NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4240 Sofware Architecture

Architectural Documentation

Group 10 - A7:

Mats Byrkjeland
Andreas Drivenes
Anders Wold Eldhuset
Stein-Otto Svorstøl
Torstein Sørnes
Håkon Meyer Tørnquist

COTS: Android SDK

Game title: Sea Battle

Primary quality attribute:
Modifiability

Secondary quality attribute:
Availability

April 26, 2015

Contents

1 Introduction . . . 1
2 Architectural drivers . . . 1
2.1 Time and platfrom contraints . . . 1
2.2 Constraints due to the Android platform . . . 1
2.3 Game idea . . . 1
2.4 Availability . . . 2
2.5 Modifiability and extendability . . . 2
2.6 Technology and COTS . . . 2
3 Stakeholders . . . 2
4 Selection of Architectural Views . . . 3
4.1 Logical view . . . 4
4.2 Process view . . . 4
4.3 Development view . . . 4
5 Architectural tactics . . . 4
5.1 Modifiability tactics . . . 4
5.2 Availability tactics . . . 6
6 Architectural and design patterns . . . 7
6.1 Architectural patterns . . . 7
6.2 Design patterns . . . 7
7 Views . . . 8
7.1 Logical View . . . 8
7.2 Process View . . . 9
7.3 Development View . . . 11
8 Consistency among views . . . 12
9 Architectural rationale . . . 13
9.1 Modifiability . . . 13
9.2 Networking, availability and performance . . . 13
10 Issues . . . 14
11 Changes . . . 14

References **15**

1 Introduction

This document describes the architectural requirements and tactics of the game *Sea Battle* for Android by A7. The game is an implementation of the famous *Battleship* board game, where two players can play against each other over the Internet. The players will take turns shooting each other's board. We will focus mainly on the networking part.

The main architectural drivers are *Modifiability* and *Availability.*

The views we have included in this report are the *Logical View* 7.1, the *Process View* 7.2 and the *Development View* 7.3 from the 4+1 View Model.

2 Architectural drivers

There are many different drivers, requirements and opinions that drive our architectural choices, not to mention our experience and knowledge on different fields and technology. In this section we try to summarize the most important drivers for our choice of architecture.

2.1 Time and platfrom contraints

First of all we're driven by some contraints given by the course this project is a part of. We have the obvious contraint of limited time, as the game and the required documentation needs to be handed in by APril 22. 2015. This means we can't plan for a huge and advanced architecture as we probably won't have the time to implement it, and in addition write the necessary documentation. We also have the obvious contraint of platform. We had the choice between Android, WIndows Phone and iOS, and went with Android. Without the contraint, we could've for instance planned for a web application, which the team has more experience with.

2.2 Constraints due to the Android platform

As we chose the Android platform, we are introduced with challenges in regard to the Android SDK, performance and how to actually get the most out of the platform, without spending too much time learning the specific APIs and workings of this specific paltforms. When deciding on the architecture, we'll how to determine if we want to use some sort of library that can abstract away some of this logic and work, or if we need to spend time in it to meet our defined requirements.

2.3 Game idea

The game idea we chose, Battleship, is a also driver for our architectural choices as it determines how the game works, and by extension how we need to develop it. For instance the workings of the game gives ut a multiplayer aspect. This means we could either do it locally on one Android unit, or networked. We want to be able to play against each other over the Internet, so the game idea makes us think about how we can built an architecture that supports this.

How the game battleship works also means that we'll probably have to have a standardized format of how the data is to be sent over the Internet. We need to decide on how the board should be represented in both the backend and the frontend. In the frontend we need to be able to show to grids (boards) at the same time. The players own board, and how he has hit or missed on the opponents board.

2.4 Availability

Because multiple players should be able to play together over a network, the players must first establish a connection between them (*matchmaking*), and their actions must then be communicated between them. As this is a turn-based game, latency is not especially important, but data consistency is. The actions must be communicated in the right order. Even though latency is not important, availability is, as the game will not function if the players can't communicate at all. There is no requirement for offline play, so if we do not let availability be of importance in the development of the architecture, then the whole game may be unplayable. This means we need to chose a backend technology that can handle this, but at the same time does not require too much code to be written. It must be a lightweight solution, and a technology that we can get into in a fair amount of time. This also goes for the choice of DBMS.

