TAMPERE UNIVERSITY OF TECHNOLOGY

MARKO TUOMINEN

MULTI-PLAYER TACTICAL ROLE PLAYING GAME WITH A JAVA SERVER AND AN ANDROID CLIENT

MSc thesis

Supervisors:

Prof. Tommi Mikkonen
MSc Juha-Matti Vanhatupa

Examiners and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 7 November 2012.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
TUOMINEN, MARKO: Moninpelattava taktinen roolipeli Java-palvelimella ja Android-asiakkaalla
Diplomityö, 88 sivua
Joulukuu 2012
Pääaine: Ohjelmistotuotanto
Tarkastajat: professori Tommi Mikkonen ja DI Juha-Matti Vanhatupa
Avainsanat: Java, MySQL, JDBC, Apache MINA, JSON, JSON-RPC, jabsorb, Android

Tietokone- ja konsoliroolipeleissä pelaaja tavallisesti ohjaa yhtä tai useampaa hahmoa. Hahmot tavallisesti keräävät kokemusta taistelemalla vihollisia vastaan. Kokemuksen kautta hahmon ominaisuudet vahvistuvat ja se saa käyttöönsä uusia kykyjä. Peleissä voi myös kerätä esineitä, kuten aseita ja varusteita.

Taktisissa roolipeleissä on myös nämä ominaisuudet, sekä taktisia elementtejä kuten liikkuminen laajemmalla taistelukentällä. Roolipeleissä juoni on tavallisesti tärkeällä sijalla, samoin kuin pelimaailman tutkiminen. Myös taktisissa roolipeleissä juoni on tärkeä, mutta tutkimista on useimmiten vain vähän. Ne eivät yleensä sisällä myöskään moninpeliä.

Työn tarkoituksena on toteuttaa aiemmin suunniteltu Mupe Force -peli: taktinen roolipeli, jonka pääpaino on moninpelillä. Moninpeli toteutetaan välittämällä pelaajien asiakaslaitteiden kommunikointi palvelimen kautta. Lisäksi palvelin tallentaa rekisteröityneiden pelaajien tiedot. Asiakkaat eivät kommunikoi toistensa kanssa suoraan.

Ennen työn aloittamista oli päätetty, että pelin palvelinohjelmisto toteutettaisiin Javalla ja asiakasohjelmisto Androidilla. Tietojen tallennukseen oli päätetty käyttää MySQL-tietokantaa ja JDBC:tä. Asiakkaiden ja palvelimen väliseen kommunikointiin käytettävää teknologiaa ei päätetty etukäteen, vaan se valittiin työn aikana tehtyjen tutkimusten perusteella.

Valittu yhteysteknologia on Apache MINA, jolla toteutettiin kommunikointi sekä palvelin- että asiakaspäässä. Viestinvälitykseen käytetty protokolla on TCP, ja viestit välitetään JSON-formaatissa. Asiakkaan palvelimelle lähettämiin kutsuihin käytetään JSON-RPC -protokollaa. Työssä tutkittiin myös useampia Java-pohjaisia JSON-RPC -toteutuksia, joista käyttöön valittiin jabsorb. Sitä käytetään muuhunkin JSON-viestien käsittelyyn, koska Java API ei sisällä JSON-tukea.

Aikomus oli käyttää JSON-RPC:tä tai vastaavaa protokollaa mahdollistamaan asiakkaan metodien kutsuminen palvelimelta. Tämä todettiin kuitenkin liian monimutkaiseksi. Aiottu käyttötarkoitus, pelin tapahtuminen välittäminen asiakkaalle, toteutettiin JSON-pohjaisella skriptillä. JSON-skriptiin suunniteltiin pelin tarkoituksiin sopiva syntaksi. Tapahtumat koodataan skriptiksi palvelimella, ja puretaan asiakkaalla animaatioiksi ja paikallista tietoa muokkaaviksi käskyiksi.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
TUOMINEN, MARKO: Multi-player tactical role playing game with a Java server and an Android client
Master of Science Thesis, 88 pages
December 2012
Major: Software Systems
Examiners: professor Tommi Mikkonen, MSc Juha-Matti Vanhatupa
Keywords: Java, MySQL, JDBC, Apache MINA, JSON, JSON-RPC, jabsorb, Android

In computer and console role playing games, the player usually controls one or more characters. The characters usually earn experience by fighting enemies. Experience causes a character's attributes to increase and enables it to learn new abilities. The games also include collecting items, such as weapons or armor.

These properties also exist in tactical role playing games, which also include tactical gameplay, such as movement on a larger battlefield. Plot and exploration of the game world are important aspects of role playing games. Plot is also important in tactical role playing games, but they usually include little exploration. They also do not usually include multi-player play.

The intent in this thesis was to implement a previously designed game named Mupe Force. It is a tactical role playing game, whose main attribute is multi-player play. To facilitate multi-player properties, communications between client devices are transmitted through a server. In addition, the server saves the data of registered players. Clients do not communicate directly with each other.

It was decided in advance that the server software would be implemented with Java, and the client software with Android. It was also decided to use a MySQL database and JDBC for persisting data. The technology for communications between clients and the server was not decided in advance, but chosen based on research during the project.

The selected communications technology is Apache MINA, which was used to implement communications both on the server and on the client. The protocol for transmitting messages is TCP, and messages are delivered in JSON format. JSON-RPC is used for requests from a client to the server. Several Java-based implementations of JSON-RPC were researched, and jabsorb was selected for use. It is also used for other handling of JSON messages, because the Java API does not include JSON support.

The intent was to use JSON-RPC or a similar protocol to enable calling a client's methods from the server. This was found out to be too complex. The intended use, delivering game events to the client, was implemented with a JSON-based script. A syntax suitable for the game's purposes was designed for the JSON script. Events are encoded into a script on the server, and decoded on the client into animations as well as commands that modify local data.

PREFACE

This thesis offered me a chance to implement a game I had been planning for some time. Unfortunately I had to leave out many of the features I had intended for the game, but on the other hand I never expected to be able to implement all of it with the amount of work estimated for a Master of Science thesis. I was also able to learn JDBC and Android, both of which I had been thinking of trying out, learn more about networking technologies, and develop a working piece of software with them.

I would like to thank the supervisors Tommi Mikkonen and Juha-Matti Vanhatupa for their advice and comments, which helped improve both the application and the thesis. Since the original version of this game was implemented as a course assignment for Game Programming, fall 2007, using the MUPE environment, I would also like to thank the people involved with the course and MUPE.

Lastly, credits to the free open source applications that I used while working on this thesis. Linux, Ubuntu and Gnome that keep the computer running and usable, OpenOffice.org Writer for text editing, Eclipse for programming, Dia for drawing UML diagrams, Firefox for browsing the Internet, Gimp for image editing, MySQL and related tools for database management, and of course the Eclipse plugin, emulator and other tools for Android programming.

Tampere 21.11.2012

Marko Tuominen

TABLE OF CONTENTS

ABBREVIATIONS, TERMS AND MARKINGS

Android	Android is advertised as the most used and fastest growing mobile operating system. It is based on Linux, uses user interface resources defined in XML and utilizes Java for applications programming. [1]
API	Application Programming Interface (API) is an application's interface which allows other applications to communicate with it. [2]
APR	Apache Portable Runtime (APR) intends to create and maintain a predictable and consistent interface to platform-specific implementations. It means to assure software developers of predictable behavior regardless of platform, and remove the need for special-case code for platform-specific features. [3]
back end	The data storage and manipulation part of an application. Receives input from the front end, and sends output to it. [4]
bytecode	Intermediary code produced by a compiler. It is designed for efficient translation into machine code and can be interpreted by a runtime environment. Runtime environments can be created for different hardware and software environments, allowing cross-platform portability. [4]
C2DM	Android Cloud to Device Messaging (C2DM) is a service for sending messages from servers to devices. [5]
Comet	Comet is a term for multiple technologies whose purpose is to enable a server to send data to a client over HTTP without a request from the client. [6, p. 1]
deserialization	The recreating of a data structure or object from serialized data. Java has automatic deserialization which requires that a class definition for the deserialized object is available. [7]
Flyweight	A design pattern for sharing large numbers of objects efficiently. A Flyweight object can be used in multiple contexts simultaneously. The Flyweight's internal state does not depend on context, allowing the object to be shared. Client objects are responsible for passing context-related properties to the Flyweight as necessary. [8, pp. 195-196]
front end	The part of an application which interacts with the user. Sends input to the back end, and receives output from it. [4]
GUI	Graphical User Interface (GUI) is a means for interacting with a computer through graphical elements using a mouse. [9]
HTTP	Hyper Text Transfer Protocol (HTTP) is an Internet communication

	protocol. It defines requests, which a server receives from a client, and responses, which the server sends to the client in reply. [10, pp. 9, 30]
JSON	JavaScript Object Notation (JSON) is a textual format which is easy to read for both humans and machines. It is language independent. [11]
JSON-RPC	A protocol for remote procedure calls using JSON. [12]
MUPE	Multi-User Publishing Environment (MUPE) is a platform for rapid mobile development of multi-user applications [13, p. 1].
MVC	Model-View-Controller (MVC) is a design pattern for relating user interface to data. The model is a representation of underlying data, the view is a representation of the user interface, and controller represents the connections and communication between them. [14]
MySQL	The most popular open source relational database management system in the world. [15]
Observer	A design pattern that defines a dependency where multiple objects are notified when the state of one object changes. A subject can have any number of observers, which are all notified when the subject's state changes. [8, pp. 293-294] The subject is implemented in the Java API class `Observable`, and the observer in the interface `Observer`. [16]
OpenGL	Open Graphics Library (OpenGL) is an API specification for rendering graphics. It is not platform-specific, and many graphics cards have an implementation of OpenGL. [17]
REST	Representational State Transfer (REST) or RESTful web services, is a design idiom for a stateless client-service architecture. A web service is viewed as a resource identifier by an URL. Clients transfer content using a small set of actions: create, read, update and delete. [18]
RPC	Remote Procedure Call (RPC) is a protocol for a program to call a procedure in another program remotely. The other program can be running on another computer or in another network. The RPC protocol hides the network details. [19]
RPG	Role-Playing Games (RPG) allow a player to experience a series of adventures in an imaginary world through a character, or a group of characters. Essential game elements are story and character growth, and combat is usually required. [20]
serialization	The process of converting an object into a byte stream that allows the object to be recreated later. Java has automatic serialization. [7]
Singleton	A design pattern that ensures there is only one instance of a class, and that a global access point for the class exists. The class controls instantiating and access to the Singleton instance. [8, p. 127]
SMS	Short Message Service (SMS) is a means for sending short text messages from a cell phone to another, and is often referred to as texting or

	text messaging. [21]
SOAP	Simple Object Access Protocol (SOAP) is a means for generalized XML messaging in various environments. [22]
SQL	Short for Structured Query Language. A standardized language for managing relational databases. It can insert, retrieve, update and delete data, and also includes schema creation and modification, as well as access control. There are multiple versions of SQL, and in addition most SQL database programs have proprietary extensions. All of them support at least the major commands, though. [23]
SQLite	SQLite is a relational database which can be embedded in an application, unlike most systems which are separate server processes. It uses non-static data types for database columns, and offers several extensions to the SQL language. [24]
TCP	The Transmission Control Protocol (TCP) provides reliable, ordered delivery of a byte stream from one computer to another. [25]
TRPG	Tactical Role Playing Games (TRPG) combine strategic or tactical gameplay with RPG elements. They commonly include chessboard-like square grids, where characters are placed. Placement is vital to gameplay, and success in the game. [26, 27]
UDP	The User Datagram Protocol (UDP) sends separate messages without guarantees for the reliability or order of delivery. [25]
URI	A Uniform Resource Identifier (URI) is a character string that identifies a resource. URI can be separated to two types, URL and URN. [28, 29]
URL	A Uniform Resource Locator (URL) is a type of URI which identifies the location of an Internet resource. [28, 29]
URN	A Uniform Resource Name (URN) is a type of URI which identifies a resource by name, independent of location. [28, 29]
XML	Extensible Markup Language (XML) defines rules for representing structured information in a textual format readable for both humans and machines. It is widely used for sharing documents, data, configurations, books and so on. [30]
XML-RPC	A remote procedure call over HTTP and with XML encoding. [31]
XMPP	Extensible Messaging and Presence Protocol (XMPP), originally known as Jabber, is a free, open standard for real-time messaging. [32]

Important terms are written in *italic* when they first appear.

The names of Java classes and primitives and other code is written in `monospace`.

Code paragraphs are separated from other text with extra space above and below.

The names of design patterns, major parts of Android and other frameworks, as well as components of the application implemented for this thesis, are written with Capital

Letters.

The term *user* is used in reference to a real person interacting with the application developed in this project. The term *player* refers to an entity in the game. A player can be the user or another real person using a different device. Multiple players are handled in the game, and the term player can also refer to their runtime representations as objects. This is only in the context of the application developed for this thesis, however.

1 INTRODUCTION

Tactical role playing games (TRPG) blend strategic gameplay with elements from role playing games (RPG) [27]. The intent in RPGs is to experience a series of adventures in an imaginary world through a character, or a group of characters. Essential game elements are story and character growth, and combat is usually required. [20, pp. 453-454] Character growth commonly occurs through earning experience points [20, p. 466]. Physical coordination challenges are rare, as the success or failure of actions depends on a character's attributes rather than those of the player. [20, p. 455; 26] RPGs are often turn-based, although this is not required. [33]

A purist interpretation defines RPGs by three factors: a set of statistics determines the attributes of a character, they can increase making the character grow stronger, and the game features a menu-based combat system that uses those statistics. Furthermore, any game that involves a grid-based map and directly placing characters on it, and where placement is vital for success is a strategy game, or a strategy-RPG fusion. [26] In another interpretation, this fusion of strategy and RPG is called a tactical RPG [27].

Besides the features mentioned above, RPGs usually include equipment for characters, as well as a world that can be explored. The world can be a dungeon, but often the game includes an open world, multiple dungeons and multiple towns. Some games aim for a believable, working world. Common parts of the world are non-player characters that can be talked to, as well as shops. A player's characters can collect treasure, acquire equipment for themselves, and use magic or similar powers, as well as other sorts of skills. [33; 34, p. 2; 35; 36]

TRPGs generally include these aspects, but also incorporate strategic gameplay, such as tactical movement on a grid [27]. An element featured in many TRPGs is the rewarding of experience to characters individually. Having a weaker character stay safely away from fighting causes it to fall behind other characters in terms of development. [35] TRPGs commonly do not include multi-player play, or exploration, which is a major feature in standard RPGs. [37]

The goal in this thesis is the implementation of Mupe Force, a multi-player TRPG. Multi-player play is the key aspect of Mupe Force, to the extent that exploration and story are completely omitted. The thesis involves research for technologies and libraries used in the implementation, as well as specification and design. The name Mupe comes from Multi-User Publishing Environment (MUPE), an open source project which was used to implement (a much simpler version of) Mupe Force as a course assignment. MUPE source code was not used in this new version of Mupe Force. The MUPE project

has been discontinued, and its website (http://www.mupe.net) no longer exists.

The basis for implementation is Mupe Force specification, which was originally written to be platform-independent. The plan is to use a client-server model, where the server stores user data, and acts as a mediator between clients.

The server is implemented with Java SE 1.7, using a MySQL database for data storage and JDBC for database connectivity. The client is implemented with Android 2.3.3 (API level 10).

MySQL is the most popular open source database management system in the world. It manages a relational database, where data is described as tables with rows and columns, and relations between them. [15]

Android is a Linux-based mobile operating system whose applications are programmed in Java [1]. The means of communication between the server and the client are not decided in advance. Possible communication methods are studied in this thesis. Application Programming Interface (API) is an application's interface which other applications can communicate with [2].

Some aspects of the application are ruled outside the scope of this project, for practical reasons. These are security, error management, player registration and sound. Graphics are also limited to simple 2D sprites, with little animation.

In a client-server application, security concerns could well grow into the size of a thesis by themselves, and likewise for handling database and network errors. Registration, on the other hand, is considered best suited for a website. Since one is not developed in this project, registration is done with a command line application which uses the server's code to create a new player and insert his data into the database.

Chapter 2 is a shortened version of the original Mupe Force specification. Implementing all of the functionality described in the specification was not plausible within the time allotted for the thesis. Various aspects of the graphical user interface (GUI) have also been removed, and adapted for Android in later chapters. A GUI is a means for interacting with a computer through graphical on-screen elements using a mouse [9].

Chapter 3 introduces Android programming. It is intended to cover the structure and main parts of an Android application, as well as techniques used in this project. Chapter 4 contains research for implementing the communication between client and server. Several technologies were researched, although most of them were ultimately not used in this project. Some comparison between the technologies and discussion of their applicability to this project are included. Chapter 5 introduces the Java technologies used in implementing the server.

Chapters 3-5 form the theory part of this thesis. Chapters 6-8 describe the application's design, and how the theory was applied. Chapter 6 has the design for the server. All of the server's design is in this one chapter, but client design was split into two chapters for clarity, and because it is much longer. Chapter 7 has the design for the client back end. Back end refers to the part of the application which stores and manipulates

data [4]. This means the parts which communicate with the server and the database, but not with the user. The client's runtime data model is also described here. Chapter 8 is the client's interface design. The design extends the interface specification in Chapter 2 with more detail. The interface is also approached in an Android-specific way, unlike in Chapter 2.

Chapter 9 finishes the thesis with conclusions and notes about further development. It includes details of Android which proved important in the project, but were not apparent from the documentation used as source material for Chapter 2.

2 MUPE FORCE: SPECIFICATION

This chapter describes the Mupe Force game from a user's point of view. Mupe Force uses a client-server model. Since the user only interacts with the client, there is only a short introduction of the server's role in the application. Interface design is also left somewhat open, so that it can be adapted to a platform's specifics. This specification tells options available to a user in a particular view, and possibly some guidelines on how to access the options. The form of input for the options is not specified.

The interface is intended to be comfortable to use, easy to learn, and it should also conform to the platform's common interface design. In each view, the user must have an option to return to the previous view, unless otherwise stated.

2.1 Application architecture

The server holds and persists the application's data. Clients connect to the server, which acts as a proxy for communication between clients. Clients do not communicate directly with each other. A diagram of the application's architecture is shown in Figure 2.1. It is intended that the server can concurrently support multiple different connection methods, which in turn can support multiple different kinds of clients.

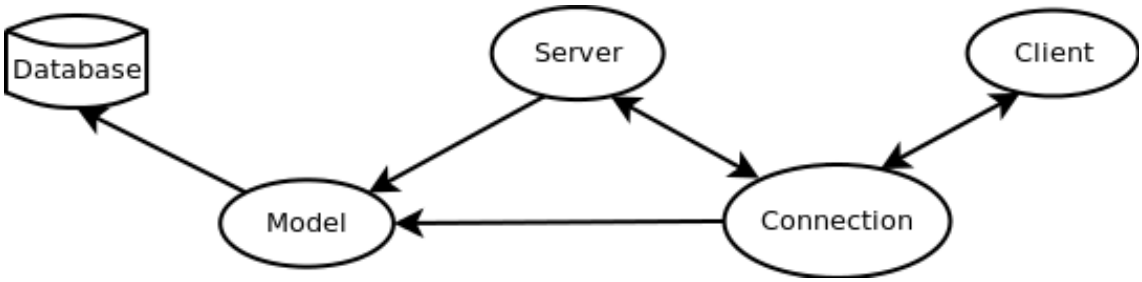

Figure 2.1: Overall application structure.

It is also intended that new content could be added on the server without changes to the client. The client is designed to emphasize executing actions received from the server over determining actions itself. Decisions made by the application, such as random values, are done on the server, and results sent to the client. This will help in maintaining equal state between the server and multiple clients.

The client needs to be able to send requests and notifications to the server, and receive data in response to the former. The server also needs to be able to send notifications to the client, although it does not need a response. The client must send in a way that allows new messages to be sent and received while waiting for a response. A re-

quest must not block the client's communication until a response is received. The client should not continuously poll the server in order to receive responses and notifications.

2.2 Data content

A Mupe Force player can have money, characters and items. The currency for money is simply called “gold”. Characters can equip and carry items. The selection of all characters owned by a player is called roster, and the selection of his items is called storage. The set of items equipped or carried by a character is called inventory.

A player is the entry point to the game's data content. A player has characters and items, and characters in turn have items and abilities. Abilities are specific to a character. Items belong to a player, and a player's storage also includes any items in characters' inventories. A diagram of the game's data content is shown in Figure 2.2.

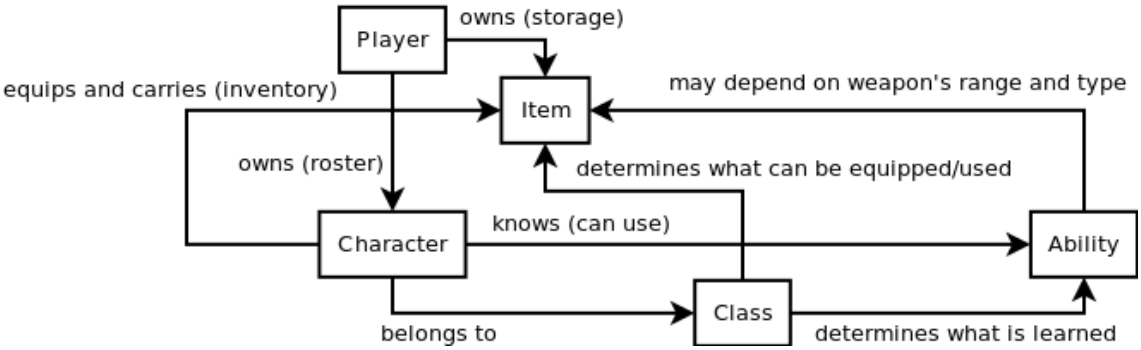

Figure 2.2: Diagram of data content.

2.2.1 Players

There are registered players and guests, whose difference is that the information of guests, and all data related to them, lasts only until the end of a session, or logging out in other words. A registered player's data is persisted, and will be available next time the player logs in. Each registered player has a unique name, while each guest player is called a “Guest-” and an ordinal number to make the name unique.

Each player initially gets up to six characters for free. He also gets some money, which can be used to buy items from shops. The player receives money from selling items, but the primary means of earning money is fighting battles. Some money is received as a prize, even if the player loses the battle.

2.2.2 Characters

Each character has a name, which is randomly selected when the character is created. The random names are checked so that a player does not get two characters with the same name. Each character belongs to a character class, which determines its starting statistics, and various other things.

There are six classes to select from, when a character is first created. A character

starts at level one, and can have some abilities from the start. A character gains experience through fighting, and once it has reached 100 experience, its level increases. At level up, the character's statistics increase and it can learn new abilities, depending on its class. Experience depends on the difference in levels of a character and the opponent. As a character becomes stronger, it will have to fight stronger enemies.

A character has ten statistics, each of which has an abbreviation to conserve screen space in user interface. The statistics are hit points (HP), magic points (MP), attack (Att), defense (Def), magic (Mag), resist (Res), hit (Hit), dodge (Ddg), speed (Spd) and move (Mov). Magic points are explained in the following subsection together with abilities. Other statistics are explained in Section 2.9 together with combat.

2.2.3 Abilities

There are two types of ability, in terms of how they work: active and passive. Passive abilities take effect as soon as they are learned, and then remain in effect. The ability can have a permanent effect, or it can take effect at random times. Active abilities must be used during battle in order to take effect.

Active abilities can have a cool down period, which limits how often they can be used. If they are magical in nature, they also consume magic points. If a character has insufficient magic points for an ability, it can not use that ability. Some active abilities can only be used when the character is equipped with a certain type of weapon. The weapon also determines the range of some active abilities.

2.2.4 Items

Characters can equip items, in a similar fashion to passive abilities, and also use items in a similar fashion to an active ability. Each character can equip a weapon, a shield, a helmet, and an armor. Double-handed weapons also require the shield slot, and prevent the character from using a shield. In addition to equipped items, a character can carry up to four other items, to be used during battle.

When an item is equipped, it increases the character's statistics, although decreases are also possible. An example of statistics changes is shown in Section 2.6. Using a miscellaneous item causes a special effect, depending on item, but also consumes the item. A character's class limits items that it can equip and use.

