Empires of Avatharia

Group 22

Christopher Raschke
Johan Wessman
Jashar Ghavampour
Johan Granberg

REQUIREMENTS DOCUMENT

Version 1

1. Introduction

This document was written as a part of the course ”Mjukvarukonstruktion” on KTH in 2008. It is meant to be read by the course examiners as well as other students taking the course. The document is a Design Description for the “Empires of Avatharia” project.

We would like recommend the readers of this document to read the Requirements Document for the “Empires of Avatharia” project before reading this. This document specifies what the SuD will contain and what will be left out. It contains some the use of the system as well as user scenarios.

The “Implementation Plan” document for the SuD is a result of this document. This document contains information about how the work shall be carried out.

This document contains information about the design of the SuD on a abstract level. It contains information about the relationships between the different classes but doesn’t include any code. There is a brief description of every class and every method in the SuD as well as diagrams of the database, classes etc. This document is meant to give an understanding of how the system will work without specific details of how it is going to be implemented.

2.1 General Description

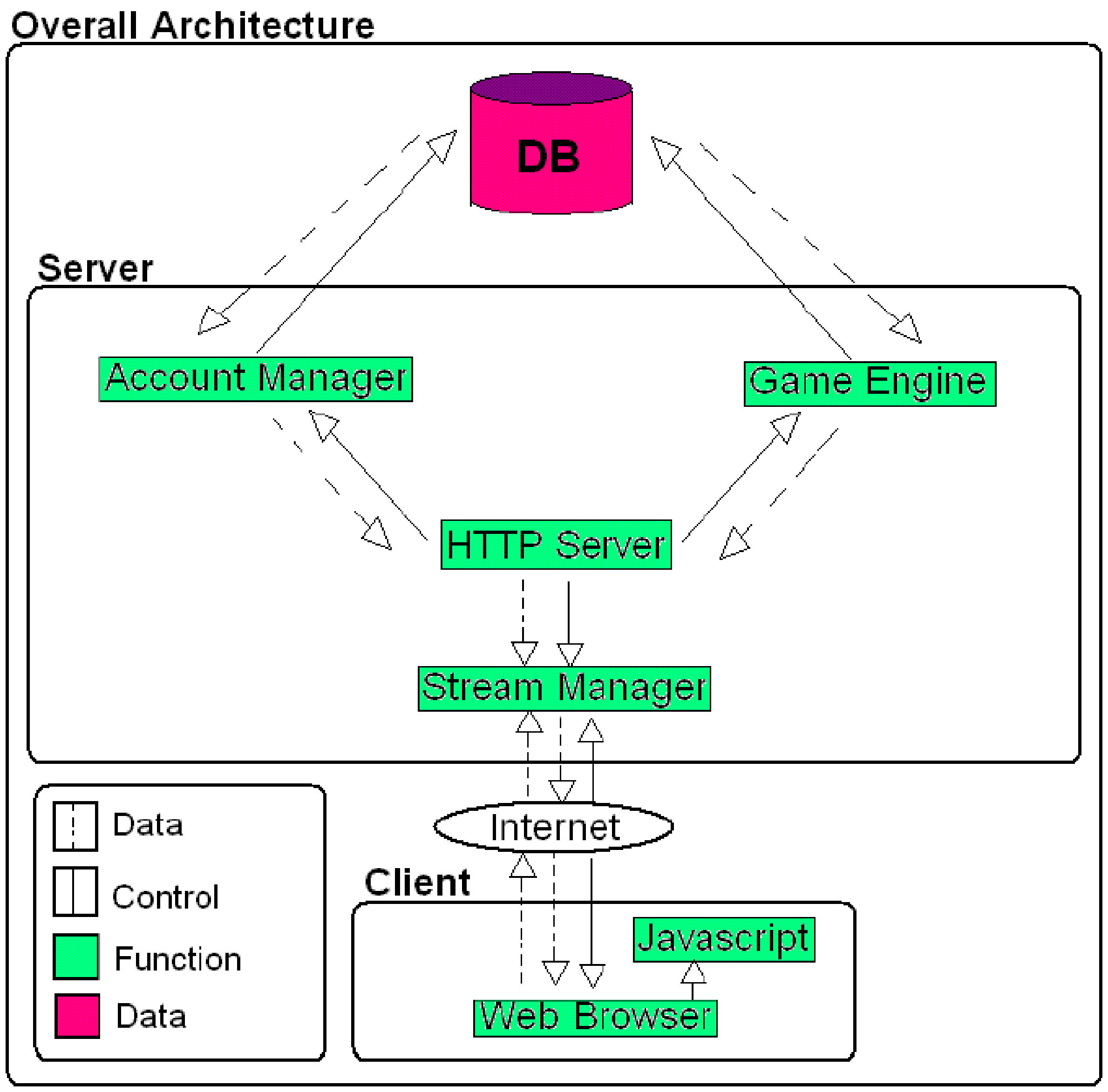
Our system is an online game with registered users as players, and it will be played in a standard web browser. The basic functionality for the user is managing his own empire and armies and sending them on missions to conquer new lands.

To make this work the system will use a database for game information storage, a HTTP-server for communication and a game engine itself running on the server.

The database is supposed to be reachable from all the parts of the system so that everything can contact the database when appropriate.
The game engine will be designed with a “main loop” going through “events” in the systems and making the necessary changes to the database.

All of these parts are explained in greater detail in the following sections.

2.2 Overall Architecture Description



All calculations in the "Empires of Avatharia" game takes place on the server side. The projects overall goal is to develop this side. We assume that the client have the necessary hardware and software installed in his or her computer to access our HTTP Server.
The client will therefore not run any code on his computer except possibly some javascripts. Please keep in mind that the arrows coming out of the "Internet" are directly connected to the client. This means that the program is not controlled by the internet but the clients. The internet only works as a "data and control"-transportation medium in this picture.

The client's web browser connects to the HTTP server via the Stream Manager. The Stream Manager is responsible for transmitting data and receiving requests for data from clients. The HTTP server sends the requested HTML files and images that the client has requested via the Stream Manager. The HTTP Server is also responsible for knowing which clients that are logged in and which level of privileges that the different clients has. It handles login requests by forwarding them to the Account Manager.

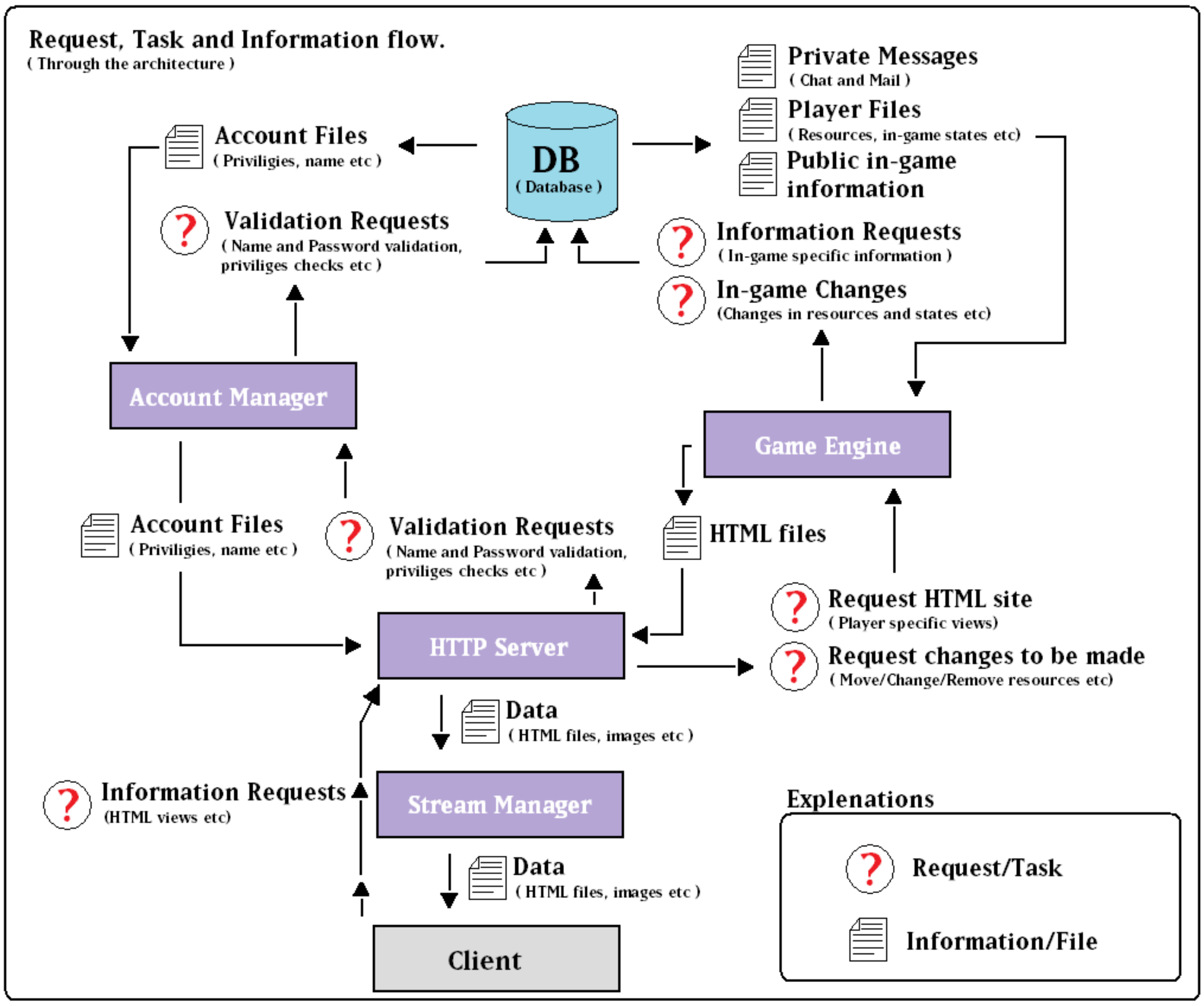
The Account Manager matches the information with the information in the database (DB) and sends a reply containing information about the user and his privileges to the Account Manager (if the information matches). The account manager then forwards this information to the HTTP Sever.

When a client is logged in the HTTP Server authorizes commands that lets the client use the Game Engine. When the client requests data that is player specific the HTTP Server needs to forward the request to the Game Engine who calculates, gathers data from the database and eventually compiles all the information into a HTML file that it sends back to the HTTP Server. The HTTP Server may also send requests to change player specific states or resources. The Game Engine checks this information, formulates a database command and last of all forwards the command to the database (if the request was considered valid). These requests come from the client originally. The Game Engine may also include JavaScript in the HTML files that it generates.

The database contains all in-game information as well as account information and pictures of the units, buildings, provinces etc.

Note that the Game Engine contains many classes but it is represented as one entity in the picture. For more detailed information about the incorporated classes see 5.2.

2.3 Detailed Architecture



Request, Task and Information flow.
(Through the architecture)
Private Messages
(Chat and Mail)
Player Files
(Resources, in-game states etc)
Public in-game information
DB
(Database)
Account Files
(Priviligies, name etc)
Validation Requests
(Name and Password validation, priviliges checks etc)
Information Requests
(In-game specific information)
In-game Changes
(Changes in resources and states etc)
Account Manager
Game Engine
Account Files
(Priviligies, name etc)
Validation Requests
(Name and Password validation, priviliges checks etc)
HTML files
Request HTML site
(Player specific views)
Request changes to be made
(Move/Change/Remove resources etc)
HTTP Server
Data
(HTML files, images etc)
Information Requests
(HTML views etc)
Stream Manager
Data
(HTML files, images etc)
Client
Explanations
Request/Task
Information/File

The HTTP Server

Main use: Act as a HTTP server. Act as a “data forwarding”-medium between the game engine and the client.
Detailed use: The HTTP-server forwards all the requests that need to be performed and changed in the game world to the Game Engine who then make the appropriate changes in the database.
Besides that the HTTP-server also requests new HTML-files that need to be sent back to the client after a specific change or update to the game world has occurred.

The Game Engine

Main use: Making all calculations needed for the game to function, to be the “game engine”.

