Settle And Destroy (SAD)

Group 13

Jonas Wikberg
Christofer Hjalmarsson
Daniel Westerberg
Saul Amram
André Sikborn Erixon

Abstract

The never ending need of entertainment and the randomness of how modern persons spend their time in front of the computer create a wish for the perfect entertainment application. This project, Settle and Destroy, seeks to fulfill this which for persons with an interest in solving strategical problems in a game situation. Not only fulfilling this first wish Settle and Destroy, also aims at giving the user an opportunity to interact with other users in a multiplayer mode further extending the entertaining value for the user.

This is the design document for project SAD which defines and outlines the project structure. The design documents is the preface to the implementation phase why thorough considerations are made concerning every design decision from top level package and class structure down to low level design decisions concerning the graphical user interface.

TABLES OF CONTENTS

DESIGN DOCUMENT

1. Introduction

1.1. Purpose and Scope

We believe that the need for entertainment is an appeal that will never change. The latest fashion and trends may change with the wind, but people will always need something entertaining to do in their spare time. In this perspective computer gaming has come to stay. Settle and Destroy (SAD) is a real-time strategy war-game with opportunities to play both multiplayer and a special training ground. The aim of our game and the incentive for a user to play our game is that the game offers entertainment for a shorter time period than general entertainment and other computer game.

The scope of this document is to provide an overall guidance to the future architecture of the software project SAD. This document will work as the underlying resource of documentation for estimating time consumption of the implementation phase of the software. Further on, this document establishes the total software outline including architectural, design and detailed design descriptions and assumptions. The design document will also include high detailed specifications provided in diagrams and charts. Generally the document should give a complete design description meanwhile maintaining a high-level view of the software.

The expected readership of the design document is concentrated to the group developing the software. This group could consist of project leaders and supervisors but also by programmers and design personnel. The document will also be read by future developing teams if a new version is planned or if the product is sold.

This development project is originally, as mentioned above, called Settle and Destroy. In abbreviated form SAD. The project will also, due to easier readership, be referred to as *the project*, *project SAD* or *the application*.

1.2. Related documentation

1.2.1. Prerequisite & Companion documents

Requirement Document (Version 1.0)

1.2.2. Context providing documents

Design Document Template[1]
Design Document Guidelines[2]
Overall Architecture Description[3]
An Example Of Object-Oriented Design: An ATM Simulation[4]
Practical UML: A Hands-On Introduction for Developers[5]

[1] R. Waltzman (2007-2008), fetched: 2008-01-20
http://www.csc.kth.se/utbildning/kth/kurser/DD1363/DesignDocument.html
[2] Waltzman(2007-2008)
[3] Waltzman(2007-2008)
[4] Russell C. Bjork (2004), fetched: 2008-01-25
http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/
[5] R. Miller (2003) fetched: 2008-02-12 http://dn.codegear.com/article/31863

1.3. Term definitions

1.3.1. Terms, abbreviations and acronyms

SAD	The application and game name **S**ettle **A**nd **D**estroy as acronym
Real-time strategy war-game	A game that is strategic game played live where all players experiences simultaneous movement and actions.
Tooltip text	The tooltip is a common graphical user interface element. It is used in conjunction with a cursor, usually a mouse pointer. The user hovers the cursor over an item, without clicking it, and a small box appears with supplementary information regarding the item being hovered over.
Player	The word “player” most often refers to the physical person playing the game.
Team	The word “team” refers to the team in the game, i.e. the player’s race, village and all his/hers armies.
Map	The map in this application refers to an overview of the game, built of square cells in which actions and graphical events can occur.
Race	The word race is used to distinguish different qualities of player teams that will be due to the race factor. Two player could have the same race and there for troops with the same qualities etc.

1.4. Abstract

The never ending need of entertainment and the randomness of how modern persons spend their time in front of the computer create a wish for the perfect entertainment application. This project, Settle and Destroy, seeks to fulfill this which for persons with an interest in solving strategical problems in a game situation. Not only fulfilling this first wish SAD, also aims at giving the user an opportunity to interact with other users in a multiplayer mode further extending the entertaining value for the user.

This is the design document for project SAD which defines and outlines the project structure. The design documents is the preface to the implementation phase why thorough considerations are made concerning everything from top level package and class structure down to low level design decisions concerning the graphical user interface.

2. System Overview

2.1. General Description

2.1.1. Basic overview

Settle and Destroy is a real-time strategy war-game with opportunities to play both multiplayer and a special training ground. A player has to do is to choose an alias to play with. Then you can choose to start your own game, or join another player’s game that’s waiting for more players. The first thing you have to do when you start your own game is to enter the number of players that will participate. In theory the number of players in a game is unlimited, but since all the slots have to be filled a maximum of eight is a rule of thumb. A player that chooses to join an already existing game has to enter the ip-address of the host to be able to connect to the game. When players join a game they must press the "ready" button to inform the host that they are ready to start. The host can not start the game until everyone

joined are ready. The game starts, and due to its nature players will die as time passes. When this happens you can either choose to stay in the game and observe, or simply leave and join another game. The game ends when all players except one are dead. This one survivor is therefore named the winner. As soon as the game has a winner, people will disconnect and then either decide whether they want to play more or perhaps get back to things they did before they started playing.

2.1.2. Functionality and design description

The emerging and most prominent functionality that also impacts the software design is the multiplayer functionality. The game is further real time based which will have further distinct implications for the system design. These two functions will impact the software design process the most.

In comparison to the requirements stated in the requirements document, referenced above, some small changed has been made in functionality. Below the principal functionality is displayed:

Multiplayer mode for both local area network (LAN) and internet	X	
Save functionality		X
Turn based		X
The game is played in real time (buildings, troops)	X	
Training mode (Single player)	X	
Artificial Intelligence (AI), Computer that join as a player		X
2D graphical view	X	
Build and expand village	X	
More than one village per player		X
Build and expand troops	X	
Move and attack using troops	X	
Map that describe the game field	X	
More then one race to play (Different races to choose from)	X	
Sound effects		X
Mouse and keyboard to play the game	X	
Simple game chat		X
Interface when creating and joining a game	X	
Tutorial that describe how to play the game		X
Observer mode for multiplayer when a player is dead	X	
Pause the game		X

2.2. Overall architecture description

When playing a multi player game, the application utilizes a client-server model.

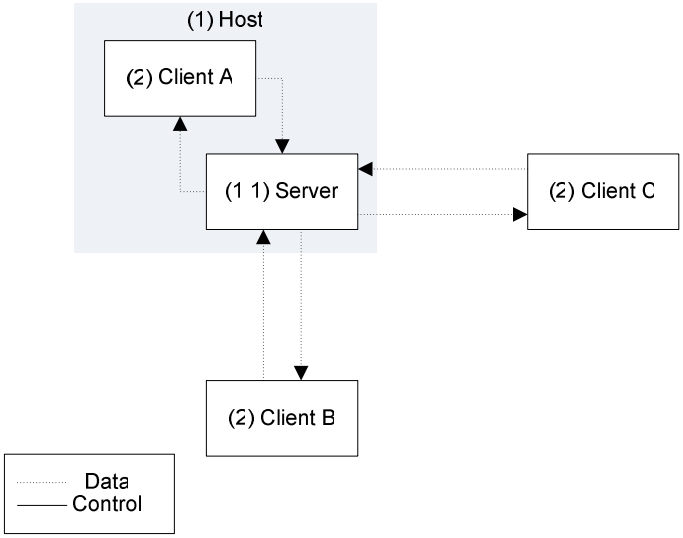


Figure 1. Application network overview.

2.2.1. Host

The host is the computer hosting a multiplayer game and is decomposed into a server and a client. The server is started as a standalone part of the host and the client of the host connects to the server as any other client. This way the hosting client doesn't have to be distinguished from the other clients.

2.2.2. Server

The server handles all connected clients, incoming connections and other events associated with the network layer of the game. All communication among the clients are sent through, handled by, and forwarded from the server.

2.2.3. Client

The client is the game application running on every connected computer.

2.3. Detailed architecture description

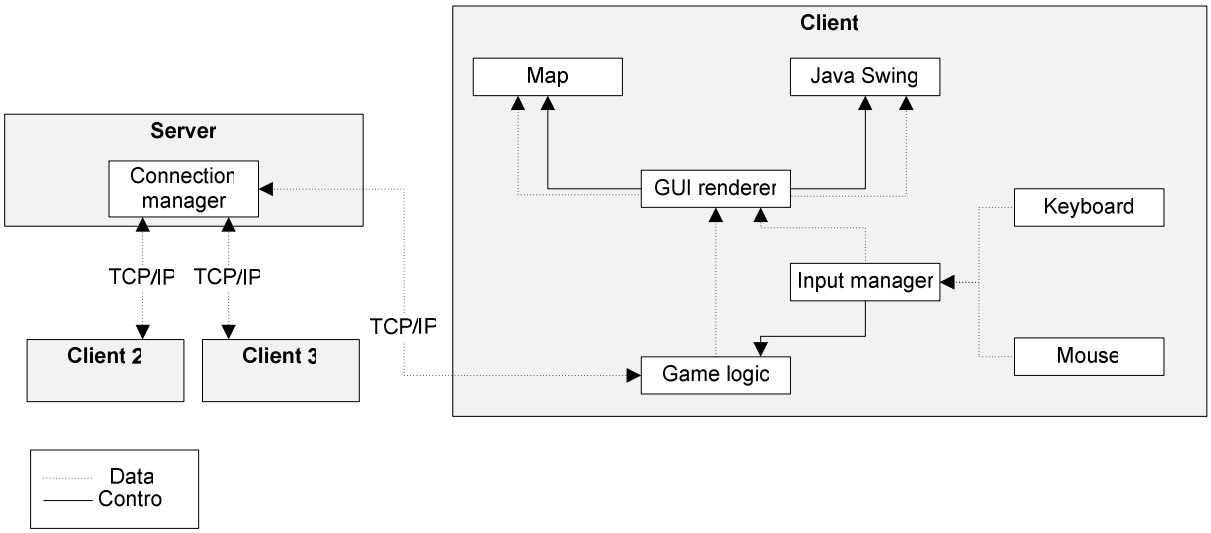


Figure 2. Data and control flow.