2.3 Logging in

At startup, a login screen with the Mupe Force logo is shown. The screen includes login options: log in with player name and password, or log in as guest. The client does not include registration functionalities, so the player name and password must already exist in the server's database.

The player is offered a choice to save the login information and automatically log in

in the future. This choice is not offered if the player logs in as a guest. With automatic login, the login screen is shown only while the client's initialization and logging in is executed.

2.4 Initial characters

When a player first logs in, either as a registered player or as a guest, he is given the chance to select one to six free characters. He must select at least one character, because otherwise he would not be able to do much of anything in the game. The character selection view opens after logging in, and has six character images and six empty slots.

The six character classes are fighter, knight, archer, rogue, mage and healer. Images of them are shown in Figure 2.3. The player can place one character into each of the six free slots, which represent their new characters. After a character has been placed, it can still be replaced with another or removed.

Figure 2.3: Screen shot: character classes.

At any time, the player can select a “done” option to close this screen and go to the actual game. They can select fewer than six characters. In that case, this screen is shown again when the player next logs in. Once all six characters are selected, this screen will never open again. A “do not show again” is available. If it is selected, this view will not be opened again later, even if the player has selected fewer than six initial characters. This option is not available for guests, since they can log in only once.

2.5 The town plaza

The town screen opens after logging in, or after selecting initial characters if they are available. The town is a five by five grid of squares, depicting an open plaza, surrounded by buildings. The player is represented by a character that can move in the squares, and enter any building which has a door on the edge of the square where the character is. The plaza itself is a three by three grid, which has twelve square edges where a building entrance can be placed.

In the center square is the portal, where the player's character appears at first. At the top center of the town screen is the way to the market square, where the shops are. At the right center is the entrance to the arena, where the fighting takes place. At the left center are the city gates, which do not open.

For visual effect, each square in the town view is represented by sixteen squares, a

four by four grid. The whole town grid becomes a twenty by twenty grid. The character on the other hand is only two by two squares. Doubling the size of the town makes the buildings larger, so that the character is not the same size as the building it is entering. Doubling the size of both the character and the town effectively allows the character to move by half squares. The character can be centered in the town view, as both the town and the character have an even width and height in terms of squares. Each building is four squares, with a character-sized two square door and a wall square on either side.

Squares with buildings are blocked and can not be entered, except the squares with doors. The town gate is one square wide (in the twenty by twenty grid), and blocks movement into those squares. The rightmost squares trigger movement into the arena. The character will walk out of the town view, and a battle will start, as soon as there is an opponent to fight. When the user leaves the arena, the character moves back into the town view, stopping so that the rightmost squares are empty. This re-enables user input, which is disabled as soon as the arena entry is triggered.

2.6 The market square

The secondary town screen, market square, is also a five by five grid. The player enters through the bottom middle square edge, which has the way back to the plaza. Buildings on the market square include weapon shop, armor shop and item shop.

Instead of a three by three square open area, the marketplace has three open squares on the left, and three at the bottom. There are five open squares altogether, since the square at the bottom left is shared. The other four squares are taken by the actual marketplace, which has no significance.

The weapon shop sells weapons and the armor shop sells protective equipment. Other than the selection of merchandise, these shops work in the same way. The shop's main view has a list of item categories, a list of the player's characters, an “all” option, and a “show own items” option. By selecting an item category, or the “all” option for all categories, the player is shown a list of merchandise and their costs. Available items depend on the types and levels of the player's characters. The selection increases as characters ascend to higher levels.

Selecting an item for sale highlights the characters that can equip it. After that, selecting a character shows the character's total statistics and changes to them. Statistics which increase and ones that decrease, and ones that stay the same are identified with different colors, and the changes to statistics are also shown. The change is not shown when the value does not change. An example is shown in Figure 2.4.

Instead of selecting an item on sale, the player can select one of his characters. Items which the selected character can use are then highlighted. When an item is selected from the list, a “buy” option becomes available, and a description of the item is shown. If the player has enough money, the buy option causes the item to be bought and added

Figure 2.4: Screen shot: equipping an ax increases attack, but decreases hit.

to the player's storage.

If a character is selected before buying, the item can be automatically equipped on the character or placed in its inventory. The player is first asked for confirmation to equip the item, and if he declines, he is asked for confirmation to have the character carry the item. The carried item is placed in the first free carry slot. In case all slots are taken, the new item replaces the item in the first slot.

The player can deselect the current item or character, which updates highlights, and hides the statistics. The “show own items” option opens a list of the items in the player's storage, and their costs. This is how much the shop is willing to pay for the items. The player can select any item, regardless of shop, and sell it. There is no way to retrieve the sold item later. The buy option is replaced by a sell option, but otherwise the items for sale in the shop and the player's own items are handled the same way.

The item shop mostly works in the same way as the equipment shops. Since it sells items that can not be equipped, changes to statistics are not shown. Characters that can use an item are highlighted, and a character can be selected in order to highlight items that the character can use, as in the other shops. The option to have a character equip an item is not available, but a character can be selected to carry the bought item.

2.7 The game menu

The game menu is visible on the town plaza and the market square, but not inside shops. The game menu has the following options: characters, items, settings and log out. The “log out” option logs out the current player, and returns the game to the login screen. This allows a different player to log in using the same device. When the log out option is selected, automatic login is ignored in the login screen.

The “characters” menu option opens a list of the player's characters, and allows the

player to select any one of them to view its status. The status view shows the character's name, level, experience, statistics, abilities and items.

There is also an “inventory” option. It opens an inventory screen with the character's equipped and carried items on the left, and all of the player's items on the right. The player can move items into, out of or between the equip and carry slots.

Items which the current character can equip or use are highlighted in the item list. When a slot is selected, items which can not be placed in that slot are removed from the list. Items that are equipped or carried by other characters are also removed.

The inventory screen has a character selector, which allows the player to select a different character's inventory without leaving the screen. When a different character is selected, or the player exits the screen, changes to a character's inventory are persisted.

The “items” menu option opens a list of the player's items, which also has a list of the player's characters. Selecting an item highlights the characters that can equip or use the item. Selecting any character highlights the items it can equip or use. There is also an “inventory” option, which is enabled only after a character is selected first. It opens the character's inventory screen.

The “settings” option in the game menu opens a player settings view. Remembering login data and automatic login can be disabled or enabled in this view. Both settings are device-specific, not user-specific, because there is no support for saving the login information of multiple users.

2.8 The arena

The arena is the place where battles take place. When the player enters the arena, a request is made to battle with another player. The player will either wait for another player to enter the arena on their respective client device, and then fight him, or immediately start a battle with another player who is already waiting. Waiting can be canceled, but this must be done before the battle begins.

Once a battle is started between the user and another player, there is a preparation phase. The user is shown a preparation view, with a list of his characters, and six empty slots. The player can move characters into the empty slots in order to have them participate in battle, and also replace and remove characters already in the slots. There is a “done” option to continue to the next phase once the player is finished with his preparations.

When a character is selected, its status is shown. The player can also access the character's inventory screen, just like in the character screen accessible through the game menu. This way, the player can select items for the battle.

After the preparation view, the player is taken to the battlefield, an eight by eight grid of squares. He can see the terrain and obstacles on the battlefield, but not the enemy's characters. The obstacles are placed in random positions on the battlefield, and

their number is random. The player can deploy his characters to the top two rows of the battlefield. The rest of the battlefield is shadowed. Once he is ready, there is a “done” option which starts the battle. However, both players must be ready, so one player might have to wait for the other to finish his preparations.

Each participant in the battle sees his characters start at the top two rows, so the battlefield is actually flipped upside down for one player. Regardless of the randomly placed obstacles, all free squares on the battlefield are reachable from any other free square, and there are at least six free squares in each player's deployment area.

2.9 Fighting a battle

A battle progresses in rounds and turns. Each character gets a turn during a round, which is composed of the turns of all characters. Once all characters have had their turn, a new round begins. The speed statistic determines turn order. Characters with higher speed will get their turns earlier, although some randomness is involved. The character whose turn it is is highlighted.

During its turn, a character can move, and then perform an action. Both of these are optional, and a character can also do nothing on its turn. In practice, a “do nothing” action, named “stay”, is performed. The move statistic determines how many squares a character can move. Moves are made either horizontally or vertically, and diagonal moves are not allowed. Obstacles block squares so that a character can not enter. Other characters also block movement, even allies.

The area where the character can move is shadowed, and the character can be moved freely around on this area. Once the player has placed the character in the square where he wants it, he can bring up an action menu. Possible actions are “attack”, “item”, “special”, and “stay”.

The “attack” action is a regular attack against an enemy. Most characters can attack an enemy in an adjacent square (horizontally or vertically), but some weapons allow a character to attack a longer distance away. The “item” action uses an item from the character's inventory, and the “special” action uses an active ability.

The “special” action is disabled if the character does not have any active abilities, and replaced with an active ability if the character has only one. The action for an ability that can not be used, due to cool down period, insufficient magic points, or wrong type of weapon, is disabled.

The “stay” action ends the character's turn without doing anything. The turn ends after any action, and the character remains in the square where it was when the action menu was opened.

2.9.1 Actions

Actions other than “stay” require a target. An ordinary attack's target is a hostile charac-

ter in an adjacent square. The user can either select a target directly, or move a cursor between possible targets and then select the current one. Any action that requires a single target works this way, even if the action can reach further away than the adjacent square. Squares within the action's range are shadowed.

Some actions can affect multiple targets. A main target is selected just like one for a single-target action, and the action will then affect other characters around the main target. Some actions can also be targeted at an empty square, and then affect characters around that square. An example of an action that can affect multiple targets and be targeted at an empty square is shown in Figure 2.5. Since the user can not move a cursor directly from target character to another, the cursor moves like a character. It can move across any obstacle, though. The cursor's shape and size indicates the area of effect.

Even though an empty square can be selected, the action must have at least one target which it can affect. An action can have separate types of target that it requires, and what it actually affects. An area effect attack spell might affect all characters, but must have at least one enemy character among its victims.

Figure 2.5: Screen shot: a mage about to blast three enemies with a Fireball.

All characters are capable of basic attacks. These can miss, depending on the attacker's hit statistic and the defender's dodge statistic. Chance to hit depends on the ratio between the two.

Attacks cause damage only if they hit. The damage depends on the attacker's attack statistic, and the defender's defense statistic. Damage is random, but close to the difference between attack and defense. Damage is more likely to be greater than the difference, than less. Damages further away from the difference are more unlikely. No matter how high the defense, minimum damage for attacks is one.

The damage caused by magical spells depends on the caster's magic statistic and the defender's resist statistic. Spells and special attacks are not as simple as basic attacks, though. They can modify statistics, as well as the damage itself, in various ways.

2.9.2 Death, experience, victory and defeat

As a character suffers damage, its hit points are reduced. When hit points are reduced to zero or negative, the character is killed and will not be able to participate in the battle anymore. The killed character is removed from battle, but will appear again after the battle ends, without lasting effects for having been killed. A dead character also keeps all the experience it had before it got killed.

Characters earn experience from damaging enemies. The stronger the enemy, the more experience is gained. Causing damage to an enemy is worth considerably less than delivering the killing blow. Minimum experience for causing damage (greater than zero) to an enemy is one. Sometimes using an item or an active ability is worth experience, even if it does not cause damage. For example healing is worth experience.

Once all characters from one player's side are dead, the other player wins the battle. Each player's characters can now gain levels, depending on the experience they gained during battle. Both players also get reward money. The reward amount depends on the number and levels of the opponent's characters. The winner gets a greater prize than the loser. Draws can happen if the battle ends because of an action that kills both players' remaining characters. After a draw, both players get a reward as if they were the loser.

2.9.3 Character status

The user can check any character's status at any time, even during the other player's turn. The status view is similar to that shown in the characters screen and the battle preparations screen. Experience is only visible for the user's own characters. Abilities, equipment and carried items are visible, but can not be changed during battle.

The status view shows the character's current hit points and magic points. They are shown in numbers, along with the maximum values. In addition, there is a bar graphic for each, which shows percentage of current value from maximum value.

3 ANDROID PROGRAMMING

The Android mobile operating system is based on a modified Linux. It includes a light-weight SQLite relational database for data storage, and supports multi-touch screens and multi-tasking applications. Android applications can be written with the Java programming language, using libraries provided by the Android runtime. [38, pp. 2-4] SQLite is a relational database which can be embedded in an application [24].

Among other things, Android has the most widely accepted Java user interface class library [39, p. 33]. Android's core libraries provide some of the Java Standard Edition (SE) classes, but not all, since for example Swing and AWT are not available. Also, classes specific to the Java Micro Edition (ME) are not available. [40, pp. 7-8]

Every Android application runs in its own process, with its own instance of Dalvik virtual machine [38, p. 4]. It runs bytecode called *dex* [39, p. 89]. Bytecode is intermediary code produced by a compiler and designed for efficient translation into machine code [4]. Dex is developed specifically for Android, and is approximately twice as space-efficient as Java bytecode [39, p. 89].

Designed for a single-user device, Android uses Linux's multi-user support to create separate privileges for different application vendors. An application is only allowed to access files created by applications from the same vendor. [39, p. 89]

3.1 Application components

The key part of an Android application's graphical user interface (GUI) is called an *Activity*. They usually fill the entire screen, and are also units of execution. A single application can contain multiple Activities, which can communicate with each other, other parts of the application, and with other applications to execute user functions. [39, p. 77]

Communication between Activities is done with *Intents*, which facilitate loose coupling between various components. Keeping references to Activity objects is discouraged and Intents should be used for communication between Activities instead. An activity chain is composed of multiple Activities, which can reside in different applications and processes. Activity chains like these are created by dispatching Intents to other Activities, and are called *Tasks*. [39, pp. 77-78]

Other important Android components are *Services*, *Content Providers* and *Broadcast Receivers*. Services perform background tasks and are not visible on the screen. A Broadcast Receiver listens for Intents broadcast in the system. It does not have a user in-

terface, but can activate an Activity when an Intent is received, for example. [39, pp. 78-79, 82]

Content Providers are similar to RESTful (Representational State Transfer) web services. They can be found with a Uniform Resource Identifier (URI), and provide operations for creating, reading, updating and deleting data. [39, p. 79] A RESTful web service is identified by a Uniform Resource Locator (URL), and provides four actions for content: create, read, update and delete [18]. A URI is a string that identifies a web resource [28]. A URL is a type of URI that identifies a resource by location. [29]

Content Providers can be used to persist data and provide it to other applications. The Android platform has several built-in Content Providers that allow access to user data. A query to a Content Provider returns the content in a Cursor, which is similar to a JDBC `ResultSet`. [39, pp. 80-81]

Content Providers are accessed through the `ContentResolver` object in a `Context`, which is the super class of `Activity`, among others [41; 39, p. 80]. The `ContentResolver` also contains `notifyChange` methods which deliver content change events to cursors. These events can be observed by registering a `ContentObserver` object to a `Cursor`. [39, p. 318]

The GUI components in Android are called *Views*. A View can contain other Views, forming a tree structure. Views that can contain other Views are subclasses of the `ViewGroup` class, and also handle screen layouts. [39, pp. 211-212]

A newer GUI feature is the *Fragment*, which was introduced in version 3.0 of Android (API level 11). Fragments are aimed for larger screens, such as tablets. Fragments are available for older versions as well, through a compatibility package. [39, p. 197]

Two important building blocks are Handler and Looper. A Looper has a task queue, which provides messages to it in order, and in a thread safe fashion. Other threads add work to the Looper's task queue, and the Looper simply runs in a loop executing the work it finds in its task queue. The Android UI thread is in fact a Looper, which gets all UI events in its task queue. A Handler receives messages and places them to the task queue of the thread where it was created. It is the means of delivering work to a Looper. The `View` class has methods for sending `Runnable`s to the main thread Looper for execution. [39, pp. 155-156, 192]

To create a Looper, the method `Looper.prepare` first needs to be called in the thread which is to be the Looper. After that, `Looper.loop` will process messages until the loop is stopped. These two methods are static, and so is `Looper.myLooper`, which returns the current thread's `Looper` object. The non-static `quit` method ends the loop. Because the `Looper` object is needed, quitting is only possible in the Looper thread, unless the `Looper` object is passed to another thread. [42]

Handler can be extended to handle messages in its queue, but there is also an interface named `Handler.Callback` which can be implemented instead. The `Handler` class has several methods for sending `Messages` and `Runnable`s, as well as a `Message` pool,

to avoid unnecessary creation of `Message` objects. [43]

3.2 Component life cycle

A component's life cycle is important for memory management, which in turn is important for multi-tasking on a device with limited memory. A component has several callback methods used in directing its life cycle. Among other things, there are methods for saving and restoring an Activity's data. [39, pp. 90-91]

When the Android system is running low on memory, the first thing it does is garbage collection. When that is not sufficient, components which are not currently visible can be destroyed. The third alternative is deleting a whole process, and all components in it. Before being destroyed, the component's state is saved for restoration at a later time. [39, p. 286]

An Activity can be destroyed at any time, and there is no guarantee that certain life cycle methods get called before this happens. Before an Activity instance is destroyed, its `onDestroy` method is called. After it is called, the instance will never be used again. When a process is killed, the `onDestroy` method might not be called. [39, pp. 287-288]

Destroying an Activity instance does not mean the end of the application, or the process running it. A new Activity instance can be created in order to continue the Activity. When a new instance is created, its `onCreate` method is called. It is called only once for an instance. [39, p. 288]

When a different Activity becomes visible, the old Activity stops interacting with the user. Then its `onPause` method is called. The `onStop` method is called after that, when the Activity is no longer visible. [39, pp. 288-289]

The `onStart` method is called when the Activity becomes visible to the user, after being created, and also when a stopped Activity restarts. In the latter case, `onRestart` is called before `onStart`. For a paused Activity, `onStart` is not called again without `onStop` being called first. Instead, only a method named `onResume`, which signifies the Activity starting to interact with the user, is called. It is also called after `onStart` for an Activity that has just been created, or is restarting after being stopped. [38, pp. 28-29]

The `onPause` method is the last that is guaranteed to be called, because paused Activities are liable to have their process killed. After `onPause`, `onSaveInstanceState` is called to persist the Activity's data. By default, the Activity calls for all its Views to persist their data, and after doing that, an Activity only needs to persist its own transient data. [39, p. 287-289]

Zechner and Green recommend saving all data in `onPause`, but make no mention of `onSaveInstanceState`. However, this is meant for securing persistent data before an application ends. [40, p. 128] The `onSaveInstanceState` method gets a `Bundle` object, whose data is passed to the `onCreate` method, or the `onRestoreInstanceState` method [39, p. 289]. This makes it much easier to migrate transient data between Activ-

ity instances. In addition, `onSaveInstanceState` is not called when there is no need to restore the activity's state later. [38, p. 108]

There is also one more method for saving state, which is invoked when device configuration changes: `onRetainNonConfigurationInstance`. It returns an object, which can be accessed later with `getLastNonConfigurationInstance`. [38, p. 109]

According to Komatineni and MacLean, “Content Providers do not have a particular life cycle. They get started when needed and stay around as long as the process stays around.” [44, p. 486] They only have one life cycle method, `onCreate`, which is called in the main thread of the application that creates the Content Provider. Content Providers are not required unless multiple applications need to share data. [45]

Like an Activity, a Service also has an `onCreate` method, which is called once when the Service is created. After that, its `onStartCommand` method is called. Once a Service is started, it can run in the background indefinitely, regardless of the component that started it. Usually, a Service executes one action and then terminates. A Service does not create its own thread, but runs in the hosting process's main thread. [46]

A Service can be bound to a client. Then the Service is created, if it is not already running, but the `onStartCommand` method is not called. Instead, its `onBind` method is called. A Service can be bound to multiple clients, and runs until all clients are unbound. Regardless of bound clients, a started Service runs until stopped. Once neither condition to keep the Service running hold, its `onDestroy` method is called, and the Service is terminated. [46]

A Service's resources are preferably not reclaimed, and they usually run until completion once started [39, p. 79]. Even if the Service's process is killed, the Service can be restarted later, but there still is no guarantee that the service can run to completion. [44, p. 485]

3.3 Data management

A Content Provider interface has methods for creating, updating, reading and deleting content. Each of these four actions takes a URI, which identifies the affected content. In addition, there are parameters similar to those used for SQL. [39, p. 315] SQL is short for Structured Query Language and is a standardized language for managing relational databases. It includes, among other things, `INSERT`, `SELECT`, `UPDATE` and `DELETE` statements for data access and manipulation. [23] Content Provider method names also reflect the SQL database which is commonly found backing up a Content Provider: `insert`, `query`, `update` and `delete`, respectively. The Android libraries provide a `UriMatcher` class, which makes parsing the identifier URIs easier, and also promotes standard URI formats. [39, pp. 315-316]

A Content Provider hides the actual location of data from clients, which do not need to know it, and do not have to be changed if the data location changes. This is especially

important for files, since the Content Provider owns the files which it has created. Another application should not access a file directly, but request it from the Content Provider instead. [39, p. 316]

The Linux file system used by the Android platform is much more efficient for storing large amounts of data than blobs in an SQLite database. Given a URI pointing to a column named `_data` in the database, the `ContentResolver` class can automatically open a file named by `_data`, and return an output stream for the file. [39, pp. 316-317]

A simpler alternative to using files or a database is `SharedPreferences`. Data is set and retrieved as key-value pairs, which are automatically saved to an Extensible Markup Language (XML) file. [38, pp. 203-204] XML is a format for representing structured information in a textual format readable for both humans and machines [30].

A `SharedPreferences` object is retrieved with an Activity's `getSharedPreferences` method. It takes a name for the preferences file, and a privacy mode. An `Editor` object from the `SharedPreferences.edit` allows the preferences to be changed, and the `commit` method saves the changes. Using another `getSharedPreferences` method which does not take a file name, the preferences are restricted to the Activity that created them. [38, pp. 207-209]

3.4 Networking

The Android platform supports sending Short Message Service (SMS) messages and e-mails programmatically, although the application needs a permission to send SMS [38, pp. 264, 267, 281]. Permissions are declared in a manifest, and shown to the user before installation, which can not proceed unless the user grants the permissions [38, pp. 280-281]. An SMS can also be sent through the built-in Messaging application, without the need for a permission [38, pp. 269-270]. SMS allows sending short text messages between cell phones, and is often referred to as texting or text messaging. [21]

The SMS manager can also be given an Intent to be invoked when the SMS is sent, and another to invoke after the SMS has been delivered [38. p. 267]. A Broadcast Receiver can listen for received SMS messages [38, p. 270].

Android libraries include the `java.net` package for basic socket and Hyper Text Transfer Protocol (HTTP) handling, as well as `org.apache.*` packages for advanced HTTP networking [47]. The Android platform supports many other networking-related Java packages as well, but not all [48]. HTTP is introduced in Section 4.1.

3.5 Graphical user interface

Android UIs are defined as a hierarchy of `View` and `ViewGroup` nodes [49]. A UI is typically defined in an XML file in the project's `res/layout/` folder [38, p. 81]. The XML offers a human-readable structure for a UI layout, where each XML element is either a

View or a ViewGroup. Views are leaves in the hierarchy tree and ViewGroups are branches [49].

The res/ folder can also include other types of resources besides layouts. Each resource receives a unique identifier in an automatically generated class named R. [39, pp. 87-88] In order to load a layout and put it to use, an Activity must call its setContentView method, and pass the UI layout reference from the R class [38, p. 25]. Layouts can also be included in other layouts using the <include> element [50].

Externalizing resources like this make them easier to maintain independent of the code. Specially named resource folders also allow providing different resources for different device configurations. Among other things, there can be different layouts and images for different screen sizes, and different strings for different languages. A resource identifier can also be an alias for another resource. [51]

An XML layout file is "inflated" into a View object. Usually this happens automatically, as an Activity gets an identifier for its layout through setContentView, inflates the layout, and adds inflated top-level View or Views to itself. Layouts can also be inflated with a LayoutInflater, or by using shortcut methods in View. [52; 53; 54]

3.5.1 Layouts

View groups determine the order of their child Views, and implement layouts. A View group can also include other View groups in order to create nested layouts. [49] Predefined View groups include (up to version 2.3) LinearLayout, TableLayout, FrameLayout, RelativeLayout and AbsoluteLayout, as well as ScrollView, which enables scrolling a single child View [38, pp. 83-96].