Detailed use: When executing tasks like updating the game world the Game Engine first makes all necessary calculations and then forwards what needs to be changed to the database. This happens typically after a battle has been calculated in the Game engine or a new building has been built or movement has been made by a client. All of these need to be updated by the game engine, and in so doing sometimes fetching information from the database that is needed to complete the task, (e.g. the Game Engine needs information about armies from the database when calculating a battle). Besides fetching information about armies and villages the Game Engine also fetches private messages and the like from the database when a user wishes to view it.

Stream Manager

Main use: Handle input streams from clients. Handle output streams for the HTTP server.
Detailed use: The client sends an HTTP request via his or her webbrowser to the stream manager. The stream just forwards all requests and sendings. All requests are forwarded to the HTTP server, which sends the input data to the game engine. The engine processes the input data and returns it in HTML form to the HTTP server. This data, along with other data such as images, is sent to the stream manager, which in turn sends the HTML to the client.

Account Manager

Main use: Handle privilege questions from the HTTP Server
Detailed use: The Account Manager handles everything that has with account security to do. It’s main purpose is verifying the username and password of a user. This request is sent by the HTTP Server when the client tries to log in to the in-game environment. It is also responsible for accessing and forwarding player’s account files to the HTTP server so that this information can be accessed both by the administrator and the player.
If a client wishes to access a restricted page via the HTTP Server then the Account Manager will deny or grant access depending on the players privileges.

Note that the javascripts are not included in this picture since they are generated by a HTML generator inside the Game Engine. It is included in the “overall architecture” under the client since the client “runs the code”.

3. Design Considerations

These issues are also mentioned in section 2 of the Empires of Avatharia requirements document.

3.1 Assumptions and Dependencies

Empires of Avatharia is an online game played using a web browser, and is therefore available to many different platforms. This also makes it lightweight in terms of required capacity in the computers on both the server and on the client side.

The server that runs the game is required to support Java Servlets and the PostgreSQL database system. It is important to keep backups of all game data to keep player accounts updated and to prevent players from having to start all over again from nothing, as this will have a heavy negative impact on the game experience. Some downtime is acceptable as long as game data is not lost.

3.1 General Constraints

The foundation for the game's interoperability is built into the internet infrastructure, but there are differences that need to be considered. The greatest issue is that of differences between browsers in rendering the content. This problem requires the game to be tested thoroughly in different browsers throughout the development stage to make sure that the game looks more or less the same for all users.

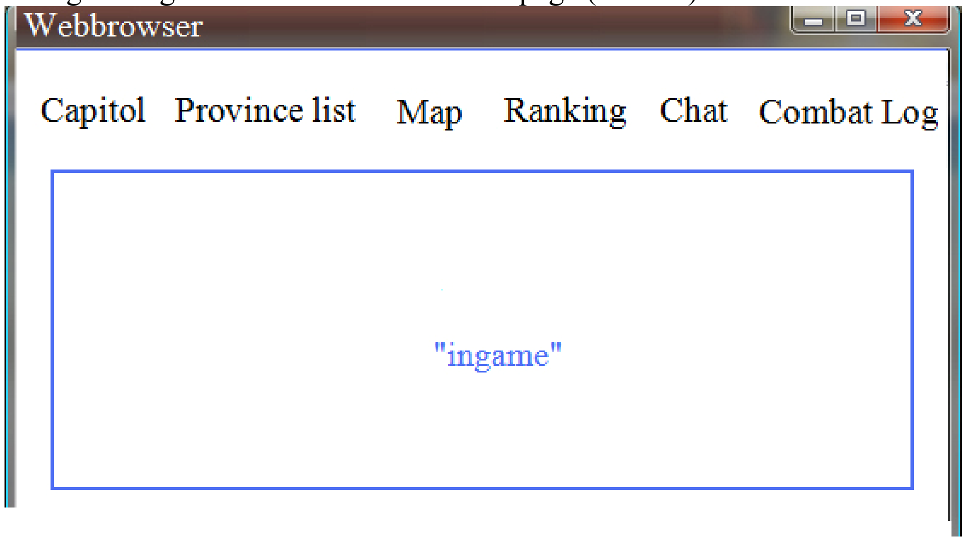
Apart from technical issues, there is also a challenge in building a game that is well balanced and fun to play. The game is designed in such a way that it is easy for the game providers to tweak different aspects of the game (unit properties, time limits and such) if it is required.

A common problem to games is that of preventing cheating. It is crucial to validate all input from a player on the server side and not only on the client side – otherwise it may be easy to send random input and gain advantages or change the game in other unintended ways. It is also important to keep user integrity as intact as possible by considering the account validation issues throughout development.

4. Graphical User Interface

4.1 General overview of the user interface

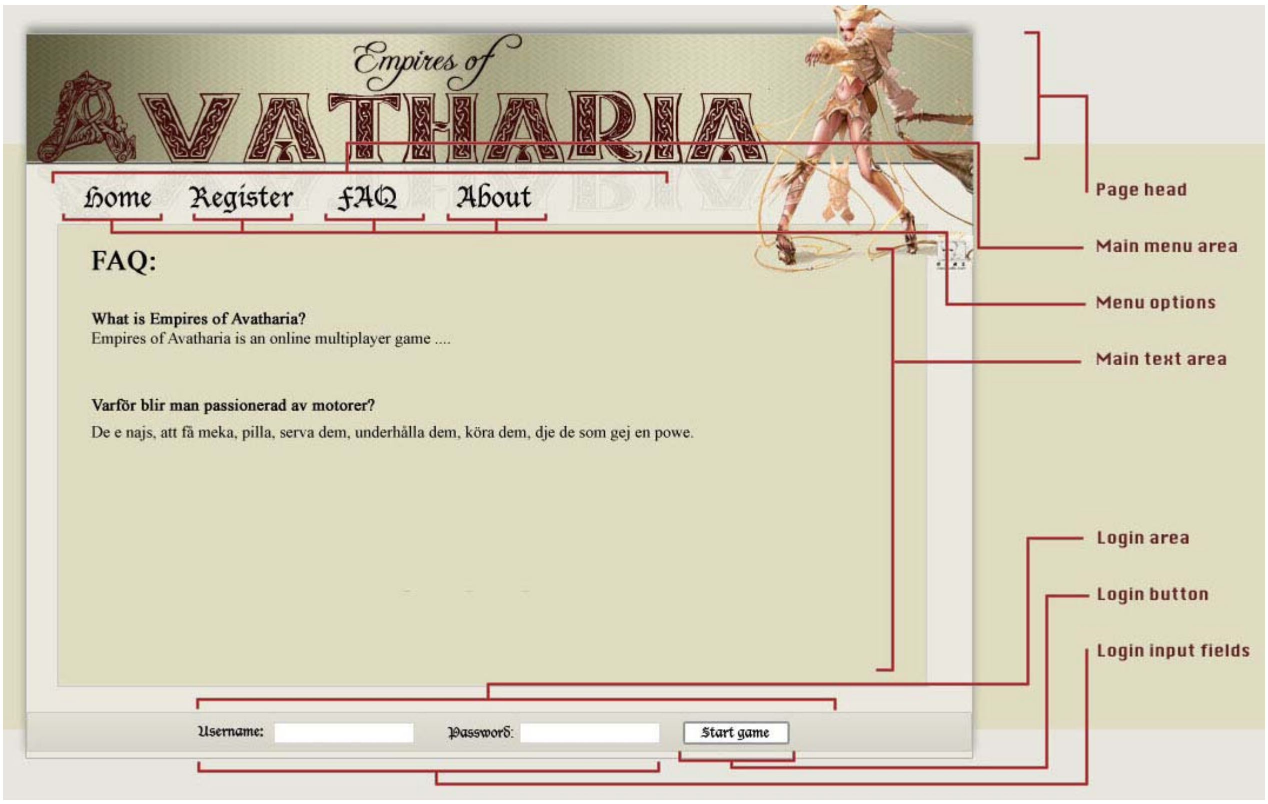
In order to access any of the game material the user has to “log in”. The user can “log in” by accessing the login-screen on the SuD’s webpage (See 4.2)



Pic 4.1 Showing the orientation of the ”tabs”

When the user has logged his main view will display his “Capitol”-tab by default. The user will be able to access any other tab at any time by clicking on these tabs that will always be positioned at the top of the gui/webpage. All these tabs will be explained in detail in the sections below. For future reference these tabs will be called the “Capitol”-tab, the “Province list”-tab, the “Map”-tab and so on. Active tabs will have their links underlined.

4.2 General (Website interface) / Login



FAQ:

What is Empires of Avatharia?
Empires of Avatharia is an online multiplayer game ....

Varför blir man passionerad av motorer?
De e najs, att få meka, pilla, serva dem, underhålla dem, köra dem, dje de som gej en powe.

Pic 4.2 Showing the index of the website when the user is not yet logged in.

This is the interface of the web site that the user comes to when coming to the game site. From here new users can find information about the game and register for an account to be able to play. Existing users can type in the login data (username and password) received when registering to start playing.

Site navigation
The site is navigated by clicking the options in the main menu area. Each option will display corresponding information in the main text area to the user.

Page head
This area contains the game logo and design elements. When clicking anywhere on the page head, the user will come to the ”Home” area.

Main menu area
All main site navigation controls will be found here.

Menu options
Items in the menu, these are clickable.

Main text area
This area contains the contents (such as images or text) that correspond to the menu option selected by the user. If these contents are too big to fit in the main text area, the browser is to display scrollbars that enable the user to view all contents without affecting the height or width of the main site.

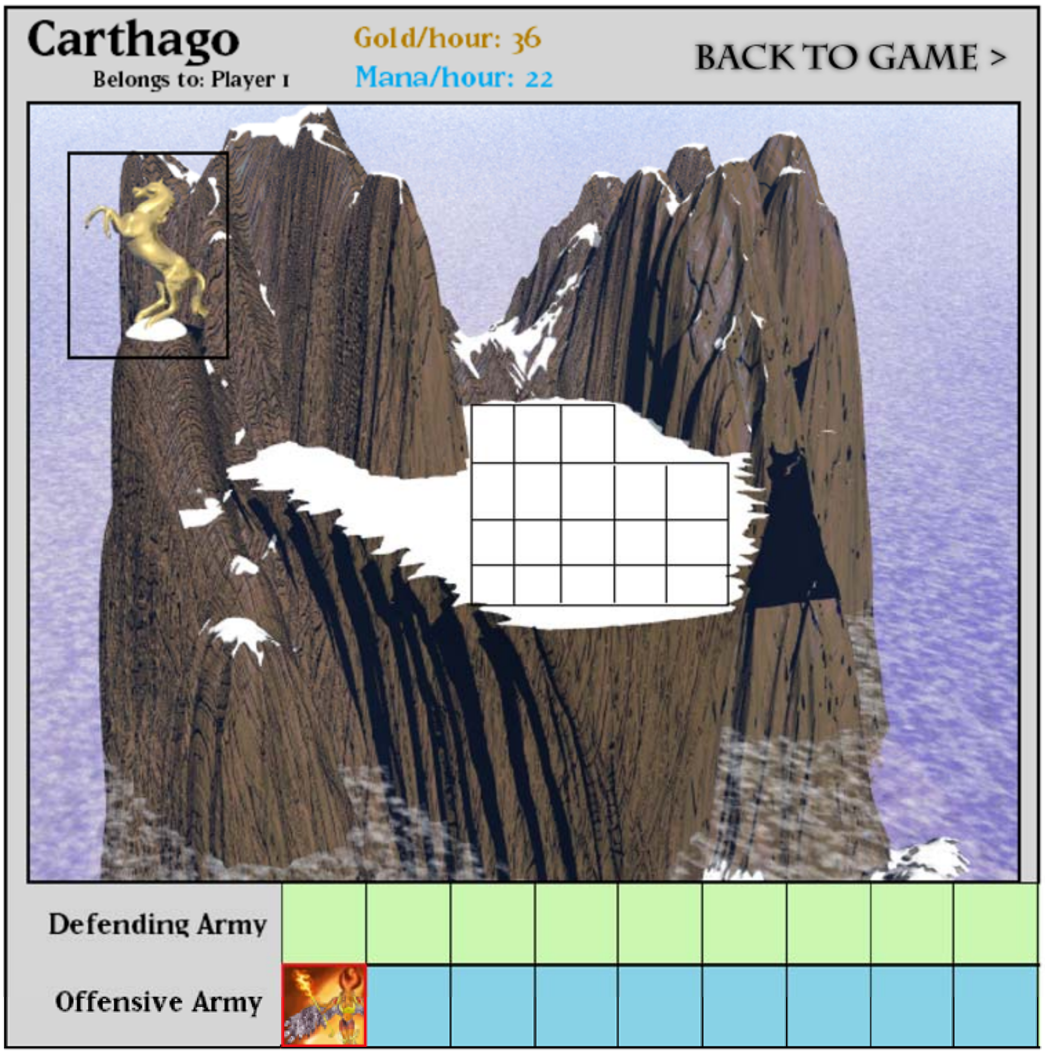
Login area
This area contains the form that users with accounts must use in order to start playing the game.

