NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4240 Software Architecture

Group 2 - Architecture
Penguins in Space
Android

Christiansen, Sander Aker
Dahl, Lars-Kristian
Klaussen, Stein-Aage Nibe
Lundenes, Morten
Sjøen, Henry Skorpe
Zhou, Shelley Xianyu

Primary Quality Attribute: Modifiability
Secondary Quality Attribute: Usability, Performance and Interoperability

April 23, 2017

Contents

1 Introduction

1.1 Description of the project and this phase

This document describes the game software architecture, the second part of the project documentation. In this phase, we identify Architectural Significant Requirements (ASRs) in Chapter 2, and design the architecture based on them. The architecture is documented through views, tactics, patterns and finally a rationale.
Notations used in this document follow recommendations from Reference [3] and Kruchten's 4+1 Framework [2].

1.2 Description of the game concept

The game is an Asteroids like game, with multiplayer functionality. Figure 2 shows a screenshot of the retro arcade game Asteroids. A lot of the game dynamics are borrowed from this game.

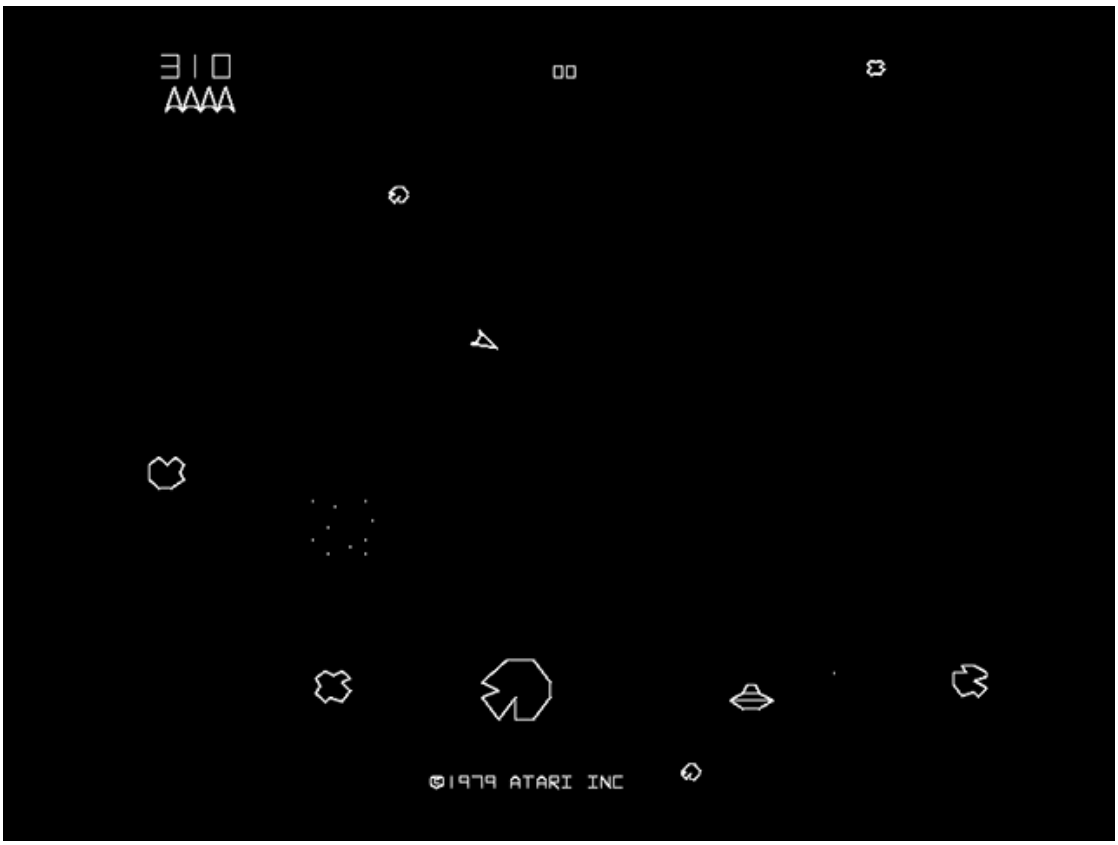

Figure 1: Asteroids, an arcade space shooter released in November 1979 by Atari, Inc.