A linear layout arranges Views in a single line, which can be either horizontal or vertical [38, p. 83]. An absolute layout places its children in exact locations, specified in pixels. It does not scale well to different resolutions and has been deprecated since Android 1.5. [38, pp. 87-88] A table layout groups Views into rows and columns. The width of each column depends on the widest of the Views in that column. [38, p. 89] A relative layout specifies the positions of its children relative to each other [38, p. 91]. Views in a frame layout overlap each other, and can have gravities that determine their position on the screen [55].

The Views in a table layout can span multiple columns, and the table can also have empty cells between Views. A column can be shrinkable, meaning that it can be narrowed to fit the table inside the parent View. A column can also be stretchable, and expand to fill excess space. A column can be both shrinkable and stretchable. [56]

All Views are required to define layout_width and layout_height attributes in XML. View dimensions can be defined using exact values, but more commonly one of two constants, wrap_content and match_parent, are used. The former makes the View just wide or high enough to contain its content, and the second makes it as wide or as high as its parent allows. [57]

There are also various other layout attributes which can affect a child's size and position within its parent. An important one is weight, which determines how excess space is distributed, but it is not available for all layouts.

3.5.2 Views

An XML element depicts a Java object of a class whose name is the same as that of the element, without package [49]. Custom Views have the entire package name in the XML element's name, in addition to the class name. An inner class requires yet another notation: a `<view>` element with a `class` attribute whose value is the inner class's name, with package and the parent class's name. As usual in Java, the parent class's name is followed by a `'$'` character, then the inner class's name. [58]

Each View has several callback methods for events, such as `onClick`. A View can react to events by overriding these methods, but it is more convenient to attach a listener to a View without having to extend the `View` class. There are nested event listener interfaces for a number of different events. [59]

A user can navigate Views using navigation keys, a trackball, or a similar device, causing different Views to be focused. From here, any such navigation device will be referred to as navigation keys. Pressing the selection key causes the focused View to be clicked, triggering an `onClick` event. With a touch screen, the user can simply touch a View to trigger an `onClick` event. For this reason, Views are by default not focusable after the device enters *touch mode* when the user touches the screen. [59]

The device exits touch mode when navigation keys are pressed. After that, Views are again focusable. It is also possible to make Views focusable in touch mode. Focus changes with navigation keys are handled automatically, and changes for Views becoming unavailable are also handled. A View can be set focusable or not focusable, and focus can be requested for a particular View. [59]

There are various attributes for Views in XML, and these also have method and variable counterparts for runtime access. A View can have layout attributes, which depend on its parent, attributes inherited from the `View` base class or another super class, as well as its own attributes. [57]

An important attribute available for all Views is the identifier. It is assigned as a string in the XML, and becomes available as an integer of that name in the `R` class. Identifiers in XML are preceded by the `'@'` character. When a new identifier is created, it must have a `'+'` character after the `'@'` character. Views with an identifier can be accessed programmatically using the `findViewById` method, and the identifier from `R`. [57]

3.5.3 Menus

In applications targeted for older Android versions, up to 2.3 (API level 10), an options menu is opened with the device's Menu button. It appears at the bottom of the screen,

and can have up to six items. Additional menu items can be made available with a More button in the menu. It is automatically included together with the first five menu items when there are more than six menu items. Since Android 3.0 (API level 11) a Menu button is no longer required on Android devices, and Android applications are recommended to migrate from the traditional six-item menu panel to an action bar. [60]

The action bar API was added in Android 3.0. Its primary goals are identifying application brand and user location, providing consistent navigation and View refinement, and making key actions prominent and accessible. The action bar also includes an "overflow menu" button, which can be used to show actions that do not fit in the action bar. It performs the same function as the device Menu button. [61]

Like other GUI components in Android, a menu can be created as an XML resource, and doing so is recommended. Menu resources reside in the `res/menu/` folder, and their root element is `<menu>`. It can include `<item>` and `<group>` elements, where the latter groups items so that menu items share properties. An `<item>` element can also include a `<menu>` element in order to create a sub-menu. [60]

The callback for creating the options menu is `onCreateOptionsMenu`. On newer Android platforms, this is called when the Activity is started in order to create the action bar. On older platforms, it is called when the user opens the menu for the first time. The menu object is retained once it has been created, and `onCreateOptionsMenu` should not be used to modify the menu. Instead, there is a method that is called every time the user opens the menu, named `onPrepareOptionsMenu`. On newer platforms which have an action bar, the `invalidateOptionsMenu` must be called first, since the options menu is always considered to be open. When the user selects a menu item, the `onOptionsMenuItemSelected` method is called. [60]

In addition to options menus, there are context menus and pop-up menus, which affect a specific item. A context menu can be provided for any View, but most often they are used for View collections. A View that has a context menu is first registered using `registerForContextMenu`. After that, the `onCreateContextMenu` method is called when the View is long-clicked. The `onContextMenuItemSelected` method is called when the user selects a menu item. [60]

Android 3.0 replaces the traditional context menu with a contextual action mode. When the user selects an item, the action mode is enabled, and a contextual action bar replaces the usual action bar. The user can then select multiple items and perform actions on all of them at once. [60]

The aforementioned pop-up menu is similar to a context menu, but is meant for actions that "relate" to specific content, while the context menu is meant for actions that "affect" the content. It was added in Android 3.0. [60]

Menu items can be grouped, making the items in a group share properties. The items can be shown or hidden, enabled or disabled, as well as checked or unchecked all at once. Menu items can also be made into check boxes or radio buttons. The group's

checkable behavior is used for this. [60]

3.5.4 Dialogs

Android API Guide recommends not instantiating the `Dialog` class directly, but instead using `AlertDialog` and other predefined subclasses. An `AlertDialog` has an optional title, a content area which can have a message, a custom layout or other things, as well as action buttons. The `AlertDialog.Builder` class can be used to configure and create `AlertDialog`s. In order to set a custom `View` for an `AlertDialog`'s content, it must first be inflated from XML into a `View` object using a `LayoutInflater`. [62]

An Activity's `onCreateDialog` callback creates a dialog, which the Activity will then retain. The dialog is created only once, but the `onPrepareDialog` callback can update the dialog every time before it is shown. In order to show a dialog, the `showDialog` method is called. Its argument is an `int` identifier, which is passed to `onCreateDialog`, if necessary, and `onPrepareDialog`. In a similar fashion, `dismissDialog` will close a dialog depending on identifier, and `removeDialog` will delete a `Dialog` object from the Activity. All of these methods are deprecated, in favor of the `FragmentManager` class, which was introduced in Android 3.0 (API level 11). [52]

A dialog can be canceled with the Back button, touching the screen outside the dialog, or by the `Dialog.cancel` method. This indicates that the user explicitly interrupted the dialog's task. [62] Listener interfaces for dialogs include `onCancelListener`, which is called when a dialog is canceled, and `onDismissListener`, which is called when a dialog is dismissed [63].

3.5.5 Animations and graphics

Android has three methods of animation: Property Animation, View Animation and Drawable Animation. Property Animation was introduced in Android 3.0, but the other two are available in older versions as well. Drawable is an abstraction for “something that can be drawn” in Android terminology, and Drawable Animation involves displaying Drawable resources, such as bitmaps, as frames in an animation. There is also the Canvas class for custom 2D rendering, and support for Open Graphics Library (OpenGL). [64, 65] OpenGL is a platform-independent API specification for rendering graphics [17]. Property Animation and OpenGL are outside the scope of this text.

View Animations can perform a series of simple transformations, such as reposition, resize and rotate, on the contents of a View. An animation sequence can be defined in XML or Android code. The instructions define what transformations are to occur, when they will occur, and how long they will last. They can be sequential or simultaneous. Regardless of how transformations alter a View's content, they do not affect the View's bounds. Content can be drawn outside a View without clipping, but content drawn outside the View's parent will be clipped. [66]

There are two ways of drawing 2D graphics onto a View: use the system's normal

View hierarchy drawing process, or draw graphics directly onto a `Canvas` in a separate thread. The system drawing process is preferable when the graphics are simple and do not need to change dynamically. The `View.invalidate` method can be called to cause it to be redrawn, and the actual drawing happens in `View.onDraw`. Drawing happens in the system main thread, which also executes other UI operations. [67]

The separate thread option allows the thread to draw as fast as it can. For this approach, a class which extends the `SurfaceView` class is needed, as well as an implementation of the `SurfaceHolder.Callback` interface. It is recommended that the `SurfaceView` subclass also implements `SurfaceHolder.Callback`, and includes the drawing thread. The callback interface is used for notifying when the surface is created, changed or destroyed. [67]

The `Canvas` used for drawing is retrieved with `SurfaceHolder.lockCanvas`. It has various drawing methods, such as `drawBitmap` and `drawText`. It is also a parameter for the `draw` method of a Drawable, which draws itself. When drawing is finished, the `Canvas` is sent back to the system with `SurfaceHolder.unlockCanvasAndPost`. The `SurfaceHolder` can be retrieved with `SurfaceView.getHolder`. The `Canvas` retains its state from the previous pass, and the entire surface must be re-painted to properly animate the graphics. [67]

4 NETWORKING TECHNOLOGIES

This chapter contains information about several networking-related technologies which were researched for this thesis. Some of them were used in implementation, others were abandoned. Explanations include benefits for the implementation of this project, as well as reasons for not using certain technologies.

Since the server is written in Java, it is preferable to use a Java-based connection with a client. When that is not possible, the communication is abstracted so that it can be used with a variety of different platforms. In that case, messages can not contain anything Java-specific, but only the necessary information in the simplest plausible form.

4.1 HTTP: Hyper Text Transfer Protocol

HTTP is a widely used protocol on the Internet. It implements a *synchronous stateless request-response model*. A client sends an HTTP request to a server, which sends back an HTTP response. Hence, a *request-response model*. Being *synchronous* means that the client connects to the server, and waits while the server processes the request. [10, pp. 9, 30]

The HTTP protocol specifies request and response structure, and certain request types [10, p. 30]. A message includes a start line, zero or more header lines, an empty line to signify the end of the headers, and possibly a message body [68].

Request types include GET, which is meant to be read-only and not modify any data on the server, and POST, which can send information to the server, and enable responses based on the sent information and some business logic on the server. [10, pp. 15, 32]

Version 1.1 of HTTP adds six new request types: HEAD, OPTIONS, PUT, DELETE, TRACE and CONNECT [10, p. 32]. HEAD is identical to GET, except that the response must not have a message body. OPTIONS requests information about available communication options. PUT sends a resource which is to be stored with the request's URI as its identifier. DELETE requests the deletion of a resource identified by URI. TRACE invokes a loop-back, receiving the same request as the request's actual recipient. CONNECT is reserved for proxies that can dynamically become a tunnel. [69]

HTTP is programming language independent. It is also *stateless*, meaning that a new connection is opened for each request, and there is no built-in way to map requests to clients, or to save state between requests. Several solutions exist to overcome this deficiency. One way is attaching a token to each request, and using that token to identify a client and manage state. Using a solution like this, the server can create and maintain a

session with a client. [10, pp. 36-37]

HTTP is suitable for simple data retrieval and updates in Mupe Force. In other cases, the server should make information available only to a logged-in player whom that information concerns. For those, as well as updates, the client needs to be identified for data integrity.

Because HTTP is synchronous, it would be necessary to create multiple threads in some situations, to avoid blocking. This is still plausible, but a larger problem is that HTTP has no support for server sending a message to a client, except in answer to a request. HTTP is an integral part of several of the following technologies, and is not discussed further on its own.

4.2 Comet

The term Comet encompasses multiple technologies, and it is also known by several other names. The Comet technologies enable an HTTP server to send data to a client without the client requesting it. [6, p. 1] There are two types of implementation: *streaming* and *long polling*. [70]

Streaming involves opening a persistent connection from the client to the server. The server uses it to send events to the client, which handles the events. Neither side closes the connection. [70]

Long polling requires the client to poll the server for events. The polling connection is kept open until the server has some data to send to the client. After the client receives the data, it opens a new long polling connection. [70]

Comet technologies are aimed for use with web browsers, over HTTP, and using common browser technologies and standards such as HTML, JavaScript and Ajax [70]. As such, Comet technologies are not well-suited for use in a non-browser Android application. Similar techniques can be utilized with different technologies.

From the point of view of Mupe Force, streaming would be a favorable connection method over long polling. During a battle, messages will be frequently transmitted back and forth between client and server. Therefore, the persistent streaming connection would be more efficient than constantly opening new connections for long polling. Long polling also presumably blocks the communication thread until an answer is received. Since HTTP needs a new connection for each request, it does not allow streaming, but long polling is possible. This is an important consideration in deciding the technology for this project's networking.

4.3 XMPP: Extensible Messaging and Presence Protocol

Extensible Messaging and Presence Protocol (XMPP), originally known as Jabber, is an XML-based open standard for real-time messaging. Anyone may implement an XMPP

service and communicate with other implementations. There is no centralization, no royalties, and no single vendor. XMPP may be isolated from the public network, and its specifications include robust security features. [32] XMPP has an HTTP binding, which uses long polling to receive messages as soon as they are sent. [71]

An example of XMPP implementations for Android is AsmackService [72]. It is lightweight and specially designed for smart phones. It does not support a wide range of features, but only basic messaging features are required for this project [73]. As for Java-based XMPP servers, an example is Apache Vysper [74]. These were researched with the intention of using them in the project, but ultimately abandoned. Vysper depends on Apache MINA [75]. It is introduced later in this chapter.

4.4 C2DM: Cloud to Device Messaging

Android Cloud to Device Messaging (C2DM) is a service for sending notifications from application servers to devices. It handles message queuing and delivery, but makes no guarantees about delivery or message order. It is not designed for sending a lot of user content. [5]

Messages can also be delivered to applications that are not currently running. As long as a suitable Broadcast Receiver and permissions exist, the Android system can start up the receiver application as needed. It requires users to set up their Google account on the device. [5]

The Mupe Force client needs to communicate with the server only while it is running. Therefore direct communication is preferred, and a third-party messaging service is unnecessary. If future developments include notifications to a user while the application is not running, C2DM is worth considering.

4.5 JMS: Java Messaging Service

Java Messaging Service (JMS) implements an *asynchronous communication model*. It allows messaging directly from a client to a receiver, as well as from a publisher to any and all registered subscribers. A sender can put a message in a queue, and does not need to wait until the receiver gets the message. The receiver can pick up the message any time, once it has been placed in the message queue. [10, pp. 42-47]

Outside of battle, updates to server data in Mupe Force can be done asynchronously. The client can notify the server of the change, but does not need to wait until the server update is completed. Likewise, it can post a request for data, which the server sends when it is ready. The amount of time needed for server communication is shortened. In some cases, the data update can affect later data queries, so the update notifications need to be processed in order.

Internet discussions uniformly agree that JMS does not work on Android, and no

claims to the contrary were found. The recommended solution is transferring messages from JMS to XMPP on the server, and communicating between the XMPP server and the Android client [76]. This solution is suitable when there is a need to communicate with an existing JMS server. Since no server for this project exists yet, this solution has no merit over using XMPP without JMS.

4.6 Java Servlets

Servlets were created for serving dynamic content for HTTP requests. Later versions introduced session management. An HTTP servlet receives an HTTP request and an HTTP response, and can create content for the latter. Details of the HTTP protocol are hidden from the programmer. Each request is handled in its own thread. [10, pp. 80-82]

The HTTP servlet base class has default implementations of methods for HTTP GET and POST, as well as other types of requests. These can be overridden as necessary. The servlet container also manages sessions transparently, and allows the programmer to save key-value pairs in the session. There is also a context where data for the entire application can be saved, and data can be associated with an individual HTTP request. [10, pp. 82-84]

Since servlets are a server side means of HTTP communication, they have the problems associated with HTTP. There is no way of sending messages from server to client, except long polling. Therefore servlets were not used in this project.

4.7 Apache MINA: Multi-purpose Infrastructure for Network Applications

Apache MINA is a network application framework which provides an abstract event--driven asynchronous API over various transports [77]. MINA is an acronym for Multi-purpose Infrastructure for Network Applications. It currently supports TCP, UDP and transports based on Apache Portable Runtime. It can be used for creating both client and server applications, and implementing protocols that keep the connection alive. [78]

The Transmission Control Protocol (TCP) provides reliable, ordered delivery of a byte stream from one computer to another. It is connection-based, and data can be sent both ways once connection is established. The User Datagram Protocol (UDP) sends separate datagrams without guarantees for reliability or order. It is faster than TCP, since there is no error checking. [25] Apache Portable Runtime (APR) intends to create an interface which allows software developers to be assured of predictable behavior regardless of platform [3].

MINA has an implementation for a low-level TCP transport server and client, among other things. These provide sufficient communication capabilities, and MINA was selected for use on both the server and the client. It is covered in more detail in Sec-

tion 5.2.

4.8 JSON: JavaScript Object Notation

JavaScript Object Notation (JSON) is derived from JavaScript for representing objects in a text-based, human-readable and language-independent fashion [11]. Its primary use is exchange of structured data. JSON is commonly used for the same purposes as XML, but is promoted as simpler and more suited for mapping objects. JSON also does not need new tags or attributes to be defined for data representation. [79]

JSON describes objects, which contain named values, and arrays, which contain ordered values. A value can be a string, a number, a boolean, a null, a JSON object, or a JSON array. Nesting of objects and arrays is therefore possible. [11] JSON is discussed further later, in the context of XML-RPC and JSON-RPC.

The Android core libraries include support for JSON objects. They are implemented as key-value pairs with get and put methods for various data types, each with a `String` key for a parameter. [80] There is also support for JSON arrays, which have similar get and put methods to those of JSON objects. The difference is that indices are used instead of keys. Put methods append an element to the end of the array without a key. [81]

4.9 Remote procedure call technologies

Remote Procedure Call (RPC) protocols allow a program to call another program's procedure remotely, abstracting network details [19]. They can be used to invoke actions on the server, but the primary need for them is the implementation of battle actions. An action, when used, does not cause a simple value to be returned. Rather, it returns a set of commands that need to be executed on the client side.

The intuitive way is to have the server remotely call methods on the client. Besides actions, RPC technologies can be used for other messaging as well. It is a simple matter of remotely calling methods on the server.

4.9.1 Java RMI: Remote Method Invocation

Java Remote Method Invocation (RMI) implements a *synchronous request-response model*. It allows a client to invoke methods on remote applications, and receive return values. Exceptions can also be thrown from remote methods. As with local methods, the client waits until it receives a response. [10, pp. 16, 19]

The client needs a reference to the *remote object* before it can call its methods. In practice, the client has a reference to a *stub*, which implements the same interface as the remote object, called a *remote interface*. The stub delegates method calls to a *skeleton*, which resides at the same location as the remote object. The skeleton takes the place of the client for the remote object. [10, pp. 17-19]

The stub and the skeleton implement the required networking, and neither the client nor the remote object need to know about it. Method parameters, return values and exceptions are also handled transparently. Remote references can be passed by reference, but any other objects and primitives need to be passed by value. Their state is converted to a raw format, which is transferred to a remote location, and converted back to the original format. These conversions are called *marshalling* and *unmarshalling*, although they are in practice Java serialization and deserialization. [10, pp. 19-21] Serialization converts an object into a byte stream that allows a copy of the object to be created. Creating a copy from serialized data is called deserialization. Java has automatic serialization and deserialization. [7]

RMI is a pure Java environment [10, p. 22]. Because of that, it is an attractive solution for communications between a Java server and Android, which is also a Java environment. RMI is an excellent solution for implementing actions, since they all share the same interface, and the client has no need to know about the implementing object.

The client needs to retrieve the correct action object, and call methods in its remote interface. The action object also requires some remote interfaces from the client, in order to execute its effects and visualizations.

Unfortunately, Android does not support RMI, as its libraries do not include the RMI packages, `java.rmi` and `javax.rmi` [48]. Including standard Java APIs to an Android application may be possible through repackaging the Java API, though [82]. Still, no reliable record of successfully incorporating RMI into an Android application was found.

It seems probable that RMI would require native code which is not present in Android. In addition, the marshalled Java objects would need to be converted to dex bytecode format for the Dalvik Virtual Machine. Therefore, no actual attempt at integrating RMI into the application was made.

4.9.2 XML-RPC

XML-RPC uses the HTTP protocol as a transport mechanism, and XML for encoding the calls. It enables a client to call a single method in a remote system, pass multiple input parameters, and receive one return value. Larger structures can be transported by using data structures and arrays. [31]

XML-RPC is criticized as “worse than all of its competitors”. Compared to RMI, it is limited in transmittable data types and has large message size. Because it is tied to HTTP, it lacks statefulness. [83] An open source library named aXMLRPC implements XML-RPC for Android. It does not depend on any Android-specific libraries so the library works also for regular Java [84].

Possible parameter types for an XML-RPC method call include numbers, strings, booleans, dates, binary data, structures or arrays. Structures contain named members, each of which belongs to one of the above types. Arrays are similar, but members are

not named. [85]

A method returns either a single value, or a fault, which is described by a structure containing an integer and a string. The return value can have any of the same types as the method parameters. A string allows all characters except "<" and "&", which are encoded as "<" and "&", respectively. [85] XML-RPC is compared to SOAP and JSON-RPC in the following subsections.

4.9.3 SOAP: Simple Object Access Protocol

Simple Object Access Protocol (SOAP) uses XML for its message format, and HTTP, among other alternatives, for transmissions. A SOAP message can be sent to a service which returns an XML-formatted document. [22]

There is no built-in SOAP library in Android. Speculations in on-line discussions offer such explanations as Google's preference for RESTful web services, the high overhead of SOAP, and JSON's better efficiency. An open source library named ksoap2-android exists for using SOAP on Android [86].

Because of its apparently heavy overhead and lack of support, SOAP was not used in this project. There is no clear advantage over XML-RPC and JSON-RPC, which are promoted as more light-weight.

4.9.4 JSON-RPC

JSON-RPC is a protocol for remote procedure calls with JSON encoding. It can be used to send notifications which do not require a response, and a batch of multiple requests which can be answered in a different order. A JSON request is a call to a single method in the remote system using any suitable transport. The request can pass multiple parameters, and receive a return value. [12]

An open source JSON-RPC, version 2.0, library named android-json-rpc exists for Android. It currently supports only JSON-RPC calls over HTTP. [87] There are numerous JSON-RPC implementations for Java.

A JSON-RPC method invocation request is a JSON object that includes the method's name, a request identifier, and an array of parameters for the method. The method response is a JSON object that includes the request identifier, and either an object returned from the method, or an error object. [12]

In addition to regular method calls, there are notifications. A notification is a single JSON object, similar to a method invocation object, but without a request identifier. Notifications do not get a response. Parameters can be omitted from a request. JSON-RPC 2.0 request and response objects contain an extra member that signifies version. [12]

4.10 Decisions

JSON-RPC was selected as the means for invoking server methods from the client. As

for invoking client methods from the server, although possible, a different method was used. The server sends a script, which the client interprets and executes. This has a similar effect, but the script is better suited for use on different client platforms than the RPC method.

Along with the decision to use JSON-RPC, JSON was selected as the application's data transfer format. It is unambiguous, simple, light-weight, has built-in support in Android, and its platform independence makes it usable for different kinds of client. Unfortunately, there is no support for it in Java SE 1.7 API, which will be addressed in the next section.

4.11 JSON-RPC implementations

In order to use JSON-RPC to call server methods, a JSON-RPC implementation is necessary for the server. Numerous JSON-RPC implementations for Java were studied, and will be described in this section.

As implementations are introduced, problems in using them for this project will become apparent. Most of these problems will also appear in other implementations, causing them to be rejected on the same basis as one or more prior implementations.

4.11.1 qooxdoo 2.0.1

The qooxdoo RPC is based on JSON-RPC as the serialization and method call protocol, and provides a Java server back end, among others. There is no mention of any front end besides JavaScript, remotely callable Java methods need to implement a special interface, and the Java back end is implemented as a servlet. [88] Front end means the part of an application which interacts with the user. [4]

The fact that the front end is JavaScript is not a major issue, since Android's built-in JSON support can be used for creating JSON-RPC messages, and MINA was intended for transportation use anyway. It would, however, be necessary to separate JSON-RPC handling from the servlet in order to use it with MINA, and the need for implementing an interface reduces code reusability and makes it more difficult to replace qooxdoo.