2.3.1. Client

The client is composed of an input manager, a GUI renderer and the game logic section.

The input manager handles and reads all the input from both the keyboard and mouse and forwards it to the game logic and the GUI renderer.

The GUI renderer visualizes the game state on the game map and in other visible components (Java Swing). It also handles some input from the input manager.

The game logic handles all input data and uses it to reflect changes in the game state. The updated game state is forwarded to the GUI so that the user sees the current game state. Some information is also sent through the network to the connected server (if playing a multiplayer game).

2.3.2. Server

The server's responsibilities are to establish incoming connections and to broadcast messages received from a client to every other client.

2.3.3. Control flow

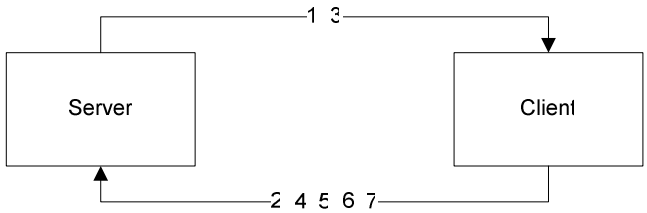


Figure 3. Control flow between server and client.

1. Accepting/Refusing connections
2. Message checking and forwarding
3. Handling network events
4. Creating a multiplayer game
5. Joining a multiplayer game
6. Connecting/Disconnecting to a server
7. Handling incoming data from server

2.3.4. Data flow

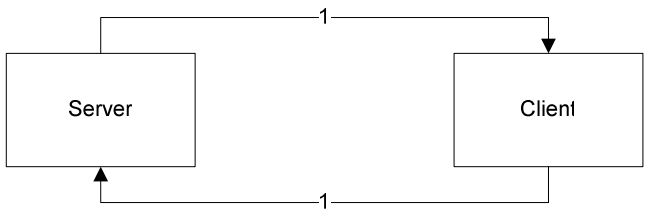


Figure 4. Data flow between server and client.

1. Reflecting game state through network

3. Design Considerations

3.1. Assumptions and Dependencies

3.1.1. Related software or hardware

The game could be played in most environments on a computer where you have access to either an internet connection or a local network for example in school, at home or at your work. If the game host has an internet connection, both players within and outside the host's local network can participate in the game. It is possible to use training mode without network connection at all.

3.1.2. Operating systems

The application will depend on that JRE 1.5 (Java Runtime Environment) is installed. JRE can be downloaded for free.

The game shall be able to run on all the major operating systems (Windows , Mac OSX, Linux, Unix) since it's will be based on java. However no testing will occur on other operating system than Windows and it will be assumed to run correctly on the others as it should thanks to JRE.

3.1.3. End-user characteristics

The end-users are likely to consist of young people, mostly guys with computer experience and more or less experience of computer games.

3.1.4. Other assumptions

The user can find other users to play against on their own. (The game will not provide any functionality to find other players)

3.1.5. Anticipated changes in functionality

Due to hardware evolution

Computer games are primarily affected by hardware evolution in the way that people want their games to support and give them benefit of e.g. their new graphics card. This won't affect this game since it's completely without focus on issues that rely on hardware performance.
However one important factor can still be identified:
Screen resolution – If users screen resolutions getting higher it will make the games user interface look very small since the game will run in the operating systems fixed resolution.

Due to changing user need

The changes in user needs in computer game in general is that users want more playable options as new characters, new levels and so on. In many commercial games changes in user needs are not considered during the life-time of the game. In other word you don't upgrade or maintains it (except errors etc.) because it will oppose the possibility to release a sequel, e.g. SAD 2. However there are some exceptions, for example online games where you pay per month and never actually buy the game. This is not the case for this game but that might change in the future and some anticipated changes in user need are:

1. Being able to play multiplayer games with more than 6 players.
2. More variety in playable options; resources, troops, buildings, etc.
3. A larger game map (playing field)
4. Users want to play the game in the web browser instead of downloading it.

Software Evolution
If the JRE version 1.5 is not available any longer due to a later versions and that version isn't backward compatible it will demand a new version of the game that rely on the either the new JRE version or that is completely independent of JRE. Since the game depends on JRE any future incompatibility between JRE and Windows or any other operating would demand a version of the game that runs independently of JRE in order to work on those operating systems.

3.2. General Constraints

3.2.1. Small maintenance costs

Expensive maintenance can be devastating to company economics. The game we supply will have low maintenance costs because of several reasons.
Impacts:
- The game shall support multiplayer network game mode without need for any online servers handling it.
- Documented and structured code to make it as easy as possible to maintain.
- Functional test cases

3.2.2. Future development should be possible

A possible business solution would be to open up an Internet portal where players meet, chat and have a graphical view of possible games to join. This would also mean supplying servers that can support this multiplayer interaction. A tournament ranking system where people can gain/lose rank points depending on their achievements could be implemented. This would make it easier to make the users spend money on the system by giving some kind of advantage, extra game functions or members only-tournaments which require real money. Commercial could then be introduced both in-game and on the Internet portal to increase profits.
Design Impact:
- Well documented and structured code.
- Functionalities that create openings for business solution as those mentioned above.

3.2.3. Verification and validation requirements (testing)

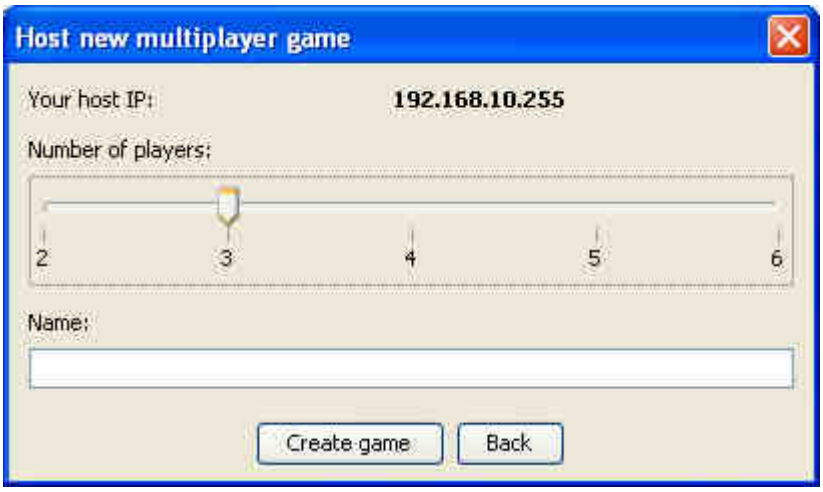
Verification and validation of the applications will be done with test cases. In order to speed up testing of softer (non-functional) requirements as the feeling of the game a training mode will be implemented without need for setting up a network connection with other players. The training mode will also function as single player mode in the final product. You will only be able to play against yourself in training mode.

3.2.4. Interface/protocol requirements

The application will be accompanied by a manual but shall still be able to learn and use without it. This will have impact on the interface requirement in order to be self explanatory.

4. Graphical user interface

4.1. Host new multiplayer game



Refers to

System requirements	6.1.5.1, 6.1.5.3
User requirements	4.4.1.1, 4.4.5.3

4.1.1. Names of the controls and fields

- labelYourHostIP
- labelIP
- labelNumberOfPlayers
- sliderNumberOfPlayers
- labelName
- textFieldName
- buttonAction
- buttonBack

4.1.2. Events, methods, or procedures that cause that form to be displayed

- new HostGameDialog(JDialog).setVisible(true);

4.1.3. Events, methods, or procedures triggered by each control

None are predetermined. Events are added from external actors (through listeners) such as the window that opens this window.

4.2. Join a multiplayer game

Refers to

System requirements	6.1.5.1, 6.1.5.2, 6.1.5.3
User requirements	4.4.5.1, 4.4.5.2

4.2.1. Names of the controls and fields

- labelHost
- textFieldHost
- labelName
- textFieldName
- buttonJoinGame
- buttonBack

4.2.2. Events, methods, or procedures that cause that form to be displayed

- new JoinGameDialog(JDialog).setVisible(true);

4.2.3. Events, methods, or procedures triggered by each control

None are predetermined. Events are added from external actors (through listeners) such as the window that opens this window.

4.3. Multiplayer mode menu

Refers to

System requirements 6.1.5.1, 6.1.5.2, 6.1.5.3
User requirements 4.4.5.1, 4.4.5.2

4.3.1. Names of the controls and fields

- buttonHost
- buttonJoin
- buttonBack

4.3.2. Events, methods, or procedures that cause that form to be displayed

- new MultiplayerModeDialog(JDialog).setVisible(true);

4.3.3. Events, methods, or procedures triggered by each control

- buttonHost - buttonHostActionPerformed(ActionEvent evt)
- buttonJoin - buttonJoinActionPerformed(ActionEvent evt)
- buttonBack - buttonBackActionPerformed(ActionEvent evt)

4.4. Start menu

Refers to
System requirements 6.2.1.4
User requirements 4.5.1.4

4.4.1. Names of the controls and fields

- buttonTrainingMode
- buttonMultiplayerMode
- buttonExit

4.4.2. Events, methods, or procedures that cause that form to be displayed

- new StartMenuDialog().setVisible(true);

4.4.3. Events, methods, or procedures triggered by each control

- buttonTrainingMode - buttonExitActionPerformed(ActionEvent evt)
- buttonMultiplayerMode-Button.MultiplayerModeActionPerformed(ActionEvent evt)
- buttonExit - buttonExitActionPerformed(ActionEvent evt)