The name of the game is "Penguins in Space".

You are a penguin floating in space. There are angry snowballs all around spawning randomly. The goal of the game is to stay alive for as long as you can, and get the highest score possible by either shooting snowballs in single-player mode or other penguins in multiplayer mode.

In singleplayer mode, power-ups will spawn throughout the game and give you abilities like dropping bombs or becoming invulnerable. The goal here is to survive and destroy as many snowballs as possible. Destroying a snowball increases ones score by one. The levels increase with your score, increasing the speed of the angry snowballs which makes it harder and harder. The goal is to achieve the highest score. In multiplayer mode there are several other players spawning that can attack each other. They will fight in a free-for-all until the last penguin is left standing.

The player controls the space-penguin using an analog stick in the lower left corner and shoots by pressing the button in the lower right corner. Below Figure 2, is a screenshot of the game in single-player mode.

Figure 2: Illustration of our game - Penguin in Space

2 Architectural Drivers / Architectural Significant Requirements (ASRs)

2.1 Functional

2.1.1 Real-time online multiplayer

The main driver of the functional requirements is the option for the user to play real-time multiplayer game over the Internet. To meet this requirement we need to have a peer-to-peer or client-server architecture. For this purpose we will use Google game play services, thus it is important that we comply to given standards for sending and receiving messages. We will also need to decide how we will structure our classes to aggregate the relevant data to be sent to the server or the other player(s). Therefore it will be important to implement a good interface to the these services.

2.1.2 Different power-ups

For the player to interact with many different power ups, we will need to make the architecture so that it is easy to add new power-ups to game.

2.1.3 Different game modes

The option to change game mode will also have an architectural impact. The game modes will share many properties, but still the rules of the game mode and logic will differ. This means we must make it so that there will be a connection between the modes, but while still being able to easily change the rules.

2.2 Quality Attribute

2.2.1 Modifiability

This is one of the main quality attributes of our game. This should be reflected in the architecture. It should be easy to add or remove new features, power-ups, playable characters and game modes.

2.2.2 Usability

This is the other main quality attribute. The game should be intuitive and easy to use. The system should provide good feedback to the user. Besides a user can go through a informative tutorial.

2.2.3 Interoperability

Real-time multiplayer is the main characteristics of our game, and hence interoperability is a significant quality attribute so that players with different clients can play together. We will use Google Play Game Services, which provides APIs supporting different client devices.

2.2.4 Performance

Performance is another significant quality attribute for real-time game, which significantly impacts player's experience. Our game will have latency no more than 200 milliseconds and at least 30 frame updates per second.

2.3 Business Goal

The primary goal will be to make a good architecture, complete all of the necessary documentation according to the assignment tasks. And implement the game with the sufficient level of completeness within the time constraints given. The architecture can thus not be too complex to finish in time.
Some of the group members already have experience with the chosen framework, Libgdx, that hopefully will speed up the implementation, and help the other group members to familiarize themselves with the framework.

3 Stakeholders and concerns

This section describes main stakeholders and their interests to the project.

3.0.1 Lecturer and teaching assistants

Lecturer is interested in how much the students have learned from the course and hence reflect the learned knowledge in the project work. In this sense we will deliver high quality documentation as well as an interesting game with well designed architecture. The high quality document will make the evaluation easier for teaching staff.
Lecturer and teaching assistants are concerned about if the selected architecture patterns are well understood and properly implemented. All architectural views are important for lecturer and teaching assistants, as they are interested in all aspects of our game.

3.0.2 Project team

Project team is concerned about good cooperation and easily task dedication, as well as delivering an interesting game and high quality documentation. In this sense, our game will be split into modules based on both functionality and architecture design, which leading to both easy task dedication and QA (Quality Attribute) improvement.
The most important view for task dedication is the development view, which contains modules of the software. The sequence diagram and process view are also important as they tell about how the different modules work together. Logical view tells what the software aims to, so it's important as well so that developers don't deviate along with implementation phase.