2.5 Modifiability and extendability

The quality requirements require that the game can be easily modified and extended with regards to graphical representation, choice of database and functionality such as implementation of AI.

2.6 Technology and COTS

We've already talked about some drivers regarding tehcnology, but another part of it si our own knowledge of what we're going to use. It's important that we select an architecture that supports the developers experience and knowledge, because if all developers has to learn some library or COTS from scratch, then we cannot follow the project schedule and deliver on time. We also have to consider how well documented the technology is, and who is backing it.We think this maybe is gonna be one of the biggest driver for our choices, as none of us really wants to spend too much time learning to libraries for this project.

3 Stakeholders

As we do not plan to release the game, we have not taken concerns like publishing or profits, or stakeholders like publishers (Google Play for Android or App Store for iOS) into consideration.

Stakeholder	**Concern**
User	**Consistency with requirements:** The game should act and feel like it is supposed to, according to the functional requirements. **Performance:** It must perform in such a way that it isn't a nuisance for the user. **Reliability:** Should crash or stall as little as possible. **Usability:** Must be easy and intuitive to use. **Playability:** Must be fun to play.

Project Manager	**Schedule estimation:** The game is to be finished in a short amount of time, and we have definite deadlines to take into consideration. **Progress tracking:** It is important to know how far the project has come at any given time. **Requirements traceability:** One should be able to easily assess if a given requirement is being met. **Effectiveness:** The team must be working effectively together and not wasting time. **Communication:** It is vital to maintain good communication within the team to avoid easily avoidable mistakes like working on the same files/issues etc.
Developer	**Readability:** When there are several developers, it is important that everyone writes code that is easy to read for others, as time flies quickly when you're trying to read bad code. **Modifiability:** Adding functions and extensions to the game should be as easy as possible. **Testability:** The game should be easy to test, to avoid bugs and errors after release. **Interoperability:** Since we have a server/client architecture, it is important that the two speaks well with each other. **Maintainability:** The program should be easy to maintain as bugs are found and updates has to be added. **Availability:** Again, because of our chosen architecture, availability is a concern as the game should be available to play as much of the time as possible (the server should not be down unnecessarily).
ATAM evaluator	**Reviewability:** The requirements have to be well-defined, and the documentation on these and on the architecture of the game has to be well-written.
Course staff	**Reviewability:** The code must be readable and "clean" to make it easy to review. **Testability:** The code must be easy and comprehensible to run and test to make the testing phase a good experience.

4 Selection of Architectural Views

We have chosen three different views from the "4+1 View Model", namely a logical view, a process view and a development view. We could have chosen a physical view as well, but the server is running in the cloud.

4.1 Logical view

Purpose To translate the functional requirements into objects interacting and sending messages to each other.

Target audience Project Manager, Developer, ATAM evaluator, Course staff

Notation Class diagram in UML, showing the main classes of the client and the server.

4.2 Process view

Purpose To present the dynamic aspects in the program like activities, states and tasks at run-time.

Target audience Project Manager, Developer, ATAM evaluator, Course staff

Notation State diagram in UML.

4.3 Development view

Purpose To give the developers an overview of the project, like how the modules and packages are organized.

Target audience Developer, Project Manager, ATAM evaluator, Course staff

Notation Component diagram in UML.

5 Architectural tactics

Describes the tactics for architecture to meet the quality requirements.

5.1 Modifiability tactics

> [. . .] high coupling is an enemy of modifiability. [2]

The goal of modifiability tactics is to make a change in the system as cheap as possible.

Our primary goal will be to reduce the coupling, and have modules with high cohesion. It should for example be possible to add new ship types without modifying many classes, by having an interface using encapsulation on a base ship object. Instead of having the computer generate random boards, it should be easy to implement that a user places all the ships. The keyword in our tactic for meeting this quality attribute, is planning. In the design process, or in the planning phase, we used a checklist as support in the design in regards to modifiability. [**126-127**, 2]

Cohesion

We must be careful to separate different concerns in the application, and one can plan this out to some extent ahead of development. This means we'll have to plan out the interfaces, and as many of the modules as possible in advance, and think through where things should be placed.[**123**, 2] In the development phase, one should also think over where one places methods, and not follow the diagrams we plan out blindly. With this tactic, we'll think through of the level of cohesion affects different modules twice, and hopefully won't have to do that much refactoring later on. An example of this tactic in practice is that it makes sense to have a distinct module for network requests on the server, and a seperate module for JSON parsing. This will increase cohesion and readability for the developer.

