UbonGo*Dream*

Architectural Description

TDT 4240 - Software Architecture

COTS: Android

Mathias Mikalsen

Julia Bröhan

Matus Smykala

Sindre O. Rasmussen

Albert Hambardzumyan

Group 19

Primary quality attribute

Modifiability

Secondary quality attribute

Usability

Spring 2016

Table of Contents

- 1. Introduction
- 2. Architectural Drivers / ASRs
  - 2.1 Functional Requirements
    - 2.1.1 Network Multiplayer
    - 2.1.2 Multiple Game Screens
    - 2.1.3 Sheep Library
  - 2.2 Quality Requirements
    - 2.2.1 Modifiability
    - 2.2.2 usability
  - 2.3 Business Requirements
    - 2.3.1 Developer experience
    - 2.3.2 Time limitation
- 3. Stakeholders and Concerns
    - 3.1 Developers
    - 3.2 Players
    - 3.3 Course staff
    - 3.4 ATAM evaluator
- 4. Selection of Architectural Views (Viewpoint)
  - 4.1 Logic view
  - 4.2 Development view
  - 4.3 Process view
  - 4.4 Physical view
- 5. Architectural Tactics
  - 5.1 Modifiability Tactics
  - 5.2 Usability Tactics
- 6. Architectural and Design Patterns

6.1 Architectural Patterns

6.2 Design Patterns

7. Views

7.1 Logic view

7.2 Development view

7.3 Process view

7.4 Physical view

7.5 Concistency between views

8. Architectural Rationale

9. Issues

10. Changes

11. References

1. Introduction

In this project our task is to plan and develop a functioning multiplayer game for Android or iPhone. The purpose of this phase, the requirement phase, is to create good and straightforwardly documents which describe the requirements for our game concept and explain our architecture and reason for our choices. Our intention with these documents is that they can be used to explain the game concept and architecture in such a way that by reading these documents, a developer can more easily start developing the game or make modifications to the game. Another important intention is that the documents can serve as a primary vehicle for communication among stakeholders.

We have chosen to base our multiplayer game on the already existing board game Ubongo. Our game will be implemented on Android by using the Sheep framework, and will be a networked game using a dedicated server. Players can create games on the server and join games on the server. A player joins a game by typing in a pin-code for the game he/she wants to join. One of the players in the game will be the owner of the game, and can at any moment start the game.

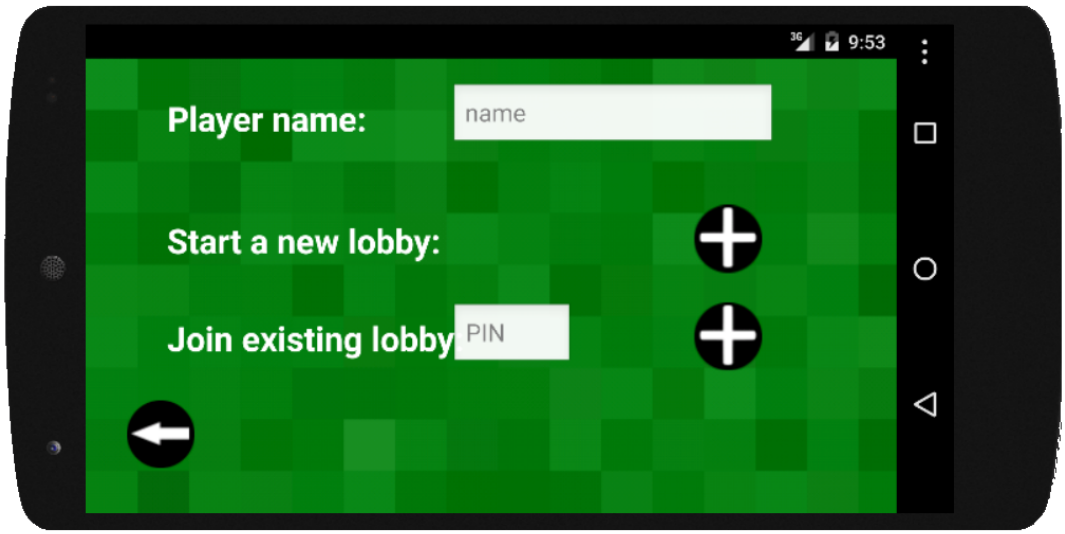


Figure 1: Ubongo game lobby

When the game is started, all players are presented with the same puzzle. The puzzle is a board consisting of a grid of empty cells and the player has to move, rotate and fit tetris-like pieces onto the board to fill all the empty cells. All the players in a game is solving the same board at the same time on their own device. The first player who solves the puzzle wins, and then the game is over for all the other players in the same game.