4.5. Waiting for players to join

Refers to

System requirements	6.1.5.2
User requirements	4.4.5.2

4.5.1. Names of the controls and fields

- labelPlayer
- labelName
- buttonToggleReady
- buttonLeaveGame

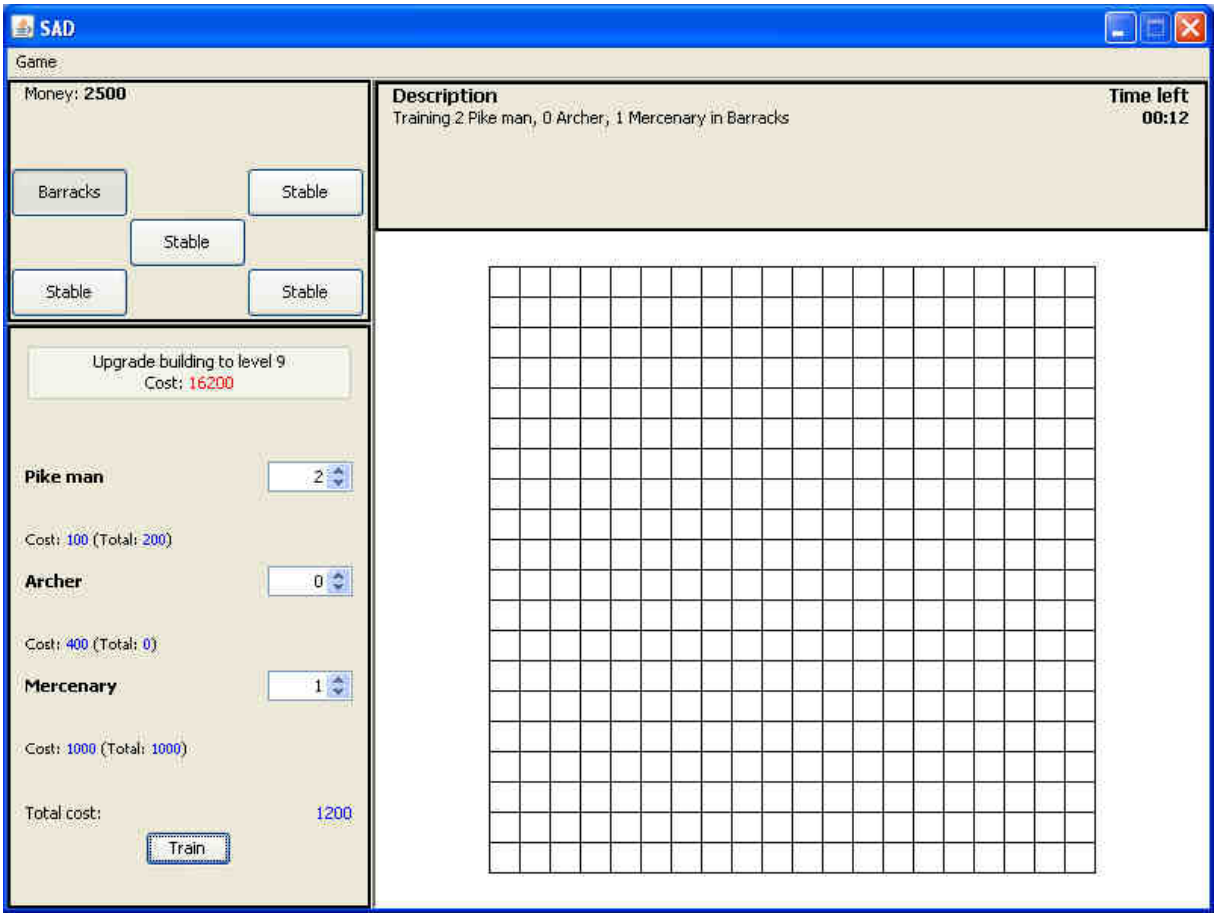
4.5.2. Events, methods, or procedures that cause that form to be displayed

- new MultiplayerWaitingReadyDialog(JDialog, int).setVisible(true);

4.5.3. Events, methods, or procedures triggered by each control

None are predetermined. Events are added from external actors (through listeners).

4.6. Main game window



Refers to

System requirements	6.1.1.2, 6.1.3.2, 6.1.1.4, 6.1.2.*, 6.1.4.1
User requirements	4.4.1.1, 4.4.1.2, 4.4.2.*, 4.4.4.* except Fog of war

4.6.1. Names of the controls and fields

- menu
- menuGame
- jSplitPane1
- jSplitPane2
- jSplitPane3
- jSplitPane4
- panelBuilding
- panelSelection
- panelInfo
- panelMap

4.6.2. Events, methods, or procedures that cause that form to be displayed

- new MainWindow().setVisible(true);

4.6.3. Events, methods, or procedures triggered by each control

None are predetermined. Events are added from external actors (through listeners) such as the window that opens this window.

5. Design Details

5.1. Class Responsibility Collaborator (CRC) Cards

Class Army	
Responsabilities:	**Collaborators:**
Know the home village of the army	Village
Keeping troops together as one unit	Troop
Knowing the army troop amounts	TroopType
To merge different armies from the same team	

Interface Building	
Responsabilities:	**Collaborators:**
Keeps track of building name, level, costs of upgrading	Village
Keeps track of its own building panel where the buildings actions are display such as different building options	BuildableItem
Decides which buildable items that are available	

Interface BuildableItem	
Responsabilities:	**Collaborators:**
Describes an item that can be built in a arbitrary building	
Checks required building level to build building	
Building cost, name and building time	

Interface Combat	
Responsabilities:	**Collaborators:**
Defines the combat inputs	CombatCalculator
	Army

Interface CombatCalculator	
Responsabilities:	**Collaborators:**
Calculates the outcome of a combat	Combat

Class Map	
Responsabilities:	**Collaborators:**
Keeps track of and displays the game terrain, villages and armies	Army
Provides possibility to move	Village

Interface Race	
Responsabilities:	**Collaborators:**
Creates and stores a Troopinfofactory that be used by the team	TroopInfoFactory
Race name	

Class Team	
Responsabilities:	**Collaborators:**
Keeps track of team information such as team name, team race etc. Keeps track of team resources	Race

Class Troop	
Responsabilities:	**Collaborators:**
Keeping track of troop characteristics such as amount, attack points and defence points	TroopInfo TroopType

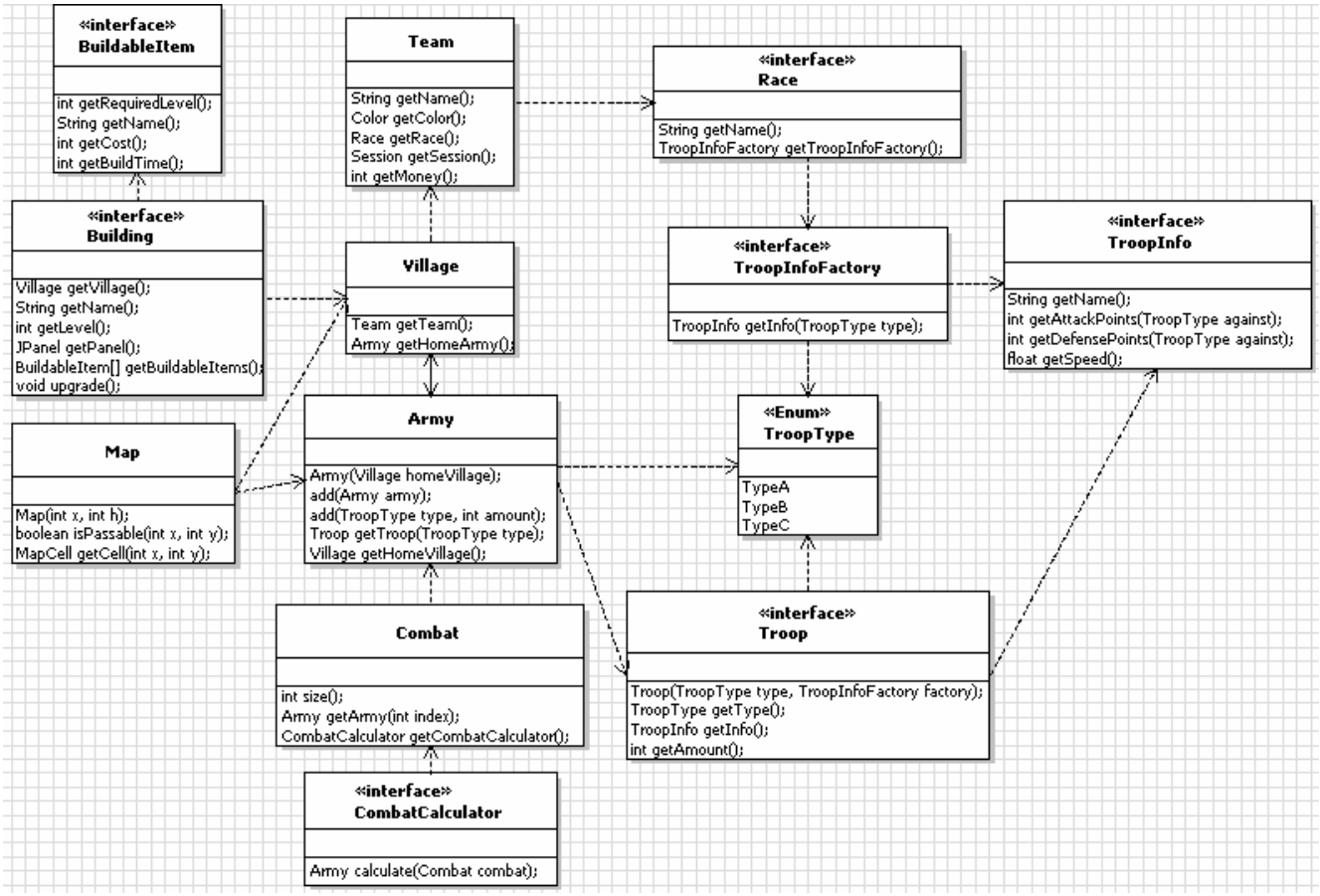
Interface TroopInfo	
Responsabilities:	**Collaborators:**
Keeps track of attributes for a specific kind of troop	

Interface TroopInfoFactory	
Responsabilities:	**Collaborators:**
Defines different attributes for different types of troops	TroopInfo

Enum TroopType	
Responsabilities:	**Collaborators:**
Classifies different troops	TroopInfoFactory Troop Army

Class Village	
Responsabilities:	**Collaborators:**
Keeps track of the village owner Keeps track of the village current army Host buildings	Army Team