The following tactics will be used to manage and maintain high cohesion:

- Plan out modules in advance, and use the plans as guidelines for the implementaiton
- Reevalute modules and where code is places at every incrementation, and split modules as needed.
- Regular system tests to ensure that modules work together
- Use an IDE that makes it easy to discover which dependenies between modules are not met, to move code and to refactor code
- Avoid having one person working on a module all by himself, so that more than one person have some sense of the modules responsibilites
- Use version control (git) and use pull requests and code reviews to ensure that adding methods/code to modules makes sense in more than one persons eyes

Coupling

We deploy the following tactics to introduce, manage and maintain loose coupling:

- Refactoring code as needed when there are new pull requests to ensure that the coupling stays loose.
- By using encapsulation we can ensure that different modules can work together in a meaningful way, and by extension taht the coupling is loose. For instance one agree upon specific APIs for the server application, but also for eg. local models and controllers. [**123**, 2]

Resources

Another part of modifiability tactics is resources. One cannot change anything unless one has resources to do it. In this case, our resource is time an energy. So how can we make our system modifiable in regards to resources? If each developer has scheduled extra hours each week, to meet or accomedate changing requirements or needs of the system, one can modify the system much easier, simply because there's time for it.

Version control

Git is used as version control system to commit changes across multiple local code branches. The concept of branches and pull requests is a nice tactic for making changes to a system where multiple developers are contributing.

5.2 Availability tactics

> Availability refers to a property of software that it is there and ready to carry out its task when you need it to be. [2]

There are three cateogories of availability tactics: Fault detection, fault recovery and fault prevention [**87**, 2]. Due to time contraints, we'll hav eto limit ourselves to some extent as to which tactics we deploy, but here are some general guidelines and tactics we'll use to keep avaialbility as high as possible:

- Fault prevention
  - Use a third party PaaS (Platform as a Service) to deploy the server, to that we do not have to make sure that our own server is down, or that there is anything wrong with the connection to that coputer itself. We can focus on the server application. In case of a growing number of users, we can pay for an upgrade in computer power.
  - On the server we plan on using a DBMS with transaction support which prevents faults caused by simultaneous use.
  - We develop the game for Andoroid, and it's written in Java. We can use exception handling to avoid faults hitting the user.
- Detect faults
  - On the client-side, we can detect faults on the server by reading the HTTP error response and give an appropriate feedback to the user. It is possible to ping the server each minute to make sure that the service is available. The JSON-to-Java object mapping can detect errors in the data sent from server, and send the network request again.
  - Monitoring of the server will help us to detect if the server goes down, or soemthing is wrong. With the Heroku platform, we can deploy addons to get notified with something is wrong with either the server or DBMS. [1]
  - We can use time stamps on creation of games so that we can easily remove old or outdated games. This can help new users have their usernames freed up, and free up resources on the sevrer.
  - We develop the game for Android, and this means it'll be written in Java. This means we can easily throw exceptions to indicate that something is wrong.
  - We'll use unit and system tests to detect and fix faults before deployment.
- Fault recovery
  - We use a DBMS on the server to have persistence so we can recover games in case of power dropout, server shutdown or other physical issues.
  - If a network requests fails or times out, the client can retry the request instead of just saying it does not work.

6 Architectural and design patterns

6.1 Architectural patterns

Our software will be built around two main architectural patterns: *Client-server*1 and *Model-View-Controller* (MVC).

The MVC-pattern will be implemented together with client-server as follows: The model contains game instances, each of which contains two players and two game boards where each board contains ships. The view presents the game screen which consists of both game boards, a button select the active (larger) view, and a button to bomb a selected coordinates. The view allows the user to choose which coordinate to bomb using the screen's touch input and by confirming the choice with a button. When a bombing is confirmed, the controller on the server is notified through an HTTP request and takes the appropriate action. This pattern allows us to cleanly separate the user input and graphical representation from the business logic. You can see the relationship in figure 2.

TThe models and controllers, which together make up the game logic, will be implemented server-side, while the client will implement the view. The clients and the server communicateso that user events in the view are reported to the controller, and the controller updates the views' contents as necessary. In actuality, the client will poll the server for changes, but logically the controller updates the view.