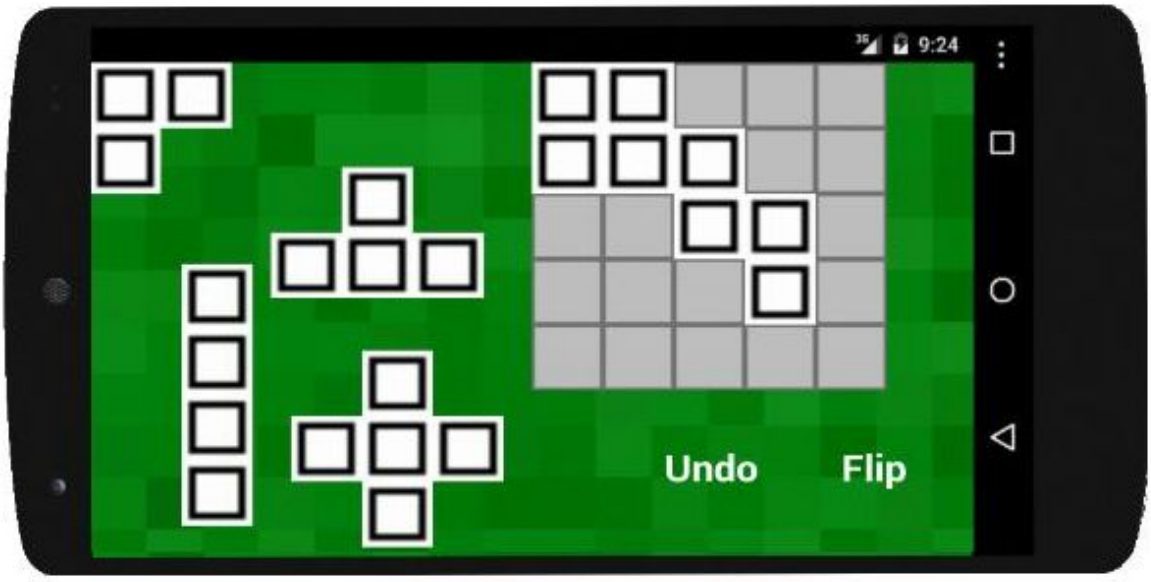


Figure 2: Ubongo gameplay

This document contains information regarding architectural drivers, stakeholders, architectural views, architectural tactics, design patterns and views.

2. Architectural Drivers / Architecturally Significant Requirements (ASRs)

A ASR can be defined as a requirement or a limitation that the system must fulfill, which will have a notable effect on architectural choices and decisions that have to be made. In this chapter, we will list the main drivers, which most affects the system architecture. These requirements can be split into three groups according to their subject - functional, quality and business.

2.1 Functional Requirements

2.1.1 Network Multiplayer

Having network multiplayer is important for the game to function, as well as important towards what architecture is used. Using wrong architecture can make multiplayer integration very difficult.

2.1.2 Multiple Game Screens

Using multiple game screens with different functionalities adds complexity to the game, and therefore must be considered when choosing architecture.

2.1.3 Sheep Library

Using the sheep framework guides the project with its architecture which we have to take into consideration when choosing the projects architecture.

2.2 Quality Requirements

2.2.1 Modifiability

Using an architecture that makes it easy to add new content to the game, such as new pieces or grids.

2.2.2 Usability

We will implement functionality which enhances the usability of the game. The architecture design must therefore consider usability tactics.

2.3 Business Requirements

2.3.1 Developer experience

The lack of much experience among the developers on our group will most certainly have an effect on the implementation and the architectural choices that have to be made.

2.3.2 Time limitation

Since the project has a relatively short time limitation considering that the task is to implement a functional game, some choices being made concerning the architecture and implementation will be made where the main reason for the choice was that it is the fastest way to solve the problem and not necessarily the best.

3. Stakeholders and Concerns

There are four stakeholders for this project: developers, players, ATAM evaluator and course staff. We will now describe the main concerns of each of these stakeholders.

3.1 Developers

- **Simple, well-structured and readable code:** Code should be easy to understand by other team members.
- **Testability:** Code should be well-structured and reasonably divided, which means easy to test.
- **Modifiability:** Developers should be able to easily extend and modify the code to add new game features according to players needs and wishes.
- **Simple cooperation:** The code should be version controlled and well documented. Other developers should be available and willing to communicate.
- **Avoid assignment misunderstandings:** The description of the application should be clear and simple.
- **Avoid time pressure:** Developers should share their code on time and keep deadlines.
- **Grade:** The developers in the project are all wanting a good grade out of their effort put into the project.

3.2 Players

- **Usability:** The game should be intuitive to play and easy to install. The rules of the game have to be easy and clear. It should be possible to run the game on many different devices without problems.
- **Availability:** It should be possible to play the game at any time.
- **Performance:** It should not take much time to connect to the server and start the game.

3.3 Course staff

- **Reviewability:** The code should be well-structured and well-documented. It should be simple to install and run the game. The documentation of the game should be clear and easy to understand.

3.4 ATAM evaluator

- **Reviewability:** The code should be well-structured and well-documented. It should be simple to install and run the game. The documentation of the game should be clear and easy to understand.

4. Selection of Architectural Views (Viewpoint)

In this section we will give a short description of the types of views we will use to present our architecture. An architectural view is a representation of a set of system elements and the relations among these elements. Different kinds of views have different focuses, and describes the architecture from different perspectives.

4.1 Logic view

Purpose	The purpose of the logic view is to create an understanding of how the system works, and how the system solves the functional requirements.
Stakeholder	Developers, course staff, ATAM evaluators
Notation	UML class diagram with packages to organize classes with similar functionality together.

4.2 Development view

Purpose	The purpose of the development view is to divide the system into chunks which can be assigned to a developer or a small group of developers, thereby making it easier to structure the development work.
Stakeholder	Developers, course staff, ATAM evaluators
Notation	UML package diagram. Each package is marked with the classes it contains from the logic view.

4.3 Process view

Purpose	The purposes of the process views are to show which states the game can be in and show how the system moves between different states when it operates.

Stakeholder	Developers, course staff, ATAM evaluators
Notation	UML activity diagram

4.4 Physical view

Purpose	The purpose of the physical view is to show how the parts of the system are distributed over different physical devices. The main focus of the physical view is to give an indication of how the non-functional requirements related to availability and performance are handled.
Stakeholder	Developers, course staff, ATAM evaluators
Notation	UML notation for node and component.

5. Architectural Tactics

Architectural tactics are primitive design techniques used to achieve the required quality attributes for the system. A tactic is usually an already well established and well tested method that has already been used in many other systems and designs before to achieve specific quality attributes. In this section we will present the tactics we have chosen for our system to achieve our quality attributes modifiability and usability.

5.1 Modifiability Tactics

Reduce Coupling

Loose coupling makes changes easier because it reduces the probability that more than one module has to be changed. We use for example separated classes (ServerManager) for the Server Communication as an intermediary.

High cohesion

By separating unrelated functionality, we make it easier to modify a specific part of the system without affecting other unrelated functionality. The separation makes functionality in the components higly related to the rest of the functionality in its component.To achieve this we have separated the functionality which generates and positions visual elements from the functionality concerning server communication and the functionality which handles the game logic. The separation can be seen in the architectural views as view, model, controller and server.