Login input fields
The username is unique for every user and is written out in plain text when the user types it into the input field. The password field will be set to not display the characters in plain text. The name of the username field is “username” and the name of the password field is “password”.

Login button
When this button is clicked the user data will be sent to the application for authorization. If the user has a correct username and password, a new browser window will open with the game interface. If the login information is incorrect, the user will be informed by information that will be displayed in the main text area when the application has denied access to the game. The name of this button is “submitLoginData” and its value is “Start Game”.

4.3 The province view

To access the province view the user has to select either the ”Capitol”-tab or select a province from the “province list”-tab, the user also has to be logged in. The “province view” displays a graphical representation of a province. Here the user can make a lot of choices concerning the province. Here is a rough sketch of how the view might look like:



Pic 4.3 The “Province view”

The two most important aspects here are the representation of the buildings and the units. The buildings will either appear in the grid (center of the picture) or somewhere around it (as the framed golden horse to the left). The player can (often) select where to put his buildings, when he builds one, by selecting the desired location in the grid.
Units are always stationed in an army, which is either the defending or the defensive one.
Options

Units:

- By clicking on the offensive army the player can send this army to other provinces. A menu is shown with the appropriate options concerning this movement.
- The user can get more information of a unit by clicking on it.
- The user can move units between the two armies by a “drag & drop”-like function.
- Train units by clicking on the appropriate building and then selecting the unit that he wants to train.

Buildings:
- By clicking on a building the user will get more information about it and sometimes be able to build units.
- Clicking on an empty grid the player will get the option to build a building there. He will be presented with a list with all the possible options of buildings.

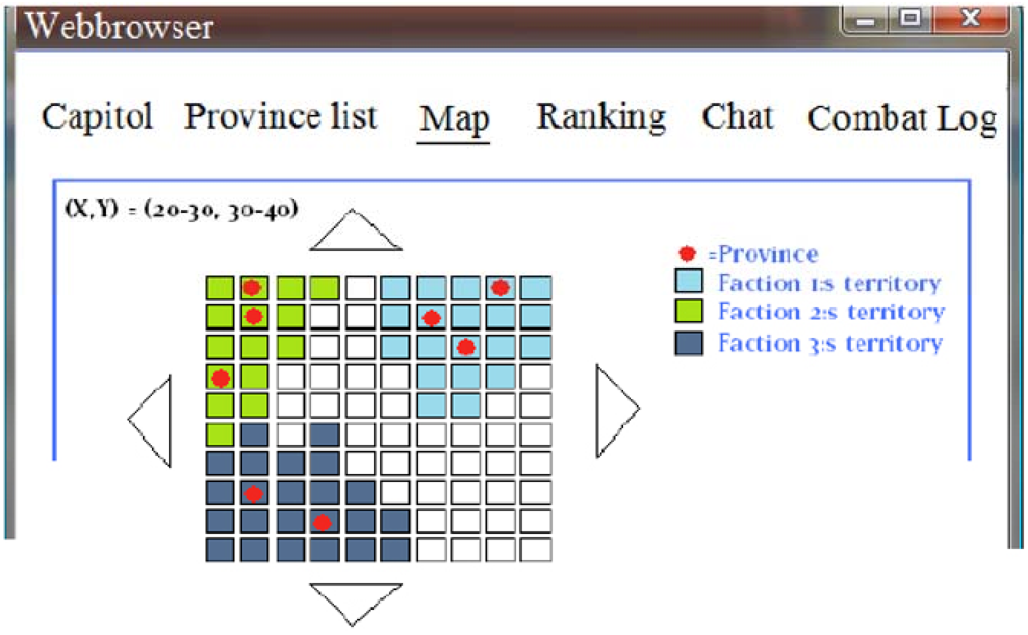
Each unit and each building is a clickable field and named after there position in the grid. Ex: “Offensive_army_spot1” or “Building_grid_x2_y1”.

The “gold and mana” (on the top of the picture) fields are updated each hour by the “Game Engine”. These fields are simply called “gold” and “mana”.

The user can return to the main interface by clicking the “Back to game” area in the top right corner.

4.4 The “Map”-tab

The map provides a general overview of a certain part of the ingame map, revealing provinces and the different factions territory. It is also a tool that let the user easily enter a region of the ingame world.



Pic 4.4 The “Map”-tab

There is a field at the top left of the screen named ”map_coordinates” which displays the current viewing coordinates to the user.

There is a field to the right of the screen named “information” displaying information of the map to the user.

The arrows are used to navigate through the map by changing the current coordinates of the top left corner of the grid:

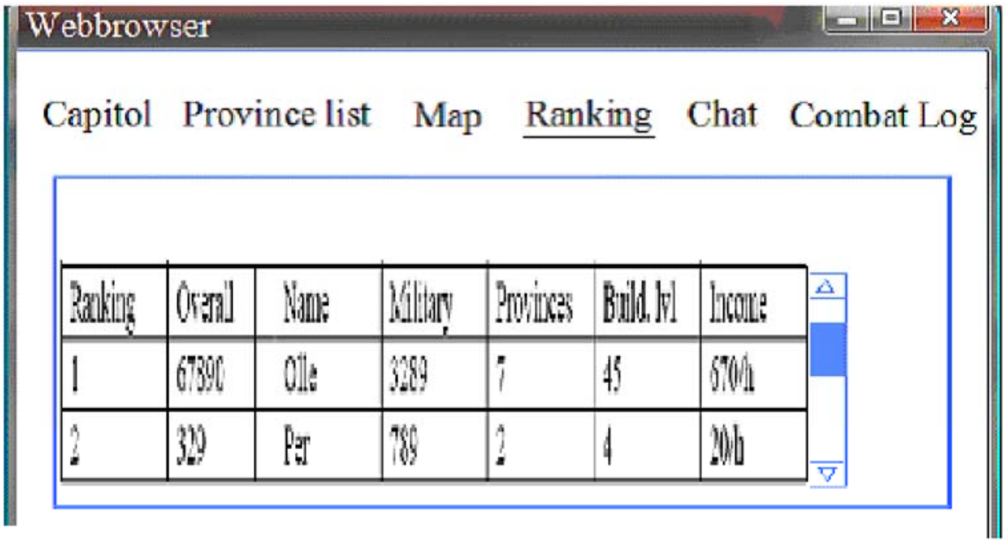
- At the top of the screen there is a control button named: map_goNorth_button map_goNorth_button can call one method: goNorth(current coordinates).
- To the left of the screen there is a controll button named: map_goWest_button map_goWest_button can call one method: goWest(current coordinates).
- At the bottom of the screen there is a controll button named: map_goSouth_button map_goSouth_button can call one method: goSouth(current coordinates).
- To the right of the screen there is a controll button named: map_goEast_button map_goEast_button can call one method: goEast(current coordinates).

There are also several controls which reveal the map to the user, these are named: region_#insert region coordinates here#.
region_#insert region coordinates here# can call one method: enter_region(#insert region coordinates here#).

There are two methods that will couse this form to be revealed to the user:
“generaltabs(Map)”.
“map_show(Coordinates)”

4.5 The “Ranking”-tab

The ranking provides a calculated measure of the players relative power in the ingame world and displays it to the user.



Ranking	Overall	Name	Military	Provinces	Build. lvl	Income
1	67390	Olle	3289	7	45	670/h
2	329	Per	789	2	4	20/h

Pic 4.5 The “Ranking”-tab

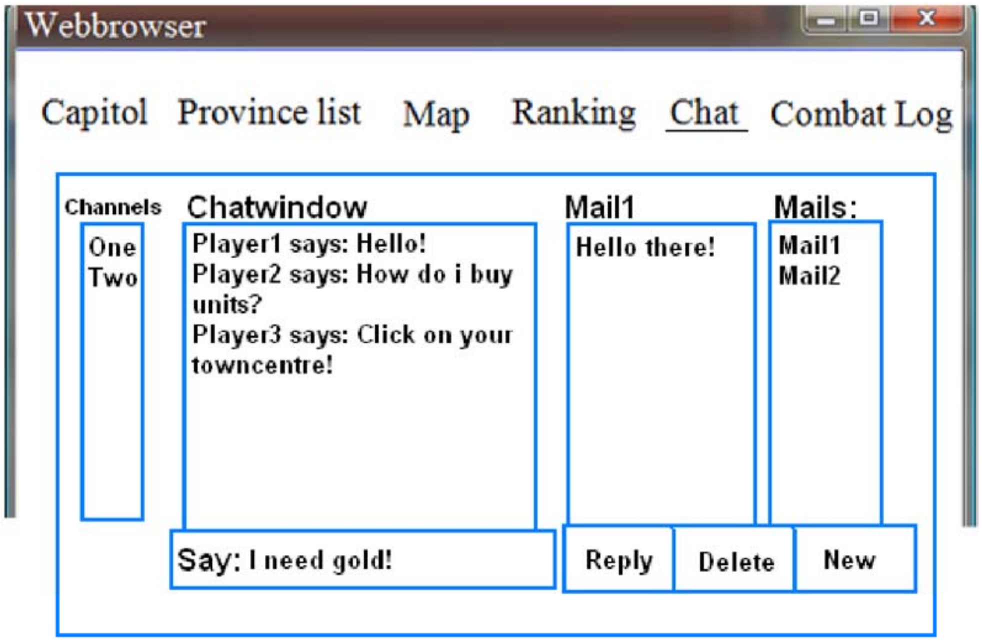
There is a scroll-list in the middle of the screen named: ”ranking_scroller”.
ranking_scroller can call two methods: scroller_up() and scroller_down().

There are several fields inside the scroll list, they are named:
“ranking_#rank#_#overall#_#name#_#military#_#provinces#_#build.lvl#_#income#”.

There is one method that will cause this form to be revealed to the user:
“generaltabs(ranking)”.

4.6 The “Chat”-tab

The in game chat is divided into two parts; the instant message chat and the private message chat. As seen in the picture below both parts appear on the same screen.



Player1 says: Hello!
Player2 says: How do i buy units?
Player3 says: Click on your towncentre!

Hello there!

Say: I need gold!

Pic 4.6 The “Chat”-tab

The instant chat will be constructed with a chat window where the actual messages appear and under it a field where the text to be sent is entered.
To the left of the chat window there is a channel control. Both the say field and channel field are control field which controls how the chat works, when new things should be displayed etc. The channel field changes which channel is used to talk in and the say field is used to send the text in it, when the ‘Enter’-key is pressed, to the chat window.

The other part which is used to manage private messages is divided into a display field, a control field and three buttons. The control field is used to choose the message to be displayed. The three buttons are ‘Reply’, ‘Delete’ and ‘New’. When a message is chosen and displayed the ‘Reply’- and ‘Delete’-buttons can be used to reply to the user who sent the message or to delete the chosen message.

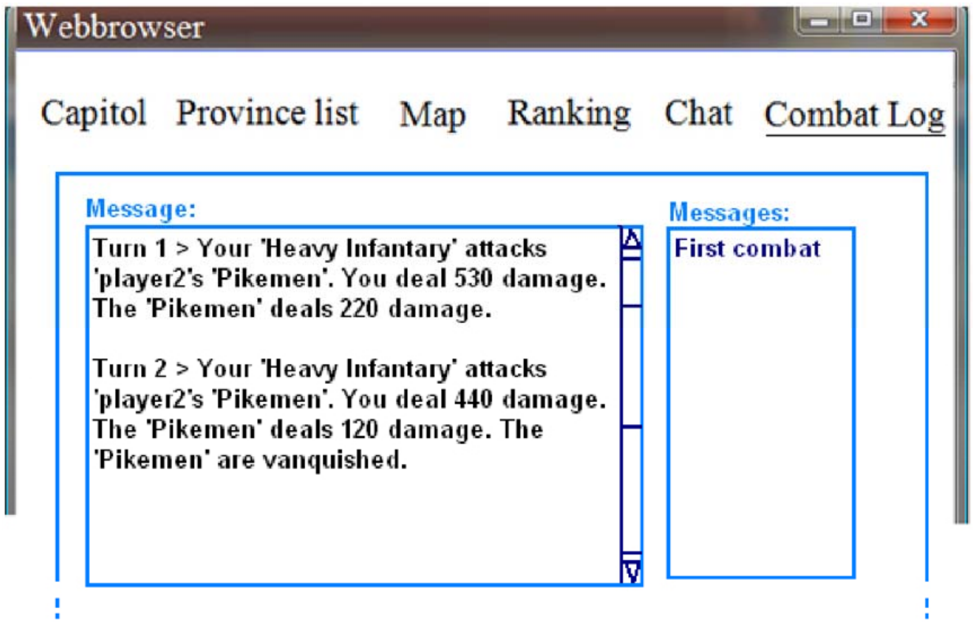
The third button helps to create a new message to be sent.