4.11.2 jabsorb 1.3.2

A major part of jabsorb is a Singleton `JSONRPCBridge` object, which unmarshalls JSON objects in JSON-RPC request format, invokes a Java method, and marshalls the returned Java objects into JSON objects. A Java object's public instance methods can be registered for JSON-RPC calls, as well as public static methods of a Java class. [89, `JSONRPCBridge`]

The jabsorb download package includes a servlet JSON-RPC implementation, as well as some JavaScript files. With some experimentation, these were found to be unnecessary for the operation of `JSONRPCBridge`, which was successfully used together

with a MINA server.

It was also ascertained that custom marshalling was possible with jabsorb's serialization framework, making it possible to transform Mupe Force objects into language-independent descriptors.

4.11.3 jsonrpc4j 0.24

The jsonrpc4j project uses another library, named Jackson, for Java/JSON transformations, and includes a client and several server implementations for JSON-RPC [90]. Its streaming server accepts a JSON-RPC message sent by the client, handles it, and returns the results [91, `StreamServer`]. No way for sending messages from the server to the client was found.

JSON-RPC handling is separated from the transport server, and implemented in the `JsonRpcServer` class [92, `JsonRpcServer`]. It should be possible to use it separately, as with the jabsorb `JSONRPCBridge` before, but this would be pointless. The interface for registering callable methods is more difficult to use than the one in jabsorb, and the Jackson library is very large in comparison. The Jackson library was not researched thoroughly enough to determine how to implement custom Java-to-JSON transformations.

4.11.4 json-rpc 1.0

This implementation requires a client interface and its server side implementation [93]. The server side implementation is then bound to an executor, which determines a Java method matching a JSON-RPC request, and executes it using a transport which reads the request and sends the response. [94, `JsonRpcExecutor`, `JsonRpcServerTransport`]

On the client side, an invoker creates a proxy instance of the client interface. It uses a transport which calls the server and returns its response. It uses a library named google-gson to convert JSON objects to Java objects and the other way around. [95, `JsonRpcInvoker`; 94, `JsonRpcClientTransport`]

MINA can be used together with json-rpc by implementing the transport interfaces `JsonRpcServerTransport` and `JsonRpcClientTransport`. However, the invoker uses a default `Gson` object for transformations between Java and JSON [95, `JsonRpcInvoker`]. Therefore it prevents the use of custom JSON serializers, and because of that, this implementation was not used in this project.

4.11.5 Others

By now, several limitations in the JSON-RPC implementations have been introduced. In order to be applicable for this project, the library must be usable without servlets, it must either be usable together with MINA or have a suitable server of its own, it must be able to handle existing Java objects without changes to the code, and it must support custom marshalling.

The other libraries that were studied, but failed to meet these requirements, are shown in Table 4.1. In the end, the only implementation that meets the above requirements is jabsorb, which was selected for use in this project. It is covered in more detail in Section 5.3.

jpoxy 1.0.17	Implements JSON-RPC in a servlet class [96].
JSON Service 1.0.0	Requires annotations for classes and methods that can be called remotely [97]. Adapting for this project would also be difficult because of bindings to servlets and the Spring framework, and use of the Jackson library.
JSON-RPC 2.0	Requires callable methods to be introduced in a class that implements the library's `RequestHandler` interface [98]. Uses yet another library for JSON encoding and decoding: JSON Smart. It is limited to only a few Java types: `Boolean`, `Number`, `String`, `List` and `Map`, as well as `null`. [99]
Java-json-rpc	A servlet implementation using Jackson [100].
libjsonrpc 1.1.2	Server side must implement an interface, and a client side proxy that implements the same interface is created. Needs annotations to determine classes and properties available for RPC invocation. [101]
simplejsonrpc 0.1.1	A servlet implementation using Jackson [102].

Table 4.1: Other JSON-RPC libraries that were researched.

5 SERVER JAVA

This chapter introduces the technologies used on the server. Server side software is mostly written in common Java Standard Edition (SE), but that will not be introduced here. Rather, Java DataBase Connectivity (JDBC) is introduced, and the Apache MINA communications library and jabsorb JSON-RPC library, which were introduced in the previous chapter, are explained in more detail.

JDBC is a part of the standard Java API, and its package is `java.sql`. It needs a separate database driver depending on the database in use. The Mupe Force server uses a MySQL database, so the appropriate JDBC driver is MySQL Connector/J [103].

MINA can be downloaded from Apache's web site. The download includes multiple JAR packages, but only the core package is necessary for this project. MINA needs some other libraries which come with the download package: Apache's Commons Logging and Simple Logging Facade for Java (SLF4J) [104, 105].

A download package for jabsorb is available at Google Project Hosting [106]. It needs Apache's HTTP Components `HttpClient` and JCP Servlet Specification 2.5 API, because of its servlet and HTTP ties [107, 108].

5.1 Database programming with JDBC

An SQL script file is used for creating the database schema. After that, the Mupe Force server also requires a user access to its database, `mupe_force_server`. User name and password, as well as database URL were hard-coded into the `main` method. The procedure for communicating with a database using JDBC is as follows [109, p. 1293]:

1 Import the necessary classes.
2 Load the JDBC driver.
3 Identify the data source.
4 Allocate a `Connection` object.
5 Allocate a `Statement` object.
6 Execute a query using the `Statement` object.
7 Retrieve data from the returned `ResultSet` object.
8 Close the `ResultSet` object.
9 Close the `Statement` object.
10 Close the `Connection` object.

The JDBC driver can be identified in a system property, "`jdbc.drivers`", which

makes it possible for the JDBC Driver Manager to load it automatically. Setting a system property might be prevented by security policy, but the driver can also be loaded explicitly using a static method in `Class`, and the driver class's fully qualified name: [109, p. 1295-1296]

```
Class.forName( "com.mysql.jdbc.Driver")
```

Once the driver is loaded, the Driver Manager can provide connections for data source identifiers [109, p. 1296]. The data source identifier is a `String` URL, which identifies both the driver and the data source. Its format depends on the driver [109, s. 1297]. The MySQL Connector/J has the following format (somewhat shortened) [110, 19.3.5.1]:

```
jdbc:mysql://[host][:port]/[database]
```

Square brackets indicate optional parts. The Mupe Force server database is named `mupe_force_server`, and runs on localhost, which is the default host, at the default port, so its URL is:

```
jdbc:mysql:///mupe_force_server
```

An open `Connection` can create `Statements`, which can be used for compiling and executing SQL queries and query batches, which will provide the results in a `ResultSet` [109, p. 1303]. Queries can be executed with `executeQuery()`, added to a batch with `addBatch()`, the batch can be executed with `executeBatch()`, and cleared with `clearBatch()`. A batch can produce multiple result sets, which can be retrieved using `getResultSet()`, and iterated using `getMoreResults()`. [109, p. 1304] Queries which do not return results, such as data manipulation, can be executed with `executeQuery()`, but also with `executeUpdate()`. It does not produce a result set, but a number of affected rows. [109, p. 1343]

A connection can also provide a more complex `PreparedStatement`. It pre-compiles an SQL statement with placeholders where parameter values can be inserted later. These are especially good when a statement is executed multiple times. [109, p. 1304]

Placeholders are marked with question marks, and indexed from left to right, starting at one. There are numerous setters for the values of a placeholder. Each takes a parameter of a certain data type, and an index to the placeholder. [109, p. 1346]

A result set contains rows, which can be navigated using `next()`, `previous()`, `first()`, `last()`, `beforeFirst()`, `afterLast()`, `isFirst()` and `isLast()`. As in a database, each row contains columns with varying types of data. A column can be identified by its name, or its index in the result set. There are various getters for each, with different data types, such as `int` or `InputStream`. [109, p. 1305]

Meta data about a result set is available through the `getMetaData()` method, which returns an object of type `ResultSetMetaData`. It can provide, among other things, the number of columns in the result set (`getColumnCount()`), and the name and type of a column (`getColumnName()` and `getColumnType()`, respectively), using a column in-

dex. [109, p. 1308]

MySQL is by default in auto-commit mode [110, 13.3.1]. The `Connection` interface has methods for enabling or disabling auto-commit, as well as executing manual commit and rollback [111].

5.2 Communications with Apache MINA

A MINA application has three layers: I/O Service, I/O Filter Chain, and I/O Handler. The I/O Service performs actual input and output. I/O Filter Chain filters or transforms data from bytes to data structures and back. The I/O Handler has the application's business logic. Several I/O Services and Filters for the I/O Filter Chain are available. The I/O Handler is application-specific, and needs to be implemented. [112]

A server has an I/O Acceptor, which listens on a port for incoming connections or packets. A new session, specific to an IP-address and a port, is created for a new connection. Subsequent requests are handled in that session. All received data traverses the I/O Filter Chain, allowing Filters to modify the content. The final destination for the data is the I/O Handler. [113]

Sending data from a client works in the opposite order. The client has an I/O Connector, which connects to the server and creates a session. Data comes from the client's I/O Handler, through its I/O Filter Chain, and is sent to the server. The response then traverses from the I/O Connector through the I/O Filter Chain to the I/O Handler, like on the server. [114]

In order to transport data in text format, the server and client both need a `ProtocolCodecFilter` using a `TextLineCodecFactory` [115]. This is not a very efficient or practical format for transporting data, but creates human-readable messages that are useful for testing.

In order to connect to the server from the client, the method `IoConnector.connect` is used. This returns a `ConnectFuture` object, which indicates connecting is an asynchronous task and may not be complete when the connect method returns. The connect future has a method named `awaitUninterruptibly`, which returns once the connection is complete. After that, the connect future provides an `IoSession` object. Messages being sent to the server are written to the session object. [116]

The `IoHandler` interface has several callbacks for tracking events: `sessionCreated`, `sessionOpened`, `sessionClosed`, `sessionIdle`, `exceptionCaught`, `messageReceived` and `messageSent`. The `sessionCreated` event is executed in a thread that handles multiple sessions, while the `sessionOpened` is called in a session-specific thread, depending on thread model. A session is “opened” after it has been “created”. [117] An adapter class, named `IoHandlerAdapter`, can be extended to selectively override methods in the `IoHandler` interface [115].

MINA has four types of Acceptor: `NioSocketAcceptor`, `NioDatagramAcceptor`,

`AprSocketAcceptor` and `VmPipeAcceptor`. The first is a non-blocking socket Acceptor, the second a non-blocking UDP datagram Acceptor, the third a blocking socket Acceptor, and the last one is for communications inside the Java Virtual Machine. There are corresponding Connectors, as well as a few others. [118]

The unreliable nature of UDP makes `NioDatagramAcceptor` unsuitable for this project. The `VmPipeAcceptor` is, of course, useless for communications over a network. Of the two socket Acceptors, the `NioSocketAcceptor` is better since it is not blocking. The client does not require an immediate response to a request sent to the server, but can send other requests and handle responses and incoming messages while waiting.

To match the `NioSocketAcceptor` on the server side, `NioSocketConnector` is used on the client side. Nothing was found in the documentation about the sending of messages from the server to the client. However, it is important to note that the socket request is non-blocking, meaning that the response from server is sent asynchronously. The method of sending the response is writing it to the server side session object [115]. An experiment showed that sending a message in this fashion caused a `messageReceived` event in the client's `IoHandler`, just like sending a message from the client to the server the same way.

As for concurrency, the default mode of `NioSocketAcceptor` is “multiple thread model”. In case more control is required, the Acceptor can be constructed with any implementation of the `java.util.concurrent.Executor` interface. [119] Same is true for the `NioSocketConnector` [120].

5.3 JSON-RPC with jabsorb

The jabsorb class which unmarshalls JSON-RPC requests, invokes methods and marshalls results into JSON-RPC responses is `JSONRPCBridge`. It has a global Singleton, and can also be instantiated. The global bridge allows objects and classes to be exported for use by all clients. Object methods are exported with `registerObject(Object key, Object o)` and classes with `registerClass(String name, Class clazz)`. [89]

The former is overridden, and can get an extra `Class` parameter, which determines exported interface. All instance methods defined in the interface `Class` parameter are exported, while without it, all public instance methods in the object are exported. For a class, public static methods are exported. The method's name for JSON-RPC is the key or name, plus a dot, and the method's name. [89]

Once methods are exported, they can be invoked using the `JSONRPCBridge.call` method. Its parameters include a context, documented as `HttpServletRequest` and `HttpServletResponse` in the case of HTTP transport. According to experimentation, the call method works fine with a `null` for context. Besides the context, there is another parameter, which is a JSON object containing a JSON-RPC method invocation. [89, `JSONRPCBridge`]

Exceptions can be transformed to JSON using implementations of the `ExceptionTransformer` interface, and setting it to the bridge using `setExceptionTransformer`. An `ExceptionTransformer` can transform a `Throwable` into any `Object`. [89, `ExceptionTransformer`, `JSONRPCBridge`] Besides exceptions, the public entry point for jabsorb's framework for marshalling Java objects is `JSONSerializer`. It is a global object with a setter and getter in `JSONRPCBridge`. [89, `JSONSerializer`, `JSONRPCBridge`]

Default serialization can be extended by adding `Serializer` implementations to the `JSONSerializer`. More specific serializers should be added after less specific serializers. The serializers available in jabsorb handle primitives, JSON types, common objects such as `String`s, various collections, arrays, and some others. [89, `Serializer`, `JSONSerializer`, `org.jabsorb.serializer.impl`]

A `Serializer` implementation needs to return the Java classes it can marshall from or unmarshall into, and classes for JSON objects it can marshall into or unmarshall from. It also has a more specific test for serialization from a Java class into a JSON type, a setter for the owning `JSONSerializer`, and of course methods for marshalling and unmarshalling objects. In order to check if a JSON object can be unmarshalled, there is a method named `tryUnmarshall`. A class named `AbstractSerializer` implements setting the owner, and a basic test of what can be serialized, based on returned classes. [89, `Serializer`, `AbstractSerializer`]

6 SERVER DESIGN

The server is built around the database, with the network connection layer as its front. Between them is a model of game contents, which provides data for the connection layer. It also handles changes that originate from the connection layer, and returns the results of the changes. There is also the battle controller and related classes.

The database can be divided into two parts: meta data and player-specific data. Meta data is loaded when the server starts up, and stays in memory until the server shuts down. Each meta data object is essentially a Singleton Flyweight, which contains the unchangeable state it was created with. Player-specific data is instantiated when a player logs in, and it remains in memory until the player logs out. It can also be altered.

Singleton is a design pattern that ensures there is only one instance of a class, and there is a global access point to it [8, p. 128]. Flyweight is a design pattern for sharing objects. A Flyweight object contains context-independent state, and receives context-related properties from clients. [8, pp. 195-196]

The communication layer is abstracted, so that the model and battle controller can work with any kind of communication implementation. The abstraction also allows the communication implementation to use a changeable component for transforming responses and outgoing messages.

6.1 Meta data

The meta data module has four classes, which correspond to database tables. A diagram of meta data tables is shown in Figure 6.1. All of the tables are not represented, though. These classes are: `AbilityTypeDB`, `CharacterClassDB`, `ItemCategoryDB` and `ItemTypeDB`. In addition, there is `StatisticsDB`, which only contains static methods for saving statistics to the database and reading them back. It can not be instantiated. Statistics belong to neither meta data nor player-specific data, or to both, depending on interpretation.

The basic operation of a meta data class consists of loading all data for that class, getting a list of `String` identifiers for the content, getting an object corresponding to an identifier, and reloading the content from database. These are all static methods, which create an encyclopedia-like interface for the meta data. Each class also has non-static getters and setters, as well as save and delete methods. Only the getters are needed in-game, since the meta data is not changed. The other methods are for internal use and content editing.

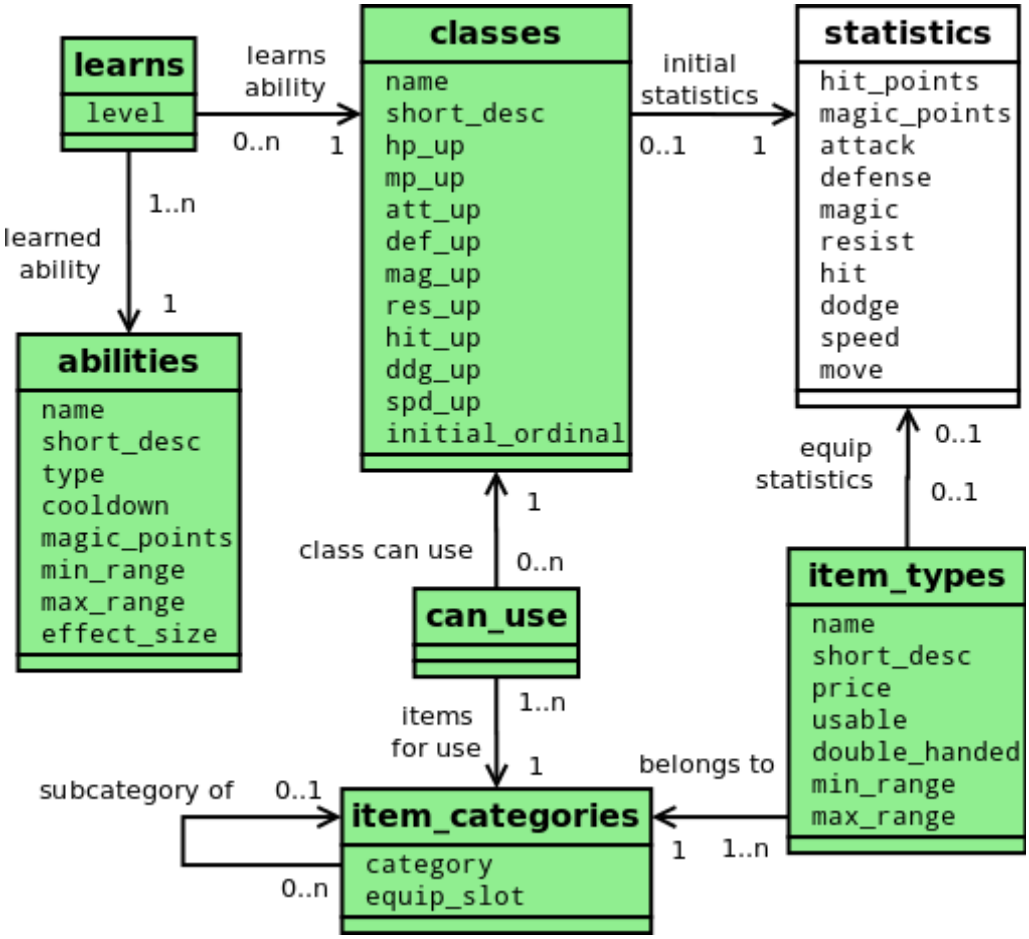

Figure 6.1: Meta data tables in database (colored).

A group of proxy classes separates the database meta data classes from the rest of the application. These are named `AbilityType`, `CharacterClass`, `ItemCategory` and `ItemType`, Each of them contains a link to a database object, and implements the Observer design pattern in order to receive a notification when the database objects are reloaded. In the Observer pattern, multiple observers can be notified of the state of a subject changing [8, pp. 293-294].

The proxies enable reloading the meta data at run time, since each database object is referenced in a single place, at most. The proxy classes have a similar static interface to the database classes. They also have getters, including some that are more specialized than those in the database classes, which only provide database data as it is stored at runtime.

All database identifiers are kept in the runtime objects for saving and deleting. When multiple operations are done during saving or deleting, they are done in a single transaction and rolled back if any of the operations fails. Rather than failing completely, individual objects can be left out during database reading, in order to cope with database errors.

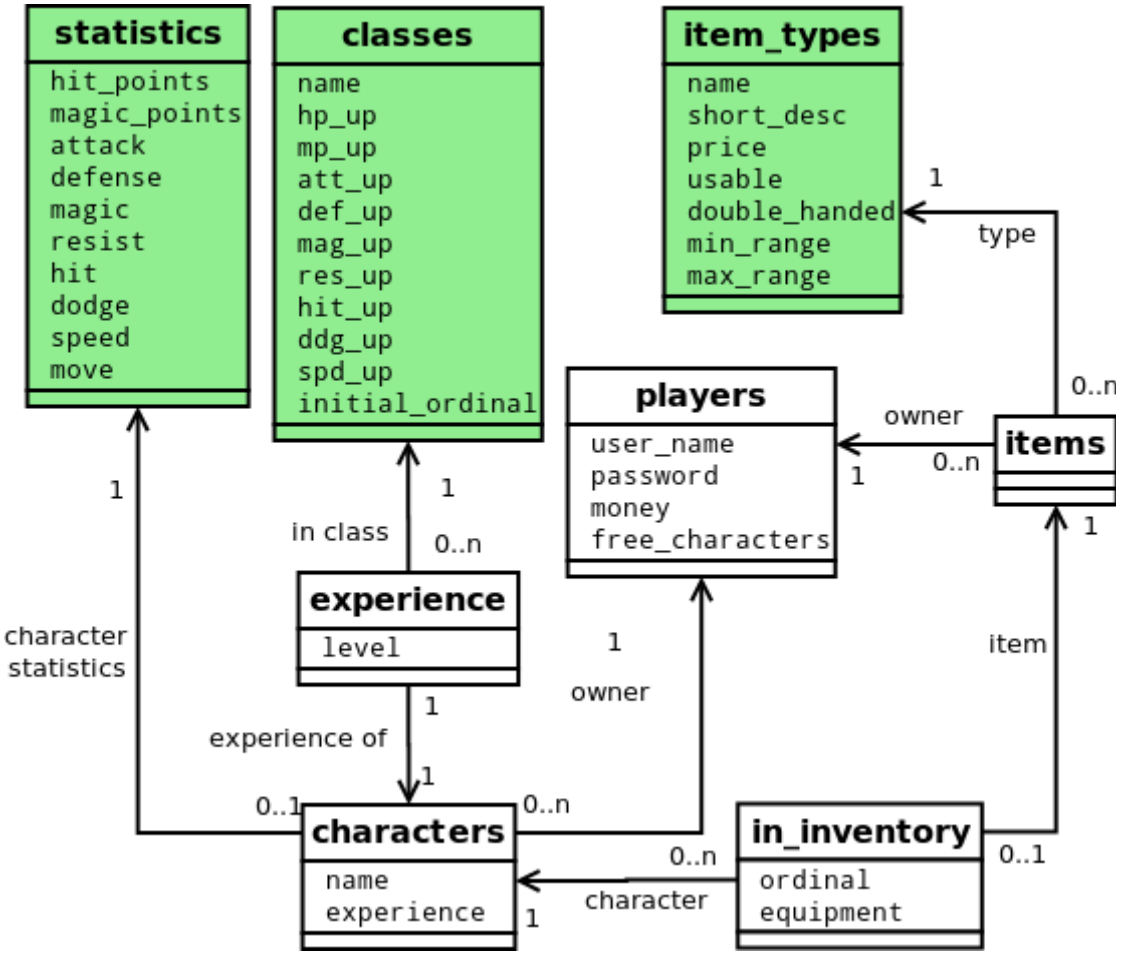

Figure 6.2: Player-specific data tables in database (white).

6.2 Player-specific model

A diagram of database tables for player-specific data is shown in Figure 6.2. Along with a player, all of that player's characters and items are created. If an item is equipped or carried by a character, it is placed to the character's inventory in the appropriate slot. Characters also have abilities, which depend on their level and class. The character class meta data object will provide a list of abilities the character has learned. New ability objects are created and added to the character.

When player-specific data objects are created, they are set with references to the appropriate meta data classes. A character belongs to a class, and an item or an ability has a type. Player-specific data objects can be added, modified and deleted at runtime. The changes are also saved to the database, if the player is registered. The data of guests is not saved, but only the runtime objects are updated. When the guest logs out, those objects are forgotten. Classes are `Player`, `Character`, `Item` and `Ability`.

A player is identified by a unique user name, and an ability's type name is required to be unique for a character. Characters and items require a means for a client to unam-

biguously refer to them. For a character or item which belongs to a registered player, its database identifier can be used. Auto-increment integer primary keys in MySQL with default settings are positive integers starting from one [110, 3.6.9].