5.2 Usability Tactics

Separate User Interface

By using MVC the user interface is separated from the other parts of the application. The advantage is that the user interface can be easily adapted to the users needs.

Support User Initiative

- Undo: The game should support undo operations, such that the user can revoke his last shape movement. This can be implemented by setting the system to save former states of the gameboard. When an undo button is clicked, the system can retrieve the former state and present it for the user.
- Cancel: The user should be able to navigate back to a former menu choice and revert the changes done. This can be implemented with a back-button in the corner which changes the game state.

Support System Initiative

- Maintain task model: The system should give feedback to the user when the user is missing inputs or has given wrong input. This will be implemented by limiting the type of keyboard the user has access to when typing in information when joining or creating a game. At the same time the system should analyse the information and throw proper exceptions which are handled by generating and displaying a simple and understandable text message about the problem to the user.

6. Architectural and Design Patterns

A pattern is a reusable solution to a known problem in a well-defined context. We can divide the patterns into two groups, architectural patterns and design patterns. While the architectural pattern describes the structural organization of the components, the design pattern describes a solution to a problem internal to one or a few components in the structure. In this section we will describe what patterns we are using, and how we intend to implement them:

6.1 Architectural Patterns

- **Client-server-pattern:** Since our game is a multiplayer networked game, which uses a dedicated server, we find the Client-Server-pattern natural to use. We will implement this pattern by using sockets for sending messages between the clients and the server. The main task of the server program is to synchronize information about running games. When a client sends a message requesting to either create a game or join a game, the server will update information about this game, stored in the server program. When updates are received, the server program will also send messages to the other clients in the same game, notifying about the change. When one of the clients manages to solve the puzzle, it sends a message to the server, which notifies all the other players that the game is over.
- **Model-View-Controller-pattern(MVC):** We will use the MVC-pattern to separate game data from the representation of the visual elements that the player sees. We find it natural to use this pattern, since the player will interact with the game through a graphical user interface. We have implemented this pattern by creating one class for view, one for controller and one for model. Our goal is to put all of the stuff related to visual positioning of graphical elements in the view-class. The model class will be a clean representation of the data model which represents the data that the view is visualizing. It is important to not have any visual elements in the model class. The controller class will have fields to hold one model-class and one view-class. The controller will then update the model based on actions done in the view. It will also use the logic in the model to evaluate validity of actions done in the view and to check for specific cases which leads to events(for example: Is the game won).
- **Singleton pattern:** We have used the singleton pattern in two cases. The class which is being used by all other classes as an intermediary for communicating with the server is implemented as a singleton. We do not see a need to have more than one instances of this class. Another use of the singleton pattern is a class we are using to generate visual elements in the view. This class is accessed by all the views when they are generating frequent visual elements that are the same all the time(for example: The back-button has the same look and position in all the views). By changing the frequent visual element in this singleton class we change it everywhere. This is easier than changing it multiple times in all the classes it is used.

6.2 Design Patterns

- **State pattern:** Since the Sheep framework is based on a state queue which decides what to be showed, we have had to adapt our architecture to fit with this system. We have therefore also chosen to use the state pattern. We have combined the state

pattern with our use of the MVC-pattern. This means that we have three types of states the game can be in. The game can either be in main menu-state, lobby-state or gameplay-state. For each of these states we have a complete set of model-, view- and controller-classes. The controller class extends the state-class in the Sheep framework, so that each of the three controller classes represent a unique variant of the main state class. To change the state of the game(move between menu, lobby or gameplay) we just need to push a controller class to the state queue. When a controller class is instantiated it will automatically create model- and view instances matching the type of state the controller represents.

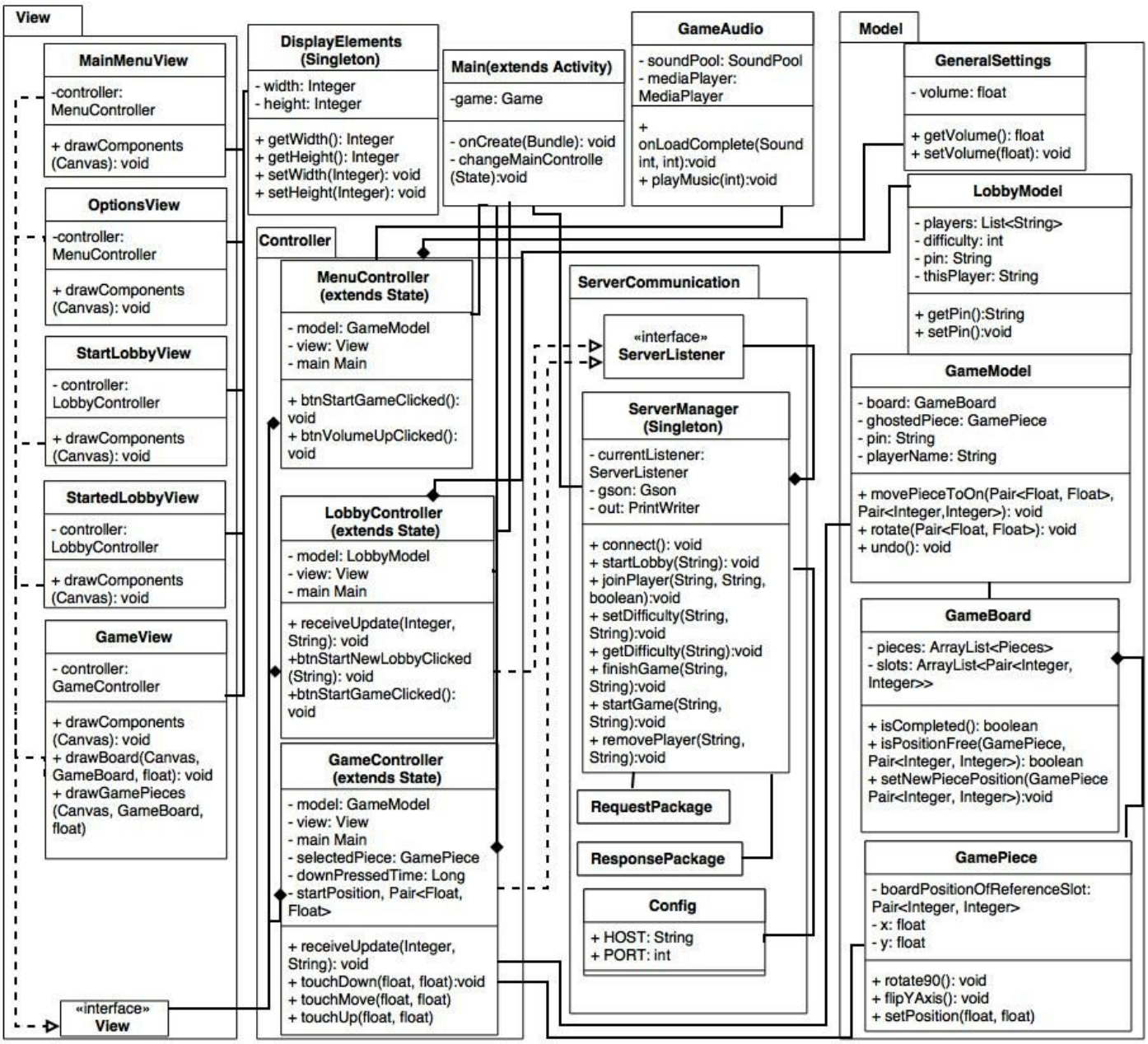
- **Observer pattern:** The different controllers should be listeners to the intermediary class that receives messages from the server. The intermediary class(ServerManager) should hold a field for the controller which is currently the active state of the application. If a message is received the controller, which listens, should get notified and initiate the appropriate actions.