To enter the in game chat section the user simply presses the ‘Chat’-button in the game menu at the top of the user interface.

4.7 The “Combat log”-tab

The combat log screen is divided into two parts; the right control field with different combat logs and the left display field which displays the chosen combat log.

First a specific log is chosen in the right field and that specific log is then displayed in the left field. The only control on this screen is the field to the right as described.



Turn 1 > Your 'Heavy Infantary' attacks 'player2's 'Pikemen'. You deal 530 damage. The 'Pikemen' deals 220 damage.

Turn 2 > Your 'Heavy Infantary' attacks 'player2's 'Pikemen'. You deal 440 damage. The 'Pikemen' deals 120 damage. The 'Pikemen' are vanquished.

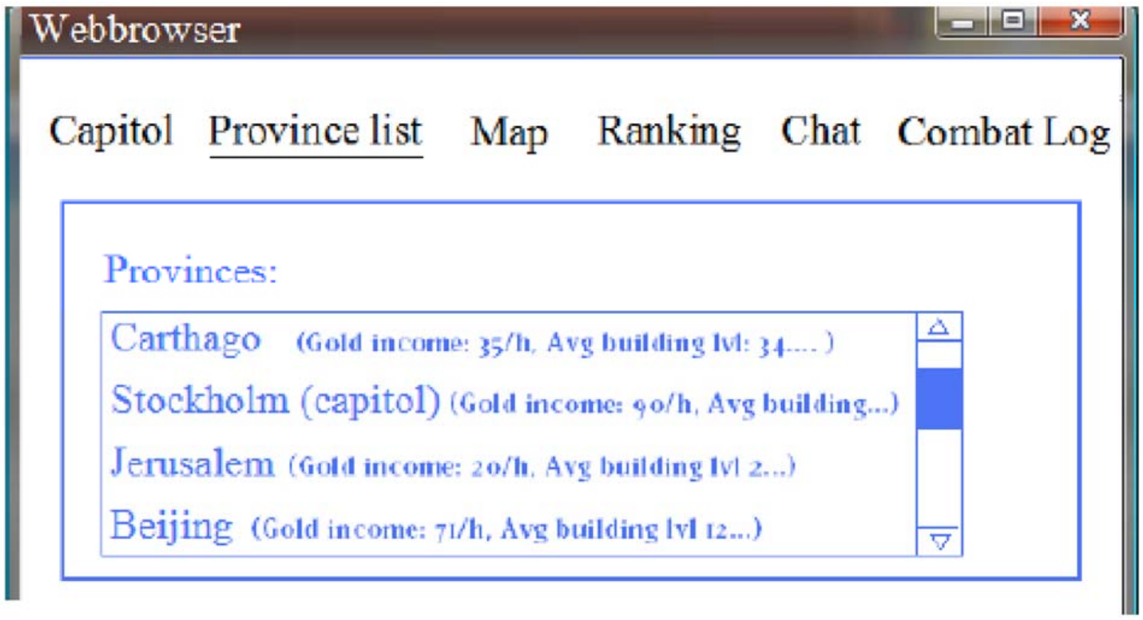
Pic 4.7 The “Combat log”-tab

The report format of the log will be on the format of turns. Each turn will be written out as how much damage a players troops did to the other players troops under that turn and vice versa, and the report contains all the turns in small text blocks until the combat is over.

The combat log is accessed by pressing the ‘Combat Log’-button on the main menu.

4.8 The “Province list”-tab

The provincelist shows the user a scroll-list of his or her provinces and relevant information concerning the provinces.
He or she can use the list to pick a province to “enter”. This will send the user to the “Province View” (see 4.3).



Carthago (Gold income: 35/h, Avg building lvl: 34....)
Stockholm (capitol) (Gold income: 90/h, Avg building...)
Jerusalem (Gold income: 20/h, Avg building lvl 2...)
Beijing (Gold income: 71/h, Avg building lvl 12...)

Pic 4.8 The “Province list”-tab

There is a scrollist is named ”Provincelist_scroller”.
Provincelist_scroller can call two methods: scroller_up() and scroller_down().

There are also several controls inside the scroll list, these are named “provinceListEntry” provinceListEntry can call one method: enterProvince(#incert provincename here#).

There is one method that will cause this form to be revealed to the user: “generaltabs(Province list)”.

5. Design Details

5.1 CRC Cards

Class GameEngine

Contains all the algorithms that are needed for the game to work. In a sense this class *is* the game

Collaborators:

All in-game classes
(see 5.2)
Startup
HTML Generator
DB

Class Startup

Contains all the startup values that are needed for a new “game session” to be created. This class accesses predefined startup variables and creates necessary instances of different in-game classes. *Note that these “sessions” are often many weeks or even months long.*

Collaborators:

All in-game classes
(see 5.2)
GameEngine

Class HTML Generator

Generates the HTML pages that eventually end up at the client. It handles the requests that are sent from the HTML server and forwarded to it via the GameEngine. It contains information about how to get pictures etc from the database as well as HTML generating methods.

Collaborators:

All in-game classes
(see 5.2)
DB

Class EventQueue

Acts like a normal queue, but for all the events in the system. When an action is performed it is added to the queue and processed in order by the GameEngine.

Collaborators:

GameEngine

Class Event

Is used as a wrapper/container for the events queued in the EventQueue.

Collaborators:

EventQueue

Class Province
A players owns provinces.
Each province contains Buildings.
A province is placed in an area, (specific coordinates)
A province contains two armies

Collaborators:
Player
Building
Area
Army

Class Area
An area is basicly square coordinates of the world-map. An area can contain a province owned by a player.

Collaborators:
Province

Class Building
The building class contains properties about a building and what units and upgrades can be bought if a player owns that kind of building.
A province contains buildings.

Collaborators:
None

Class Upgrade
An upgrade contains some specific property for either a unit or a building that is gained when it is bought.
An upgrade also contains a cost.

Collaborators:
Unit
Building

Class Unit
The class's main use is to store unit specific data.
Store unit upgrades

Collaborators:
Army
Upgrades

Class UnitShell
The class' main use is to store unit specific data.
Store unit upgrades.

Collaborators:
Unit
Army

Class Player
Store information about the players armies.
Store information about the players Provinces
Store the player's faction.
Store other player specific information.

Collaborators:
Faction
Province

Class Army
Collect units and group them into an army
Store information about the army's current mission.

Collaborators:
Unit
Player
Sides

Class Sides
Collect armies attacking and defending the same target and sort them into two sides

Collaborators:
Army
Combat

Class Faction
Store information regarding class features.
Store information regarding class game stats.

Collaborators:
Player

Class Combat
Calculate combat results
Report combat results to CombatLog

Collaborators:
Sides
CombatLog

Class CombatLog
Generate a combatlog for the players to read.

Collaborators:
Combat

Class User
Handles login procedure, sessions, user permissions and user information.

Collaborators:
Player
Db

Class Db
Handles database connection

Collaborators:
User

Class AccountManager
Manages users, handling their authentication and construction as well as setting up and checking their permissions.

Collaborators:
User

Class HTTPServer
Acts as an interface between the application and the rest of the system (web server, file system), creating an abstraction level for some logic.

Collaborators:
StreamManager
GameEngine

Class StreamManager
Manages communication between the player and the application on the server, handling input data and sending output data.

Collaborators:
HTTPServer
GameEngine

Class Messages
Handles the chat and email messages in the application.

Collaborators:
User
Channel
Email

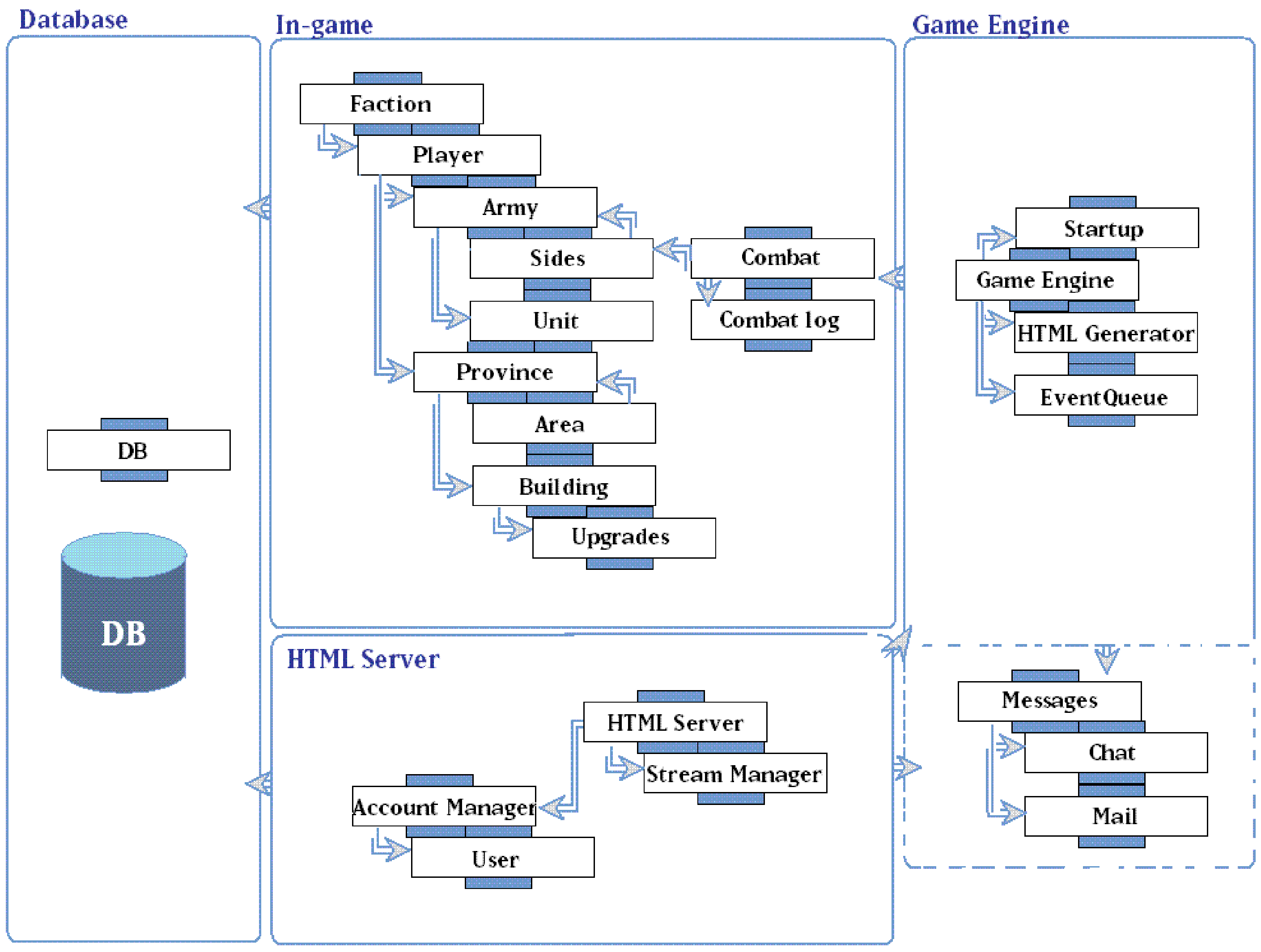
Class Channel
Stores data associated with the chat function.
Players use channels when talking in the chat.

Collaborators:
Messages

Class Mail
Stores the data associated with the chat function.
Used for sending private messages between players.

