regions]
[no=0; ul=F145B6; ur=FA0B50; dl=FB2152; dr=E9025C]

note index, upper left color, upper right color, lower left color, lower right color

(Server -> Client)
[res]
[no=0; result=64; duration=10]

note index, resulting midivalue, length

(Client -> Server)
[done]

(Server -> Client)
[end]
[score=423; rank=Neat;
   comment=Your current highscore is 612 Try again]

(Server -> Client)
[time]
[left=2]

minutes

(Client -> Server or Server -> Client)
[invalid]
[msg=Message did not contain a valid game]
E Database Structure

Each of the four prototypes makes use of a database hosted on the external server. The database runs MySQL, and the servlets connect to it by using the Hibernate object-relational mapping library. This appendix shows conceptual schemas of each prototype’s database, represented using Entity-relationship (ER) diagrams, along with short descriptions.

E.1 The Tourist Guide

The Tourist Guide database structure is shown in Figure 65. The core of this database model is the Tour table. Each Tour has a number of Places and a Place can belong to many Tours. A Tour has a map which is basically a Picture and its coordinates. Both Tours and Places have Pictures.

Another important part of the database is the User table which contains user info. The coordinates of the user is kept up to date in the database as well. Since the Tourist Guide prototype supports user-added content, the database keeps track of who added content (such as a Place or a Picture) and when it was added.

E.2 Agent SQ

The structure of the Agent SQ database is shown in Figure 66. The table that keeps track of the game state is the CurrentQuest table, which is used to keep track of which Mission and Quest a Player is on. The Clue table contains all the clues a Player has to find and what clues he already have found. Each mission in the Mission table has a Place similar to the Tourist Guide. This means that part of the database can be shared with the Tourist Guide, which makes it possible for a user to easily add a Place using the Tourist Guide client and then create a mission to be used in the Agent SQ game with it.
E.3 Vos

The structure of the Vos database is shown in Figure 67. As can be seen, there is only one table, called Datagram, used in this database. The reason for this is that we store all the data in a binary format, in the data column of the table. The type column contains information about what type of data is stored in the data column. The user column contains the SIP address of the user whose information is stored in that row.

E.4 CamGame

In the CamGame database, shown in Figure 68, there are three tables: Users, Highscores and Songs. The Users table contains all users that have
played the game, and information about them such as their SIP address, nickname and a unique ID. The Songs table contains all playable songs. Every song has a name, a melody, a unique ID and may also store who added it. The melody of each song is stored as a binary large object (or BLOB), which technically allows the contents to be anything, but in order for the program to work it has to be on this form (bytewise):

| tone | length | tone | length | ... |

Every user may have one high-score per difficulty and song. The best score, and when it was played, is stored in the Highscores table.