7. Views

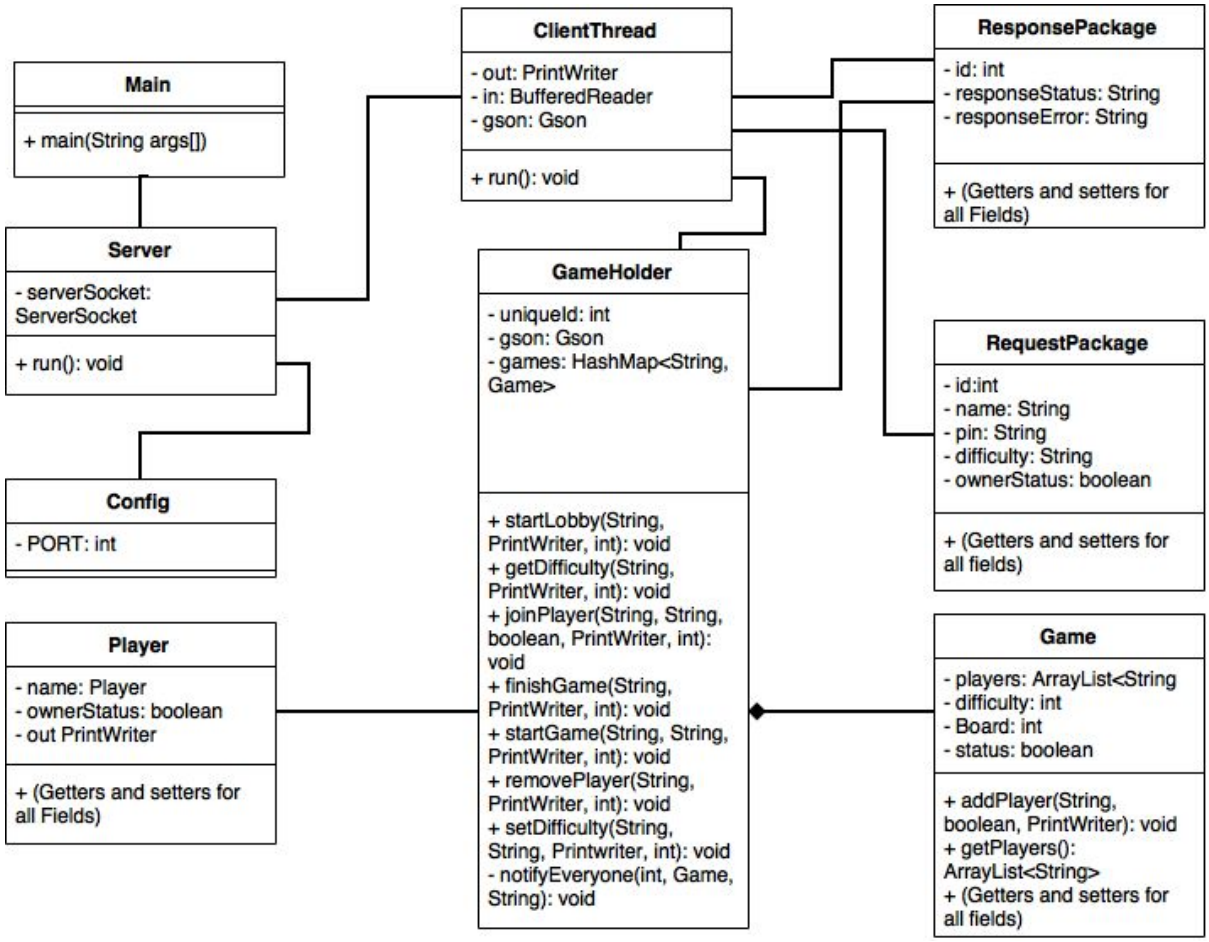
7.1 Logic view

The purpose of this logic view is to give a description of our main structure. We have divided the class diagram into two parts. One for the client side which is running as an app on a phone, and one for a server program which should run on a dedicated server. Note that not all methods and fields are included in this diagram due to space considerations, but the most important methods for the playing of the game and navigation in menus are included. For more details about each class we refer to the implementation document. For more details about methods in the class, we refer to comments in the code.