Another means of generating identifiers is needed for characters and items belonging to guests. Each new guest gets a number from one to the maximum 15-bit value, or in other words the maximum value of `short`. Each character or item for a guest also gets a number from one to the same maximum value.

These are then combined with the guest number in the 16 most significant bits and the character or item number in the least significant 16 bits of a 32-bit `int`. The result's opposite is used for the identifier. Therefore, negative numbers are identifiers for the characters and items of guests, positive values for those of registered players, and zero is neither. Zero is the identifier for “no character” or “no item”.

6.3 Database helper

The database helper is a utility class for handling database operations. It wraps an open JDBC connection. It also has a list of `ResultSet` objects, and a list of `Statement` objects. When called to do so, it will close all `ResultSet`s and `Statement`s. `ResultSet`s and `Statement`s can be added. The helper also has methods for creating `PreparedStatement`s, which add the new statement to the list, and methods for executing queries, which add the query `ResultSet`s. The query execution method uses a `Statement`, which the helper creates internally when necessary. The same `Statement` can also be used to execute updates, which do not return a `ResultSet`.

The helper does not handle transactions, but provides access to the connection it wraps. Therefore the connection's auto-commit can be disabled, and changes committed manually. The helper has a rollback method for easier exception handling.

The Observer system for database meta data is implemented in the `DBHelper` class. It has a static `Observable`, and static methods to attach observers to it, remove observers, and notify observers.

6.4 The facade

The runtime model is accessed through a facade, which has three classes: `CharacterActions`, `ItemActions` and `PlayerActions`. Each has a set of static methods for manipulating the model and getting data from it and the meta data classes. The facade classes hold the JDBC connection necessary for persisting changes. Each static method encapsulates one game action and affects only the objects it gets as parameters, as well as the database.

A second facade layer acts as a proxy between a non-Java client and the facade. It takes parameters as `String`s and primitives, matches them to Java objects, and delivers

those objects to the real facade. The proxy facade methods are also static, and affect only those objects identified by the parameters. Any method in either facade layer can throw an `SQLException`, if the action changes database content.

`PlayerActions` is the entry point onto the server because it has actions for logging in and creating new players. Registered players have a create action, which performs an initial database save and then logs the new player in. Guests are only logged in, and remain in existence only until logged out. All logged in players are stored in a map keyed by name, and removed from it when logged out.

When a guest is logged in, a new name is generated. A new player's name is expected as a parameter. Either way, the names of players that are already logged in are checked, as well as those saved in the database. The new player's name must be different from all registered players whose data exists in the database, and all active guests. In the case of guests, a new name is generated until an unclaimed one is found. For a new registered player, the user must provide a different name.

A `Player` object has either direct or indirect references to all the model objects specific to that player. The `Player` object can be retrieved from `PlayerActions`. This is the first thing to do for most data requests and updates coming from a client.

6.5 Communication abstraction

The primary abstraction for separating the communications from other parts of the server is an interface named `ServerMessageHandler`. It contains a method for handling incoming messages from the client, and several methods for sending messages to the client. There is also an abstract class named `PlayerHandler`, which implements the server messaging interface, and is in addition tied to a player.

The incoming message handler has an `Object` parameter, allowing for any format of message from the client, and an `Object` return type, allowing any type of response. Parameters for the messages to be sent to the client are Mupe Force model objects, and return type for each is `Object`. The interface can be implemented for transforming messages, returning them in any type, or for actually transmitting the messages to a client, in which case there is no need for returning anything.

The player handler holds the name of the player it is associated with, and has a static interface for registering player handlers. This registry provides the player handler when there is need for sending a message to a player. The player handler also provides an `ActionExecutor`, which is used for executing actions in battle.

One player handler is implemented for this project: `MinaPlayerHandler`. It is created with an instance of `ServerMessageHandler`, and one of `ActionExecutor`, which are used for the actual handling of messages and actions. The Mina player handler simply gets and returns a response to an incoming request, or writes an outgoing message into a MINA `IoSession`.

6.6 JSON-RPC component

An instance of `JSONRPCBridge` is used for JSON-RPC invocations, with custom exception transformer and custom serializers in addition to default ones. The exception transformer extracts a `Throwable` object's message, which is sent to the client. The client is notified when errors occur on the server, but there is no need to send stack trace or other information contained in the `Throwable`.

The custom main serializer marshalls Mupe Force model objects into JSON objects. The point here is not to serialize them in a manner that allows the objects to be recreated on the client, but to send only the necessary data. This way, the JSON-RPC server is usable with a wider variety of clients, the amount of transferred data is reduced, and the client can use different classes for its model.

The built-in `ListSerializer` in jabsorb serializes a list into a JSON object with key `"javaClass"` for the object's class name (although this can be turned off with serializer settings) and key `"list"` for a JSON array with the list's contents. [89, `JSONSerializer`; 121, `ListSerializer`] For this project, the list's content in a JSON array is sufficient and preferable for its simplicity. The same things apply to jabsorb's `MapSerializer` and `SetSerializer` [121, `MapSerializer`, `SetSerializer`]. Custom serializers are implemented for lists, maps and sets.

The custom serializers do not unmarshall anything. This is because the data expected from clients is in the form of `String`s and primitives, which have different meanings depending on context. Instead of attempting to unmarshall them with jabsorb's framework, the proxy facade introduced in Section 6.4 handles them depending on action. This is much simpler, and also more reusable, since the facade can be used with other protocols besides JSON-RPC.

With the custom serializers and exception transformer in place, the only configuration left is registering the proxy facade for JSON-RPC invocations. The `JSONRPCBridge` can then find the method matching an incoming message, which it receives from the JSON message handler, which implements the `ServerMessageHandler` interface. The JSON message handler then returns the call's result which has been automatically marshalled into JSON format.

The server-to-client messaging methods each create a JSON object, and put the message name into it with the key `"message"`. Parameters for the message are also placed into the JSON object and marshalled using the jabsorb serialization framework. The JSON handler returns the messages in JSON format, and knows nothing about sending them to the client.

6.7 The MINA server

The MINA server has the `main` method for the server application. It creates the MySQL

Connector/J JDBC driver, loads meta data from the database, initializes the facade with a JDBC connection, and binds a MINA `NioSocketAcceptor` to a port. The acceptor will then service incoming requests.

The `MinaServer` class also extends the MINA `IoHandlerAdapter`, and handles incoming requests. When a session is opened, the server creates a new `MinaPlayerHandler` with JSON message handler and action executor. The server does not read the contents of a message, but expects it to be in JSON format and checks the message for a `"userName"` parameter. That user name is used for registering the player handler. A new player handler can also replace an existing one, if the session is interrupted and recreated. The player handler is unregistered when the session is closed.

The server main class is the only one that uses both MINA classes and JSON. This is because of the need for connecting player handlers to user names. The message handler could get the user name from the JSON object, but does not get a reference to the `IoSession`. Rather than tie any message handlers with MINA, the `IoHandler` is tied to JSON.

6.8 Battle controller

The battle controller has a static method for starting battles. A new battle controller object is created for a player looking for battle. When another player starts looking for a battle, the new player is added to the waiting battle controller, and a battle is started. The next player again gets a new battle controller instance and waits for an opponent. In a multi-player environment, multiple battle controller instances can exist at a time, so there is a static method for getting the appropriate instance for a player.

Starting a battle involves randomly generating a battlefield, and creating a battle player handler for each participant. The handler's main purpose is transforming coordinates between the battle controller's system and a player's system. The first player uses the same coordinates as the battle controller, and the second has the coordinates flipped upside down. A battle player handler receives a coordinate transformer object from the battle controller. It is used to reverse transform coordinates from the client to battle controller coordinates, and transform coordinates from the battle controller to player coordinates.

6.8.1 Generating a battlefield

The first step in generating a battlefield is setting a random terrain identifier for each square, after which obstacles are placed. Each square has a 20% chance of getting an obstacle. Placing obstacles randomly can cause the battlefield to be split into multiple areas that are not connected with each other. This is not wanted, so any separate areas are connected.

First, the number of separate areas is counted. If there is more than one, squares

blocked by obstacles are checked to find cut points, where removing the obstacle connects two separate areas. Blocked squares are checked in random order. If clearing all cut points does not connect the battlefield's open squares into a single area, obstacles are removed randomly.

The random clearing algorithm ignores squares which have all their neighbors blocked by obstacles. Other than those, squares with the most obstacles in surrounding squares are favored. One of those squares is selected at random, then cleared, and its blocked neighbors checked to find more cut points. Any cut points are cleared, and the random clear process is repeated until open squares form a single area.

Another problem that can arise from randomly placing obstacles is that a player's deployment area might not have enough squares for their characters. To clear sufficient area, random blocked squares are cleared on each deployment area until there are at least six free squares. The maximum size of a battle force is used here, not the number of characters the player actually selected for battle.

Deployment areas are cleared first, before clearing cut points, for two reasons. First, randomly clearing squares in the deployment areas can cause disconnected open areas. Second, in an extremely unlikely case (probability of one in approximately $5.4*10^{44}$) every single square on the battlefield can be blocked by an obstacle. In that case, the cut point algorithm would fail.

6.8.2 Battle progress

The battle controller waits for messages from clients, reacts to them, and sends results to both participating players. At first, it waits for deployment positions for each player's characters. These must come from the battle handlers, because of coordinate transformations. In fact, anything that has to do with coordinates must come through the battle handlers and not directly from a client. A battle facade gets calls from clients, finds the appropriate battle controller and battle player handler, and delegates calls to the latter. The facade, like the ones introduced before, has static methods for receiving `String`s and primitives from a client and finding the corresponding server-side model objects.

The true start signal for the battle controller is an “end deployment” message. Once both players have finished deployment, the battle controller notifies each that the battle is about to start for real, and immediately progresses to the first turn.

At the beginning of each turn, the battle controller first checks if its turn order list is empty. If yes, a new round begins. The battle controller selects a random speed modifier for each of the characters that were deployed on the battlefield earlier. The characters are then placed into the turn order list, and sorted in descending order by the product of speed and speed modifier.

The first character in the turn order gets the next turn, and is removed from the turn order. Both players are notified that a new turn begins, and also get the character whose turn it is. The handler for the character's owner creates a movement radius through the

owner's battle handler, and sends it to the client.

Movement radius generation starts in a character's current position. All unblocked neighbors are added to the movement radius. The same is then repeated to their neighbors, except that squares which have already been added to the movement radius are ignored. Squares whose distance to the starting square is greater than the character's movement statistic are also ignored.

On the client side, the player decides where the character will move, what it will do, and where the action is targeted. These are then sent back to the server and, through a battle handler, to the battle controller. The battle controller moves the character, and calls the action to determine its results. Those results are then sent to each player, using action executors. This sequence is depicted in Figure 6.3.

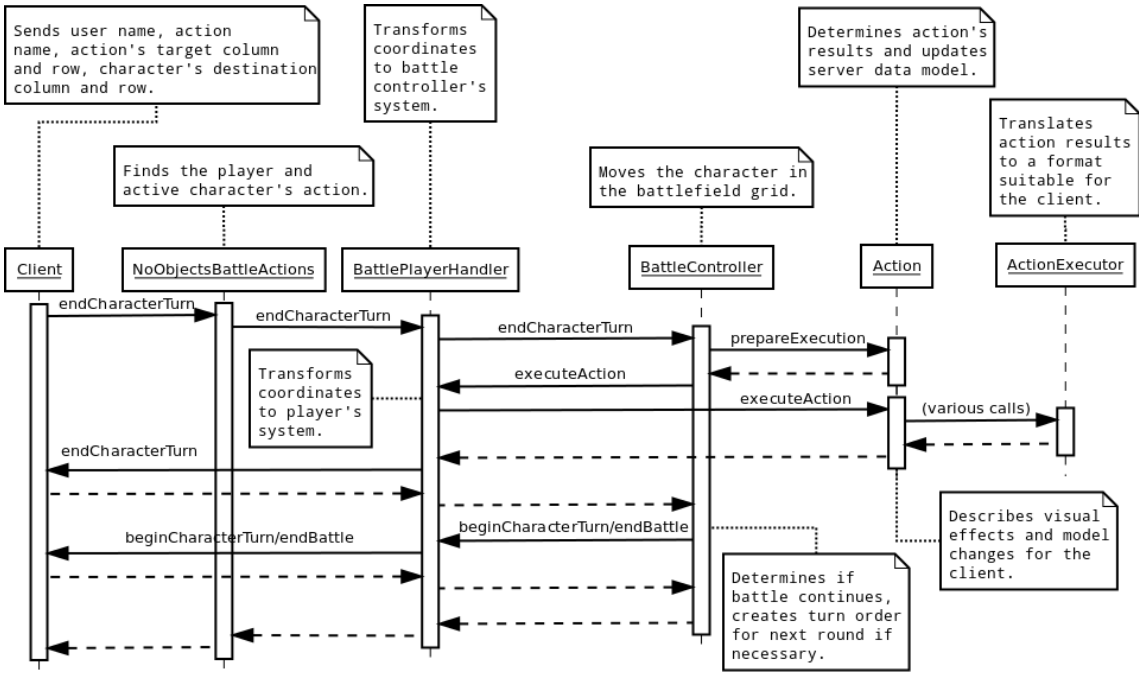

Figure 6.3: Sequence diagram for executing a character's action (simple getters are omitted).

After the action, the character's turn ends. The battle controller will now begin the next turn, as explained before. There is also the chance that the action ended the battle, and the players are notified of that instead of a new turn. The battle controller object is destroyed when the battle is over.

A battle controller's last functions are determining reward money for each player and leveling up their characters. The `Character` class has a level up method, which checks if the character has enough experience. If yes, it gains one or more levels. Statistics increases are then randomly selected for each level, and the appropriate `CharacterClass` object is checked to determine if the character has learned any new abilities. Changes are then saved to the database, which is why the level update is actually done by the `CharacterActions` facade. The battle's result, reward money and level up data are sent to the players in the end battle message.

6.9 Animation scripts

During battle, the server sends the effects of a character's action to each player, in the form of a script. The script defines an animation the client is to show, and updates to local model. The animation script is similar to the XML-definitions of View Animations in Android, although a simpler subset is sufficient.

Defining the animations in Android format reduces the server's usefulness for different clients, and is also contrary to the JSON format of other messages. Furthermore, the XML animation files are loaded through Android's resource system, requiring them to exist in files in the application's installation package. Methods for creating animations from dynamically created XML were not researched, as there was no intention of creating the scripts in Android format, or in XML. Android's animation XML files are a useful template for defining a JSON-based script language, though.

An animation file must have a root element, which is either `<alpha>`, `<scale>`, `<translate>` or `<rotate>`, for transformations of the same name, an interpolator, or a `<set>`, which can contain a group of elements, including other `<set>` elements. An interpolator determines how a transformation is applied over time. [66]

The Mupe Force animation script has a single root element, named `"animations"`, which is a key for a JSON object of animations. An animation has an identifier, which is its key in the animations object. Its type, one of the above transformations, is the value for a key named `"animation"`. Other parameters depend on the type of animation, including `"set"`, which has a JSON array of animation identifiers. The animation XML elements' attributes, defined in [122], become keys in a JSON object, without name space.

Animation XML only defines transformations, though. The JSON script also needs to define what is being transformed. For this, Android's `Canvas` class is used for a template. It has a large number of drawing methods, as well as ones for clipping and transformations [123]. A subset of these actions is used for the animation script.

Coordinates in the script are defined using square grid columns and rows, and relative positions within a square, rather than absolute coordinates. The size of squares depends on client's resolution, but the number of squares and their content does not.

Similar to animations, animatable graphics are in an object whose key is `"graphics"`. Each graphic has an identifier, which is its key in the graphics object. A graphic is defined as an array of drawing operations which are to be executed in order. These include `setColor`, `drawText`, and so on. Each drawing operation can have a position, which is defined relative to square dimensions.

In addition to animations and graphics, there are interpolators. They determine the speed of an animation's change, such as constant speed, acceleration or deceleration. Interpolators are in a JSON object, whose key in the script object is `"interpolators"`. Each interpolator has an identifier, which is its key in the interpolators object. Each one

also has a type, whose key is `"interpolator"`, and parameters depending on type.

Any animation can have an interpolator, whose key is `"interpolator"` in the animation's JSON object. Other properties common to all animations are `"duration"` and `"offset"`. The latter determines the time from the animation's start time to the time when it actually begins and becomes visible.

Often a character will move to a different square before executing the action which ends its turn. Therefore the script has a `"moveCharacterTo"` JSON object, which has the character's identifier and destination square column and row. This movement is meant to be executed before the action.

The action itself is a series of operations, a JSON array whose key is `"actionQueue"`. The action queue can contain such operations as altering a character's current hit points or magic points, or removing a character from the battle. Each operation has a type, whose key is `"action"`, and parameters depending on the operation.

Visual effects are controlled by `"startAnimation"` and `"wait"`. Starting an animation needs the identifier of one of the animations that was defined elsewhere in the script. It can be a set, for executing multiple animations simultaneously. It also needs an identifier for a graphic, which is animated, and the drawing area in terms of square grid columns and rows. A graphic defines its position within that area, and an animation can move it from its default position. Any animation, graphic or interpolator can be used multiple times when the script is executed. Each operation in the action queue is executed only once.

A `"wait"` operation of course makes the execution wait for a specified amount of time. Without it, the action queue's execution will continue without waiting for an animation to complete. Wait times can therefore be used to control simultaneous animations, or prevent beginning next character's turn while an animation is still active.

7 CLIENT BACK END DESIGN

Client design is mostly based on “Network MVC” by Mednieks et al [39, pp. 329-334]. Model-View-Controller (MVC) is a design pattern for relating user interface to data. The model is a representation of the application's data, the view is a representation of the user interface, and the controller is a representation of the connections and communication between them. [14]

The key element in Network MVC is a Content Provider which returns cached data at first, then retrieves more data from the network asynchronously, updates the cache, and notifies clients when new data is available. The view and controller parts do not make network requests directly, but only through the Content Provider. [39, p. 331] This chapter is about the part of the Mupe Force project which has the Content Provider's role.

Perceived performance is increased, since the UI thread never waits for a network request to complete. Data is cached in a local database as soon as it is received from the network, minimizing the risk of data loss. The platform's built-in capabilities are used for update notifications, and there is no need to write custom systems for polling and updates. Since the Content Provider handles network responses on the background, the risk of results being dropped because there is no active Activity to receive them is removed. Also, thread safety and encapsulation are improved, applications are easier to write, and server load is reduced. [39, p. 333]

Since Mupe Force does not provide its data to other applications, a Content Provider

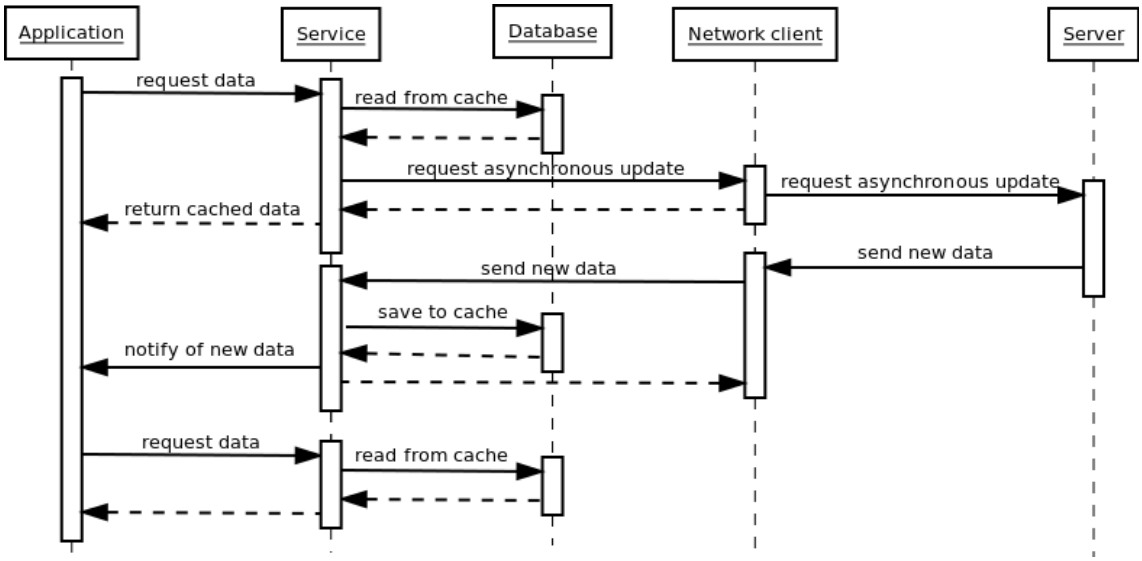

Figure 7.1: Network MVC for Mupe Force

is unnecessary, and is replaced with a Service. Because most of the data is connected, the Service maintains a model instead of just providing database cursors to Activities. A copy of player data is kept in the local cache, and notifications about changes are sent to the server in the background. A sequence diagram for retrieving data from the “Network MVC” Service for Mupe Force is shown in Figure 7.1.

Each Activity requests the data it is going to use. Cached data is used initially, and data from server becomes available later. For example in a shop, the player at first sees the same items that were available last time he was in the shop. New items might have become available since then, and they become visible after an asynchronous server request completes. Once an item has become available, it will remain available, so the player can buy items without having to wait for the server request to complete.

7.1 Local cache

Being a cache, the client side database is smaller and simpler than the server database. Especially meta data tables are much simpler. Schemata for player-specific data are closer to those on the server, but the amount of data is less. Most database tables have a corresponding model class.

The client database is designed so that a data set can be read with a single SQL query, and preferably from a single table. Data is read from the database into a cursor, which is then returned. It would be more complex to handle a situation where it would be necessary to read data from one table, and then another based on that. Nested queries and joins have their limits.

7.1.1 Model classes

The model classes generally have a `JSONObject` constructor for data received from the server, and a `Cursor` constructor for data read from the database. All of them have a method for putting their data in a `ContentValues` object for saving to the database. These methods are meant to be used only by the database handler, which resides in the same package as the model classes, and are therefore package private. The `Cursor` constructor and the `ContentValues` method keep the mapping between object properties and database columns inside the class.

Public methods include getters for other object properties, some data manipulation utilities, and very few setters. Most of the time, the objects are not changed except when new data is received from the server, and then a new object is created. This does not include relations between instances of the model classes.

Statistics have their own model class, but no database table, as statistics are saved in character, character type and item type tables instead. Instead of a `JSONObject`, a `Statistics` object is constructed using a `JSONArray`. Statistics are not named, but instead rely on their order in the array. Statistics are optional for character types and item types,

but that is not an issue. If the model object does not have statistics, then they are not saved, and when they are read back later, the `Cursor` will give zeros for their values. Unlike regular model classes, the `Statistics` class has a constructor without parameters. It constructs a set of statistics with zero values, and can be used in calculations. The `Statistics` class also includes various methods for calculations, as well as static fields and methods for identifying, ordering and displaying statistics.

Meta data tables and model classes include character type, item category and item type. Other classes save meta data references as a string, which is required to be unique. The string can be displayed to a user in some places, without needing the meta data object, and the string can be sent to the server as an identifier. Other classes need to get meta data objects for the names, so the meta data classes maintain static maps of existing objects, and have a getter that maps a name to an object.

A character type has an extra difficulty in its variable length usable item category and statistics increase arrays. The item categories are saved as a single string, with category names separated by newline characters. Statistics increases are saved as a byte array, with each increase in four bits. Each statistic's bytes start with a count of increases, also in four bits, followed by the increases themselves. If a statistic has an even number of increases, the second four bits in its first byte are ignored for easier bit arithmetics.

Item categories form a hierarchy, which is difficult to manage if objects are created as needed, like most model objects. Instead of constructors, the `ItemCategory` class has static factory methods for `JSONObject` and `Cursor`. They will find an existing `ItemCategory` object, update it with the given data, and return it, or if one does not exist, create a new object. Each object has references to its parent and children, and the factory methods also update those relations.

Besides the classes described above, all cached data is player specific. A player object is the root of a tree of characters, items and other model objects. The player object includes lists of the player's characters and items, as well as characters selected for battle.

A character object has references to equipped items, carried items, actions and abilities. The character class has some methods for calculating the character's statistics with equipped items. It can also calculate new statistics after changing equipment, without actually changing it.