3.0.3 ATAM evaluators

ATAM evaluator requires necessary information in order to evaluate the architecture. High quality documentation is very important, especially the architectural views, which provide high level view of the architecture. Clarification from architects during ATAM session is also important if any information is left out from the documentation.
For ATAM evaluators, all architectural views are important, since all together would help them understanding our game better and more easily.

3.0.4 End-users

End user is concerned about how easy and interesting the game is. To strengthen this, the user has option for an informative tutorial when starting the game and opportunity to select different character appearances. End user is mainly interested in the architectural logical view, which provides information of game features and functionality.

4 Selection of architectural views

The selected architectural views are listed in Table 1.

Table 1: Architectural views

View	**Purpose**	**Target audience**	**UML Notation**
Logical View	To describe what the system will provide in terms of features for the end user	Developers, course staff, ATAM evaluators, End user	Class Diagram
Process View	To describe the processes and workflows in a system, as well as the communication between them.	Developers, course staff, ATAM evaulators	Activity Diagram, Sequence Diagram
Development View	To describe the major components of a system, eg. the packages, libraries and sub-systems used.	Developers, course staff, ATAM evaluators	Component Diagram
Physical View	To map the software components and subsystems to physical hardware and the physical topology.	Developers, course staff, ATAM evaluators	Deployment Diagram

5 Architectural tactics

This section describes which architectural tactics are being used to meet the quality requirements: modifiability, usability, interoperability and performance.

5.1 Modifiability tactics

Modifiability tactics are being used to reduce the cost of changes made to a system. The tactics that are being implemented help us create a system that is able to accommodate changes with ease. The two main concepts for achieving high modifiability are: coupling and cohesion. Our main goal should be to reduce coupling and maximize cohesion in a modular design. In other words; maximizing the cohesion in each module and minimizing the coupling between modules.

We'll define cohesion and coupling from the concept of responsibilities. A responsibility is an action, a decision, or knowledge that is being preserved by a system or an element of the system. Coupling is the strength of the relationship between responsibilities. By knowing the strength of the relationship between responsibilities, the likelihood

of propagating changes can be calculated. To reduce coupling we can either reduce the relationship between modules or maximize the relationships between elements in the same module. Maximizing the relationships between elements in the same module is defined as cohesion. Next we'll define some of the tactics to be used for maximizing cohesion and reducing coupling.

5.1.1 Cohesion

To increase cohesion, we need to get the strongest possible relationship between responsibilities in one module. This can involve moving responsibilities between modules, or splitting responsibilities. By moving or splitting responsibilities we reduce the likelihood of side-effects to other responsibilities in the original module. We will use the following tactics to move or split responsibilities:

1. **Maintaining semantic coherence**
   Collocating responsibilities that are affected by a single modification such that the cost of modifying the new "collocated" module is less than modifying the original one.

2. **Abstracting common services**
   If two or more modules essentially provide a variant of the same service, this service could be implemented in a single module in a more general form. By doing this, any modification to the service would only need to occur once. This tactic is often used when refactoring code.

5.1.2 Coupling

To reduce coupling we can either reduce the relationship between modules or maximize the relationships between elements in the same module. Some of the best ways to reduce coupling are to generalize, abstract or to break dependencies by adding layers. We will use the following tactics to achieve these effects:

1. **Encapsulation**
   By introducing an explicit interface for a module, you reduce the chance of changes propagating to this module. In other words, you reduce the strength of coupling between the modules by placing an interface working as an API in front of the original module. By doing this you enforce information hiding and limit the ways other modules can interact with the original module.

2. **Wrapping**
   A form of encapsulation where the interface also transforms the data to "fit" the original module. The cost of changes made to the wrapper must obviously be less than the changes made to the original module, or else it makes little sense to implement a wrapper.

3. **Higher abstraction levels**
   Makes propagating changes less likely by implementing parameterized modules that are already equipped to handle differences in data-flow.

4. **Intermediates**
   Breaking dependencies by implementing an intermediate layer between the modules. Intermediates remove the knowledge modules have of each other, and creates a standard way of interaction for all modules requiring the same service from each other.