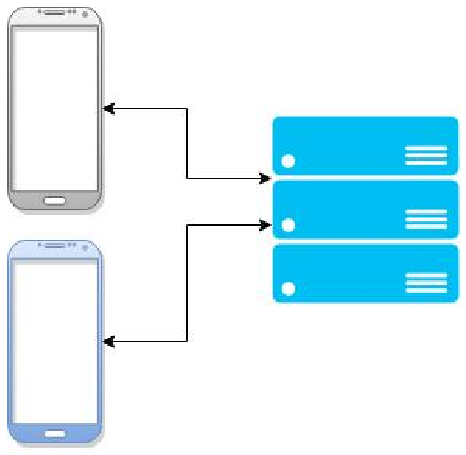

Figure 1: A figure describing the client-server architecture, with two Android clients.

6.2 Design patterns

We will in addition to the above mentioned architectural patterns, we also plan to make use of the following design patterns:

State-pattern Both views on the client and game-object on the server will have states.

Template-pattern With the libgdx-library we'll have to extend the abstract Game- and Screen-classes to get a working Android application.

Observer-pattern Buttons and other graphical elements will make use of this pattern to react on events from the user.

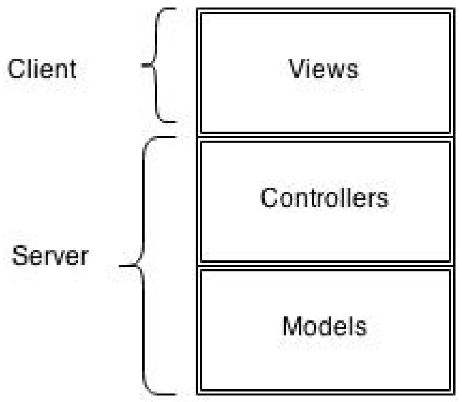

Figure 2: A figure describing how the MVC architectural pattern will work witn the client-server architectura in our planned architecture.

7 Views

7.1 Logical View

Below is a class diagram for our program. We split the Client and the Server parts, to better show how the classes interact on their respective sides. For information of how the client and server communicate, please take a look at the the Component Diagram 7.3.

As you can see, the Client side classes are only Graphical User Interface classes, that is, they belong to the View part of MVC. The actions performed by the player in the GameGUI will trigger events handled on the server. The GameGUI will then in turn display the consequences.

The Server is responsible for the game logic and consistency. The Game class holds most of the game logic, including the “run loop” which keeps the game going and changes player after each iteration. After each turn, it saves the game state to the database so that a game can be resumed after a connection loss. A Game instance consists of two Board objects – one for each player, and each Board object is a grid of Cell objects.

The Player objects are also saved in the database. As we do not provide safe login with a password, this means that a user can log in as another, provided he knows the username of another player, and can consequently resume the other player’s games. This is of course a huge security flaw, but as security is not one of our main concerns, we have chosen to implement user authentication only if we find that we have time to spare towards the end of the development process.

When a user clicks "Play!" in the Menu view, the Matchmaker class will be notified. The Matchmaker class has a FIFO queue of all players that are waiting for a game. Theoretically, this queue should never contain more than two elements (users), as if there are two players in the queue, the two players will be popped, and a game will be started featuring these two players. However, due to the possibility of two players requesting a match at (practically) the same time, we will not constrain the length of the queue or in other ways assume that only two elements can exist in the queue simultaneously.