The client implementation of a character action uses data from the server to determine if the action can be used and to find targets. This involves deciding its range dynamically depending on an item or the host's weapon, but does not have other functionality. The client-side action is only a representation for showing to the user and selecting targets. Actual execution of the action happens at the server.

7.1.2 Database operations

The client database uses an `SQLiteDatabase` object for lower level SQL operations, and

has getters for various types of data, an insertion method for each model class except statistics, and some update and delete methods. Statistics do not have an insertion method because they are inserted as part of another object.

Objects are inserted simply by placing their data in a `ContentValues` object, and using the `SQLiteDatabase.replace` method. The name column in meta data tables is defined as unique in the database creation SQL script. When inserting data with the same name, it will replace the existing data. A player's user name is similarly unique, and so are the server identifiers of characters and items.

An action's name is unique among the actions of a character. An action can also belong to an item type, when the item can be used. Therefore, the character's identifier, the item type's name and the action's name together create a unique identifier for the action. Either of the first can be null, but not both. On experimentation, it was discovered that SQLite allows three column primary keys, but allows multiple rows to have the same primary key in this case. Because of this, a key is generated by appending a new line character and the action's name to either character identifier or item type name.

An ability's name is unique among a character's abilities. Therefore, primary key is the combination of character identifier and ability name. As with actions, SQLite did not maintain unique primary keys. The cases where it did not enforce uniqueness are not allowed in game data, however, so the two-column primary key is used.

Each of the database's getter methods returns a `Cursor` with the requested data. In many cases, getting the data only needs a simple query. The more complex ones include character type and item type getters whose parameter is a list of names of the types to get. For these, an “OR” clause is added for each name, using a `StringBuffer`. The names are then inserted to the query using `SQLiteDatabase`'s selection argument mechanism.

Getting the items or actions for a group of characters works the same way, except that instead of names, server identifiers are used. They are added directly to the `StringBuffer`, as integers do not have the security issues of strings, and they would otherwise have to be converted into a string array.

The getter for item types by category is the most complex one, because of the hierarchical form of item categories. This getter creates a list of string arrays, which initially has category names received as parameters. For each string in each array, it gets the corresponding `ItemCategory` object, and adds the names of its children. The names are then used like the character type names before. This way, all items are retrieved recursively from subcategories.

Other methods include updating a player's last on-line date, and some purge methods. A purge method intended for use after a battle removes all players who have logged in from a different device, as well as all their data. Another purge method is intended for use when the application shuts down. It deletes guests and all of their data.

7.2 Network client

A MINA `NioSocketConnector` is used for connecting to the server. The client class has methods for connecting and disconnecting, extends `IoHandlerAdapter` for handling network events, and implements `Handler.Callback` for handling local messages.

The handler callback gets a message, which includes the name of a server remote method, and its parameters. If the current session is not connected, the client attempts to reconnect, and when it has a connection, sends a message to the server. The client creates the JSON-RPC request, and converts parameters to JSON format. It also adds an extra value to the JSON-RPC request object: user name.

When the client receives a message from the server, it forwards it to a `Handler` object. The message for the Handler has the server remote method's name, which is used as the request identifier in the JSON-RPC request, and the result JSON object. Responses have the same identifier as the request, but incoming messages do not have any identifier.

The client has identifiers for JSON-RPC messages and notifications, and expects to get one of them in the incoming local messages. The only difference in their handling is that the request identifier field is left out from notifications.

There is also a quit identifier, which can replace remote method name in the incoming local message. When the client receives a quit message, it disconnects, forwards the quit message to the outgoing local message handler, and ignores subsequent incoming messages, both network and local ones.

7.3 The Service

The Service is the connection point between the user interaction classes, the database and the network client. It takes data requests from the front end, relays them to the server via the client and returns cached data to the front end. It also receives incoming data from the server, puts it in the cache, and notifies the front end about new data.

When it is created, the Service creates its own thread, and a client. It then creates a Looper for its thread, and directs the client to try and connect to the server. If a connection can not be made, the Service broadcasts the error. There is little the Service can do without a server connection, but it can not do anything about the error either. Errors in server responses and messages are also broadcast.

The Service creates a Handler with the client as its callback, and makes itself the handler for the client's outgoing local messages. Both Handlers are created in the Service thread, so sending messages to server and handling messages from the server are both handled in the Service thread. Only the actual receiving of a server message happens in the client's thread.

It would make more sense to both send and receive messages in the client's thread,

and the use of Handlers would easily facilitate this. However, the client thread, created by MINA, is not easily accessible, can change on reconnect, and causes a need to use the client's Handler before it can be created. There is also no particular need for sending and receiving messages in the same thread.

The Service has methods for logging in and out. When the user successfully logs in, a player object is created to represent them. It is maintained until the user logs out. The current player has a getter, since it provides an access point to all of the user's characters, items and so on. A successful login is broadcast.

When the Service is being destroyed, in the `onDestroy` method, it logs out the user and waits until the logout is complete. Then it sends a quit message to the client. When the client forwards it to the Service's message handler, the Service quits the Looper. This is because the Looper can only be accessed from its own thread, while the `onDestroy` method is executed in the Android main thread. Meanwhile, the `onDestroy` method waits for the Service thread to die, which happens after the Looper quits.

7.3.1 Data retrieval

The Service has various getters, which will send a request for up-to-date data to the server and provide cached data in the return value. Each getter takes a `ContentObserver` as a parameter. It is notified when the Service gets new data from the server, and its absence signifies that there is no need to get new data from the server. Whether or not the Service sends a request for new data, it retrieves available data from the cache, creates model objects from it, and returns them.

The `Cursor` from the database is used for the notifications. The `Cursor` gets a notification URI, depending on the data being retrieved, and the `ContentObserver` is registered to it. When new data is received, the Service inserts it into the cache and sends a notification with the appropriate URI through the `ContentResolver`.

The Service keeps open `Cursor`s which have registered `ContentObserver`s, and has a public method for unregistering a `ContentObserver`. After unregistering, the `Cursor` is closed. If a `Cursor` already exists for a `ContentObserver`, the old `Cursor` is closed first, and then the `ContentObserver` is registered to the new `Cursor`, which is kept. Without a `ContentObserver`, the `Cursor` is simply closed.

Activities should notify the Service to get the necessary data, and should not rely on other Activities having readied the data in advance. Each Activity extends a custom `MupeForceActivity`, which has utilities for binding to the Service. Once the Service is bound, the Activity will be notified, and can start readying the data it needs.

The Activity first sends a `ContentObserver` to the Service, and gets cached data in response. It can then use the cached data, and will later receive a notification for new data. It will then request the data, without a `ContentObserver` since the data is now cached locally. If the data does not need to be updated later, the `ContentObserver` can be unregistered and destroyed.

Before the Service returns cached data, it will also make connections between the new model objects, and existing ones. The new model objects will replace older instances of the same data. The model will retain the new objects, and there is no need to get the data again from the server, unless it changes.

7.3.2 Updates

Updating the server data only needs the sending of a message to the server. In some cases, no response is needed. In others, the server sends back updated data, which is then cached. Many of the battle updates are notifications, because the server responds to them by sending the next battle event to both participating players.

When the user buys an item, its cost is first deducted from the player's money, so that it is shown correctly in Activities. If the buy action fails for some reason, the Service puts the item's cost back to the player's money. After a successful buy, the player's updated money is saved. Selling an item works the same way, except that money is of course added instead of deducted.

7.3.3 Incoming messages from the server

When the Service gets a message from the server, it will act almost the same as it does with responses. Any data that comes with the message is inserted to the cache, and the front end is notified. Since there is no dedicated `ContentObserver` for the notification, the Service broadcasts the message. The broadcast Intent also includes the data from the server, or a suitable identifier to retrieve it through the Service.

Before a battle, the server sends the opposing player's data. The opponent is made available in the same way as the logged in player. The server also sends the other player's battle force, which is put to the player object. Any items and abilities retrieved for the other player's characters are also added to the character objects.

8 CLIENT INTERFACE DESIGN

This chapter is about the design for the client's user interface and the implementation of the user interface classes. The interface was described in Chapter 2, but this chapter expands on that description in an Android-specific manner. Android's requirements for inputs are studied briefly, and then the more complex interfaces are designed based on that. Parts of the user interface which can be implemented by simply placing ready-made Android Views on the screen and reacting to the user's interaction with them are omitted.

This chapter is mostly about the animated Town View and Battle View. Another major part is a set of GUI components that combine Android Views with a class that controls their behavior. Activities are also outlined, although they have little functionality beyond retrieving data from the Service, which was introduced in Section 7.3, and transmitting user input between components. Lastly, there is the design of the action executor, which executes the action scripts received from the server. The scripts were introduced in Section 6.9.

8.1 Interface design guidelines

Interface design relies on inputs specified in Android's Compatibility Definition Document (CDD). A device must provide at least one software keyboard, even if it has a hardware keyboard. A hardware keyboard is optional, but if present, must be one of specified formats. A non-touch navigation option (such as a d-pad or trackball) is optional. [124, pp. 18-19]

The Home, Menu and Back functions must be provided, but are not necessarily bound to physical buttons. CDD for the latest Android version (4.1) states that they "may" be physical buttons [124, p. 19]. The CDD for the target Android version (2.3.3) states they "should" be physical buttons. In addition, there is a requirement that even if they are not physical buttons, they must always be accessible and may not obscure application display, or interfere with it. [125, p. 16]

In the newer CDD, this requirement has been expanded. Even if the Home, Menu and Back keys are shown on the screen, the remaining screen area must meet the CDD's requirements for screen configuration. These keys must be visible for applications which do not specify a system UI mode. The Menu key must be available to applications whose target version is 2.3.3 or less. [124, p. 19] The system UI mode (system UI visibility) was introduced in version 3.0 [54]. All devices must have a pointer input sys-

tem, either mouse-like, or touch [124, pp.18-19]. In version 2.3.3 CDD, a touch screen is mandatory [125, p. 16].

Despite differences between versions, a pointer system is mandatory. Also, the Home, Menu and Back keys are guaranteed to be always available, and not interfere with the application's screen area. Non-touch navigation might not be always available, but at least some sort of keyboard will be. Non-touch navigation is considered to include direction keys and a selection key. Based on this, three guidelines for the user interface were decided on. They are explained in Table 8.1.

1. Guideline:	The client software must be fully usable with only the mandatory pointer input system, except for text input. Text input is left to the underlying software and hardware.
2. Guideline:	Home, Menu and Back keys are supported. The Home button will perform its default function and will not be interfered with. The function of the Menu button will depend on current Activity, and can be omitted. The Back button will also perform its default function, except that escaping from battle is not allowed. Some Activities are also removed from the current Task and can not be reopened with the Back button. Altering the Task in this fashion does not affect the Back button's behavior, which depends on Activities in the Task. [126, 127]
3. Guideline:	Non-touch navigation is supported. The user must be able to navigate Views in a sensible order, focus on Views which can be clicked, and execute a click using the non-touch navigation. Clicking with the non-touch navigation will cause the same effect as clicking by touching. Views which can not be clicked can not be focused, either. When a focusable View is touched, it will also gain focus, despite touch mode.

Table 8.1: User interface design guidelines.

Using a pointer system is called touching from here, even if the system is not actually a touch screen. Keyboards are not used, except for text input and in the context of non-touch navigation. Other inputs or sensors are not used.

8.2 The Town View

The user can move the character in the town by touching or with the navigation keys. Touching directs the character to a square, and it will move there automatically. The character can be given a new destination while it is moving, and will start moving towards it without going to the old destination first.

Navigation keys move the character towards the direction of the key. It stops moving when the key is released. When multiple keys are held down, the character moves according to the one that was pressed last. If movement towards that direction is

blocked, the character moves according to the second last key press, and so on.

Movement according to touch control takes precedence over keys that were pressed before the touch. If the keys are still pressed when the character reaches the destination set by touching, it will again move according to the keys. Pressing a key while the character is moving towards a touched destination cancels the destination, and the character will start moving according to the pressed key.

The character does not have to follow square columns and rows, meaning that it can move partially in one and partially in another. It can not stop its movement that way, though. Releasing the navigation keys can leave the character between squares. It will then move to center itself according to squares, preferring the direction where it was moving before the user released the navigation keys. This also means that a single short key press will move the character exactly one square, assuming the direction is not blocked.

Using navigation keys in this fashion prevents moving focus to another View in the normal way. Instead, the town's menu is focused by pressing the selection key or the Menu key. Once focus is moved away from the Town View, the character can still move towards a touched destination, or to automatically center itself to a square, but movement according to navigation keys stops.

8.3 The Battle View

The Battle View's main part is the battlefield. There are other Views as well, depending on the battle's mode and current character. There are six modes to the battle: deploy, wait, observation, turn, targeting and execution. The Battle View is initially in deploy mode, and goes to wait mode when the user finishes deploying his characters. Wait mode ends when the other player has finished deploying, and observation mode is activated. When a character belonging to the user gets a turn, the Battle View goes to turn mode, and then to targeting mode when an action is selected. When the action's target is selected, the Battle View goes to execution mode. On an opposing character's turn, the Battle View is in observation mode. The user can also switch to observation mode in deploy mode and turn mode, and back. Targeting mode can be canceled, returning to turn mode.

Each mode has something that moves on the screen according to the user inputs. In most cases, key controls work in the same way as in the Town View. Touching, on the other hand, mostly puts the moving thing directly to the touched square, rather than having something moving from its former square to the touched square. The selection key can still be used for focusing a menu in some modes, but it has other uses as well. The usage of the Menu and Back keys varies as well. The Back key is not used for its default purpose because escaping from a battle is not allowed.

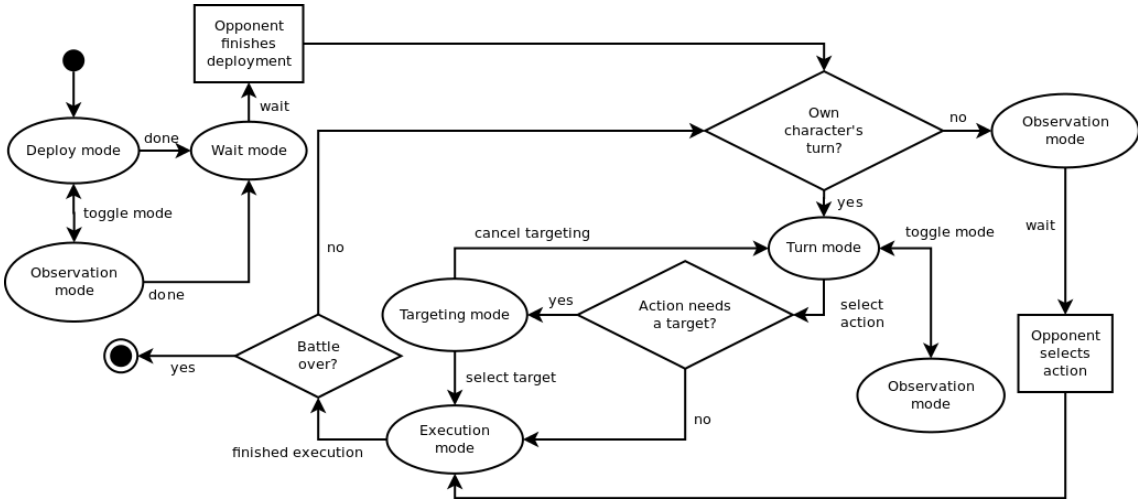

Figure 8.1: State diagram of battle modes.

8.3.1 Deployment

In deploy mode, the user controls a focus cursor, and can select squares. Touching a square will select it, and move the focus cursor to it. The selection key will select the square where the focus cursor is, and a selection cursor is shown on the selected square. Squares with battlefield obstacles can be focused, but not selected. The focus cursor can not move onto the shadowed area, and shadowed squares can not be selected. The focus cursor is hidden when focus moves out of the Battle View.

A set of slots with the user's characters participating in the battle is shown outside the battlefield. They can also be focused and selected in the same way as battlefield squares. When a selection is made, if there is already a selection, the characters in the selected slots or squares switch places. After that, the selection is removed.

There is also a “done” button, which becomes visible when all characters have been moved onto the battlefield. It will be hidden again if characters are removed from the battlefield. Pressing it will finalize deployment, and the user can not change it later. The Battle View now goes to wait mode, which is similar to observation mode, except that the shadow is still present and limits selections and cursor movements as before.

If the other player has not finished deployment, the user is shown a message to wait for it. When the opposing player also finishes deployment, the shadow is removed and the battle starts in earnest. The battlefield is mirrored vertically so that each player sees their own characters starting at the top two rows.

8.3.2 Observation mode

The user can toggle between deploy mode and observation mode by pressing the Back button, or a View button outside the battlefield. Toggling between turn mode and observation mode can be done the same way when one of the user's characters has a turn. On the turns of opposing characters, observation mode can not be exited, and the toggle

button is hidden. The done button in deploy mode will remain visible if the user goes to observation mode, if it was visible in deploy mode.

In observation mode, the user controls a selection cursor, and can select squares in the same way as in deploy mode. Unlike in deploy mode, the selection does not remain, but moves according to the navigation keys. Selecting a character shows a status dialog of it. Selecting a square without a character does nothing. Long-clicking a character will show its status in any mode other than the execution mode.

8.3.3 A character's turn

Turn mode is activated when one of the user's characters gets a turn. The user is also shown a message about the character's turn beginning. Movement control works like in the town, except that the character can not move outside its movement radius.

Pressing the selection key or the Menu key both move focus to an action menu. When the user selects an action which requires a target, targeting mode is activated. Each action has minimum and maximum range, counted in squares, and the squares within range are shadowed. Movement radius shadow is not shown in targeting mode. The player controls a selection cursor.

The selection cursor and selecting a target work in the same way as in observation mode, if the action can be targeted at empty squares. The cursor can be moved to squares where there are no targets, but not outside the target radius.

Most actions can not be targeted at empty squares, and for them the navigation keys move the cursor directly from one viable target to another. They can also be selected by touching, but touching a square without a target has no effect.

Once a target is selected, the action is executed, and the character's turn ends. The stay action does not require a target. Then the action is executed as soon as it is selected from the action menu, and targeting mode is not activated.

When an action is executed, the battle goes to execution mode and shows the action's effects. User input is ignored. Once the action is executed, the Battle View goes to either turn mode or observation mode, depending on whether the next character belongs to the user or the other player. It is also possible that the action ends the battle.

The user can cancel targeting by pressing the Menu key or the Back key. A cancel button is also shown outside the battlefield. In targeting mode, the action menu is shadowed, can not gain focus, and touching it has no effect.

Key-controlled movement in the Battle View stops when the Battle View loses focus. Automatic and touch-controlled movement can continue. In that case, targeting mode is delayed until the movement is finished. It depends on the character's position after it stops moving.

8.3.4 The action menu

At the top is “attack”, the basic way of damaging enemies and winning battles. At the

bottom is “stay”, which ends the character's turn without doing anything. On the left is “special”, whose workings depend on the character's abilities. On the right is “item”, which allows a character to use items from its inventory. Selecting the item action opens a dialog with the character's inventory. The user can select an item to use. Items that can not be used are shadowed and can not be selected.

The special action is disabled if the character has no active abilities to use. If there is only one, it is shown as the special action on the left of the action menu. If there are more, the special action opens a sub-menu with all of the character's active abilities. An ability which can not be used is disabled.

8.4 Animated views

The animated Town View and Battle View are each a large and important part of the application. They are created using Android's `SurfaceView`, in order to gain control of frame rate and provide better support for dynamic changes. The `TownView` class, which is used for walking a character around the town, follows the 2D graphics animation guidelines introduced in Subsection 3.5.5, as it extends `SurfaceView`, implements `SurfaceHolder.Callback` and includes an animation thread which extends `Thread`. The `BattleView` extends the `TownView`.

The Town View's function is providing a configuration interface for other classes, and relaying user input to the animation thread. It also provides its subclasses and the animation thread with access to its configuration. Configuration options include making any squares inaccessible, and setting a movement listener. The movement listener implements the `TownView.MovementListener` interface and is notified when the moving character moves into or stops in an active square.

Movement listener events are transferred from the animation thread to another thread using a Handler. The movement listener's actual function in the animation thread is only to determine if a square is active and there is cause for sending an event. This reduces the number of messages delivered by the Handler.

8.4.1 Basic graphics

For initialization, the Town View needs the number of squares in the grid, and the available screen size. The grid is square, so the smaller of width and height is used. A single square's width and height are equal to the smaller view dimension divided by the number of squares. The View's size is the product of square count and size, which can be less than the screen dimension.

The number of squares is set by the Activity which includes the View, while the screen dimensions are received through the `SurfaceHolder` callback. The View does not rely on the order in which square count and screen dimensions are received. When either one becomes available, the View will attempt to configure itself, but postpone the

initial configuration if the necessary properties are not yet available. This solution also allows the View to reconfigure itself later, if the number of squares or the size of the screen is changed. It is used in various other parts of the animation framework as well, when necessary configuration options are received from multiple sources.

For actually drawing anything, the Town View needs background and character graphics. Interfaces for background animations and character animations are created, and the animation thread starts with an instance of each. In addition, it has a separate default animation for each. A temporary animation can be inserted, and gets replaced by a default animation once it finishes. Each animation returns a boolean, which indicates whether or not the animation will continue. A default animation can always return false, since it will just get replaced by itself.

Other configuration options include character size, background image and character image. These will be used for creating the default background and character animations. They too depend on screen size, which determines how big squares are. Therefore an attempt is made at animation configuration every time any of those properties changes.

The animation thread checks that both default animations are ready and the surface has been created before it can be started. Once started, it will redraw the view at a rate of 25 frames per second. Redrawing involves updating the moving character's position. If after the redraw the character is not moving and no animations are active, the animation thread sleeps until it receives something new to draw. A default animation which actually draws a static image must return false, so that the animation thread does not keep redrawing the view needlessly.

8.4.2 User controls

Updates to the character's position depend on user input via touching and the navigation keys. Interfaces are created for handling touch and key input. The Town View creates a default touch handler, and a default key handler. They can be changed, but the Town View itself only uses the default input handlers.

An interface is created for movement controllers, as well. The Town View creates a default movement controller, which can not be changed. However, it can have three movement controllers of its own, and they can be changed. They are, in order of preference, automatic movement, touch movement and key movement. The default movement controller also implements centering the character to a square, which is used when none of the others move the character. Pressing a key will cancel the touch destination, so this order implements the precedence described in Section 8.2.

The default touch movement controller finds a path to a destination square, which it gets from the default touch handler. The default key handler keeps track of what keys are currently held down, and the times when they were pressed. The key press times are stored in an array, and movement direction preference is decided by order from highest to lowest, ignoring zeros. The key handler sets the preference order of directions to the

default key movement controller. It tries each direction in order of preference until it finds a direction where movement is not blocked, if there is one.

The touch and key handlers keep the latest user input. Before an animation frame is drawn, they are prompted to update the state of the movement controllers. Synchronization on the animation thread is used to prevent this state update from occurring simultaneously with user input. After drawing and before going to sleep, the animation thread synchronizes again and checks if new user input has been entered, if there is no ongoing movement or animation. When user input is received, the animation thread is notified in case it has gone to sleep. The `TownView` holds the animation thread's monitor while the touch and key handlers process input events.

Pressing the Menu key or the selection key clears key states, because that will move focus away from the Town View. The View would not receive events for key releases, and would keep moving according to the keys that were held down at the time focus was lost. Of course, the character's movement according to navigation keys should stop anyway, when the navigation keys start affecting another View.

User input can be disabled and is so when the View is constructed. The Activity which configures the Town View needs to enable user input once it has finished all necessary configurations. Touches are ignored while input is disabled, but keys are not, because key controls depend on maintaining a record of key presses and releases. Instead, key movement is ignored while input is disabled. Regardless of the setting, input is disabled for automatic movement and certain animations.

An interface named `AccessibleSquares` is used to determine which squares can be entered and which can not. An instance of it is placed in the Town View configuration, and it can also be replaced with another.