Collaborators:
Messages

5.2 Class Diagram



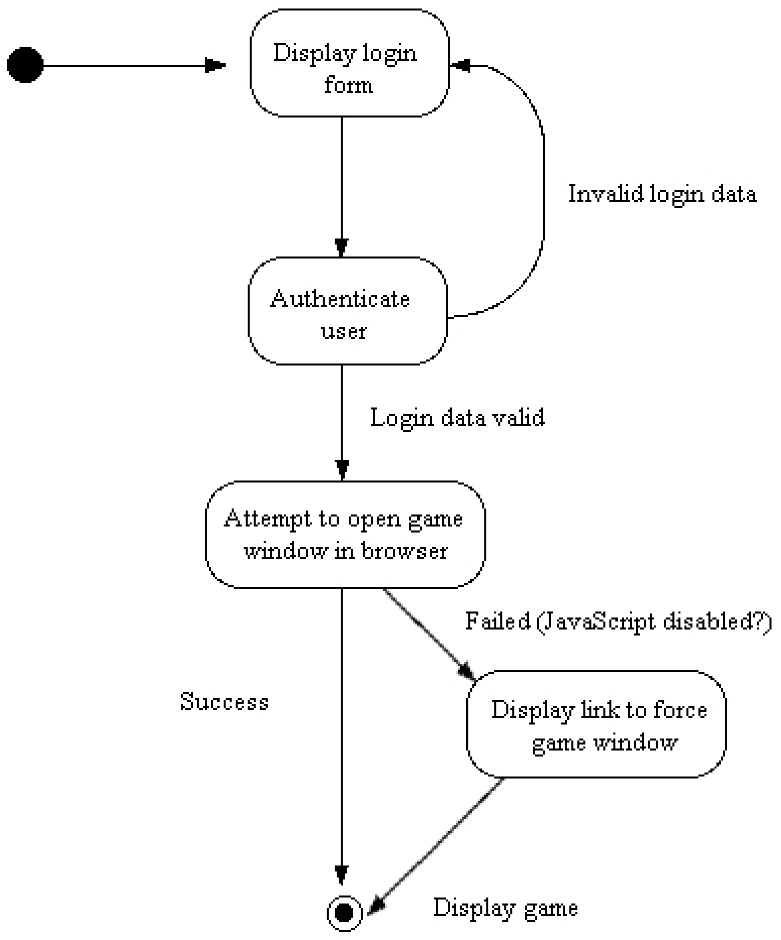
The class diagram should be interpreted as:

- Each box with a name on it is a class
- An arrow between classes basically means "contains". For example a "faction" contains a lot of "players" and a "player" has "armies" but the when the player is "in combat" the "sides" also contains "armies".
- Boxes surrounding classes' means that they belong to the same group. Classes in the same group are strongly connected and are more likely to appear in the same "folder"/"level of architecture" on the server.
- The arrow between the "boxes" indicates how requests are sent between the different groups. These arrows do represent arrows from all classes in the group. The reverse of these arrows could be interpreted as responses or data transfers.

Example: all classes in the In-game group need to be able to send requests to the database. If the Game Engine wants to generate a HTML page, via the HTML generator, containing “the province view” it will send a request to the province to send pictures of all its buildings. The province will then send this request forward to all its buildings who will then ask the “DB” class for the pictures from the database.

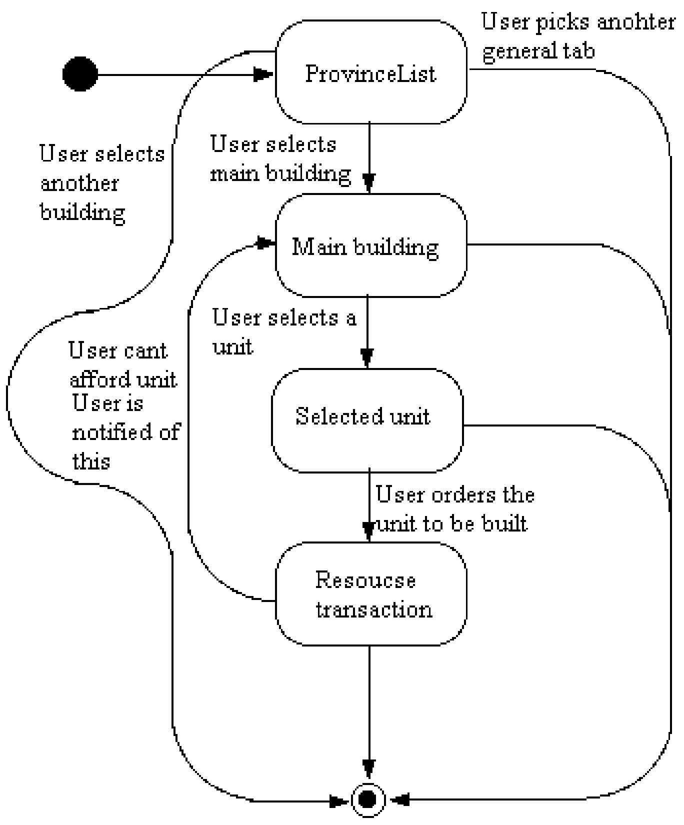
5.3 State charts

State Chart, Logging in


Display login form
Invalid login data
Authenticate user
Login data valid
Attempt to open game window in browser
Failed (JavaScript disabled?)
Success
Display link to force game window
Display game

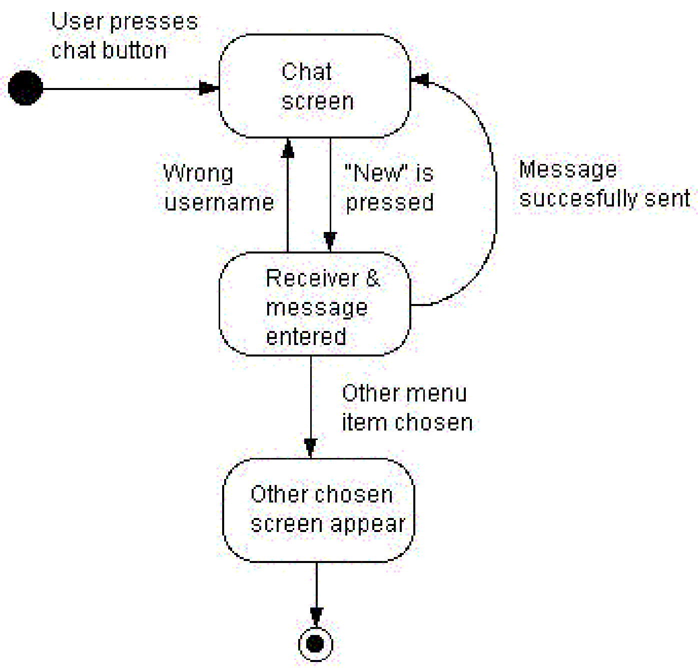
State Chart, Training a unit

It is assumed that the player is logged in into the game.


ProvinceList
User picks anohter general tab
User selects another building
User selects main building
Main building
User selects a unit
User cant afford unit User is notified of this
Selected unit
User orders the unit to be built
Resoucse transaction

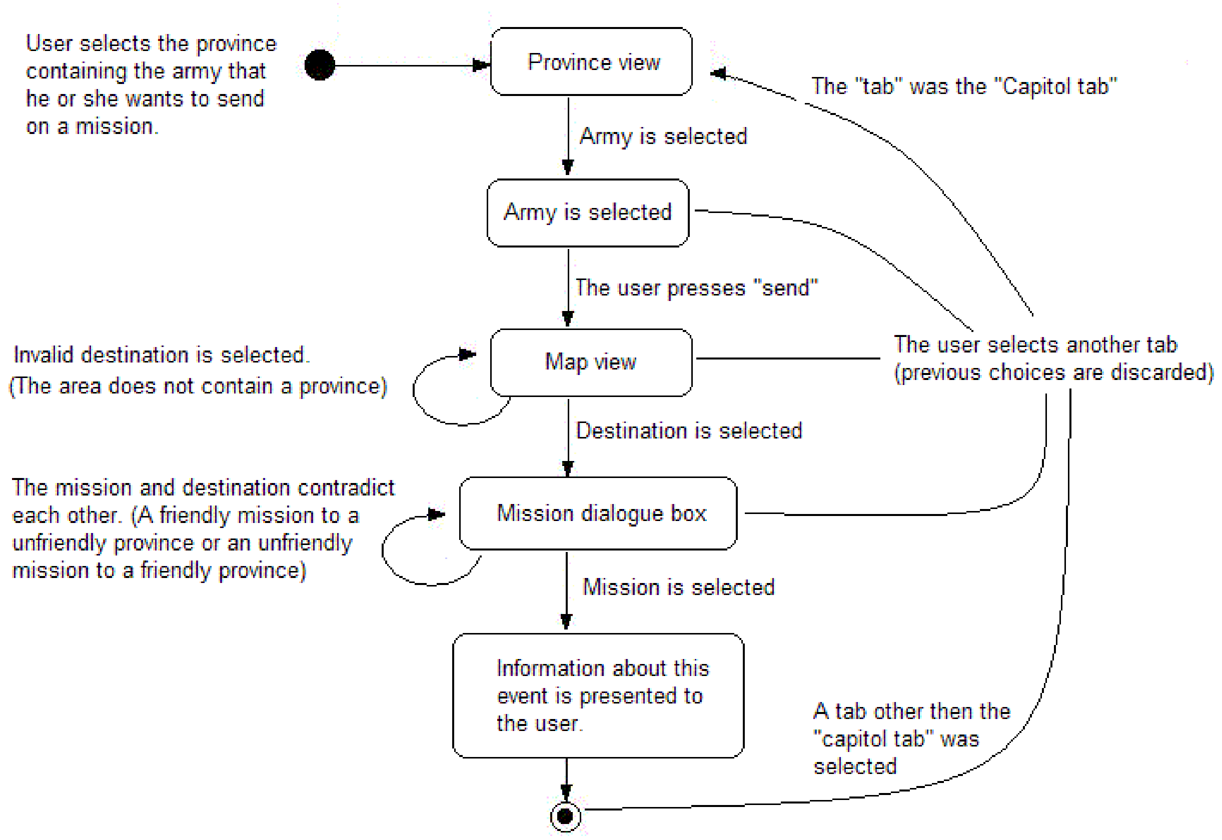
State Chart, Sending Mail

It is assumed that the player is logged in into the game.



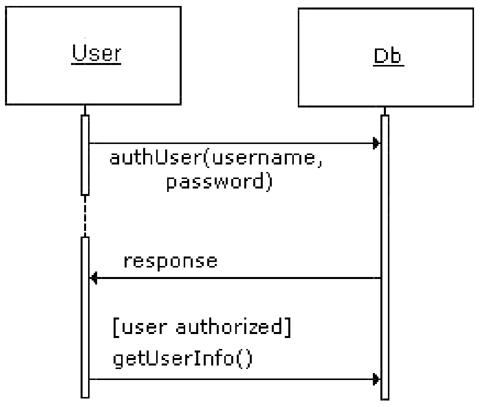
State Chart, Sending an army to a mission

It is assumed that the player is logged in into the game.