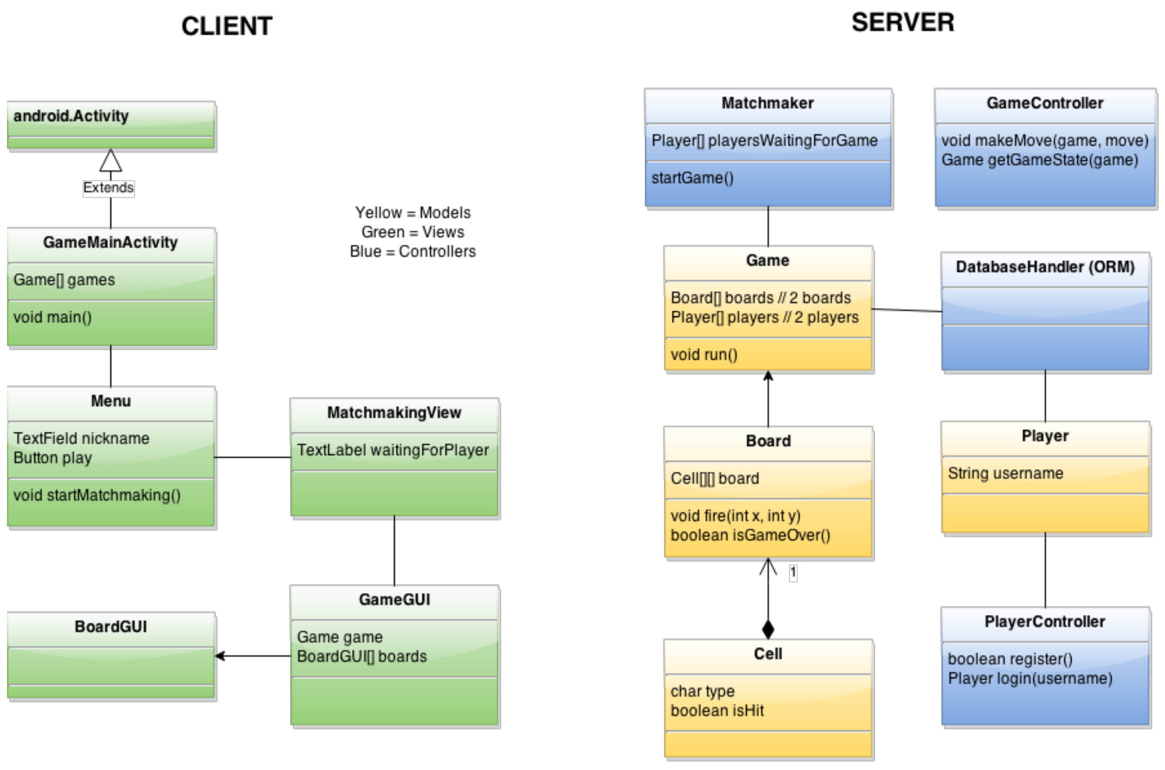

Figure 3: Class Diagram for the system

7.2 Process View

The following state diagram shows the transition between the three different views that our app consists of. You should notice that registration and login are the the same operation. The user simply writes his desired username/nickname, and clicks play. The user will then be registered in the database, if it is not there already. The user will then be able to resume his game, or start a new one if he has no ongoing games.

The Matchmaking View will be shown in the case where the current player is the only one in the Matchmaker's queue. This will probably not be a problem, as our game will be massively popular. Therefore, the MathmakerView will rarely be visible for the player.

The GameGUI will show the boards, some status text and buttons. If the game is won/lost (game over), or one of the users taps "Quit", the game is ended and the Menu will be shown again.

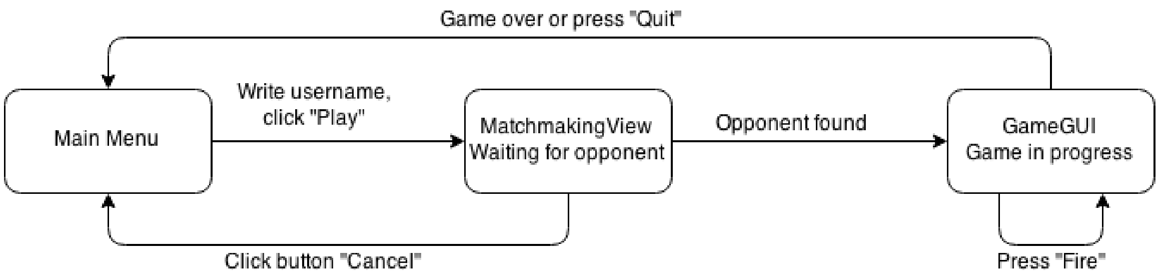
Game over or press "Quit"
Main Menu
Write username, click "Play"
MatchmakingView Waiting for opponent
Opponent found
GameGUI Game in progress
Click button "Cancel"
Press "Fire"