5.2 Usability tactics

The QA of Usability is about "how easy it is for a user to accomplish a desired task on a system and the kind of user support the system provides" (Bass, L., Clements, P., & Kazman, R. (2013). *Software architecture in practice* [chapter 11, pp. 175]. Upper Saddle River, NJ: Addison-Wesley). Tactics for this are split into **supporting user initiative** and **supporting system initiative**.
A properly designed GUI that follows design guidelines regarding the **affordance** and **consistency and standards** of elements on the screen goes a long way [4] [5]. For instance through allowing the player to control the sprite through using a "joystick"-element and buttons that have the physical affordance that allows control as well as following standards for how controls work in games.

5.3 Interoperability tactics

Interoperability is about how well two or more systems in a given context can usefully exchange information through interfaces. This is especially relevant for the multiplayer-feature of our game. Tactics related to this are split into the categories of **locate** and **manage interfaces**. As we have already decided on a server to use (for which the API is available at design time), we are aware of the interface that our application has to communicate with, so the relevant tactics will be about managing interfaces.
Tailor interface is probably most relevant to us. This allows adding and removing capabilities to an interface. Among these are buffering, which is relevant in our case as the clients and servers can have problems with variations in their connection. On the other hand all irrelevant capabilities can be "removed" through modifying their visibility in the interface.

5.4 Performance tactics

Performance tactics are implemented to make sure a system is being able to meet all it's timing requirements. Being able to meet timing requirements means being able to generate a response to an event arriving at the system within a given time constraint. "Events" are the triggers for the computation that needs to happen before a response is sent, and it can be single or a stream of requests. "Latency" is the time from the event

has arrived to a response has been generated.

When an event arrives at the system, it is either processed or blocked. This means that there are two main factors contributing to the latency of a system: resource consumption and blocked time. Resource consumption contributes to the latency of processing an event by physically or strategically restricting access to resources, these resources can be memory, CPU, data stores or bandwidth. The blocked time is increased if there are events competing for the same resource, if a resource is unavailable, or if the computation of an event is dependent on the results of other computations. Knowing which problems might occur, we can take a closer look at the three main categories for handling the resources of a system: demand, management, and arbitration.

5.4.1 Resource Demand

Demand is characterized by the frequency of events and how much resources each event consumes. To reduce latency, one tactic is therefore to reduce the resources required to process the events. This can either be done by implementing and/or improving the algorithms used for processing events, or removing intermediaries/layers for that are not needed, but can increase modifiability. The trade off between performance and modifiability is a common one when optimizing system architecture.
Other ways to reduce the latency can be to reduce the sampling rate for monitoring environmental variables, controlling the sampling frequency of external events, bounding the execution time for responding to an event, or bounding the queue size for event arrivals.

5.4.2 Resource management

Sometimes you cannot control the demand for resources, but you can control how those resources are managed. To reduce the time used to process each event you can:

1. **Use concurrency**
   Reduce the time to process events by processing them in parallel. You can either process similar streams of events one the same threads or by creating new threads to process different types of activities.

2. **Cache data or distribute computations**
   By caching data that doesn't change too often you can reduce the stress on the server. It is also good practice to distribute computations that are not critical to exploit the processing power of each clients computer.

3. **Increase available resources**
   Increasing computing power by using faster/more processors, adding memory or increasing network throughput can reduce latency. When scaling computing power it is generally considered most cost-effective to distribute the incoming events on several computing nodes (load balancing) rather than increasing the computing power of a single node.

5.4.3 Resource Arbitration

Whether it's processing power, network throughput or memory usage, the resources needs to be scheduled properly. The most effective way to schedule these resources is to know your system well and customize the scheduling strategies to the resources you know your system will need. Some of the most common scheduling policies are:

1. **First-in/First-out**
   All requests are processed in the order they joined the queue. This can be problematic if some requests are of higher priority than others.

2. **Fixed-priority**
   Requests are being rated by the source of the request, this ensures better service than the FIFO strategy because it ensures lower latency for higher-priority requests. This strategy also runs the risk of not providing fast service to low-priority, but important requests which have been downgraded because of the fixed priorities.