User selects the province containing the army that he or she wants to send on a mission.
Province view
The "tab" was the "Capitol tab"
Army is selected
Army is selected
The user presses "send"
Invalid destination is selected. (The area does not contain a province)
Map view
The user selects another tab (previous choices are discarded)
Destination is selected
The mission and destination contradict each other. (A friendly mission to a unfriendly province or an unfriendly mission to a friendly province)
Mission dialogue box
Mission is selected
Information about this event is presented to the user.
A tab other then the "capitol tab" was selected

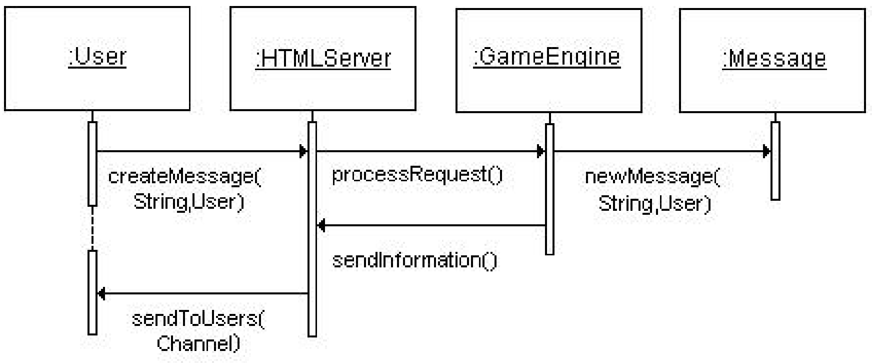
5.4 Interaction charts

Interaction Chart, Logging in



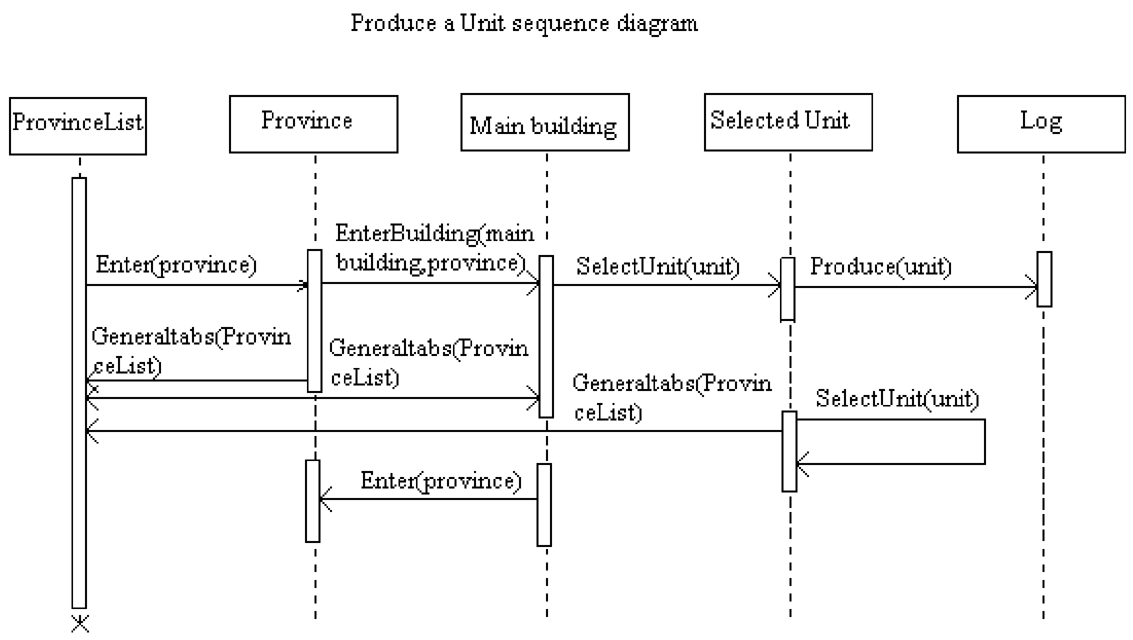
Interaction Chart, Instant Chat.

It is assumed that the player is logged in into the game.



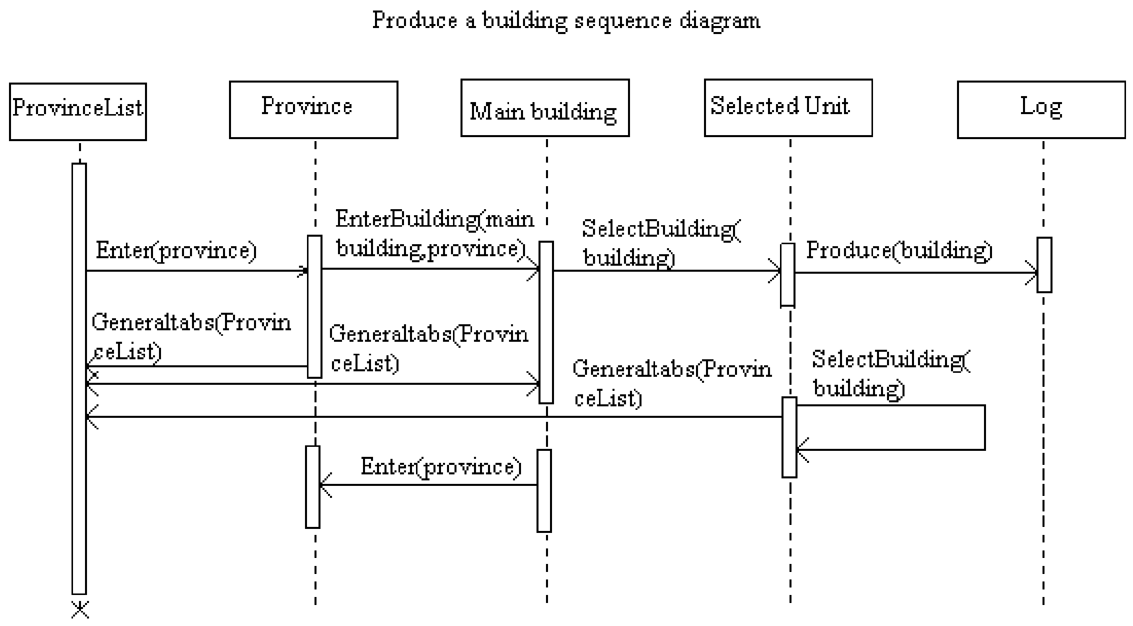
Interaction Chart, Train unit

It is assumed that the player is logged in into the game.


Produce a Unit sequence diagram
ProvinceList
Province
Main building
Selected Unit
Log
Enter(province)
EnterBuilding(main building,province)
SelectUnit(unit)
Produce(unit)
Generaltabs(ProvinceList)
Generaltabs(ProvinceList)
Generaltabs(ProvinceList)
SelectUnit(unit)
Enter(province)

Interaction Chart, Build Building

It is assumed that the player is logged in into the game.


Produce a building sequence diagram
ProvinceList
Province
Main building
Selected Unit
Log
Enter(province)
EnterBuilding(main building,province)
SelectBuilding(building)
Produce(building)
Generaltabs(ProvinceList)
Generaltabs(ProvinceList)
Generaltabs(ProvinceList)
SelectBuilding(building)
Enter(province)

5.5 Detailed Design

About the classes:

In the following pages each class will be presented in the following format:

Class Name
(References to RD)

Int ***variableName***; // Comments that clarifies the use of the variable (if necessary)

/*
* Comments about the method
*/
methodName(String input)
-pre: Prerequisites.
-post: Any post-method conditions
-return: What the method returns
-data access: Database accesses
-called by: The classes that calls this method
-calls: The classes (and/or methods) this method calls
It is also assumed that each class contains “setters” and “getters” for each local variable. (“Setters” and “getters” are methods that sets a variable respective returns a variable).
Local variables are marked with bold and italic characters.
Some methods may be split into two or more methods when they are implemented but they may, for simplicity reasons, be viewed as one single entity.

Cross-reference chart

Should be read as:

RD section : classes implementing the contents of this section

4.1.1 Registering: class User.
4.1.2 Logging in: class User, class Player, class accountmanager.
4.1.3 Recover password: class User, class accountmanager.
4.1.4 The in-game-environment: class GameEngine, class Building, class UnitShell, class Province, class Area, class CombatLog and class Upgrades.
4.1.5 Build buildings: class Buildings, class Upgrades.
4.1.6 Train units: class UnitShell, class Unit and class Building.
4.1.7 Chat with other players: class Channel, class Messages.
4.1.8 Send private messages to other players: class Mail, class Messages.
4.1.9 Manage armies: class Province, class Combat, class Unit, class UnitShell and class Area.
4.1.10 Select a view: class CombatLog, class Province, class Area and class Game Engine.
6.1.1 Registering: class User.
6.1.2 The in-game-environment: class GameEngine, class Building, class UnitShell, class Province, class Area, class CombatLog and class Upgrades.
6.1.3 Build buildings: class Buildings, class Upgrades.
6.1.4 Train units: class UnitShell, class Unit and class Building.
6.1.5 Chat with other players/ Send private messages to other players: class Channel, class Messages and class Mail.
6.1.6 Manage armies: class Province, class Combat, class Unit, class UnitShell and class Area.

Class Building

(Partly implements the Build Buildings (4.1.5), Build Building (4.3.3) and “Build buildings” (6.1.3) requirements in the RD)

Int ***level***;
LinkedList ***upgrades***;
String ***name***;
String ***image***;
Int[] ***coords***;

/*
* Increments the level of a Building by one and updates the image
*/
levelUp()
-pre: The building is level X and has ***image*** Y.
-post: The building is level X+1 and has ***image*** Z, where $Z \neq Y$.
-return: Void
-data access: Reads ***image*** string Z from the Database via the class DB.
-called by: class GameEngine
-calls: None

/*
* Applies an upgrade to a building
*/
applyUpgrade(Upgrade upgrade)
-pre: The building has ***upgrades*** $\{X_1,\ldots,X_n\} = Z$ where Z may be {Ø}
-post: The building has ***upgrades*** $\{X_1,\ldots,X_n,X_{n+1}\}= X$
-return: Void
-data access: None
-called by: class GameEngine
-calls: None

/*
* Creates a new building object
*/
Building(String n, int[] co)
-pre: No building at these coordinates exists
-post: A building with ***name*** = n, ***coords*** = co, ***level*** = 1, **upgrades** = {Ø}, ***Image*** = Z exists
-return: Void
-data access: Reads ***image*** string Z from the Database via the class DB.
-called by: class GameEngine
-calls: None

Class Upgrade

(Partly implements the Build Building (4.3.3 (see Variations 7)), Build Buildings (4.1.5) and the “Build buildings” (6.1.3) requirements in the RD)

String ***name***;
Int[] ***bonuses***; //Contains bonus percentages

/*
* Creates a new upgrade object
*/
Upgrade(String n, int[] b)
-pre: This upgrade object does not exist
-post: An upgrade object exists with ***name*** = n, **bonuses** = b.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

Class Area

(Partly implements the View Map (4.3.9 (see “Main Succcess scenario” 2 and “Extensions” 2b1)) and the “In-game environment” (6.1.2) requirements and use case in the RD)

Int[] ***coords***.
Province ***p***; //May be { Ø }
Int ***factionId***; //The faction it belongs to (0 if no faction)

/*
* Creates a new area object
*/
Area(int[] c)
-pre: This area object does not exist
-post: An area object with ***coords*** = c, ***factionId*** = 0, ***p*** = null exists.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

Class Province

(Partly implements the “Build buildings” (4.1.5) , “Train units” (4.1.6), “Manage Armies” (4.1.9) , “Select a view” (4.1.10) and “In-game environment” (6.1.2) requirements)

String ***name***;
Int[] ***coords***;
Int ***playerID***;
Building[][] ***buildings***;
Sides ***armys***;
Int ***gold***;
Int ***mana***;

/*
* Creates a new province object
*/
Province(String n, int[] c, int p)
-pre: This province object does not exist
-post: A province object with ***name*** = n, ***coords*** = c, ***playerID*** = p exists.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

/*
* Adds a building to a specific coordinate.
*/
addBuilding(int[] c, Building b)
-pre: There is no building at coordinate c.
-post: ***buildings***[c_x , c_y] contains building b
-return: Void
-data access: None
-called by: class GameEngine
-calls None

Class Army

(Implements the “Manage Armies” (4.1.9) and the “Manage armies” (6.1.6) requirements in the RD)

Int[] ***coords***;
Int ***factionId***;
UnitShell[] ***units***;
Int ***mission*** // If the army has a ***mission*** then ***mission*** ≠ 0
// Ex 1100 , 1 = Defend, 100 = 100 minutes

/*
* Creates a new army object
*/
Army(int[] c, int f)
-pre: This army object does not exist
-post: An army object with ***coords*** = c, ***factionId*** = f and ***units*** = {Ø} exists.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

/*
* Adds a new unit to the army
*/
addUnit(UnitShell u, int pos)
-pre: No unit shells exists at position pos in ***units***;
-post: A unit shell u is added to position pos.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

/*
* Removes a unit from the army
*/
removeUnit(UnitShell u, int pos)
-pre: A unit shell u exists at position pos.
-post: No unit shells exists at position pos in ***units***.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

Class Sides

(Partly implements the War (4.3.9 (see “Main success scenario” 5) use case) and the “Manage armies” (6.1.6) requirement)

ArrayList ***defendingSide***;
ArrayList ***attackingSide***;
Int ***defendingSideFactionId***;