Figure 4: State Diagram for the system

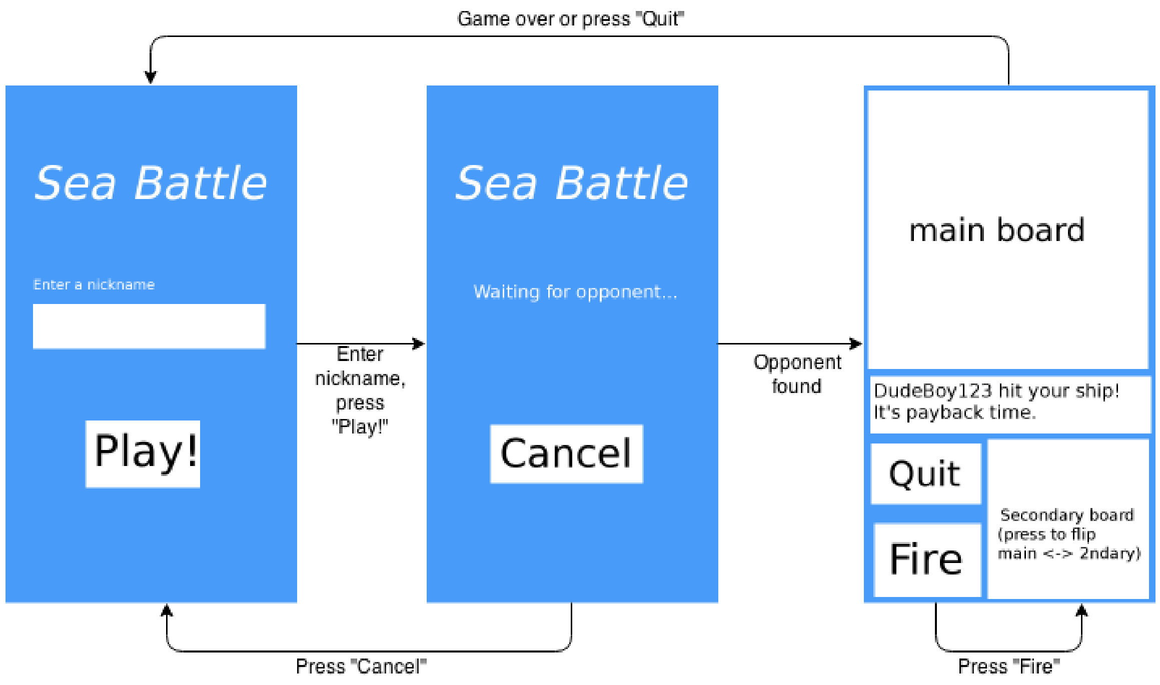
Game over or press "Quit"
Sea Battle
Enter a nickname
Play!
Enter nickname, press "Play!"
Sea Battle
Waiting for opponent...
Cancel
Opponent found
main board
DudeBoy123 hit your ship! It's payback time.
Quit
Fire
Secondary board (press to flip main <-> 2ndary)
Press "Cancel"
Press "Fire"