Class diagram for client:



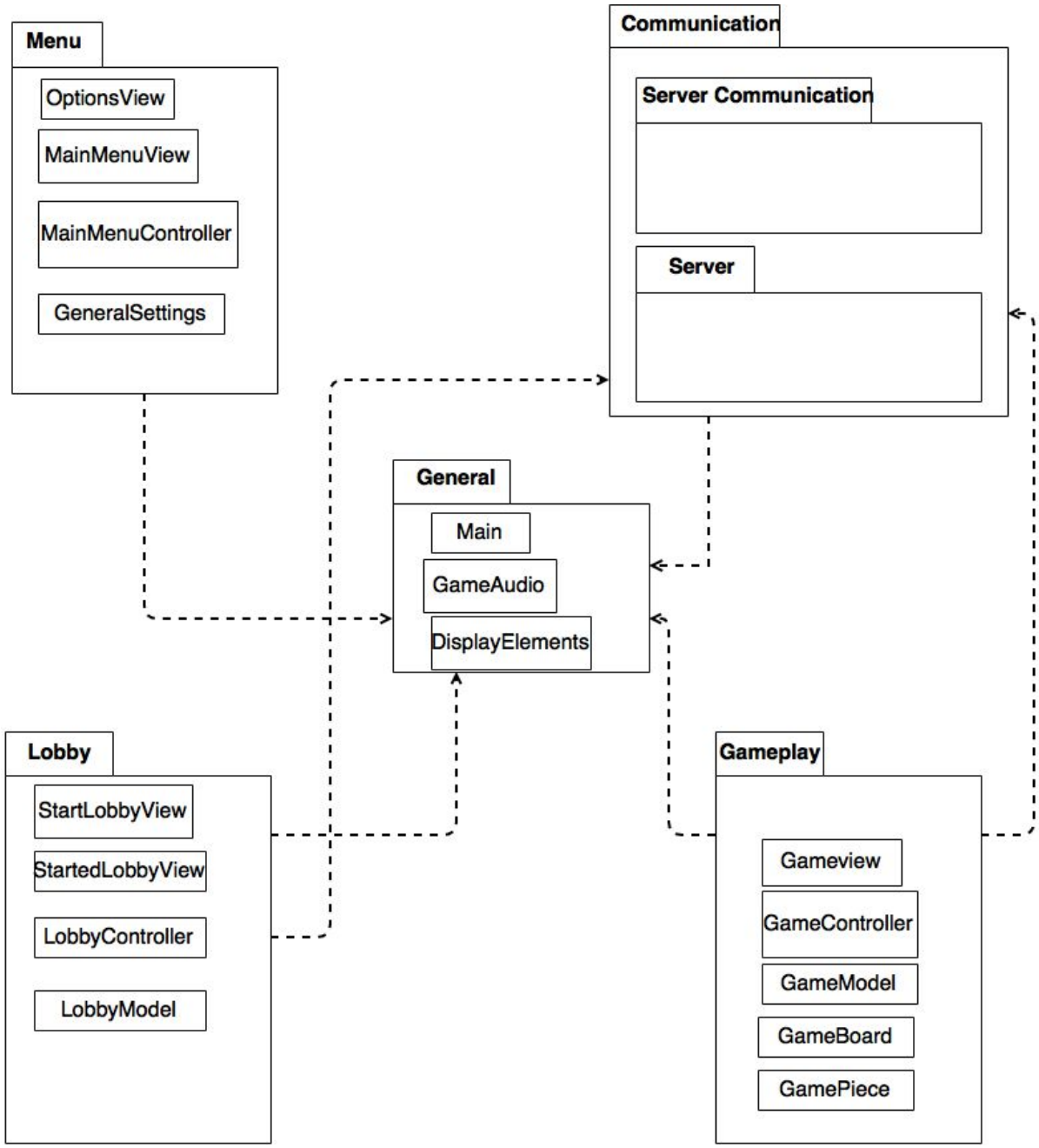
Class diagram for server:



7.2 Development view

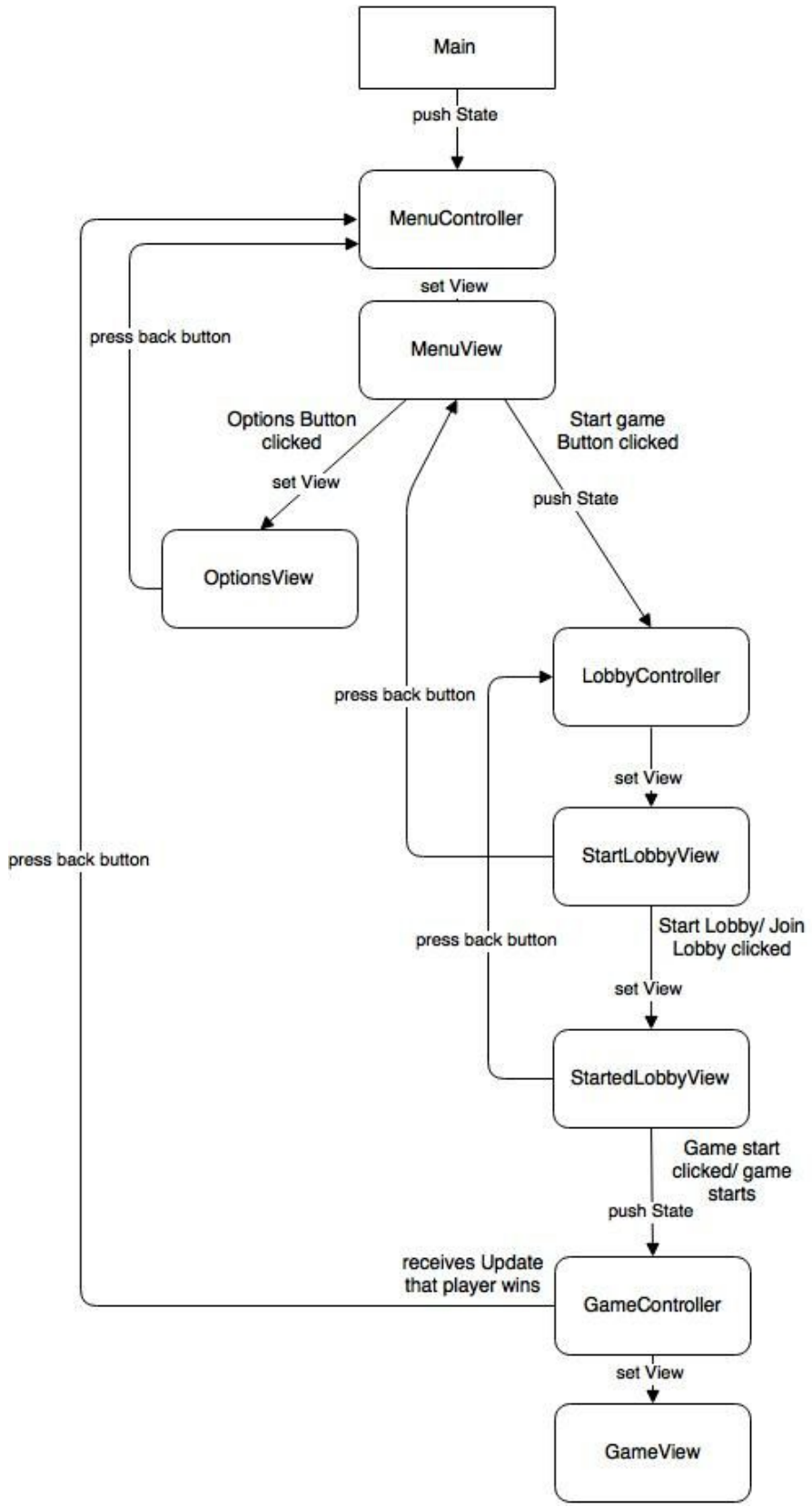
This development view organizes the classes from the logic view into development packages. Each package contains classes which are tightly connected. The reason for dividing the classes into development packages is because the classes in a package should be developed by the same developer, or by developers working closely together. By dividing the system in this way, we make it easier to assign tasks to different programmers on the team based on who are working with who. The arrows signalizes dependency between packages. For example; Before you can develop the LobbyController, you have to be finished with the server in order to get responses to

show in the view.

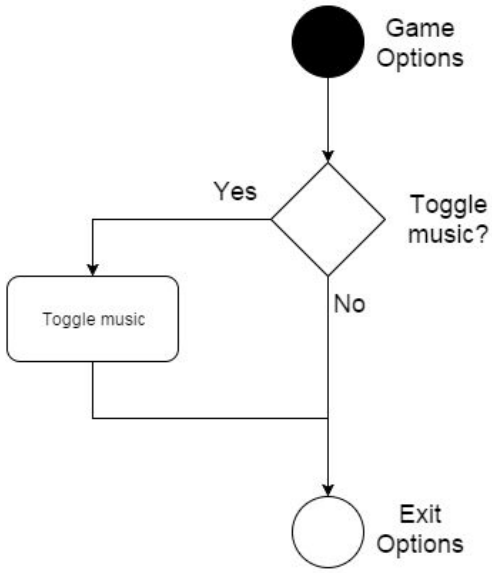


7.3 Process view

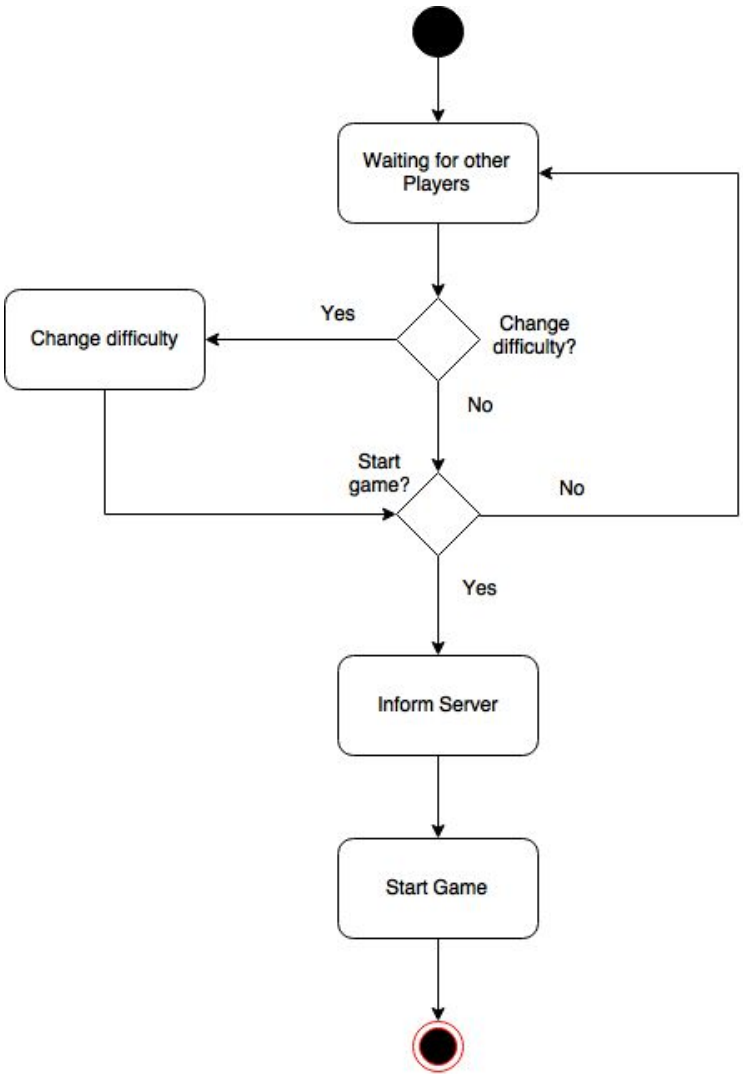
The diagram gives an overview over the navigation through the game starting from the main menu and how the controller and view classes are connected.