/*
* Creates a new sides object
*/
Sides(int def)
-pre: This side object does not exist
-post: A side object with ***defendingSideFactionId*** = def.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

/*
* Adds a new army to the sides
*/
addArmy(Army a, int i)
-pre: No side contains a.
-post: If I == ***defendingSideFactionId*** then the defending side contains a, else the attacking side does.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

Class Faction

(Partly implements the “In-game environment” (6.1.2) requirement)

Int ***number***;
Unit[] ***unitPool***;
HashMap ***players***; // Consists of all the players in that faction

/*
* Creates a new faction object
*/
Faction(int n)
-pre: This Faction object does not exist
-post: A Faction object with ***number*** = n exists.
-return: Void
-data acess: None
-called by: class Startup
-calls: None

/*
* Adds a new player to the faction
*/
addPlayer(Player p)
-pre: Player p is not in ***players***
-post: Player p is in ***players***.
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

/*
* Removes a player to the faction
*/
removePlayer(Player p)
-pre: Player p is in ***players***.
-post: Player p is not in ***players***
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

/*
* Returns a player object from the faction
*/
getPlayer(int id)

-pre: No prerequisites.
-post: Player p is returned where p has id ***id***.
-return: Player p
-data acess: None
-called by: class GameEngine
-calls: None

Class Mail

(Partly implements the “Send private messages to other players” (4.1.8) and the “Chat with other players / Send private messages to other players” (6.1.5) requirements as well as the “Private Message” use case (4.3.6))

String ***sender***;
String ***receiver***;
String ***message***;
Date ***timestamp***;
boolean ***read***;

/*
* Creates a new Mail object
*/
Mail(String s, String r, String mess)
-pre: There exists no mail object
-post: There exists a mail object with ***sender*** = s, ***receiver*** = r, ***message*** = mess, ***timestamp*** = current date, ***read*** = false
-return: Void
-data acess: None
-called by: class GameEngine
-calls: None

Class Combat

(Partly implements the “Manage Armies” (4.1.9) and the “Manage armies” (6.1.6) requirements as well as the “War” (4.3.5) use case)

Sides ***s***;
HashMap ***combatlogs***;

/*
* Calculates the outcome of a combat based on two different sides and calls the
*CombatLog class to print a combat log for the combat.
*/
Combat(Sides s)
-pre: Two different sides are in the same province.
-post: Combat has been calculated and one side left victorious. this.***s*** = s;
-return: Void
-data access: None
-called by: class GameEngine
-calls: class CombatLog

/*
* Computes a turn of combat. Called every ten minutes by the GameEngine.
* Writes down reseults in a HashMap, this is saved until the combat ends.
*/
doCombat()
-pre: A combat has been started.
-post: One more turn of combat calculated. Returns true if one side has been vanquished.
-return: boolean
-data access: None
-called by: class GameEngine
-calls: class CombatLog

/*
* After combat has ended the HashMapp containing the logs for the different players is
*returned to the GameEngine.
*/
finishCombat()
-pre: A combat has ended, a doCombat()-call has returned true.
-post: A HashMap, ***combatlogs***, containing all of the logs for the players is returned to the GameEngine.
-return: HashMap
-data access: None
-called by: class GameEngine
-calls: None

Class Messages

(Partly implements the “Send private messages to other players” (4.1.8), “Chat with other players” (4.1.7) and the “Chat with other players / Send private messages to other players” (6.1.5) requirements as well as the “Private Message” (4.3.6) and the “Chat Message” (4.3.6) use cases)

/*
* Searches for mails within the specified interval and after a certain timestamp for the
*chosen player.
*/
mail[] getMail(String PlayerID, int from, int to, int timestamp)
-pre: The user has accessed the mail-section in the game.
-post: The mails within the specified parameters are showed to the user.
-return: mail[] , returns all the relevant mails in a vector.
-data access: Accesses the DB for mails via the DB-class.
-called by: class GameEngine
-calls: class DB

/*
* Returns how many mails a player has in his/her mailbox.
*/
int howManyMail(String PlayerID)
-pre: The user has accessed the mail-section in the game.
-post: How many mails the players has in his/her inbox is displayed.
-return: int, the # of mails.
-data access: Accesses the DB and mail-table and returns the number of posts in it.
-called by: class GameEngine
-calls: class DB

/*
* Sends a mail to the user specified in the Mail-object being sent.
*/
sendMail(Mail a)
-pre: The user has pressed “New” in the mail section of the game.
-post: The mail ***a*** is sent to according to the information contained in the Mailobject.
-return: None
-data access: Accesses the DB and enters the mail to the correct players mail table.
-called by: class GameEngine
-calls: class DB

Class Unit

(Partly implements the “Manage Armies” (4.1.9), “Train Units” (4.1.6) and the “Manage armies” (6.1.6) requirements as well as the “War” (4.3.5) and the “Train a unit” (4.3.4) use cases)

String ***name***;
Int ***unitID***
Int ***specialFeat***;
Int[] ***stats***; //{maxHealth, attack, defence}
Int[] ***unitTypeModefiers***; //damage modifiers against other units depending on unittypes

/*
* Creates a new unit object
*/
Unit(String n, Int id Int sFeat, Int[] startstats, Int[] startUnitTypeModefiers**)**
-pre: No unit.
-post: A unit with ***name*** = n, ***unitID*** = id, ***specialFeat*** = sFeat, ***stats*** = startstats, and ***unitTypeModefiers*** = startUnitTypeModefiers
-return: Void
-data access: None.
-called by: class Startup
-calls: None

Class UnitShell

(Partly implements the “Manage Armies” (4.1.9), “Train Units” (4.1.6) and the “Manage armies” (6.1.6) requirements as well as the “War” (4.3.5) and the “Train a unit” (4.3.4) use cases)

Int ***level***;
Int ***health***;
Int ***originalUnitID***;
Int ***experience***;

/*
* Increments the level of a UnitShell by one
*/
levelUp()
-pre: The unit is ***level*** X
-post: The unit is ***level*** X+1
-return: Void
-data access: None
-called by: class Combat
-calls: None

/*
* Creates a new unitshell object
*/
Unit(Int orgID)
-pre: No unit.
-post: A unit with ***originalUnitID*** = orgID, ***level*** = 0, health equal to the original units ***maxHealth*** and ***experience*** = 0
-return: Void
-data access: None.
-called by: class GameEngine
-calls: (faction.unit(orgID)).getMaxHealth();

Class Event

Int ***evenType***;
Object[] ***eventParameters***;

/*
* Creates a new event.
*/
Event(Int event, Object[] parameters)
-pre: No event.
-post: An event that stores relevant data that applies to the event.
-return: Void
-data access: None.
-called by: class Combat.
-calls: None

Class EventQueue

Timer ***timer***;
Heap ***queue***;

getEvent()
-pre: the timer tells GameEngine that its time to get the next event.
-post: The event has been sent to the GameEngine and the event is removed from the EventQueue.
-return: Event
-data access: None.
-called by: class GameEngine.
-calls: None.

addEvent(Event insertevent, Date eventdate)
-pre: The eventqueue does not contain the event.
-post: The eventqueue contains the event and the timer is set accordingly after eventdate if needed.
-return: Void
-data access: None.
-called by: class GameEngine.
-calls: None.

/*
* Starts up an eventqueue.
*/
EventQueue(Date enddate)
-pre: No eventqueue.
-post: A Eventqueue started with a running timer set to an "end game event" with the time set to enddate that is inserted into the heap as the only element so far.
-return: Void
-data access: None.
-called by: class Startup.
-calls: None.

Class DB

String ***name***;
String ***password***;
Connection ***conn***;

/*
* Opens connection to the database
*/
init()
-pre: There is no connection to the database.
-post: A connection is established to the database if the username and password are correct and the database is online.
return: Void
-data access: None
-called by: class Startup
-calls: None

/*
* Checks if there is a connection to the database
*/
isConnected()
-pre: Some system component requests connection status.
-post: The method returns status as a Boolean value.
-return: True or false
-data access: None.
-called by: All classes requesting access to the database.
-calls: None

/*
* Closes database connection
*/
destroy()
-pre: There is an active database connection.
-post: The active connection is disconnected.
-return: Void
-data access: None.
-called by: All classes requesting access to the database.
-calls: None

/*
* Executes a query that modifies the database by inserting, deleting or updating data.
*/

executeModifyQuery(String query)
-pre: There is an active database connection and a valid query is sent as an argument.
-post: A database query has been performed
-return: Int queryResultState
-data access: Tables affected by query.
-called by: All classes requesting access to the database.
-calls: None

/*
* Executes a query that fetches data.
*/
executeSelectQuery(String query)
-pre: There is an active database connection and a valid query is sent as an argument.
-post: A database query has been performed
-return: Array resultSet
-data access: Tables affected by query.
-called by: All classes requesting access to the database.
-calls: None

Class CombatLog

(Partly implements the “Select a view” (4.1.10) (the “combat log” is a view) and the “In-game environment” (6.1.2) requirements)
Int ***time***;
String ***mission***
String ***province***;
String ***log***;

/*
* Creates a new combatlog object from the information given by combat.
*/
CombatLog(String[ ] n)
-pre: No combatlog.
-post: A combatlog with the information from n.
-return: Void
-data access: None.
-called by: class Combat.
-calls: None

Class Channel

(Partly implements the “Chat with other players” (4.1.7) and the “Chat with other players / Send private messages to other players” (6.1.5) requirements as well as the “Private Message” (4.3.6) use case)

String ***name***;
String ***password***;
LinkedList ***messageLog***;
ArrayList ***playersInChannel***;
HashSet ***playerNamesToPlayersSet***;

/*
* Used by players to join a channel.
*/
addPlayer(String playerName, String channelPassword)
-pre: Player is not in the channel;
-post: If the password was correct the player is in the channel and has loaded the messageLog.
-return: Void
-data access: None.
-called by: class Channel.
-calls: None

/*
* Used by players to leave a channel.
*/
removePlayer(String playerName)
-pre: The player is in the channel;
-post: The player is in the channel
-return: Void
-data access: None.
-called by: class Channel.
-calls: None

/*
* Used by players send a message to the channel.
*/
sendMessage(String playerName, String message)
-pre: The player wants to send a message
-post: The has sent a message to the channel(updating ***messageLog***), which in turn sends the message to the other players in the channel telling them that there has been a message sent.
-return: Void
-data access: None.
-called by: class Channel.
-calls: None

/*
* Used by players to receive messages sent by other players.
*/
recieveMessage()
-pre: The player has the old message log;
-post: The player has the new message log and have thus received the message.
-return: LinkedList messageLog
-data access: None.
-called by: class Channel.
-calls: None

/*
* Creates a new empty channel, if password is empty it's a new public channel
*/
Channel(String channelName, String channelPassword)
-pre: No channel with ***name*** = channelname;
-post: A channel with ***name*** = channelname and ***password*** = channelpassword.
-return: Void
-data access: None.
-called by: class Messages.
-calls: None

Class GameEngine

(Partly implements “The in-game-environment” (4.1.4), “Build Buildings” (4.1.5), “Train units” (4.1.6), “Manage armies” (4.1.9), “Build a building” (4.3.3), “Train a unit” (4.3.4), “War” (4.3.5) requirements and usecases in the RD)

Faction[] ***factions***
HashMap ***activePlayers***

/*
* Takes an event from the httpserver and checks if it would be legal do add the event to
*to the EventQueue. Player pays the costs for putting the event on the EventQueue. This
* method together with handleEvent will probably be split up into many methods but you
* may view them as one entity.
*/
putEvent(Event inevent)
-pre: No inevent in EventQueue.
-post: If the event is legal, there will be an Event object with inevent data in the EventQueue. Made necessary adjustments to the database for the events to be allowed to occur.
-return: Void
-data access: Reads relevant data to verify that the Event is legal(gold in a province etc) and writes new relevant information to the database.
-called by: class HTTP server
-calls: class EventQueue, class DB

/*
* Gets an event from the EventQueue and handles it appropriately. This method together
* with putEvent will probably be split up into many methods but you may view them as
* one entity.
*/
handleEvent(Event eventToHandle)
-pre: Event eventToHandle has not been handled and was in the EventQueue
-post: The object eventToHandle has been handled and the appropriate methods in other classes has been called.
-return: Void
-data access: None
-called by: class EventQueue
-calls: class Building, class UnitShell, Class Combat, …