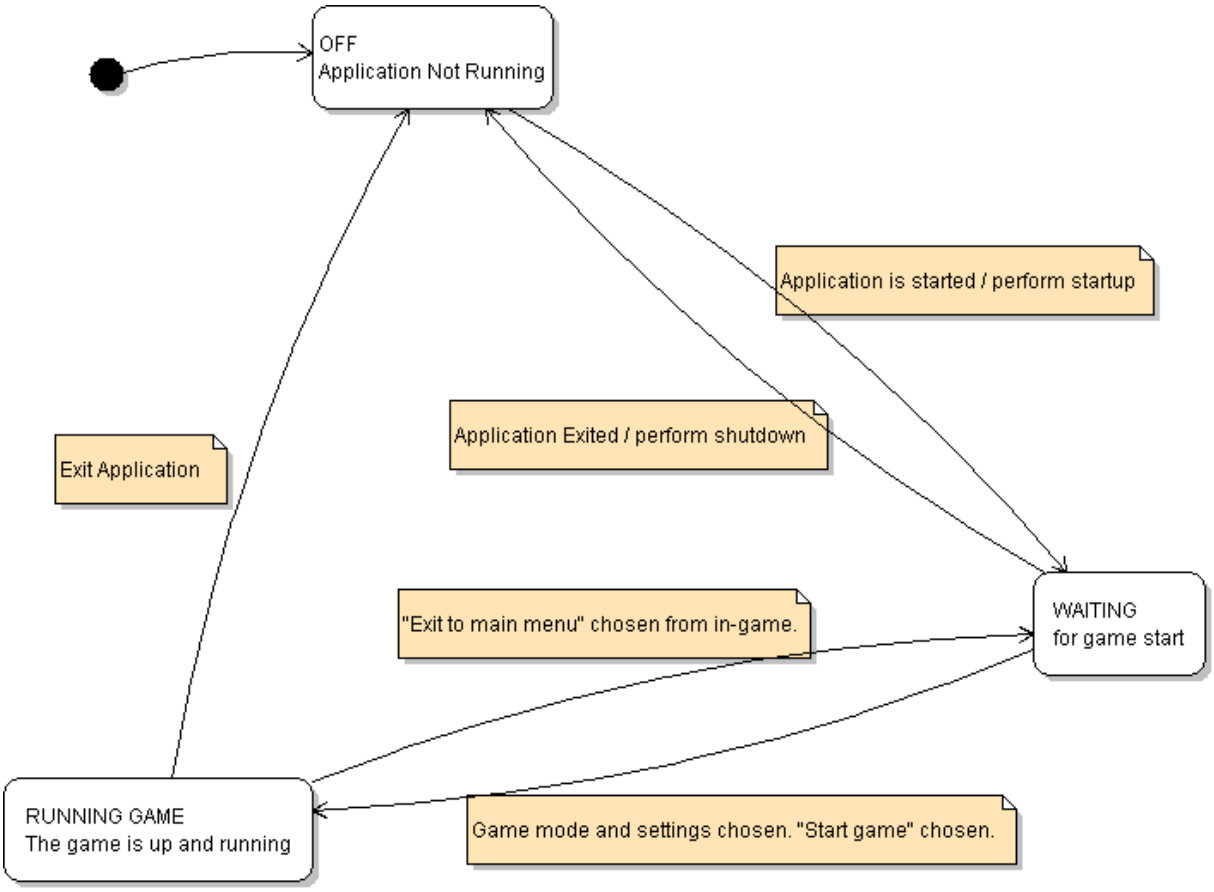
5.2. Class Diagram



Figur 5. Class Diagram

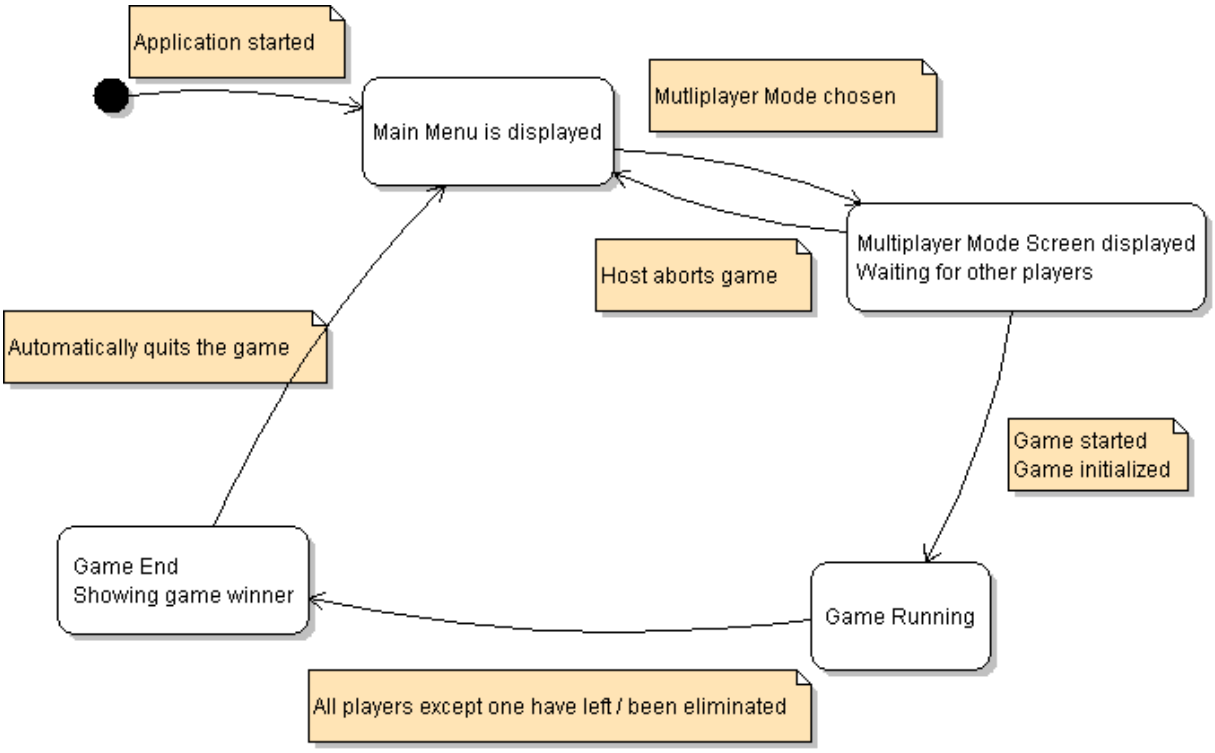
5.3. State Charts

5.3.1. Application state overview



Figur 6. State Chart 1 - Overview

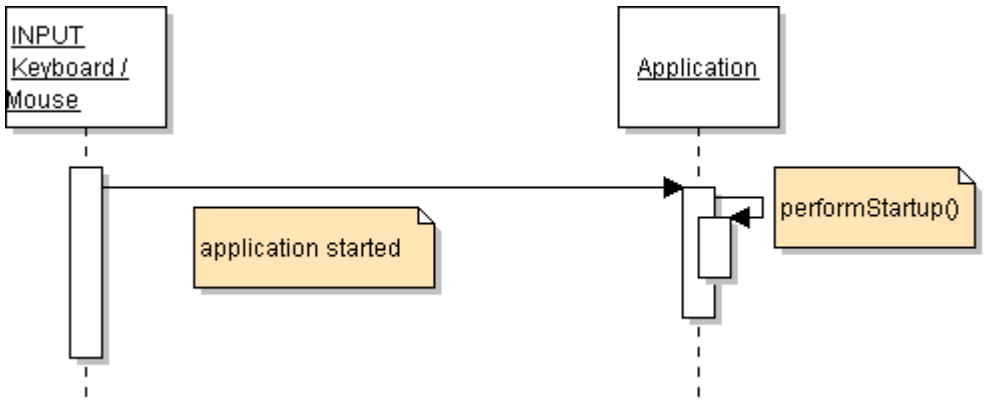
5.3.2. Application flow



Figur 7. State Chart 2 – Staring, running and ending a game

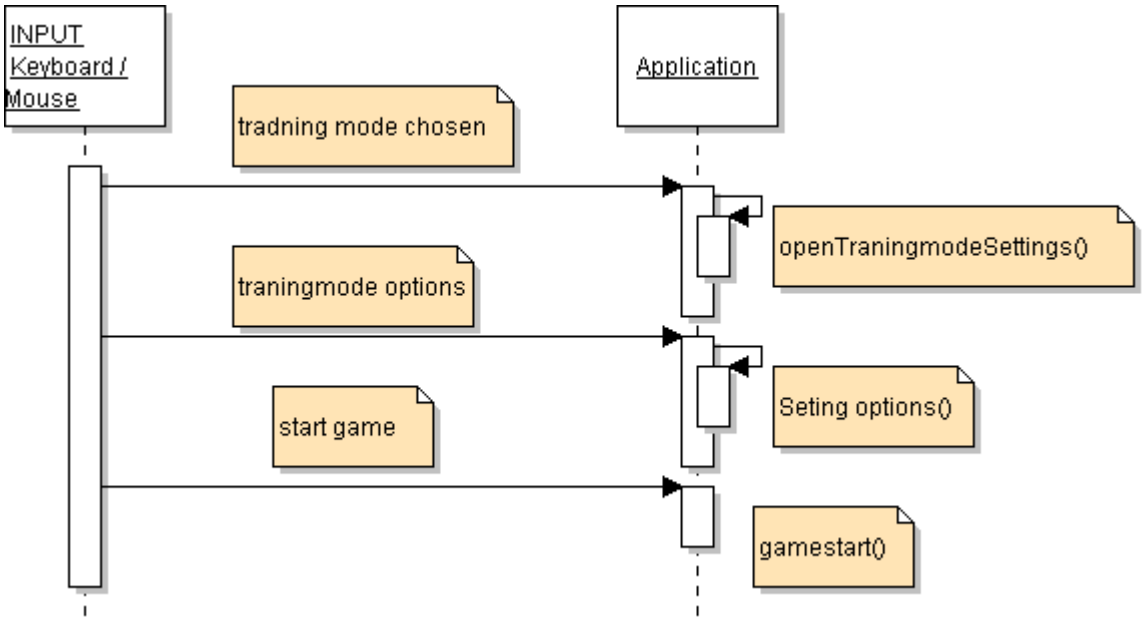
5.4. Interaction Diagrams

5.4.1. Launch game application



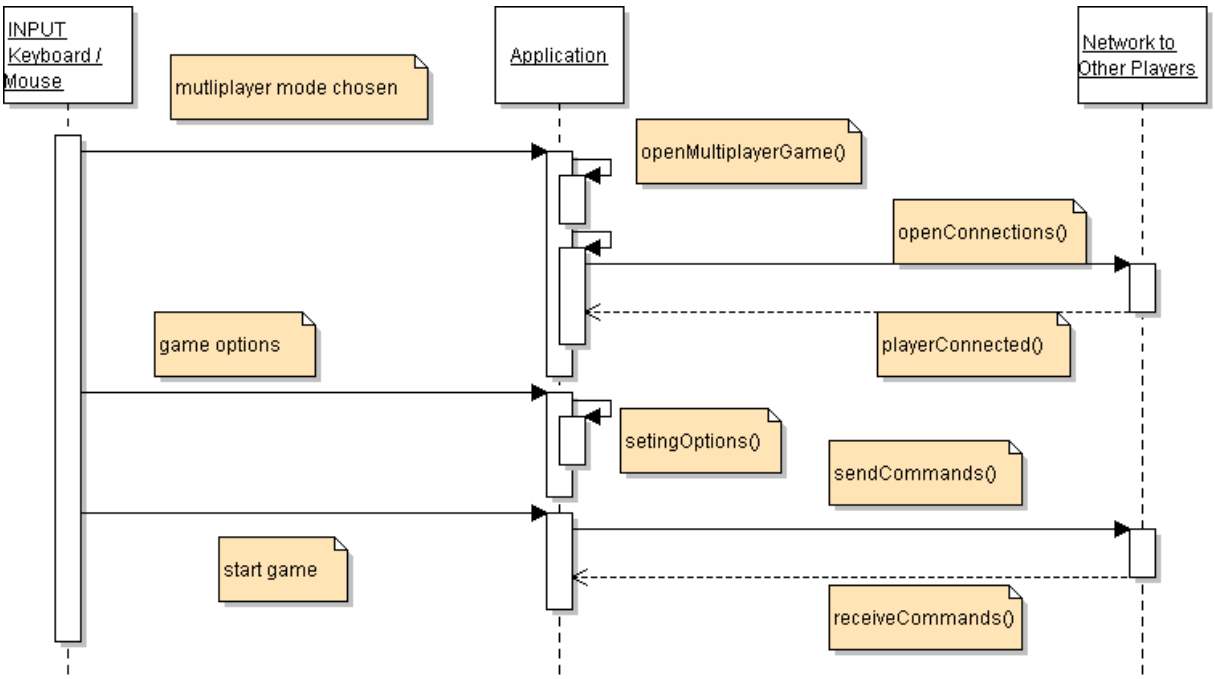
Figur 8. Launch game

5.4.2. Start a training mode game



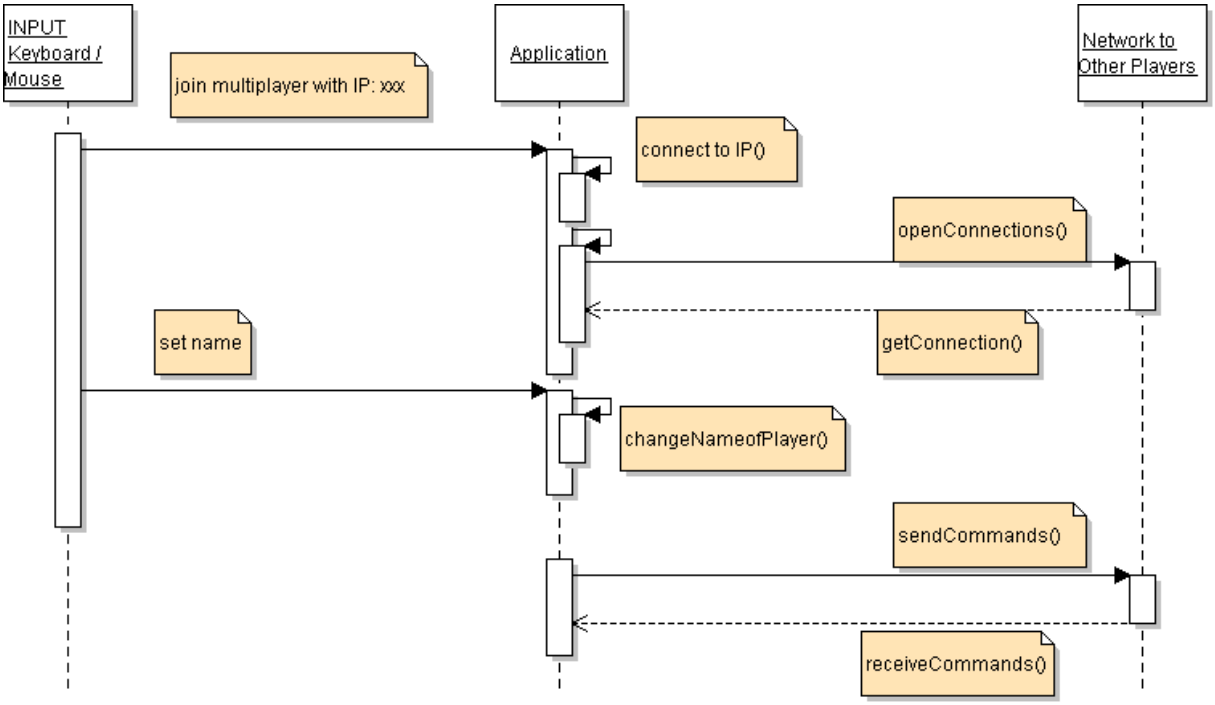
Figur 9. Training mode

5.4.3. Host a multiplayer game



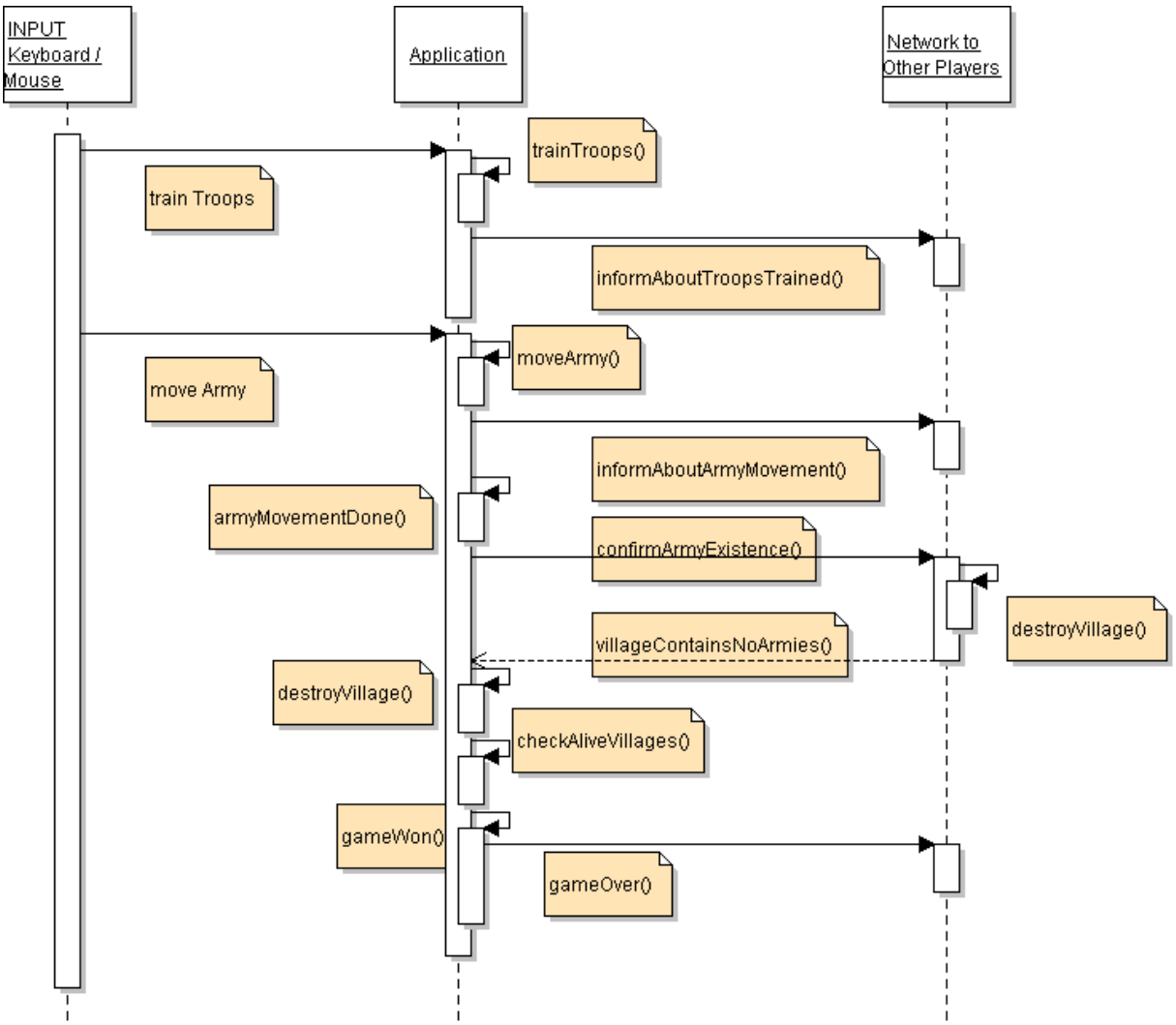
Figur 10. Multiplayer game hosting

5.4.4. Join a multiplayer game



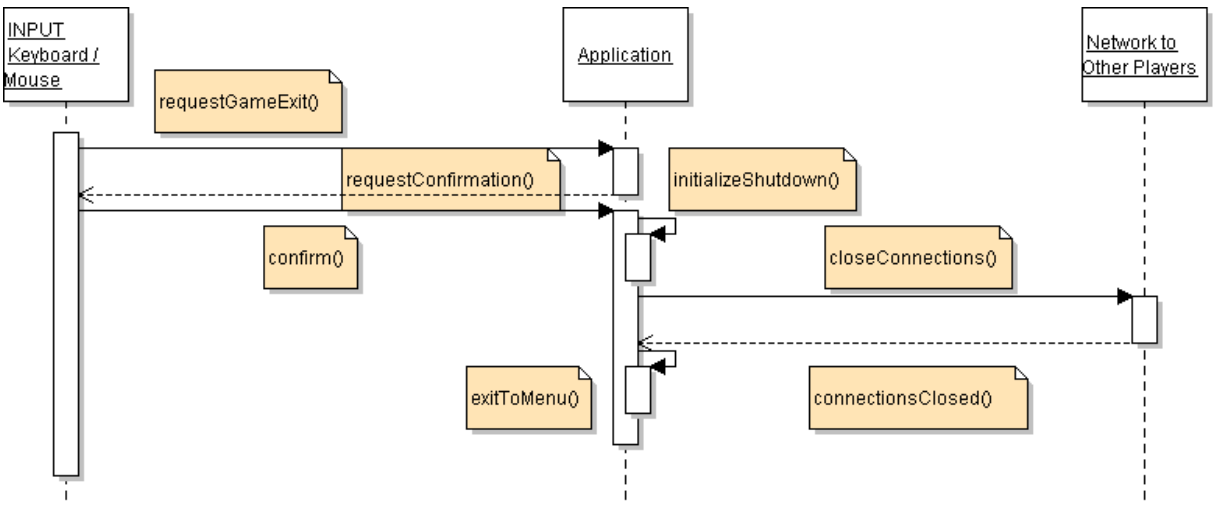
Figur 11. Join a multiplayer game

5.4.5. Win a multiplayer game round



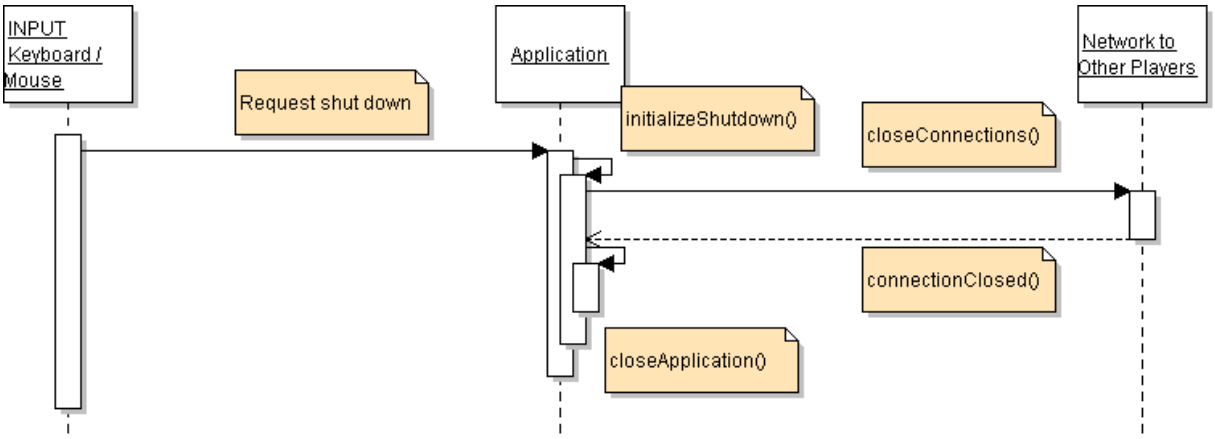
Figur 12. Win multiplayer game

5.4.6. Leave a multiplayer game



Figur 13. Leave a multiplayer game

5.4.7. Application shutdown



Figur 14. Game shutdown

5.5. Detailed Design

Provided below are the detailed descriptions of each class of the logical part of the game application.

5.5.1. Class Army

References: Functional Requirements
Army consists of troops 4.4.3.3.
Conflict of armies 4.4.3.6
Move armies around the map 4.4.3.7
Armies consists of troops 6.1.3.3.
Perform movement of player armies 6.1.3.6.

Field:

Attribute: unitA
Type: int
Usage: Is used to know how many troops of the type unitA this army consist of.

Attribute: unitB
Type: int
Usage: Is used to know how many troops of the type unitB this army consist of.

Attribute: unitC
Type: int
Usage: Is used to know how many troops of the type unitB this army consist of.

Attribute: speed
Type: float
Usage: Is used to know the speed of the army, the speed is equal to the slowest troop in the army.

Methods:

getSpeed()

Method Name: getSpeed()
Parameters: -
Return Value: int speedValue
Description: This method is used to get the speed of the army
Data structures: -
Pre-condition: The army consists of at least 1 unit of any kind.
Validity Checks, Errors, and other Anomalous Situations: -
Post-condition: The team has the speed of the slowest unit in the army.
Called by: Pathfinder.findPath()
Calls: -

setSpeed(int unitA, int unitB, int unitC)

Method Name: setSpeed(int unitA, int unitB, int unitC)
Parameters: unitA – how many troops of the unitA
unitB – how many troops of the unitB
unitC – how many troops of the unitC
Return Value: -
Description: This method is used to set the speed of the army.