8.4.3 Movement controllers

The Town View uses four movement controllers: preferred directions, movement paths, automatic movement, and the default controller, which controls the order of preference among the others, and centers the character to a square. The preferred directions movement controller is the simplest of these. It gets an order of preference for directions, which does not need to include all four directions. It then tries to move the character to the most preferred direction first. If movement to that direction is blocked, it will be ignored and the direction with the next highest preference is searched, and so on. It will not move the character if movement is blocked in all directions that have a preference.

The path move controller finds a path from the character's current position to a destination square. If a path exists, the character will move along it to the destination. The path search algorithm is a fairly simple depth-first search that starts from the character's position and progresses to all neighbors. The search will continue to the square with the shortest distance to the destination. It also prefers squares that are on the same row or column as the destination, in order to create paths with less turns to them.

Each square has a path data object with the length of the path from the starting square to that square, and distance to the destination. The previous path square for each searched square is added to a 2D-array corresponding to the square grid. This approach reduces the creation and disposal of path data objects, but prevents the storage of multiple paths to any single square. Given the simplicity of the square grid and character movement, that is not an issue. The character might not always move along the path a human would find most sensible, but it gets to its destination.

The automatic movement controller implements walking out of the view, and walking in from outside of it. Touch destination is cleared when these automatic movements are activated. The controller determines the direction to walk out, or the entry point when walking in, depending on the character's position.

The automatic movement parameters are set in the main thread, by the current Activity. The Town View puts them to the automatic movement controller in its synchronized update phase. This way, internal changes can not override external input, and external input can not interfere with the drawing of an animation frame.

The default movement controller centers the moving character into a square when no user controls are in effect. The character will continue to move to the direction where it was moving before. It will stop so that its leading edge is along a square edge.

It might be that the character needs to be centered both horizontally and vertically. The default movement controller tracks the character's movements, and records last system time when it has moved to each direction. These times are then used to decide preference in the same way as key press times.

8.4.4 Battle View implementation

The Battle View provides support for multiple characters and various battle graphics. It does not use an existing image for a background, but instead generates a new bitmap using terrain data from the server. In order to draw the background, it implements the background animation interface and sets itself as the default background animation.

There are six mode objects in the Battle View, each implemented in a subclass of a `Mode` class, which extends `Observable`, and implements the touch and key handler interfaces, as well as `AccessibleSquares`. The Battle View replaces the superclass's input handlers with its current mode. Modes also affect graphics. There is no object for wait mode, which is simply observation mode with deployment shadow.

Deploy mode sets a focus cursor as the default character animation, and allows it to move only in the top two rows. The Battle View knows nothing of the deployment mode character slots, and can only move characters between squares. It has methods for setting a character in the selected square, and removing the character in it. These methods can only be used in deployment mode.

The Battle View remembers the square which has been selected, and draws it separately from the moving focus cursor. The selection is used for moving characters be-

tween squares, and for the setter and getter introduced above. The selection is only used in deploy mode.

When a square is selected, the deploy mode notifies its observers without data, and when characters are moved, it notifies with those characters. The data can be either a character object, or an array of two character objects.

The second mode is observation mode, which is activated as soon as the user has finished deploying the characters. The deploy mode object is removed, and it can not be activated again. The deployment shadow still exists, until the server notifies that the other player has also finished deployment. While it remains, the observation mode makes the squares below the first two rows inaccessible.

The default character animation in observation mode is a selection cursor. Clicking selects a square. If that square has a character, observers are notified with it as data. The observation mode also notifies its observers whenever it becomes the current mode, or is replaced by another mode.

The third mode is turn mode. The default character animation for it is a `BattleObject`, which draws the image of the active character, as well as a highlight. The movement controls from the superclass are used, since the character moves the same way as the character in the town. Squares outside the character's movement radius are not accessible, including all squares with an obstacle.

The fourth mode is the targeting mode. The default character animation for it is a targeting cursor, whose size and shape depends on the action. The targeting mode creates a list of squares within the action's range, and then narrows it down into another list which has squares with valid targets. The former is used for drawing shadows on the squares, as well as determining where the targeting cursor can move. The latter is used for determining where the cursor moves if it should move directly between targets, and checking if a selected square has targets.

The direct targeting mode is an extension of the targeting mode for controlling cursor movements when it moves directly from target to target. The cursor does not move at all if there is only one target. Touch control moves the cursor directly to the touched square, if it has valid targets.

The direct targeting tracks key presses through a default key handler, which does not have a movement controller attached. When a direction key is pressed, the direct targeting mode sets itself as character animation. It does not draw animations itself, but updates the state of a direct movement animation, which calculates coordinates for a selection cursor, which handles the drawing.

The direct movement animation first gets the next target square as its destination. It calculates movement speed so that the movement from one target to the next takes the same amount of time as a character moving one square. It then animates movement from the starting square to the destination, and can also recalculate its properties and switch animation direction on the move.

The latter capability is used when a key is pressed while the animation is going on. No update is made if the user presses to the same direction where the animation is already moving. Otherwise the animation's direction is reversed.

Once the animation reaches its target, the direct targeting mode will end the animation if no keys are pressed. While a direction key remains pressed, the animation is given the next target square as its new destination. Selecting a target by touch clears key states and stops key movement, until next time a key is pressed.

8.4.5 Concurrency

Many of the Town View's configuration properties are used in the animation thread, as well as animations. Updates to them, on the other hand, mostly happen in the Android main thread, which handles user input. All variables that can be set in both threads are marked with the keyword `volatile`, but that is insufficient by itself.

In the main thread, updates to animations and other properties are done while holding the animation thread's monitor. Before drawing a frame, the animation frame acquires its monitor and updates its state. It will then use that state for the next frame, ignoring state changes in objects related to the main thread. Synchronizing on the animation thread allows multiple changes to be done atomically, which is necessary especially when changing battle modes.

Variables are divided into four groups according to their intended use. The first group is written in the main thread, and read in both the main thread and the animation thread. These are placed in a configuration object, which has getters, setters and some utility methods, including copying. They are changed while holding the animation thread's monitor, and the animation thread then copies them to a different object while holding the same monitor. The second group can be set in both threads, but is only read in the animation thread. They are handled like the first group.

The third group is set in the animation thread, and read in both threads. Actually, this group only includes the moving character's position, which can be set in the main thread as well. However, it is not necessarily possible to set the position, because View configuration may be incomplete. The position is saved, and then set in the animation thread, which does not start before all necessary View configurations are complete. Both getters and setters are synchronized in order to prevent concurrent reads while the animation thread is updating the current position.

The fourth group includes the animations, which have their own properties class. They can be set in both threads, but are only read in the animation thread. Most of the time, they are set in the main thread as a response to user input, events from the server, or View configuration. Movement and animation events are also transferred to the main thread. All of those make their changes to a main thread properties object. The animation thread copies the changes to its own object, and then clears the relevant fields in the main thread object. If there are no changes for a variable waiting in the main thread ob-

ject, the animation thread can do automatic changes to it.

The boolean for enabling or disabling user input belongs to the first group. Regardless of its setting, the animation thread can disable input due to automatic movement and ongoing animations.

The animation properties class has a `ReentrantLock`, which the animation thread holds while drawing a frame. It is meant for changes to the internal states of animation objects. Changes should be made while holding the lock, to prevent concurrent reads in the animation drawing methods.

In addition to the main thread and the animation thread, the Battle View deals with an action executor thread. Its purpose is to control animations and delays associated with them, while leaving the main thread and the animation thread free. It transfers messages from the main thread to its own thread using a Handler.

The Battle View has some variables which are set in the main thread and accessed in both threads. They are copied for the next draw as in the super class. Properties related to the square grid battlefield have their own class, `Battlefield`. The properties which are accessed by both the action executor thread and the main thread are synchronized.

8.5 View components

Several View components can be reused between different Activities, or are separated from a single Activity in order to promote encapsulation and clarity. These components are made of one or more Android Views, as well as code to control the components' functionalities.

The user interface description of a component is written as an Android XML layout, which can be included in other XML layouts. The controller part is written in Java, and needs the Java object inflated from the XML. In many cases, a component also includes custom sub-components, which are written in their own XML layout files. Any number of sub-components can be inflated dynamically at runtime.

8.5.1 Highlight Views

These add an additional state besides those already available in Android Views: highlighting. The View will change its background drawable according to its highlight state. Highlighting does not affect any of the default states, and a View can be both selected and highlighted, for example. There are two of these Views: a highlight button, which only adds highlighting to default button behavior, and a list View.

A list View is a linear layout with text Views for name and price, as well as two image Views. It can be configured to show or hide any of the others, but name is always visible. Clicks on the name are handled as if the whole View was clicked, and the other sub-Views are not clickable.

Both the button and list View override the callback method which handles touches.

By default, a View which is focusable in touch mode will request focus when touched, and trigger a click only when it already has focus. If the highlight View does not have focus, it requests it, and then lets the superclass continue from there. The View will get focus, the superclass will acknowledge that it is focused, and trigger a click.

8.5.2 Item and character list

Shows a list of items and a list of characters, as well as an item description. An item and a character can be selected separately. The item and character list is `Observable`, and notifies observers when selected item or character changes. The item list can be configured to show or hide item prices, and can have a multiplier for them. Item category icon is shown on the left of each View. The list can also have a filter to hide some of the items in the list, and both the item list and the character list can be hidden separately.

Items are shown in a linear layout of list Views, and characters in a linear layout of highlight buttons. When a user clicks on an item or character, it is selected. Selecting an item highlights characters that can use the item, and the other way around. A selected item's description is shown. When a user clicks on an item or character which is already selected, it is unselected. If item or character is set programmatically, its selection state is not changed. Unselecting characters can be disabled. A character will always be selected, and clicking the selected character again has no effect.

The item list can have either individual items or item types. Changing the list of items or item types does not change the selected item, unless the new list does not include the selection. Then the selection is removed. The same is true for characters. Selected item is also unselected if it is hidden because of the filter.

8.5.3 Statistics changes

Shows changes to a character's statistics in a table of 40 cells, which are by default ordered in ten rows and four columns. The first column has a label text, the second has the original value of a statistic, the third has the statistic's changed value, and the fourth column has the difference between the two. Statistics which increase are shown in green and statistics which decrease are shown in red.

The grid can also be put to a wide mode, with five rows and eight columns. The statistics are then ordered so that the first five statistics are in the first four columns, and the other five statistics are in the last four columns.

Statistics changes are initially invisible, and are shown only when the component has a character, as well as statistics changes to show. Changes can be for an item or item type to equip, with an equipment slot, or a set of changes to the statistics as an integer array. If the character can not equip the item or item type, the statistics will remain invisible to indicate this.

8.5.4 Character inventory and abilities

The character inventory component shows slots for equipped and carried items. It has a setter for the character, whose items are immediately shown when set. Besides displaying the character's items, the component has methods for changing equipment and carried items. These are delegated to the character class, but the inventory also updates itself according to changes. The inventory also tracks changes to the character's items, and can be queried to determine if the character's items have changed since the character was set.

A user can select item slots, which causes the selected item's (if any) description to be shown below the item slots. The inventory is `Observable`, and notifies observers of the item slot selection changing. The inventory is a linear layout of list Views. It can alternately show a list of the character's abilities.

8.5.5 Character selector

Shows a character, and arrow buttons to the left and right. These can be used to select the previous or next character, respectively. The left arrow is disabled for the first character, and the right arrow for the last character.

Once given a list of characters, the character selector handles changing the character when the buttons are clicked. It is an `Observable`, and notifies observers with the new character when selection changes. It has no means of deselecting a character, unless the character list is empty.

8.5.6 Character slots

Shows six character types in a row, a title and six characters in another row. The component can be set to allow or prevent user from removing characters from the slots. If allowed, characters can be removed by long-clicking. Similarly, it can allow or prevent selecting empty slots. The character type row and title can be hidden. Any slot can be disabled, so the user can not select it.

The character slots component is `Observable`, and notifies observers when a character is removed, or the selection changes otherwise. It handles moving characters from slot to slot, and also has utilities for counting the characters on the character row and getting the names of their classes.

8.5.7 Character status

Shows character's picture, name, level, experience, statistics, items and abilities. Also includes a button for opening the character's inventory. The character's hit points and magic points can be shown either like the other statistics, with a label and a value, or in an alternate mode that shows current hit points and magic points.

A label is shown for current values, and another text View shows them as “current value slash maximum value”. There is also a graphic bar which shows the percentage of

current from maximum. The bar is originally hidden, and shown only when the current value mode is enabled. The bar gets horizontal space left over from the text Views.

Since they are hidden at first, the image Views for the bars are sized 0x0 pixels. The bars need to be drawn after layout completes, and the bar's width has been determined. There is no event to respond to layout being completed, but instead the character status component is made a `Runnable`, and posted to be run in the main thread.

A layout is scheduled first, when the image Views' visibility is changed, and the `Runnable` is posted after that. Therefore it is in the main thread's queue after the layout. The layout schedules a screen update, which is queued after the `Runnable`. The `Runnable` can therefore draw bitmaps for the image Views using their dimensions after layout, but before the updated Views are shown on the screen.

8.6 Activities

Most of the functionality of any Activity resides in the `ContentObserver`s, `Observer`s for the components it uses, and event-handling methods for clicking components. There is very little anything else.

Each Activity binds itself to the Service using the `MupeForceActivity` superclass. When binding is completed, the Activity sends requests for data it needs. After a server update, the Service sends notifications through `ContentObserver`s, and the Activity makes new requests based on the new data. This is repeated until all necessary data is available.

The Login Activity, being the first Activity in the application, is the first to bind to the Service. The Login Activity also starts the Service, so that it remains running after the Login Activity is exited. The Login Activity destroys itself on exit, so the Service would also stop if it was only bound.

The Login Activity saves user name, password and login settings to shared preferences after a successful login. Of course, user name and password are only saved for a registered user, and they are not saved if the remember login setting is disabled. User name and password are private to the Login Activity, but the Settings Activity can change the other settings.

The Shop Activity requires two parameters in its starting Intent: shop title and root item category. These are set in the Town Activity, because the shops are entered from the town. The title becomes the Activity's title, and the root item category determines what the shop sells. The Shop Activity determines items on sale by that category's child categories.

8.6.1 The Town Activity

The Town Activity has an animated View for the town itself, and an action bar. The action bar buttons start new Activities, and new Activities are also started when the char-

acter is moved to town locations.

The marketplace is not a separate Activity. The Town Activity only uses an animation in the Town View to transition from one background image to another, and replaces `AccessibleSquares` and `MovementListener` implementations in the Town View's configuration. The Town View has a nested class for each of the town plaza and the marketplace, implementing the above interfaces.

The background transition animates the character walking from one section of town to the other. When moving to the marketplace, the town plaza image is moved down and off the screen, while the marketplace image moves in from the top. When leaving the marketplace, the animation's direction is reversed. The background transition uses an Android `TranslateAnimation` to move the graphics. The character graphic gets the same translation as the old background, making the character move along with it. When starting the animation, the Town View disables user input and also starts an automatic walk out move, which causes the character to move onto the new background. When the background transition stops, the Town View moves the character's position to coordinates matching the new background, and starts an automatic walk in move. The character moves further onto the new background, leaving one empty row of squares between it and the town edge. Then it stops, and user input is enabled again.

When new Activities are started, the town Activity remains in the background, and is activated again when the user returns from the other Activity. Since the Login and Initial Character Selection Activities do not remain in history, the town Activity is the root of the application's Task. When it is closed, the application ends. Therefore the town Activity, when destroyed, logs out the user and then shuts down the Service.

Logging out through the action bar also destroys the town Activity, but does not stop the Service. The town Activity starts the login Activity, but first adds an override to the shared preferences, so that automatic login is not executed.

8.6.2 The Battle Activity

Battle modes are implemented in the Battle View where they concern the Battle View itself, and in the Battle Activity where they concern other Views. The Battle Activity updates other Views and components depending on mode changes in the Battle View, and they do not need to know about current mode.

Besides the Battle View, the Battle Activity includes character slots for deployment, two action menus, a done button, a cancel button, and a mode toggle button. These are all placed in a frame layout outside the Battle View, one on top of the other. Visible Views change depending on mode and other things, and it does not matter if Views that can not become visible at the same time are overlapping.

The done button is visible in deploy mode when all characters are on the battlefield. It will remain visible if the player switches to observation mode during deployment, but be hidden if characters are removed from the battlefield. Pressing it hides the done but-

ton itself, as well as the character slots, and sends the battlefield positions of the player's characters to the server. The Battle Activity then notifies the battle View to end deploy mode.

The mode toggle button is visible in deploy mode, observation mode and turn mode, except during opponent's turns. It will switch from deploy or turn mode to observation mode and back. The cancel button is visible in targeting mode. It causes a return to turn mode.

The Battle Activity also has a Broadcast Receiver for incoming messages from the server. These are used to update the Battle View's state, and execute the actions of both players. Actions are of course executed in the action executor thread, and it is also used to schedule events. When a battle end message is received, the Battle Activity starts the Battle Results Activity, which shows reward and level ups. The Back button finishes this Activity, and returns to the Town Activity.

8.7 The action executor

The Battle Activity creates the action executor, which starts its own thread as a Looper, and creates a Handler for transferring messages to it. When starting up, it waits until it has both created its Handler and received a notification from the Battle Activity that the battle is ready to start. This is to ensure that the Battle Activity can not send messages to the Handler before it exists, and that the first begin turn event is delayed until all preparations for the battle are completed.

After that, the action executor receives notifications for beginning turns and executing actions, which it places to a queue through the Handler. The queue ensures that the next turn does not begin before the previous turn's action has been completed. Eventually the action executor receives a battle end notification and stops its Looper.

When the action executor receives an action script, it will first move the character whose turn it is to the location in the script. If the character belongs to the user, it is already in the given position, but if the character belongs to the opponent, it needs to be moved. The path move controller, which is also used for touch-controlled movement, handles the move. It is set as automatic movement, and the action executor waits for the movement to complete. The Battle View is now in execution mode, which decides the accessibility of squares based on obstacles and characters, preventing the character from moving across them.

The action executor interprets action scripts into `ActionDraw` objects which draw graphics, as well as `Interpolator`s and `Animation`s. Then it executes the script's action queue, which involves starting animations, waiting for them to complete, and making changes to the local model.

A script-based animation is drawn over anything else in the Battle View. A class is implemented to draw the graphics using the `ActionDraw` objects, but interpolators and

animations are provided by Android. An Android `Animation` can only be applied to a View, which is not how they are used here. Instead, the `Animation`'s `Transformation` object and alpha value are copied to the `Canvas` and `Paint` used for drawing the animated graphic.

9 CONCLUSIONS

The Mupe Force project can be called a success in that a functional application was developed. Unfortunately, it includes only the properties which were selected as the minimum set from the original specification. It was intended to implement more of the game's planned features, but development took too much time for that.

There was some unnecessary (in a manner of speaking) tweaking of the user interface, as was expected in the early stages of this project. The biggest difficulties that slowed down the project, however, were difficulties in the concurrency and graphics of the Town and Battle Views, certain less informed decisions regarding the client-side data model, and the peculiarities of Android layouts. These will be discussed in more detail later in this chapter.

The next section discusses the technologies used in the project. After that, more detail about GUI programming with Android is included. These are things learned during development, in contrast to Chapter 3, which is based on documentation. At the end of this chapter, there are some mistakes which were repeatedly made during development, and some points about developing Mupe Force further.

9.1 About the project

Networking for the project was considerably easier than expected, even though Java-based technologies JMS, RMI and Servlets were found to be ill suited for this project. After that, several networking technologies were studied, starting from HTTP and progressing into more complex ones like XMPP. In the end, something as basic as TCP was found to be sufficient. Also, this was learned through the discovery of Apache MINA, which implements TCP communication for both the server and the client.

The initial idea of having the server remotely call methods on the client to execute character actions was found to be unnecessarily difficult. An RPC-type solution would have required much longer messages than the script solution which was decided on. The JSON server's cross-platform compatibility would also have suffered, and it would have placed more requirements for theoretical future client implementations.

As for Android, which was the primary technology studied in this project, it is good but has its quirks. The life cycle of an Activity is very important to learn, but can be difficult to understand. The same can also be said of the entire structure of Activities, Intents, Content Providers and so on. The structure promotes modularity, and the easy navigation between Activities from different applications is very interesting. Even

though it was not needed in this stand-alone project.

After studying Android, the initial plan was to have each Activity retrieve only the model objects it needed, and then discard them. This idea turned out to be the biggest mistake in the project, as it produced multiple problems.

First, it became obvious that additional meta data was required to properly connect different model objects to each other. A solution compliant with the original idea would have been to include meta data in object data, but this would have caused considerable redundancy and made the JSON messages much bigger. Instead, in addition to player--specific data, each Activity also retrieves the meta data for making connections in the model. This led to redundancy and copy-paste programming as more Activities were implemented.

Each Activity retrieves all the data it needs, requesting updates from the server for all of it. At first, data is read from the cache and new model objects are created. When data is received from the server, new objects are created for insertion into the cache, the data is read back, and new model objects are created from it. Although not erroneous, this behavior causes a lot of unnecessary database reading and writing, as well as server communication and object creation. Updating Activities with the new model objects also becomes a problem in many cases.

9.2 About the Android GUI API

A recurring issue during the project was Android's incomplete documentation. There were mistakes, such as discrepancies between the common layout tutorial and detailed documentation for the layouts. All in all, these did not pose major problems, but extra work was certainly required.

A more difficult but less common problem is the calculation of view dimensions. None of the layouts in Android supports setting view dimensions proportionally to the parent view group. The closest alternative is creating zero-sized views in a linear layout and using weights to distribute extra space between them.

There were also several situations where a “wrap content” text View would not use available horizontal space to fit the text. Instead, the text would be folded to a second line or cut completely depending on settings. Image Views, in contrast, would occasionally reserve much more space than needed, even with a static pixel size, or a zero size. These are likely due to layouts that calculate View dimensions and distribute space.

Out of the five available layouts, and excluding the deprecated `AbsoluteLayout`, only `LinearLayout` can be used to constrain a View's size. `TableLayout` can be used to some extent, but only horizontally, as its rows extend `LinearLayout`. This becomes visible at times, when columns are different widths on different rows. The table layout supposedly supports column spanning [56]. The XML attribute for that was treated as an invalid parameter, though. Also, margins and paddings for table cell Views were ignored

in several cases.

The relative layout does not support layout weight or any similar mechanism. In case the children are too large to be positioned within the available space, which is often the case, they are simply placed one on top of the other. Also, according to experimentation, a relative layout with “wrap content” dimensions would spread to cover all of the available area, and stretched the child views accordingly. In other words, it makes no distinction between “wrap content” and “match parent”.

The `GridView` was found to be very difficult to understand. Depending on settings, it would for example make columns too narrow to fit their content, while leaving excess empty space for another, wider, column. Focus for the grid was shown as an orange box, much larger than the focused View. None of the settings changed in attempt to make the grid and its cells not focusable had any effect.

Despite all of the above, Android's GUI API was found to be preferable to Swing and lwuit (Lightweight UI Toolkit for Java ME). The program structure, Activities, Services and so on, as well as various mechanisms for doing things behind the scenes, such as Content Observers and Handlers, are great. With a better understanding of them at the beginning of this project, the biggest difficulties in the client's model and animated Views would have been greatly reduced.

9.3 Pitfalls

Here are a few things which repeatedly caused mistakes and annoyances during the project. First the most common one, which was routinely corrected later in the project: Eclipse's automatic features for managing `import` statements can not manage a class named `Character` because of the class `java.lang.Character`.

The Java editor in Eclipse is not meant for editing “xml_style” names. The XML editor, on the other hand, is not meant for editing “JavaStyle” names. Both types of names appear in both types of files, making text editing noticeably clumsier than it needs to be.

The XML editor could also benefit from tighter integration with the Java side. For example, an Android application could not be launched directly from Eclipse's Run and Debug buttons when an XML file was open. Instead of the Android application being launched, an XSL transformation was done on the XML file. This caused an error report and generated an empty result file, which Android could not compile. The output file had to be removed before the project could be compiled. Automatic refactoring also does not carry between Java and XML very well.

Android expects colors in a four-byte ARGB (alpha, red, green, blue) format. Therefore, setting a background or text color, for example, programmatically in RGB (red, green, blue) format makes the color invisible. This is because the alpha component in a three-byte number is zero and the color is fully transparent. In XML, colors can be writ-

ten in RGB format.