3. **Dynamic**
   Assigns the request to the next resource possible by ordering the incoming requests. An example of this is a load-balancer distributing the incoming requests in a cyclic order at fixed time intervals.

4. **Static**
   A cyclic scheduling strategy where pre-emption points and the sequence of assignment to the resource are determined offline.

6 Architectural patterns

The architecture will combine several different architectural patterns at different abstraction levels, to fulfill the requirements dictated by the quality attributes and architecturally significant attributes.

6.1 Backend as a Service (BaaS)

At the highest level, the game will have a cloud based client-server architecture. The game will utilize Google Play Game Services to manage authentication, authorization, peer-to-peer connections and possibly cloud-based storage and achievements.

6.2 Peer-to-Peer

Google Play Games Services internally sets up a peer-to-peer mesh network between all participants where clients can communicate directly with each other, rather than through the Google Play Games Services servers. This allows for lower latency between devices once the connection has been established as the devices can send and receive data directly.

6.3 Model-View-Controller (MVC)

Model-View-Controller and it's siblings Model-View-Presenter (MVP), Model-View-View Model and their variations are software architectural patterns aimed at separating an application's concerns. They describe ways to separate between user interface views, logic and the business models. The game client will use the MVP pattern at it's core to navigate between screens, such as when navigating the menu. This will aid in achieving the modifiability requirements.

6.4 Entity Component System (ECS)

Entity-Component-System is an architectural pattern that is mostly used for games. It follows the composition over inheritance principle, and allows greater flexibility by composing entities out of smaller individual components that can be mixed and matched. An ECS makes it easy to add new entities, and makes it easier to create complex entities. It also improves performance compared to MVC by allowing tight coupling between systems and the components they act upon. The client will use the ECS pattern to manage game entities, behaviour and relations between these. This should also aid in achieving the modifiability requirements.

7 Design patterns

Software design lower level patterns provide general solutions to problems that commonly occur in software development. All design patterns used in our game code are described in this section. The detail of where/how the patterns are used is described in Implementation document.

7.1 Creational patterns

Singleton The singleton pattern should be used when there is a need to guarantee that only one object of a certain class will exist at any time. This pattern will be utilized for application-wide configuration, settings and logging. This may be considered an anti-pattern by some, and usually don't contribute much to modifiability, but it does simplify some aspects of the implementation.

Factory pattern The factory pattern can be used to create based on configuration or external state. This pattern will be used to manage and instantiate various modifiable components of the game such as audio, textures, entities, animations. This pattern will help achieve modifiability.

Object pool Object pools promote object reuse by recycling them instead of disposing and creating new objects. This is especially useful for Android applications where allocating and freeing memory frequently can have a large impact on performance because of how the garbage collector works. This will be used for game entities and components

that appear and disappear often, such as projectiles and asteroids. This pattern will help achieve the required performance and user-experience by limiting garbage-collection jitter.

7.2 Behavioural patterns

Observer The Observer or publish-subscribe pattern allows an object to notify observers of changes in it's state without depending explicitly on the observers. The Observer pattern will be used as part of the ECS implementation to allow systems to observe changes in game entities, so the systems know which entities they need to update on the next iteration. It will likely also be used elsewhere in the implementation. This will help achieve modifiability.

State The state pattern allows objects to seemingly change class by altering it's behaviour at runtime using polymorphism. This pattern will be used to allow entities to change their state during their lifecycle, and is achieved through manipulating the components of entities. This will help achieve modifiability and simplify the class hierarchy.

Strategy The strategy pattern deals with how an object performs a certain task, and encapsulates the behaviour. Some entities will use the strategy pattern to change their behaviour during run-time, such as asteroids, NPCs and the player. The strategy will change through use of power-ups, and can have effects such as invulnerability and modified guns. This will also help achieve modifiability and simplify the class hierarchy.

7.3 Optimization patterns

Optimization patterns are optimizations that can be considered when and if performance becomes an issue. Investing a lot of effort on optimizations ahead of time is often considered bad practice, but one should have an overview of possible improvements and solutions if issues should arise.