Figure 5: A more graphical State diagram

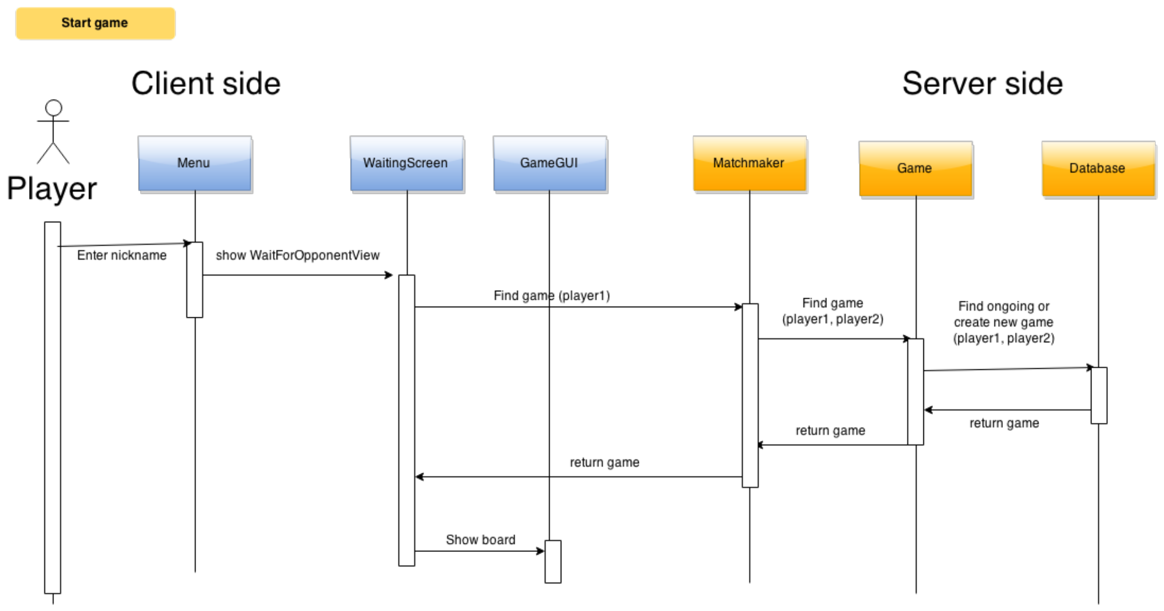

Figure 6: Sequence Diagram for starting the system

7.3 Development View

The diagram shows the main modules and components of the system. Mainly, it shows that the server (Node, Express) provides a routing-based API that the NetworkRequests module needs. The received data needs parsing on the client-side.

Our ORM provides an interface to Express and communicates with a database.

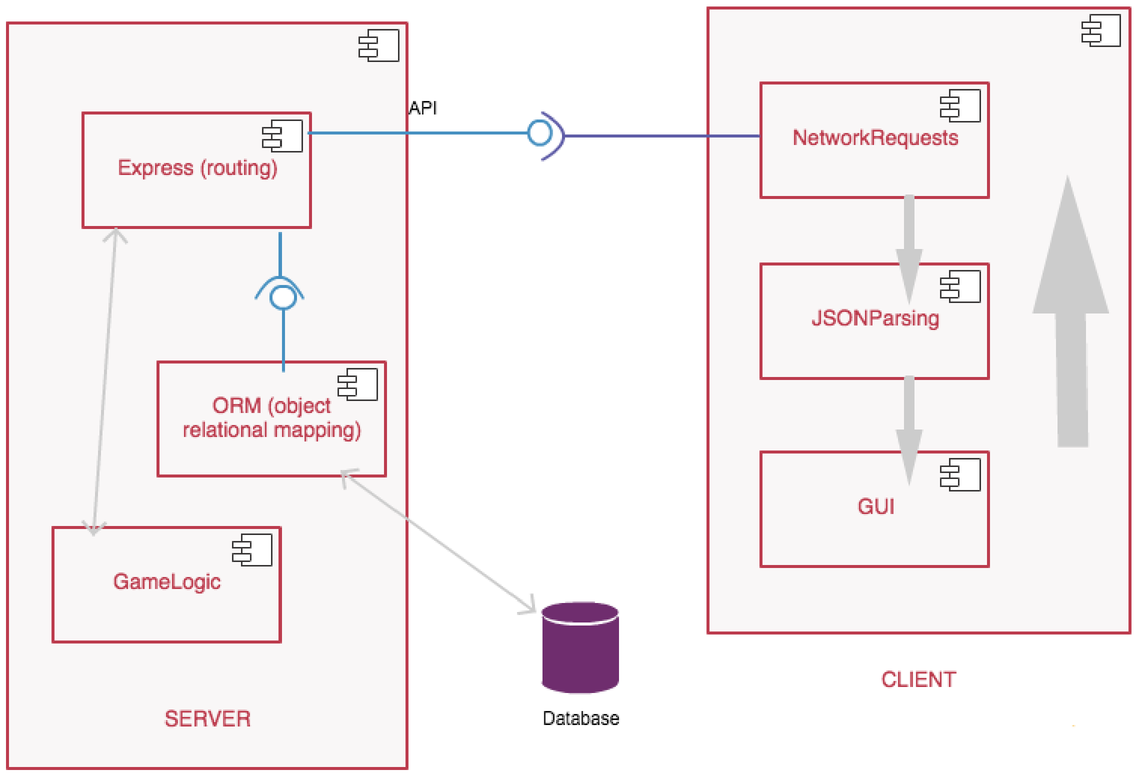

Figure 7: Component Diagram for the system

By separating into these modules as seen in the diagram, we can split the implementation work on the group members. One could do the ORM controller (handling saving to and fetching models from the database), one could do the game logic (models and their methods), one could do the routing on the server, making sure the incoming API network requests trigger the right actions. The client side could also be separated in network handling (controller), and the different views could be delegated to different persons.

In order to make this above tactic plausible, good communication and planning are crucial factors. The database handler must know how the models are implemententeded to know how to store them in the database. The server network handler implementor and the client side network handler implementor must know what data to expect from each other.