/*
* Creates a new GameEngine object
*/

GameEngine(Faction[] f)

-pre: No GameEngine exists.
-post: A GameEngine oject exists and it has started necessary game threads.
-return: Void
-data access: None
-called by: class Startup
-calls: None

Class Player

(Partly implements the “Logging in” (4.1.2), “To login to the game”(4.3.2), “Private message”(4.3.6), “Chat message”(4.3.7) requirement and usecases in the RD)

ArrayList ***provinces***; //Contains all the province-id’s that a player controls.
int ***faction***;

/*
*Creates a new player object and takes the id of the faction that the player play as, as
*parameter.
*/
Player(int faction)
-pre: None
-post: A new player object is created and successfully logged in.
Return: Void
data access: None
called by: The object itself when created.
calls: None

Class StartUp

/*
* Initiates the game engine and all other nessecary components
*/
StartUp()
-pre: The game is not started.
-post: The game engine and all other nessecary components has been created and initiated.
-return: Void
-data access: None
-called by: class EmpiresOfAvatharia.main
-calls: class Factions, class HTTPserver, class GameEngine, class Messages, class DB

Class User

(Partly implements the Registering (4.1.1)(6.1.1), Logging in (4.1.2) and Recover Password (4.1.3) requirements in the RD)

String[] userinfo;
String ***playerid***;

/*
* Returns the Player from the database whose id is the same as the User-objects.
*/
getPlayer()
-pre: Player information is needed.
-post: The corresponding player info is returned.
-return: Player p.id == ***playerid***;
-data access: The database for player information.
-called by: class GameEngine
-calls: None

Class AccountManager

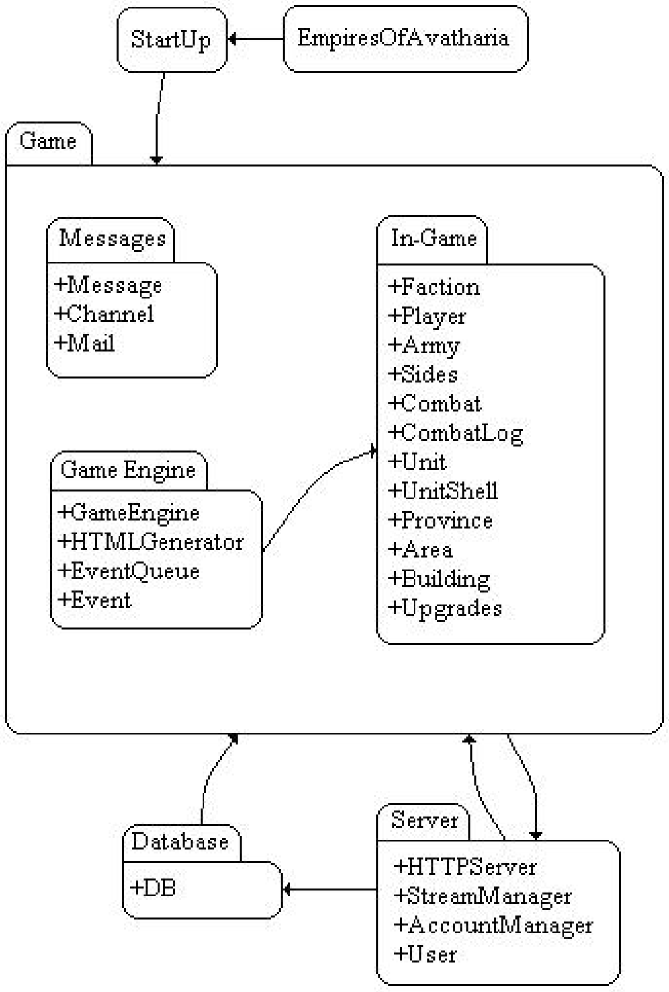
(Partly implements the Logging in (4.1.2) and the Recover Password (4.1.3)
requirements in the RD)

/*
* Logs in a user and returns the corresponding user-object.
*/
login (String name, String pass)
-pre: A user is not logged in and tries to log in.
-post: The user is logged in.
-return: User u.
-data access: None
-called by: class HTTPServer
-calls: None

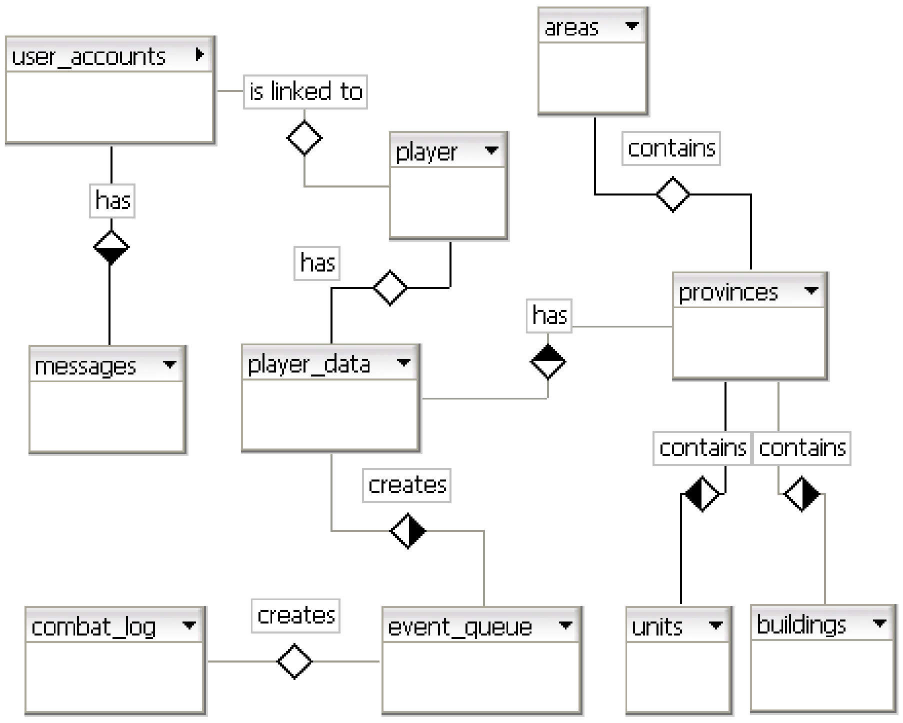
/*
* Registers a new user and ads the entered information to the database.
*/
register(String[] info)
-pre: A user who's not registered tries to register.
-post: The user is now registered.
-return: None
-data access: Contacts the database and enters the new information for the new user.
-called by: class HTTPServer
-calls: None

/*
* Checks if a certain player has the privilege to perform a certain event, and returns to
*true or false depending on the result.
*/
hasPrivilege (Event e, Player p)
-pre: A user tries to perform something in the system, clicking a button etc.
-post: The user gets to perform the action if true is returned and the other way
around if false is returned.
-return: True/False
-data access: Accesses the database to check the users privilege.
-called by: class HTTPServer
-calls: None

5.5 Package Diagram



5.6 Database table



6. Functional Test Classes

Build building

(See RD section 4.1.5)

Explanation:

Test if you are able to build a building in a province spending X resources where X is the buildings cost.

Input:

The building that the user wants to build from the town centre where it's supposed to be built and where in that province it is to be built.

Output:

The appropriate amount of gold and mana is subtracted from the player and the building that was chosen is built on the selected coordinate.

Step-by-step procedure:

1. Click on the "Province list"-tab.
2. Click on the appropriate "Province", (the province you want to build a building in).
3. Click on the "Town centre"-building.
4. Click on the building that you want to build.
5. Click the coordinates where the building is to be built.

Train a unit

(See RD section 4.1.6)

Explanation:

Test if you are able to train a unit in a province spending X resources where X is the training cost of the unit.

Input:

The unit that the user wants to train.

Output:

The appropriate amount of gold and mana is subtracted from the player and the unit is starts its training process.

Step-by-step procedure:

1. Click on the “Province list”-tab.
2. Click on the appropriate “Province”, (the province you want to train a unit in).
3. Click on the “Town centre”-building.
4. Click on the unit that you want to train.

Recover Password

(See RD section 4.1.3)

Explanation:

Test if you are able to recover a users password given a user name.

Input:

The input required to recover a password is simply the user's user name.

Output:

The output of this function is an e-mail containing the user's password and sent to the user's specified e-mail address in the database.

Step-by-step procedure:

4. First enter the main page of the game.
5. Click the recover password button on the main page.
6. Enter the user name of the password to be recovered.
7. Check the, with the user associated, e-mail for the password.
8. Enter the user name and recovered password in the login fields and continue playing.

Move a Unit Between Armies

(See RD section 4.1.9 , first part in the Manage armies-section)

Explanation:

Test if you are able to move a unit between armies stationed in the same province.

Input:

The user must be positioned in the correct province where the armies are situated. One army in the defensive army position and one in the offensive army position in the province.

Output:

The output should be a unit being moved from one army to the other in the province, specifically from the defensive army to the offensive, or the other way around.

Step-by-step procedure:

4. Enter the province screen from the main game menu.
5. Select the province where the armies are situated.
6. Select and drag the unit in question from the defensive army to the offensive or vice versa.
7. The unit is moved.

Send an Army on a Mission

(See RD section 4.1.9, second part of the Manage armies-section)

Explanation:

Test if you are able to send an army on a mission, which includes pillaging, defending or conquering.

Input:

The input required is an offensive army from a province.

Output:

The selected army is sent on one of the three missions correctly. That is, either defending a friendly province, pillaging an opponents province or conquering an opponents province.

Step-by-step procedure:

1. Enter the province screen from the main game menu.
2. Select a province containing the offensive army that you want t send on a mission.
3. Select the offensive army and select that you want to send the army on a mission from the army-menu.
4. Select one of the three mission alternatives and then select the province you want the army to go to.
5. The army is sent on the selected mission to the selected province.

Chat with other players

(See RD section 4.1.7)

Explanation:

Test if a player P can send a message X to the channel Y and test if the players in channel Y can revieve message X.

Input:

A player P, a message X and a channel Y.

Output:

The players in channel Y recieves message X.

Step-by-step procedure:

1. Click on the “Chat”-tab.

2. Click on the appropriate “Channel”, (the channel you want to send a message to).
3. Type the message you want to send in the textfield.

Click on the “send”-Button.

Send private messages to other players

(See RD section 4.1.8)

Explanation:

Test if a player P_1 can send a message X to a player P_2 and test if the player P_2 can revieve message X.

Input:

A player P_1, a player P_2 and a message X.

Output:

The player P_2 recieves message X from the player P_1.

Step-by-step procedure:

1. Click on the “Chat”-tab.
2. Click on the “New message”-button.
3. Type a message and a reviever in the textfields that are revealed.
4. Click on the “Send message”-Button.

Registering

(See RD section 4.1.1)

Explanation:

Test if you are able to register as a new player.

Input:

The registration form on the main website (pict. 4.2). Required input is full name, unique username, unique e-mail-address, desired password and confirmation of desired password. The registration also requires the user to choose in-game faction and in-game start location.

Output:

The registration form will display information about insufficient input if the form is not completely filled, if the username does not fit in with the restrictions, if the passwords mismatch or if the in-game start location is invalid.

If the registration form is correctly submitted, the user will receive an e-mail to the posted address containing a link to amongst other things ensure that the user is not at least the simplest of mass registration bots and that the user has access to the mail account. When the link in the e-mail has been opened, the registration will be complete and a user will be created with the desired username and password. When the user has received a confirmation on the page that is displayed when clicking the link, the registering process has been completed.

Step-by-step procedure:

1. Enter the main site.
2. Click on the "Register" menu option on the main site.
3. Fill in the registration form.
4. Check the e-mail account that was given for new messages and wait for.
5. Click the confirmation link that is in the message.
6. Verify that account creation confirmation is displayed in the web browser window opened by clicking the link.

Logging in

(See RD section 4.1.2)

Explanation:

Test if you are able to log in with the registration data that you got when you signed up for the game.

Input:

The input fields in the login area (pict. 4.2).

Output:

If the login data is incorrect (the user is not registered or is using an incorrect username or password) the user will be notified by new highlighted text in the form. If the login data is correct, a new browser with the game interface will be opened.

Step-by-step procedure:

1. Enter the main site.
2. Type in the username and password from the registration step in the login area.
3. Click Start Game.
4. Verify that the game interface has been displayed in a new browser window.