Pre-condition: The speed of the army is set to the slowest unit in this army.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: -
Called by: Building.trainTroops(), Map.formMergeArmy
Calls: Army.addArmy(int unitA, int unitB, int UnitC)

initializeCombat(Army1, Army 2)

Method Name: initializeCombat(Army1, Army2)
Parameters: Two Armies
Return Value: One winning army
Description: This method starts a combat
Pre-condition: Two armies move to same cell of the map..
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: One Army is destroyed
Called by: -
Calls: CombatCalculator.calcuateCombat(Army1, Army2)

5.5.2. Interface Building

References: Functional Requirements
Building slots 4.4.2.1.
Differnet types of buildings 4.4.2.2.
Construct Buildings 4.4.2..3.
Upgrade Buildings 4.4.2.4
Upgrade Buildings 6.1.2.3.

Fields:

Methods:

getVillage()

Method Name: getVillage()
Parameters: -
Return Value: Village
Description: The method returns the village a building belongs to
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The village is returned
Called by: TotalCostPanel.buildItemNumberChanged()
Calls: Team.addMoney()

getName()

Method Name: getName()
Parameters: -
Return Value: String name
Description: The method returns the building name
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The name of the building is returned
Called by: UpgradeBuildingPanel.buttonUpgradeActionPerformed(), BuildingPanel.update(), BuildingButton()
Calls:

getLevel()

Method Name: getLevel()
Parameters: -
Return Value: int
Description: The method returns the building level
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The level of the building is returned
Called by: UpgradeBuildingPanel.updateButton(),UpgradeBuildingPanel.buttonUpgradeActionPerformed(),BuildingPanel.update()
Calls: -

isUpgradable()

Method Name: isUpgradable()
Parameters: -
Return Value: boolean
Description: The method checks if the building is upgradable
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: A true or false is returned
Called by: BuildingPanel.update()
Calls: -

getUpgradeCost()

Method Name: getUpgradeCost()
Parameters: -
Return Value: int
Description: The method checks the upgrade cost of the building
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The cost of upgrade is returned
Called by: UpgradeBuildingPanel.updateButton()
Calls: -

getUpgradeTime()

Method Name: getUpgradeTime()
Parameters: -
Return Value: int
Description: The method checks the upgrade time of the building
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The time of upgrade is returned
Called by: UpgradeBuildingPanel.buttonUpgradeActionPerformed()
Calls: -

getBuildableItems()

Method Name: getBuildableItems()
Parameters: -