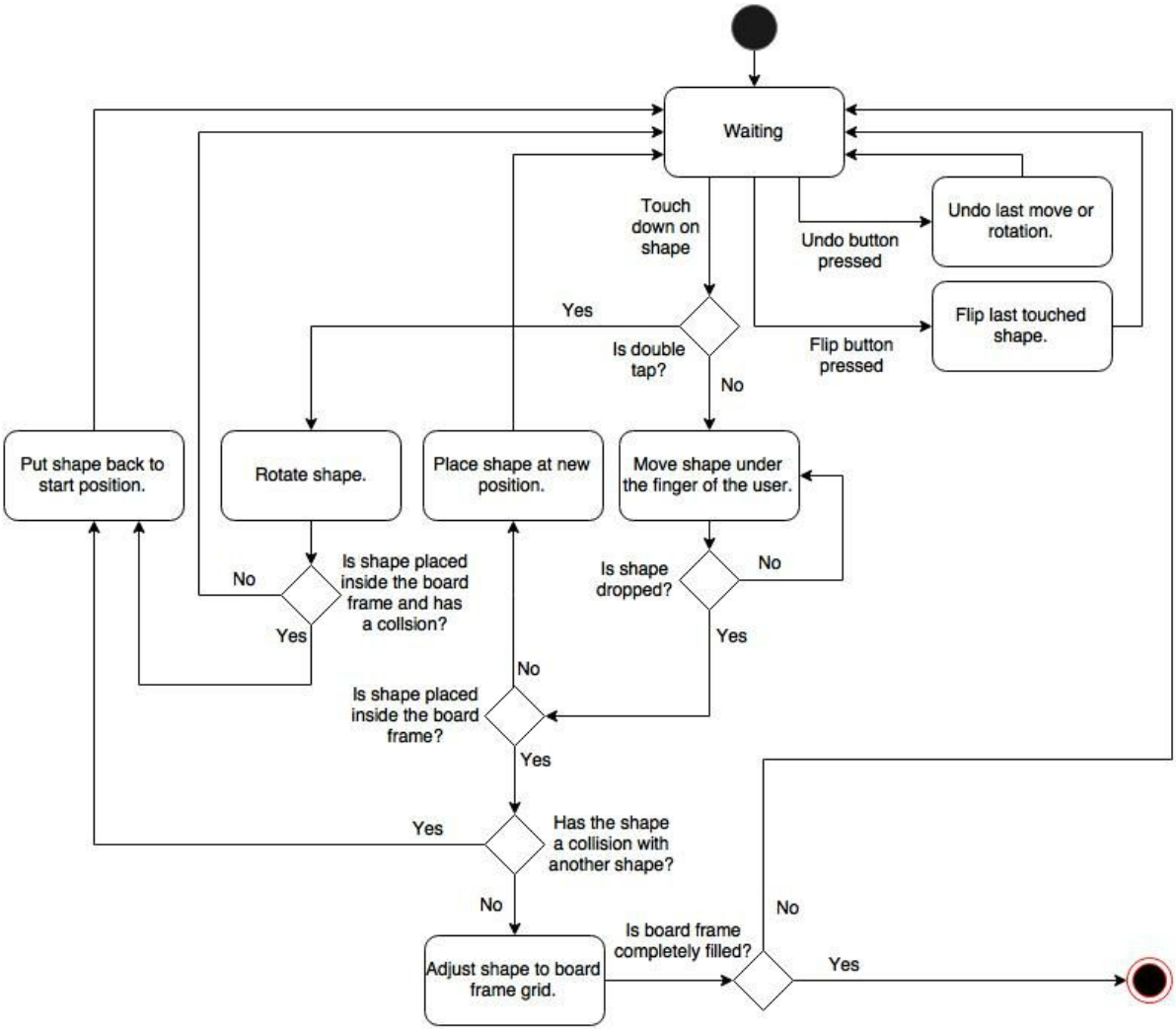
Here we have the activity diagram for the options screen, where the user can turn on or off the music.



Here we have the activity diagram for the lobby. The owner of the lobby waits for other players and can change the difficulty the game should have.