Data locality Data locality can be used to optimize performance by utilizing the principle of locality. This is especially relevant when using a game loop combined with the update method to update sequences of objects. By considering how the objects are laid out in memory, one can predict and optimize cache usage. This pattern may be considered if the game has performance issues caused by cache misses.

Dirty flag Dirty flags can avoid unnecessary work by deferring calculations until the result is needed. This can be used to increase computational efficiency as part of performance tactics. This pattern is especially useful for eg. physics calculations, where objects with a velocity of zero do not need to be processed until the velocity changes.

Spatial partition Spatial partitioning allows efficient localization of objects by storing them in data-structures organized by their position in the game world. This can be used to limit the amount of objects that need to be processed for eg. collision detection and other entity-to-entity interactions. Like the dirty flag pattern, this can be used to increase computational efficiency as part of performance tactics.

7.4 Sequencing patterns

Sequencing patterns are mostly game-specific patterns that generalizes methods for processing objects in sequence.

Game loop The game loop is one of the most important patterns for most modern games. It allows one to decouple the progression of game-time from user input and processor speed. The game loop pattern will be used to update and render the entities of the game to the device screen at set intervals, and is part of the LibGDX framework.

Update method The update method pattern allows a game to simulate a collection of independent objects by requesting each object to process one frame of behaviour at a time. The ECS system will utilize the update method pattern to let each system process all related entities one step at the time.

8 Views

8.1 Logical

The overall logical view of figure 3 describes the main application classes of the game, in a class diagram. The Asteroids class is where it all starts by setting the main menu screen and loading assets.

The Presenter package contains two classes. The BasePresenter is used for menus. It is extended by screens that implement the menus and the tutorial for the game.

The BaseGamePresenter is extended by the singleplayer and multiplayer playscreens. The classes should have some similarities since both classes extends the GDX Screen class(Methods like pause, render, resume etc.). The main difference between them is that the BaseGamePresenter should handle the gameplay by updating the gameworld through the ECS engine that in turn updates and renders the gameworld. It is also responsible for updating the view that takes care of UI for the game, e.g. score, health. The BasePresenter on the other hand should only update the view(e.g. UI like text and buttons) since it should be used for the menus and tutorial screen.
The model package will primarily be the component part of the ECS framework, since they are the ones that contain the data.

There will be different views for each screen, and they will all extend the BaseView. Each view will be instantiated in their respective presenter class. The views contain UI interfaces.