Return Value: BuildableItem[ ]
Description: The returns all buildable buildings for a specified village
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The buildable buildings are returned
Called by: BuildingPanel.update()

Calls: -

5.5.3. Interface BuildableItem

References: **Functional Requirements**
Different types of buildings 4.4.2.2.
Construct different kinds of buildings 6.1.2.2.

Methods:

getRequiredLevel()

Method Name: getRequiredLevel
Parameters: -
Return Value: int - The minimum level of a building required to build this item
Description: Returns the required minimum level
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The required level is returned
Called by: BuildingPanel.update
Calls: -

getName()

Method Name: getName
Parameters: -
Return Value: String - The name of the item
Description: Returns the name of the item
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The name is returned
Called by: BuildingPanel.update, BuildItemPanel constructor
Calls: -

getCost()

Method Name: getCost
Parameters: -
Return Value: int - The cost to build this item
Description: Returns the cost
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The cost is returned
Called by: TotalCostPanel.getTotalCost, BuildItemPanel.updateCost

Calls: -

getBuildTime ()

Method Name: getBuildTime
Parameters: -
Return Value: int - The time it takes to build on of this item
Description: Returns the build time
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The cost build time is returned
Called by: -
Calls: -

5.5.4. Class Combat

References: Functional Requirements
Conflict of armies 6.1.3.5.

Methods:

Size()

Method Name: Size()
Return Value: int size
Description: Defines the combat size in number of participants
Pre-conditions: To armies has initialized a combat
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: -
Called by: Army.initializeCombat()
Calls: -

getArmy(int index)

Method Name: getArmy(int index)
Return Value: Army Army
Description: Collects the combat participants
Pre-conditions: To armies has initialized a combat
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: -
Called by: Army.initializeCombat()
Calls: -

getCombatCalculator()

Method Name: getCombatCalculator ()
Return Value: CombatCalculator CombatCalculator

Description: Collects the combat logic from Combat calculator
Pre-conditions: To armies has initialized a combat
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: -
Called by: Army.initializeCombat()
Calls: -

5.5.5. Class CombatCalculator

References: Functional Requirements
Conflict of armies 4.4.3.6.
Attack villages with armies 4.4.3.5.
Conflict of armies 6.1.3.5.