8 Consistency among views

The state, class and component diagrams are consistent. We can see that the server components shown in the Logic and Development views are not shown in the Process view, but this is due to the nature of the diagrams, and what they aim to describe.

Regarding the differences between the component and class diagrams, the "GameLogic" component in the server component in figure 7 consists of most of the classes, matchmaking etc. from the class diagram for the server. The Express component simply takes the requests and calls methods and returns results based on the request.

9 Architectural rationale

9.1 Modifiability

The primary quality attribute of our software architecture, is that it should be modifiable – that is, it should be easy to add or change functionality. We've achieved this through several architecture and technology choices.

Our application implements a **client–server** architecture, where all of the game logic is on the server and the client is a relatively simple view. This architectural choice entails that adding support for multiple platforms should not require changes to the game itself; we could add an arbitrary number of different client applications that all use the same networked API. This ties in to our choice of an **MVC** architecture, where the models and controllers are on the server and the views are on the clients. Decoupling the game logic from platform-specific representation makes it easy to add other clients. This is made even simpler through our use of the libGDX framework [3], which allows us to package our client application for Windows, Mac, Linux, Android, iOS, BlackBerry or HTML5 using the same codebase.

Furthermore, our server application does not communicate directly with its data store, but does so through an **object-relational mapping (ORM)**. This means that we only deal with objects native to the programming language of our server application, and the ORM converts these objects to the form required by our database, as well as handling reading and writing to the database. By using an ORM, we can easily change from one database management system (DBMS) to another, given that they are both supported by the ORM in question. Not having a strong dependence on a single DBMS, means that it is easier to switch to another form of hosting where another DBMS may be preferred, or to switch to another DBMS if we find that this is advantageous to, say, performance. A nice side-effect of using an ORM, is that many developers find it to be more readable and understandable when compared to writing plain SQL or other database-specific languages in the application code.

9.2 Networking, availability and performance

Because our product is a multiplayer game, where each player uses his own device, we have two options when it comes to communication: Using a game server, as we have decided to do, or using peer-to-peer connectivity. In the latter case, Bluetooth or a similar short-range, radio-based protocol is the only feasible solution as peer-to-peer connectivity over the Internet is fairly difficult, especially on mobile devices, due to security settings on the devices. Such a solution implies that the players need to be physically close to each other, which seems an unwanted requirement, hence the game server.

Using a client–server architecture, as we do, has the advantages mentioned in the section on modifiability (9.1) when it comes to decoupling game logic and user interfaces. However, it also has its own set of challenges as discussed in section 5.2.

As a side note, because our game will run on Android devices with varying technical specifications, there is an element of uncertainty when it comes to device performance. As our game is turn-based and very simple, performance should not be a problem on any modern device; however, if this were to be an issue, running the business logic on the server would make the performance depend (almost) only on the speed of the network connection, resulting in a more predictable user experience and fairly low battery usage.

10 Issues

One issue we faced when describing our software architecture, is the logical separation of concerns between the client and server applications and its relation to the MVC pattern. We wish to implement the models and controllers server-side, and the view client-side, as is common with web applications. However, as the two applications are completely separate and only communicate over the Internet, there is a need for model-like objects (of deserialized data delivered over the network) on the client-side for the client application to construct its graphical presentation in a sensible way. There is also a need for controller-like event handlers in the client that trigger the appropriate HTTP requests. The group discussed whether this implies that we have a complete MVC architecture on the client-side alone. However, we found that as these "models" and "controllers" only mediate between the client's view and the *actual* models and controllers on the server, they are an implementation detail rather than a complete MVC architecture. For example, the client's "models" are only deserialized JSON data, and do not contain the business logic of the server's models.

11 Changes

In table 11 you can see what changes we've carried out with this document from first draft to the final delivery.

Date	**Change**
02.03.2015	Delivery of the first draft of the report.
20.03.2015	Made architecture tactics and drivers more specific and more directly related to our project, and added design patterns to the section about archiectural and design patterns, due to feedback from the course staff.
22.03.2015	Delivery of final draft.

Table 2: Changes made to this document.

References

[1] *Heroku StillAlive Addon*. Mar. 20, 2015. URL: https://devcenter.heroku.com/articles/stillalive.

[2] R. Kazman L. Bass P. Clements. *Software Architecture in Practice*. Addison Wesley, 2013.

[3] *libGDX – Java game framework*. Feb. 23, 2015. URL: http://libgdx.badlogicgames.com/.