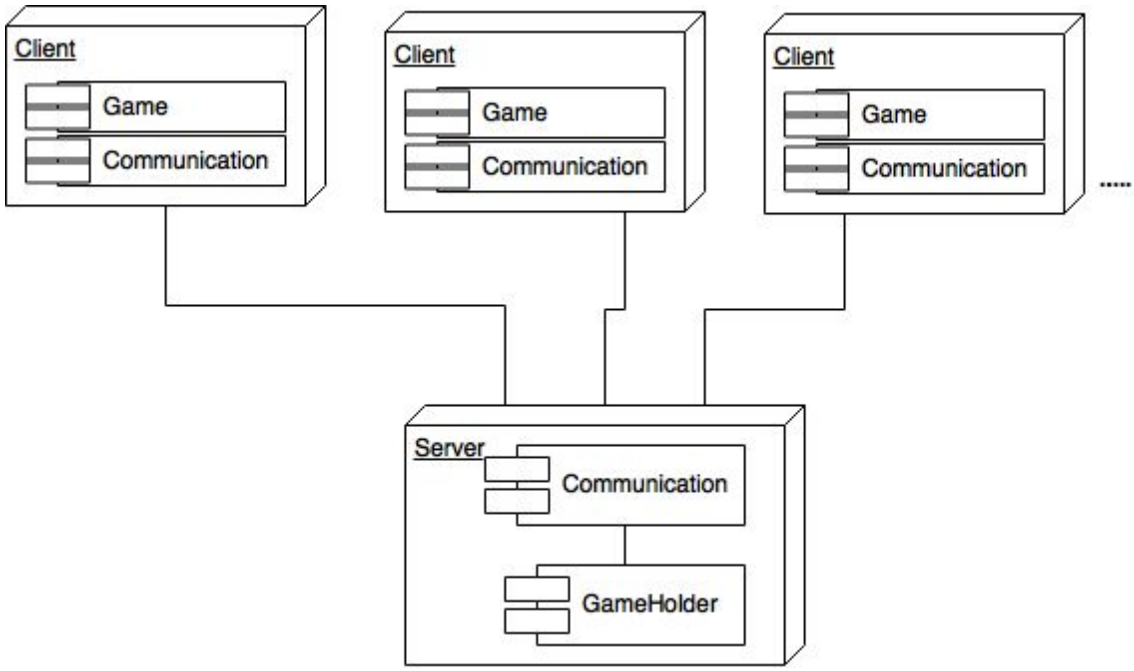


Here we have the activity diagram for the gameplay. Additionally an observer will monitor, if another player wins the game.



7.4 Physical view

This physical view describes how the system will run on different physical devices. The game-component in each client represents a running MVC-architecture of the game. This component will consist of a running controller with its associated view and model. The communication-component in a client sends and receives messages from the communication-component in the server. The GameHolder-component consists of the logic for updating information about which clients are playing against each other and who has won the game.



7.5 Consistency between views

All our views are describing our architecture from different perspectives. In the class diagram we have splitted the classes into the packages view, model, controller and server communication, based on the type of class it is. In the development view on the other hand, we have used another package arrangement based on what classes are connected to each other. Each of the packages menu, lobby and gameplay represent a state the client program can be in and contain a set of model-view-controller needed in that state. The class names used in the development view are the same as the class diagram and represent the same class in the implemented code. The communication package in the development view contains all classes related to the ServerCommunication-package in the client class diagram and the server class diagram. The physical view divides the client into game and communication. Game represents the packages menu, lobby and gameplay in the development view and communication represents the ServerCommunication-package in the class diagram. The server in the physical view represents the server class diagram. The process views are more a description of how choices made by the player changes the game state and how the models and controllers makes changes to the views based on user interaction.

8. Architectural Rationale

In this section we will reason for our architectural choices and explain how our architecture fulfills our primary and secondary quality attribute.

How do we accomplish our goals for modifiability?

Use of modifiability tactics:

In our architecture we have focused on having high cohesion. By for example making sure that all visual elements like buttons, text, textboxes etc. are placed in view classes, separated from the game logic in the model. This makes it more easy and understandable for a developer to change the visual presentation of the game. Another example of high cohesion is that we have separated functionality for server communication into a intermediary singleton class(ServerManager) which the rest of the classes that need to communicate with the server can use. The focus on high cohesion leads to a more modifiable architecture.

Adopting the MVC-pattern:

As mentioned, the MVC-pattern with the separation of logic and visual elements contributes to high cohesion which again leads to a more modifiable architecture. It also makes it easier for the developer to understand the code, because it is a well known pattern, so when the developer sees the terms view, model and controller, he will know where to find the functionality he is looking to moderate.

Adopting the State-pattern:

By using the state pattern to move between game states, we also make it easier to add new game states. A new game state can be added easily by just creating a new controller class which extends the State class from the Sheep framework. This class can then be pushed to the stack, and display its own specified views. To make sure that the views are compatible, we have also added an interface for these. By using the state patterns makes it more recognizable for a developer how to add new game-states, and thereby making the architecture more modifiable.

Adopting the Observer-pattern:

By using the Observer-pattern we make it easier to add new components which should get data from the server. A new class that is added only has to implement the ServerListener-interface, and be set as the currentListener for the ServerManager to get data from the server. By using this pattern for getting server-data we introduce a clear set of procedures a component must follow in order to get server-data. Therefore it becomes more modifiable, since a developer can check components already getting server-data to understand the procedures, which follows the well known observer-pattern.

Adopting the Singleton-pattern

We have used the singleton-pattern for the DisplayElements-class. By putting frequent elements in a class that is reachable to all classes in the game, we make it easier to change these visual elements that are occurring often. Instead of changing an element many

places, it could be changed only one place, in the DisplayElements-class, and the effect would be in all places the element is referenced through the DisplayEleemtns-class. Thereby making it easier and faster to do modifications on the views.

Other:

By making the boards as text-files, it is much easier to add boards, and the developer does not have to edit the source code of the client to add a new board.

How do we accomplish our goals for usability?

Use of usability tactics:

As mentioned, we have used several tactics in our architecture to make the game as useable as possible. In the gameplay we have implemented buttons and methods for undoing actions. In the menus we have implemented back-buttons which change the state of the game to the former state. Maybe the most important tactic is the maintain-task-model-tactic, which each time the server returns an error message, or the models cast exceptions, because of an user action, an error message is printed to help the user to solve the problem.

9. Issues

We had problems to decide which parts of the system should be on server side and what should run on the client. In the beginning we were also not sure, if we need a persistent storage on the server side.

It was challenging to get a common understanding of the architecture in the group. Different understanding lead to some problems when work from different group members were combined.

10. Changes

03.03.2016	First draft
15.04.2016	Based on the evaluation the following improvements have been added: - A more detailed introduction - Some reformulation in the ASRs - More concerns for the stakeholders - Introduction to Viewpoints - Removed scenarios - Introduction to tactics - Added more usability tactics - Added high cohesion as a tactic - Introduction to patterns - Rewritten the entire patterns-section
22.04.2016	- Added updated class diagram - Removed “defer bindings” in the tactics because it was implemented differently. This because of misunderstandings in the group, it did not get implemented. - Wrote architectural rationale - Added some issues - Updated process views

11. References

[1] Len Bass, Paul Clements, Rick Kazman, “Software Architecture in Practice – Third edition”, Addison Wesley, September 2012.

[2] Buschmann, F. ,Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996). “Pattern-Oriented Software Architecture”. John Wiley and Sons. ISBN 0-471-95869-7