Methods:

calculateCombat(Army armyB, Army armyC)

Method Name: calculateCombat(Army armyB, Army armyC)
Parameters: armyB – An army
armyC – An army of another player
Return Value: Army winningArmy
Description: Calculates who is the combat's winning army, dependant on army factors, troops relations and some random factors.
Pre-conditions: Two different player's armies meet at the same map cell.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: One army has been eliminated,
Called by: Cell.calculateCombat(Army armyB, Army armyC)
Calls: -

5.5.6. Class Map

References: Functional Requirements
Game map 4.4.4.1.
Maps cells 4.4.4.2.
Perform movement of player armies 6.1.3.6.
Provide interactive game map 6.1.4.1.
Construct a map with different kinds of map cells 6.1.4.2.

Fields

Attribute: grid
Type: Cell[ ] [ ]
Usage: All map cells are stored in this cell-matrix.

Attribute: randomizer
Type: Random
Usage: Machine for producing random seeds for the map creation process. This is needed to make each game map unique.

Methods:

generateRivers()

Method Name: generateRivers()
Parameters: -
Return Value: -
Description: Generates and randomizes amount of rivers that should exist on the map. Also randomizes how long each river should be.
Data structures: -
Pre-conditions: A game is launched and a map is needed.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: A map with a cell matrix full of different cells is created.
Called by: Game.generateMap()
Calls: createRiver(int n)

createRiver(int riverSize)

Method Name: createRiver(int riverSize)
Parameters: riverSize – specifies how many cells this river should be
Return Value: -
Description: Randomizes rivers positioning and generates the related cells in the cell matrix.
Data structures: -
Pre-conditions: Rivers are being created.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: A river is created.
Called by: generateRivers()
Calls: -

generateRocks()

Method Name: generateRocks()
Parameters: -
Return Value: -
Description: Generates and randomizes amount of rocks/mountains that should exist on the map. Also randomizes how big each rock should be.
Data structures: -
Pre-conditions: A game is launched and a map is needed. Rivers are created.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: A map with a cell matrix full of different cells is created.
Called by: Game.generateMap()
Calls: createRock(int n)

createRock(int rockSize)

Method Name: createRock(int rockSize)
Parameters: rockSize – specifies how many cells this rock should be
Return Value: -

Description: Randomizes rock positioning and generates the related cells in the cell matrix.
Data structures: -
Pre-conditions: Rocks are being created.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: A rock is created.
Called by: generateRocks()
Calls: -

generatePlains()

Method Name: generatePlains()
Parameters: -
Return Value: -
Description: Generates and creates plains cells in the empty cells of the cell-matrix.
Pre-conditions: A game is launched and a map is needed. Rivers and rocks are created.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: A map with a cell matrix full of different cells is created.
Called by: Game.generateMap()
Calls: -

5.5.7. Interface Race

References: Functional Requirements
Playable Races 4.4.1.5.
Choosing a player race 6.1.1.5.

Methods:

getName()

Method Name: getName()
Parameters: -
Return Value: The name of the race
Description: The method returns the name of the race
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The name of the race is returned
Called by: BuildingPanel update
Calls: -

5.5.8. Class Team

References: Functional Requirements
Player specific colors 4.4.5.6
Assign each player a unique player name 6.1.5.1.
Player specific colors 6.1.5.6

Fields

Attribute: name
Type: String
Usage: Every team has a unique name to separate them from each other.

Attribute: money
Type: int
Usage: This is used to keep track of a teams money they can spend.

Attribute: color
Type: Color
Usage: Every team has a unique team color to separate them from other teams.

Attribute moneyListeners
Type: List<MoneyListener> - List of moneyListeners

Methods:

addMoney(int money)

Method Name: addMoney(int money)
Parameters: money – how much you should add to the team money.
Return Value: -
Description: This method is used to add money to the team. The amount of money added is told by the parameter.
Data structures: -
Pre-condition: A team has gain money in some way and need to add it to there team money.
Validity Checks, Errors, and other Anomalous Situations: -
Post-condition: The teams money has change.
Called by: Village.increadeMoney()
Calls: -

5.5.9. Interface Troop

References: Functional Requirements
Different types of troops 6.1.3.1.
Train military troops 6.1.3.2

Methods:

Troop(Troop type, TroopInfoFactory factory**)**

Method Name: Troop
Parameters: Troop type, TroopInfoFactory factory
Return Value: -
Description: The method defines a troop
Data structures: -

Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: -
Called by: BuildingPanel
Calls: -

getType()

Method Name: getType
Parameters: -
Return Value: TroopType
Description: The method
Data structures:
Pre-conditions: Two or more troops exists and an army is about to form or a combat takes place.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The type of the troop is retuned.
Called by: CombatCalculator, BuildingPanel, Army
Calls: -

getInfo();

Method Name: getInfo
Parameters: -
Return Value: TroopInfo
Description: The method returns info about the troop.
Data structures: -
Pre-conditions: Two or more troops exist and an army is about to form or a combat is about to take place.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: Troopinfo is returned
Called by: CombatCalculator, Army
Calls: -

getAmount()

Method Name: getAmount
Parameters:
Return Value: Int
Description: Returns the size of the troop
Data structures:
Pre-conditions: Two or more troop exists and an army is about to form.
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The amount of troops is returned
Called by: CombatCalculator, Army
Calls: -

5.5.10. Interface TroopInfo

Methods:

getName()

Method Name: getName()
Parameters: -
Return Value: Troop name
Description: Returns the troop name
Data structures: -
Pre-conditions: The player has a troop
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The name of the troop is returned
Called by: TroopInfoFactory.getInfo(TroopType type)
Calls: -

getAttackPoints(TroopType against)

Method Name: getAttackPoints(TroopType against)
Parameters: TroopType
Return Value: Int
Description: Assembles the troop attack points
Data structures: -
Pre-conditions: The player has a troop
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The attack points of the troop is returned
Called by: TroopInfoFactory.getInfo(TroopType type)
Calls: -

getDefensePoints(TroopType against)

Method Name: getDefensePoints(TroopType against)
Parameters: TroopType
Return Value: Int
Description: Assembles the troop defense points
Data structures: -
Pre-conditions: The player has a troop
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The defense points of the troop is returned
Called by: TroopInfoFactory.getInfo(TroopType type)
Calls: -

getSpeed()

Method Name: getSpeed()
Parameters: -
Return Value: float

Description: Returns the troop speed
Data structures: -
Pre-conditions: The player has a troop
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The speed of the troop is returned
Called by: TroopInfoFactory.getInfo(TroopType type)
Calls: -

5.5.11. Interface TroopInfoFactory

Methods:

getInfo(TroopType type)

Method Name: getInfo(TroopType type)
Parameters: -
Return Value: TroopInfo
Description: Returns the full troop information
Data structures: -
Pre-conditions: The player has a troop
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The full info of the troop is returned
Called by: -
Calls: TroopType

5.5.12. Class Village

References: **Functional Requirements**
Player controlled villages 4.4.1.1.
One village per player 4.4.1.2
Establish village control in multiplayer mode 6.1.1.1.
A player shall have one village 6.1.1.2.
Buildings slots 6.1.2.1

Fields:

Attribute: team
Type: Team
Usage: Assigns a village name

Attribute: homeArmy
Type: Army
Usage: The stationary army where new built troops are gathered

Methods:

increaseMoney()

Method Name: increaseMoney

Parameters: -
Return Value: -
Description: The method increases the player money production
Data structures: -
Pre-conditions: -
Validity Checks, Errors, and other Anomalous Situations: -
Post-conditions: The player money amount is increased
Called by: MainWindow.createThread()
Calls: Team.addMoney()

5.6. Cross-referenced index

5.6.1. User Functional Requirements

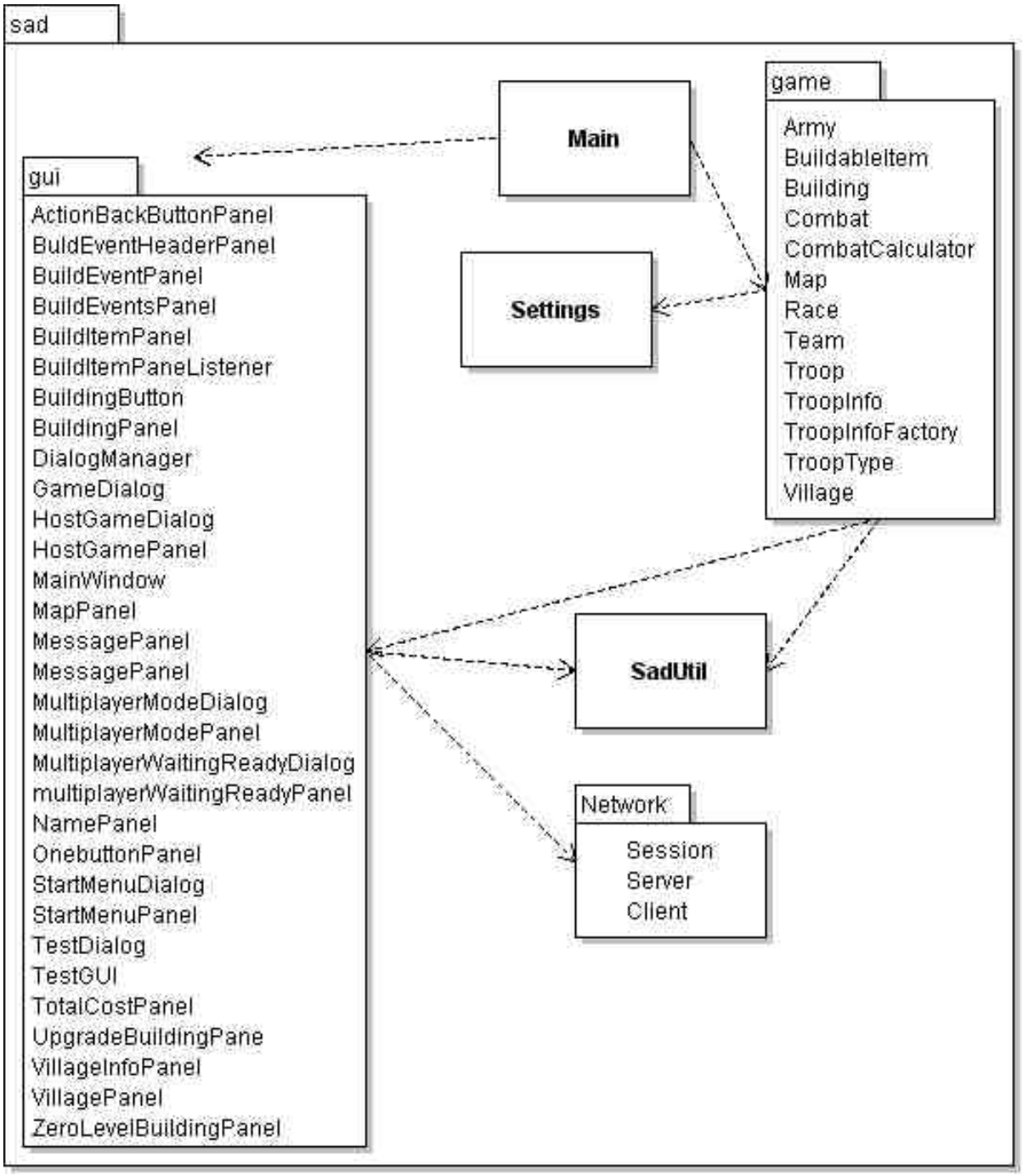
Army consists of troops 4.4.3.3.	5.5.1. Class Army
Attack villages with armies 4.4.3.5	5.5.5. Class CombatCalculator
Building slots 4.4.2.1.	5.5.2 Interface Building
Conflict of armies 4.4.3.6	5.5.1. Class Army, 5.5.5. Class CombatCalculator
Construct Buildings 4.4.2.3.	5.5.2 Interface Building
Different types of buildings 4.4.2.2	5.5.2 Interface Building, 5.5.3. Interface BuildableItem
Game map 4.4.4.1.	5.5.6. Class Map
Maps cells 4.4.4.2.	5.5.6. Class Map
Move armies around the map 4.4.3.7	5.5.1. Class Army
One village per player 4.4.1.2	5.5.12. Class Village
Playable Races 4.4.1.5.	5.5.7. Interface Race
Player controlled villages 4.4.1.1.	5.5.12. Class Village
Player specific colors 4.4.5.6	5.5.8. Class Team
Upgrade Buildings 4.4.2.4	5.5.2 Interface Building