The `JSONObject` defined in jabsorb automatically converts an `int` array into a `JSONArray` of integers. Android's `JSONObject`, on the other hand, simply uses `toString` to convert the array into a `String`, and therefore it is necessary to manually convert the `int` array into a `JSONArray`.

An important aspect of Android GUI programming is finding views by identifier. Views are searched with a depth-first search, rather than a breadth-first search. In practice, if a layout includes other layouts from other files, the view identifiers set in the included files can override view identifiers in the including higher-level layout. This can be avoided by using different identifiers, but it still requires a knowledge of what identifiers appear in the included files.

9.4 Further development

Before any further development, the data retrieval model should be improved. The Service should take more control of the data, and only access the server when necessary. Except in battle, any change at runtime begins in the client, and it would be sufficient to retrieve each data item once, and then only react to changes from the user. The Service could also take control of preparing the meta data for connecting model objects that an Activity has requested.

The Service returns cached data when an Activity asks for it, then contacts the server, and notifies the Activity to get the new data from the server. This causes redundancy in the Activity, because the same data is handled when it is first retrieved, and then again when a Content Observer notifies. Instead, the Service should be modified to return nothing, and only notify Content Observers about available data, which would then be retrieved through the model. This goes well together with the previously mentioned improvement.

Since Activities call for data in the main thread, and the Service handles incoming data in its own thread, there is a risk of concurrent modification of the model. Such problems were avoided by designing data requests to be sequential, but that hinders the usage of cached data while the Service gets new data from the server. Concurrency should be better handled in the Service and inside the model so that Activities would not have to take it into consideration.

As for new features, there is much room for improvement in the game itself. Many of the features in the original specification were left out due to time constraints. Graphics could obviously be improved by quite a bit, and sounds could be added.

It was initially intended to create a Content Provider for images. All of the images are now packaged with the client and resized as necessary. The plan was to get the images from the server, which would also allow them to be replaced by new ones without having to re-install the client.

Considering the addition of new character classes, items, abilities and so on, it is unnecessary to store a growing number of images on the client. Instead, the Content Provider could retrieve necessary images for example before a battle, and delete images that have become unnecessary.

An important aspect in downloading images from the server was to be re-colorization of character images. Currently, all characters of the same class look the same, and they can not be told apart during battle without checking their status. It would be much better if each player could set a color scheme for their characters. It would add personalization possibilities to the game, and make it possible to recognize which player a character belongs to.

REFERENCES

1 Android Developers. Android, the world's most popular mobile platform. [WWW]. [Referenced: 13.11.2012]. Available: http://developer.android.com/about/index.html.

2 Kashyap, V. MakeUseOf. 2010. What Is An API & What Are They Good For? [Technology Explained]. [WWW]. [Referenced: 13.11.2012]. Available: http://www.makeuseof.com/tag/api-good-technology-explained/.

3 Apache Portable Runtime Project. 2012. Welcome! - The Apache Portable Runtime Project. [WWW]. [Referenced: 5.8.2012]. Available: http://apr.apache.org/.

4 Daintith, J., Wright, E. A Dictionary of Computing. 2008. Oxford University Press. 608 p.

5 Google Code. Google Projects for Android: C2DM. [WWW]. [Referenced: 31.3.2012]. Available: http://code.google.com/android/c2dm/.

6 McCarthy, P., Crane, D. Comet and Reverse Ajax: The Next Generation Ajax 2.0. 2008. Apress Media. 100 p.

7 The Java™ Tutorials. Serializable Objects. [WWW]. [Referenced: 15.11.2012]. Available: http://docs.oracle.com/javase/tutorial/jndi/objects/serial.html.

8 Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. 1994. Addison Wesley Longman, Inc. 395 p.

9 Rouse, M. SearchWinDevelopment. 2006. GUI (graphical user interface). [WWW]. [Referenced: 13.11.2012]. Available: http://searchwindevelopment.techtarget.com/definition/GUI.

10 Allamaraju, S. 2001. Java Server Programming: Principles and Technologies. [PDF]. [Referenced 13.11..2012]. Available: http://www.subbu.org/articles/j2ee/java_server_programming.pdf.

11 JSON. Introducing JSON. [WWW]. [Referenced: 31.3.2012]. Available: http://www.json.org/.

12 JSON-RPC. 2011. JSON-RPC 2.0 Specification. [WWW]. [Referenced: 17.11.2012]. Available: http://www.jsonrpc.org/specification.

13 Suomela, R., Räsänen, E., Koivisto, A., Mattila, J. Open-Source Game Development with the Multi-User Publishing Environment (MUPE) Application Platform. [PDF]. [Referenced: 13.11.2012]. Available: http://scholar.google.com/scholar_url?hl=fi&q=http://pdf.aminer.org/000/464/326/open_source_game_development_w

ith_the_multi_user_publishing_environment.pdf&sa=X&scisig=AAGBfm0pxAsCdRrptRfr3QvIDSQoIMJ0VQ&oi=scholarr.

14 Rouse, M. WhatIs.com. 2011. model-view-controller (MVC). [WWW]. [Referenced: 13.11.2012]. Available: http://whatis.techtarget.com/definition/model-view-controller-MVC.

15 MySQL. Why MySQL? [WWW]. [Referenced: 15.11.2012]. Available: http://www.mysql.com/why-mysql/.

16 Paul, J. Javarevisited. 2011. Observer design Pattern in Java with Real world code Example. [WWW]. [Referenced 3.11.2012]. Available: http://javarevisited.blogspot.fi/2011/12/observer-design-pattern-java-example.html.

17 OpenGL.org. 2012. FAQ. [WWW]. [Referenced: 13.11.2012]. Available: http://www.opengl.org/wiki/FAQ.

18 Tyagi, S. Oracle Technology Network. 2006. RESTful Web Services. [WWW]. [Referenced 13.11.2012]. Available: http://www.oracle.com/technetwork/articles/javase/index-137171.html.

19 Rouse, M. SearchSOA. 2009. Remote Procedure Call (RPC). [WWW]. [Referenced: 13.11.2012]. Available: http://searchsoa.techtarget.com/definition/Remote-Procedure-Call.

20 Adams, E. Fundamentals of Game Design. 2010. New Riders. 675 p.

21 Fendelman, A. About.com. Definition of SMS Text Messaging: What is SMS Messaging, Text Messaging? [WWW]. [Referenced: 13.11.2012]. Available: http://cellphones.about.com/od/phoneglossary/g/smstextmessage.htm.

22 Skonnard, A. MSDN. 2003. Understanding SOAP. [WWW]. [Referenced: 13.11.2012]. Available: http://msdn.microsoft.com/en-us/library/ms995800.aspx.

23 W3Schools. Introduction to SQL. [WWW]. [Referenced: 15.11.2012]. Available: http://www.w3schools.com/sql/sql_intro.asp.

24 SQLite. Distinctive Features of SQLite. [WWW]. [Referenced: 16.11.2012]. Available: http://www.sqlite.org/different.html.

25 Diffen. TCP vs UDP. [WWW]. [Referenced: 17.11.2012]. Available: http://www.diffen.com/difference/TCP_vs_UDP.

26 RPGFan. The Definition of a Role-Playing Game! [WWW]. [Referenced: 13.11.2012]. Available: http://www.rpgfan.com/editorials/old/1998/0007.html.

27 Everything2. 2003. tactical RPG. [WWW]. [Referenced: 13.11.2012]. Available: http://everything2.com/title/tactical+RPG.

28 Connolly, D. W3C Architecture domain. 2006. Naming and addressing: URIs, URLs, … [WWW]. [Referenced: 13.11.2012]. Available: http://www.w3.org/Addressing/.

29 W3C. 2001. URIs, URLs, and URNs: Clarifications and Recommendations 1.0. [WWW]. [Referenced: 13.11.2012]. Available: http://www.w3.org/TR/uri-

clarification/.

30 W3C. XML Essentials. [WWW]. [Referenced: 16.11.2012]. Available: http://www.w3.org/standards/xml/core.

31 Tutorialspoint. XML-RPC Introduction. [WWW]. [Referenced: 17.11.2012]. Available: http://www.tutorialspoint.com/xml-rpc/xml_rpc_intro.htm.

32 XMPP Standards Foundation. XMPP Technologies Overview. [WWW]. [Referenced: 13.11.2012]. Available: http://xmpp.org/about-xmpp/technology-overview/.

33 Komppa, J. Adventure Classic Gaming. 2006. How to design an ideal computer role playing game? [WWW]. [Referenced: 19.11.2012]. Available: http://www.adventureclassicgaming.com/index.php/site/features/118/.

34 Barton, M. Gamasutra. 2007. The History of Computer Role-Playing Games Part 1: The Early Years (1980-1983). [WWW]. [Referenced: 18.11.2012]. Available: http://www.gamasutra.com/view/feature/3623/the_history_of_computer_.php.

35 Sinister Design. 2011. What makes an RPG an RPG: a universal definition. [WWW]. [Referenced: 19.11.2012]. Available: http://sinisterdesign.net/?p=785.

36 Doucet, L. Gamasutra: Lars Doucet's Blog. 2011. Rebooting the RPG. [WWW]. [Referenced: 18.11.2012]. Available: http://www.gamasutra.com/blogs/LarsDoucet/20110309/7182/Rebooting_the_RPG.php.

37 Encyclopedia Gamia. Tactical role-playing game. [WWW]. [Referenced: 19.11.2012]. Available: http://gaming.wikia.com/wiki/Tactical_role-playing_game.

38 Lee, W-M. Beginning Android Application Development. 2011. Wiley Publishing, Inc. 428 p.

39 Mednieks, Z., Dornin, L., Meike, G. B., Nakamura, M. Programming Android. 2011. O'Reilly Media, Inc. 482 p.

40 Zechner, M., Green, R. Beginning Android 4 Games Development. 2011. Apress Media. 677 p.

41 Android Developers. 2012. Content Providers. [WWW]. [Referenced: 31.3.2012]. Available: http://developer.android.com/guide/topics/providers/content-providers.html.

42 Android Developers. 2012. `Looper`. [WWW]. [Referenced: 7.10.2012]. Available: http://developer.android.com/reference/android/os/Looper.html.

43 Android Developers. 2012. `Handler`. [WWW]. [Referenced: 7.10.2012]. Available: http://developer.android.com/reference/android/os/Handler.html,

44 Komatineni, S., MacLean, D. Pro Android 4. 2012. Apress Media. 1020 p.

45 Android Developers. 2012. `ContentProvider`. [WWW]. [Referenced 26.3.2012]. Available:

http://developer.android.com/reference/android/content/ContentProvider.html.

46 Android Developers. 2012. Services. [WWW]. [Referenced 27.3.2012]. Available: http://developer.android.com/guide/topics/fundamentals/services.html.

47 Ableson, F. IBM developerWorks. 2009. Networking with Android. [WWW]. [Referenced 27.3.2012]. Available: http://www.ibm.com/developerworks/opensource/library/os-android-networking/.

48 Burnette, E. ZD Net. 2008. Java vs. Android APIs. [WWW]. [Referenced 27.3.2012]. Available: http://www.zdnet.com/blog/burnette/java-vs-android-apis/504.

49 Android Developers. 2012. User Interface. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/guide/topics/ui/index.html.

50 Android Developers. 2012. Layout Resource. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/guide/topics/resources/layout-resource.html.

51 Android Developers. 2012. Providing Resources. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/guide/topics/resources/providing-resources.html.

52 Android Developers. 2012. `Activity`. [WWW]. [Referenced: 7.10.2012]. Available: http://developer.android.com/guide/topics/ui/dialogs.html.

53 Android Developers. 2012. `LayoutInflater`. [WWW]. [Referenced: 10.10.2012]. Available: http://developer.android.com/reference/android/view/LayoutInflater.html.

54 Android Developers. 2012. `View`. [WWW]. [Referenced 26.3.2012]. Available: http://developer.android.com/reference/android/view/View.html.

55 Android Developers. 2012. `FrameLayout`. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/reference/android/widget/FrameLayout.html.

56 Android Developers. 2012. `TableLayout`. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/reference/android/widget/TableLayout.html.

57 Android Developers. 2012. XML Layouts. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/guide/topics/ui/declaring-layout.html.

58 Android Developers. 2012. Custom Components. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/guide/topics/ui/custom-components.html.

59 Android Developers. 2012. Input Events. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/guide/topics/ui/ui-events.html.

60 Android Developers. 2012. Menus. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/guide/topics/ui/menus.html.

61 Android Developers. 2012. Action Bar. [WWW]. [Referenced: 8.5.2012]. Available: http://developer.android.com/guide/topics/ui/actionbar.html.

62 Android Developers. 2012. Dialogs. [WWW]. [Referenced: 7.10.2012]. Available: http://developer.android.com/guide/topics/ui/dialogs.html.

63 Android Developers. 2012. `DialogInterface`. [WWW]. [Referenced: 7.10.2012]. Available: http://developer.android.com/reference/android/content/DialogInterface.html.

64 Android Developers. 2012. Animation and Graphics Overview. [WWW]. [Referenced: 13.11..2012]. Available: http://developer.android.com/guide/topics/graphics/overview.html.

65 Android Developers. 2012. `Drawable`. [WWW]. [Referenced: 13.11.2012]. Available: http://developer.android.com/reference/android/graphics/drawable/Drawable.html.

66 Android Developers. 2012. View Animation. [WWW]. [Referenced: 22.8.2012]. Available: http://developer.android.com/guide/topics/graphics/view-animation.html.

67 Android Developers. 2012. Canvas and Drawables. [WWW]. [Referenced: 21.8.2012]. Available: http://developer.android.com/guide/topics/graphics/2d-graphics.html.

68 Fielding et al. HTTP/1.1: HTTP Message. [WWW]. [Referenced: 8.10.2012]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html.

69 Fielding et al. HTTP/1.1: Method definitions. [WWW]. [Referenced: 8.10.2012]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

70 Gravelle, R. Webreference. Comet Programming: Using Ajax to Simulate Server Push. [WWW]. [Referenced: 13.11.2012]. Available: http://www.webreference.com/programming/javascript/rg28/index.html.

71 XMPP Standards Foundation. 2010. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). [WWW]. [Referenced: 13.11.2012]. Available: http://xmpp.org/extensions/xep-0124.html.

72 Treffer, R. github. 2011. AsmackService. [WWW]. [Referenced: 12.5.2012]. Available: https://github.com/rtreffer/AsmackService.

73 stackoverflow. 2011. Android and XMPP: Currently available solutions. [WWW]. [Referenced: 12.5.2012]. Available: http://stackoverflow.com/questions/4769020/android-and-xmpp-currently-available-solutions.

74 Apache MINA. 2011. Welcome to Apache Vysper Project!. [WWW]. [Referenced: 12.5.2012]. Available: http://mina.apache.org/vysper/index.html.

75 Apache MINA. 2011. Dependecies – Apache Vysper. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/vysper/dependencies.html.

76 Hachicha, N. 2012. XMPP client with Android. [WWW]. [Referenced: 5.8.2012]. Available: http://nhachicha.wordpress.com/2012/03/14/xmpp-client-with-android/.

77 Apache MINA. 2011. Welcome to Apache MINA Project! [WWW]. [Referenced: 13.11.2012]. Available: http://mina.apache.org/.

78 Apache MINA. 2011. FAQ. [WWW]. [Referenced: 5.8.2012]. Available: https://mina.apache.org/mina/faq.html.

79 JSON. JSON: The Fat-Free Alternative to XML. [WWW]. [Referenced: 17.11.2012]. Available: http://www.json.org/xml.html.

80 Android Developers. 2012. `JSONObject`. [WWW]. [Referenced: 30.3.2012]. Available: http://developer.android.com/reference/org/json/JSONObject.html.

81 Android Developers. 2012. `JSONArray`. [WWW]. [Referenced: 4.8.2012]. Available: http://developer.android.com/reference/org/json/JSONArray.html.

82 Dalvik. 2010. Including additional `javax.*` packages in your Android App. [WWW]. [Referenced 27.3.2012]. Available: http://code.google.com/p/dalvik/wiki/JavaxPackages.

83 Mertz, D. IBM developerWorks. 2001. XML Matters: XML-RPC as object model. [WWW]. [Referenced: 20.11.2012]. Available: http://www.ibm.com/developerworks/xml/library/x-matters15/index.html.

84 Roes, T. GitHub. 2012. aXMLRPC. [WWW]. [Referenced: 30.3.2012]. Available: https://github.com/timroes/aXMLRPC.

85 Winer, D. XML-RPC.com. 2003. XML-RPC Specification. [WWW]. [Referenced: 30.3.2012]. Available: http://xmlrpc.scripting.com/spec.html.

86 Google. 2011. ksoap2-android. [WWW]. [Referenced: 30.2.2012]. Available: http://code.google.com/p/ksoap2-android/.

87 android-json-rpc. 2012. Getting Started. [WWW]. [Referenced: 13.5.2012]. Available: http://code.google.com/p/android-json-rpc/wiki/GettingStarted.

88 qooxdoo 2.0.1 documentation. 2012. RPC (Remote Procedure Call). [WWW]. [Referenced: 4.8.2012]. Available: http://manual.qooxdoo.org/current/pages/communication/rpc.html.

89 jabsorb 1.3.2 API. 2009. [ZIP/HTML]. [Referenced: 4.8.2012]. Available: http://code.google.com/p/jabsorb/downloads/detail?name=jabsorb-1.3.2-javadoc.zip&can=2&q=.

90 JSON-RPC for Java – Google Project Hosting. 2012. jsonrpc4j. [WWW]. [Referenced: 4.8.2012]. Available: http://code.google.com/p/jsonrpc4j/.

91 jsonrpc4j-0.24-sources. 2012. [JAR/Java]. [Referenced: 4.8.2012]. Available: http://jsonrpc4j.googlecode.com/svn/maven/repo/com/googlecode/jsonrpc4j/0.24/jsonrpc4j-0.24-sources.jar.

92 JSON-RPC for java 0.24 API. 2012. [JAR/HTML]. [Referenced: 4.8.2012]. Available:

http://jsonrpc4j.googlecode.com/svn/maven/repo/com/googlecode/jsonrpc4j/0.24/jsonrpc4j-0.24-javadoc.jar.

93 json-rpc - Easy to use JSON-RPC Client/Server – Google Project Hosting. 2012. Usage. [WWW]. [Referenced: 4.8.2012]. Available: http://code.google.com/p/json-rpc/wiki/Usage.

94 JSON-RPC 1.0 API. 2011. [ZIP/HTML]. [Referenced: 4.8.2012]. Available: http://code.google.com/p/json-rpc/downloads/detail?name=jsonrpc-1.0.zip&can=2&q=.

95 jsonrpc-1.0-sources. 2011. [ZIP/JAR/Java]. [Referenced: 4.8.2012]. Available: http://code.google.com/p/json-rpc/downloads/detail?name=jsonrpc-1.0.zip&can=2&q=.

96 jpoxy - Simple JSON-RPC framework for Java apps – Google Project Hosting. 2011. RPC.java. [WWW]. [Referenced: 4.8.2012]. Available: http://code.google.com/p/jpoxy/source/browse/trunk/jsonrpc/src/main/java/org/jpoxy/RPC.java.

97 GitHub. 2012. stefaniuk/com.code4ge.json.service. [WWW]. [Referenced: 4.8.2012]. Available: https://github.com/stefaniuk/com.code4ge.json.service.

98 JSON-RPC 2.0. 2012. Java Server Framework. [WWW]. [Referenced: 5.8.2012]. Available: http://software.dzhuvinov.com/json-rpc-2.0-server.html.

99 JSON-RPC 2.0. 2012. Base Java Classes. [WWW]. [Referenced: 5.8.2012]. Available: http://software.dzhuvinov.com/json-rpc-2.0-base.html.

100 Java-json-rpc library. [WWW]. [Referenced: 5.8.2012]. Available: http://java-json-rpc.sourceforge.net/.

101 json rpc on java. - Google Project Hosting. 2010. libjsonrpc. Available: http://code.google.com/p/libjsonrpc/.

102 Google Project Hosting. 2011. UsageGuide – simplejsonrpc. [WWW]. [Referenced: 5.8.2012]. Available: http://code.google.com/p/simplejsonrpc/wiki/UsageGuide.

103 MySQL. 2012. Download Connector/J. [WWW]. [Referenced: 6.8.2012]. Available: http://dev.mysql.com/downloads/connector/j/.

104 Commons Logging. 2008. Overview. [WWW]. [Referenced: 6.8.2012]. Available: http://commons.apache.org/logging/.

105 SLF4J. 2012. [WWW]. [Referenced: 6.8.2012]. Available: http://www.slf4j.org/.

106 Google Project Hosting. 2012. Downloads – jabsorb. [WWW]. [Referenced: 6.8.2012]. Available: http://code.google.com/p/jabsorb/downloads/list.

107 Apache HttpComponents. 2012. HttpComponents Downloads. [WWW]. [Referenced: 6.8.2012]. Available: http://hc.apache.org/downloads.cgi.

108 JAR Search. 2010. `servlet-api-2.5-6.1.6.jar`. [WWW]. [Referenced: 6.8.2012]. Available: http://www.findjar.com/jar/org/mortbay/jetty/servlet-api-2.5/6.1.6/servlet-api-2.5-6.1.6.jar.html.

109 Horton, I. Ivor Horton's Beginning Java™ 2, JDK™ 5 Edition. 2005. Wiley Publishing Inc. 1470 p.

110 MySQL 5.0 Reference Manual. 2012. [WWW]. [Referenced 18.2.2012]. Available: http://dev.mysql.com/doc/refman/5.0/en/.

111 JavaTM Platform, Standard Edition 7 API Specification. 2012. `Connection`. [WWW]. [Referenced: 6.8.2012]. Available: http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html.

112 Apache MINA. 2011. MINA based Application Architecture. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/mina-based-application-architecture.html.

113 Apache MINA. 2011. Server Architecture. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/server-architecture.html.

114 Apache MINA. 2011. Client Architecture. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/client-architecture.html.

115 Apache MINA. 2011. Sample TCP Server. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/sample-tcp-server.html.

116 Apache MINA. 2011. Sample TCP Client. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/sample-tcp-client.html.

117 Apache MINA. 2011. Chapter 7 – Handler. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/chapter-7-handler.html.

118 Apache MINA. 2011. Chapter 8 - Acceptor. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/chapter-8-acceptor.html.

119 Apache MINA 2.0.4 API Documentation. 2011. `NioSocketAcceptor`. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/report/trunk/apidocs/index.html?org/apache/mina/transport/socket/nio/NioSocketAcceptor.html.

120 Apache MINA 2.0.4 API Documentation. 2011. `NioSocketConnector`. [WWW]. [Referenced: 5.8.2012]. Available: http://mina.apache.org/report/trunk/apidocs/index.html?org/apache/mina/transport/socket/nio/NioSocketConnector.html.

121 jabsorb 1.3.2 source code. 2009. [ZIP/Java] [Referenced: 6.8.2012]. Available: http://code.google.com/p/jabsorb/downloads/detail?name=jabsorb-1.3.2-src.zip&can=2&q=.

122 Android Developers. 2012. Animation Resources. [WWW]. [Referenced: 27.8.2012]. Available: http://developer.android.com/guide/topics/resources/animation-resource.html.

123 Android Developers. 2012. `Canvas`. [WWW]. [Referenced: 27.8.2012]. Available: http://developer.android.com/reference/android/graphics/Canvas.html.

124 Android Compatibility Program. 2012. Android 4.1 Compatibility Definition. Revision 1. [PDF]. [Referenced: 7.8.2012]. Available:

http://source.android.com/compatibility/4.1/android-4.1-cdd.pdf.

125 Android Compatibility Program. 2010. Android 2.3 Compatibility Definition. [PDF]. [Referenced: 7.8.2012]. Available: http://source.android.com/compatibility/2.3/android-2.3.3-cdd.pdf.

126 Android Developers. 2012. Tasks and Back Stack. [WWW]. [Referenced: 7.8.2012]. Available: http://developer.android.com/guide/components/tasks-and-back-stack.html,

127 Android Developers. 2012. `<activity>`. [WWW]. [Referenced: 7.8.2012]. Available: http://developer.android.com/guide/topics/manifest/activity-element.html.