5.6.2. System Functional requirements

A player shall have one village 6.1.1.2.	5.5.12. Class Village
Armies consist of troops 6.1.3.3.	5.5.1. Class Army
Assign each player a unique player name 6.1.5.1.	5.5.8 Class Team
Buildings slots 6.1.2.1	5.5.12. Class Village
Choosing a player race 6.1.1.5.	5.5.7 Interface Race
Conflict of armies 6.1.3.5.	5.5.4. Class Combat, 5.5.5. Class CombatCalculator
Construct a map with different kinds of map cells 6.1.4.2.	5.5.6. Class Map
Construct different kinds of buildings 6.1.2.2	5.5.3. Interface BuildableItem
Different types of troops 6.1.3.1.	5.5.9. Interface Troop
Establish village control in multiplayer mode 6.1.1.1.	5.5.12. Class Village
Perform movement of player armies 6.1.3.6.	5.5.1. Class Army
Perform movement of player armies 6.1.3.6.	5.5.6. Class Map

Player specific colors 6.1.5.6	5.5.8 Clas Team
Provide interactive game map 6.1.4.1.	5.5.6. Class Map
Train military troops 6.1.3.2	5.5.9. Interface Troop
Upgrade Buildings 6.1.2.3.	5.5.2 Interface Building

5.7. Package Diagram



Figur 15. Package Diagram

6. Functional Test Cases

6.1. Different types of troops

Description: Test to see if you can create different types troops

Reference: Functional Requirements
User - 4.4.3.1
System – 6.1.3.1

Input: None
Expected output: Different type of troops are constructed

Procedure:
1. Upgrade village barracks
2. Construct troops of the at least two different types that now is possible.
3. Compare that the two troops are different by comparing the info that appears when clicking on them. E.g. names.

6.2. Train military troops

Description: Test if it is possible to build troops

Reference: Functional Requirements
User - 4.4.3.2
System - 6.1.3.2

Input: Amount of troops
Expected output: Troops are constructed

Procedure:
1. Enter the barracks
2. Click to build a troop
3. Enter the amount of troops need
4. Push the button to start building troops
5. Hold the mouse over the village
6. A pop-up appears showing the troops that have been built.

6.3. Resources

Description: Test that resources are collected.

Reference: Functional Requirements
User - 4.4.1.3, 4.4.1.4
System – 6.1.1.4

Input: None.
Expected output: Resources are increased at a constant rate.

Procedure:

4. Choose “Training mode” from the start menu.
5. Wait for the game to start.
6. Wait a few seconds and you will see that your resources are increased.

6.4. Different kinds of buildings

Description: Test that different buildings can be constructed and upgraded.

Reference: Functional Requirements
User - 4.4.2.2, 4.4.2.3, 4.4.2.4
System – 6.1.2.2.

Input: A village
Expected output: Upgraded buildings; barrack, stable, bank, stronghold and hospital

Procedure:

1. Create a new barrack, stable, bank, stronghold and hospital
2. Upgrade each building
3. Control that the buildings has been upgraded by reading the text tool tip and check what level they are.

6.5. Map

Description: Test the map and that it has different cells.

Reference: Functional Requirements
User - 4.4.4.1, 4.4.4.2
System – 6.1.4.1, 6.1.4.2

Input: None.
Expected output: A map with different cells.

Procedure:

1. Choose “Training mode” from the start menu.
2. Wait for the game to start.
3. A random map will be generated and drawn into the map panel.

6.6. Connect to a multiplayer game

Description: Test to connect to a multiplayer game.

Reference: Functional Requirements
User - 4.4.5.2
System – 6.1.5.2

Input: A player name and a host name.
Expected output: A connection to a multiplayer game is made OR

No connection is made because no multiplayer game exists at the location.

Procedure:

1. Choose “Multiplayer mode” from the start menu.
2. Choose “Join a game” from the menu.
3. Enter a player name and an ip/host name to connect to.
4. Click “Connect”.

6.7. Multiplayer game settings

Description: Test to create a multiplayer game with different settings.

Reference: Functional Requirements
User – 4.4.5.1, 4.4.5.3
System – 6.1.5.1, 6.1.5.3

Input: Different game settings and a player name.
Expected output: A multiplayer game is started.

Procedure:

1. Choose “Multiplayer mode” from the start menu.
2. Choose “Host a game” from the menu.
3. Enter a player name and change the game settings.
4. Click “Create”.

6.8. Player specific colors

Description: Tests that all players have unique colors.

Reference: Functional Requirements
User - 4.4.5.6
System – 6.1.5.6.

Input: An initialized game, several players
Expected output: That all teams/players have different colors

Procedure:

1. Chose one of the teams
2. Look at team village on the map
3. Compare village color with the other players village color
4. Decide whether any teams have the same colors

6.9. Assign each player a unique player name

Description: Test to see if you can assign unique player name.

Reference: Functional Requirements
User - 4.4.5.1
System - 6.1.5.1.

Input: Player names
Expected output: Messages informing that the name is already taken.
A game with two or more players with different names.

Procedure:
1. Create a new game with two or more players
2. Choose a name for the first player
3. Try to choose the same name for the second player.
4. Check that you get an error message.
5. Try to choose a different name.

6.10. A player has a race

Description: Tests that a player has a valid race.

Reference: Functional Requirements
User – 4.4.1.5
System – 6.1.1.5

Input: A player
Expected output: The player race

Procedure:
1. Create a team
2. Look at village panel at the top left of the screen
3. The player race should be stated here

6.11. A player shall have one village

Description: Test to see that a player has one village.

Reference: Functional Requirements
User – 4.4.1.2.
System – 6.1.1.2

Input: None.
Expected output: A game is created and the player has one village.

Procedure:
1. Choose “Training mode” from the start menu.
2. Wait for the game to start.
3. Watch the map and look for your one and only village.

6.12. Move armies around the map

Description: Test that an army can be moved.

Reference: Functional Requirements

User - 4.4.3.7.
System – 6.1.3.6.

Input: An input and mouse actions.
Expected output: An army is moved.

Procedure:
1. Left click on one of your armies on the map.
4. Right click on another cell in the map.
5. Watch the army move.

6.13. Join two armies

Description: Tests that two armies could be joined

Reference: Functional Requirements
User - 4.4.3.4.
System - 6.1.3.3

Input: Two armies
Expected output: One army

Procedure:
1. Check the troop amounts within the two armies
2. Join the two armies by moving them to the same map cell
3. Check that the troops amounts within the new single army is consistent
4. Move the new army to a new map cell to see that the army acts as one.

6.14. Armies never separate

Description: Tests that an army never could be separated

Reference: Functional Requirements
User - 4.4.3.4.
System - 6.1.3.3

Input: Two armies
Expected output: The army has moved to another position and still consists of the amount of troops.

Procedure:
5. Join two armies by moving them to the same map cell
6. Check map position of the army
7. Check the troops size of the armies different troops
8. Move the army one step.
9. Check if the troops size are consistent

6.15. Attack village with armies

Description: Test that a village can be attacked.

Reference: Functional Requirements

User - 4.4.3.5
System - 6.1.3.4.

Input: An army and a hostile village.
Expected output: The hostile village has been attacked.

Procedure:

1. Left click on one of your armies on the map.
2. Right click on a hostile village on the map.
3. Watch the army move.
4. When the army reaches the village:
    a. The village is destroyed if the army is big enough.
    b. If the army is too small: The village is partially destroyed and the army is destroyed.

6.16. Conflict of armies

Description: Test to see that two armies battles with each other when entering the same map cell.
Reference: Functional Requirements
User - 4.4.3.6
System - 6.1.3.5

Input: Two armies from different players
Expected output: One army remaining.

Procedure:

1. Move one of the armies to the spot on the map where the other army is located.
2. Check that there is only one army left.