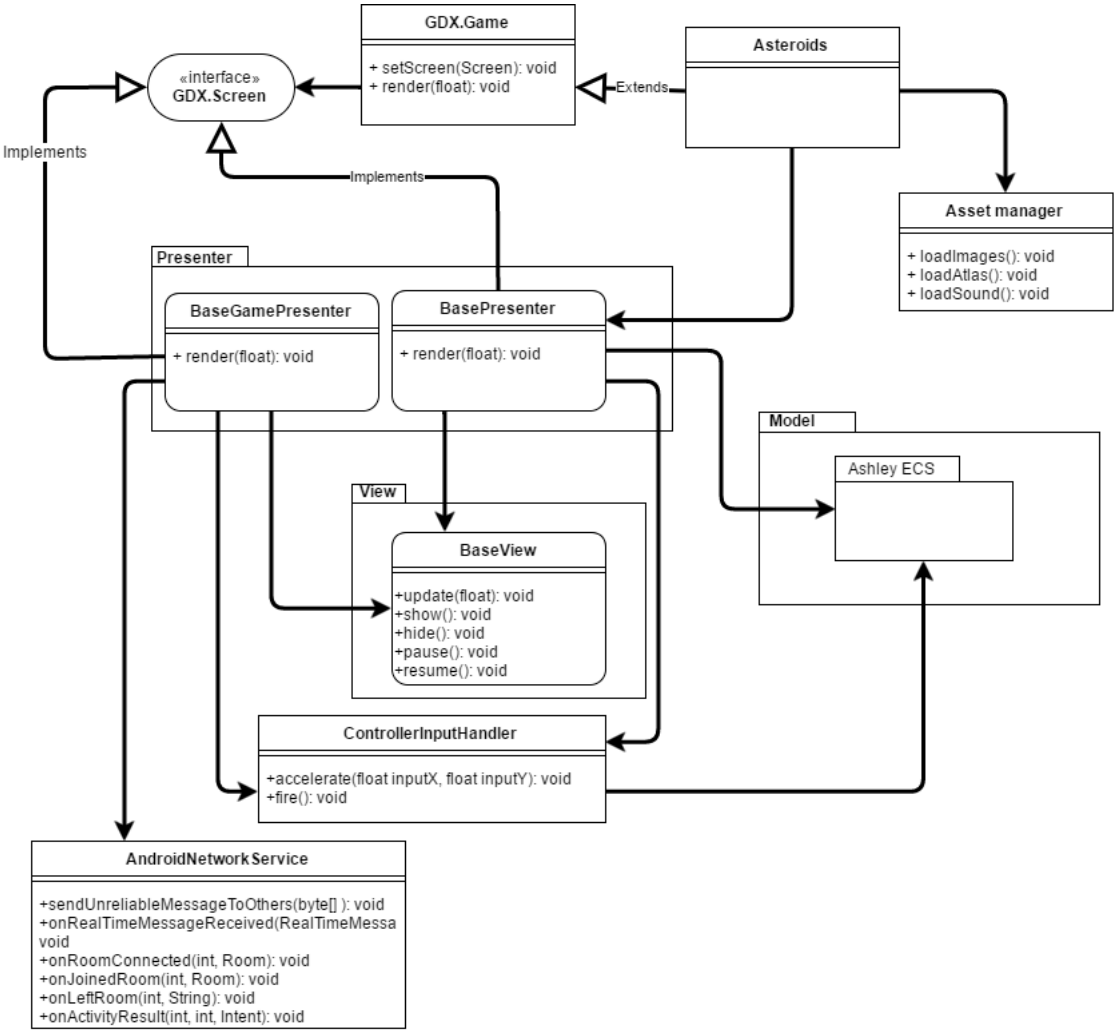

Figure 3: Logical View

The application will be utilizing the Ashley ECS framework. The following diagram (figure 4) depicts the classes in the framework, and how they relate to each other.

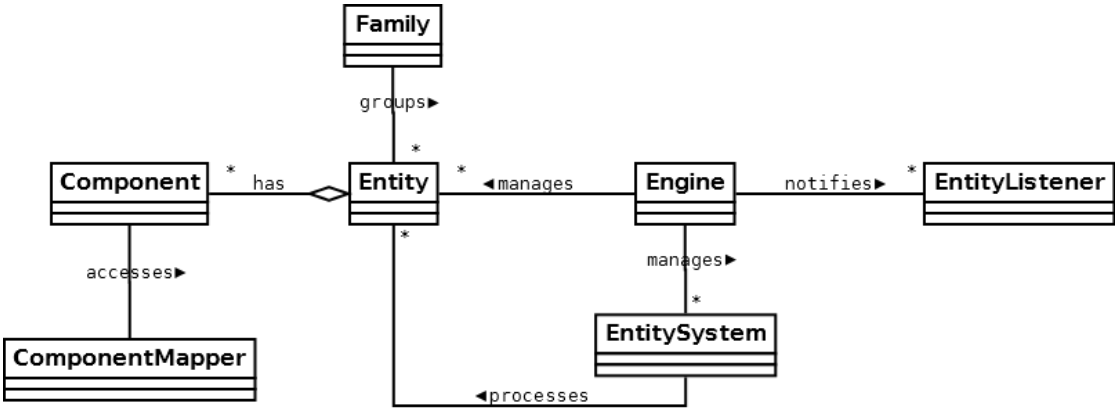

Figure 4: Ashley Framework

The ECS class diagram describes how the implementations of the ECS relate to each other. Each entity has a set of components. Each component contains a piece of data. Below is a figure with an early example of how we believe our implementation of ECS would be like. A player would for instance require both a position, velocity, a sprite-graphic etc., hence these components are added to the player entity (as illustrated by the arrows from the player entity to the components). Arrows from a system to entities displays which entities have components in the family of this system.

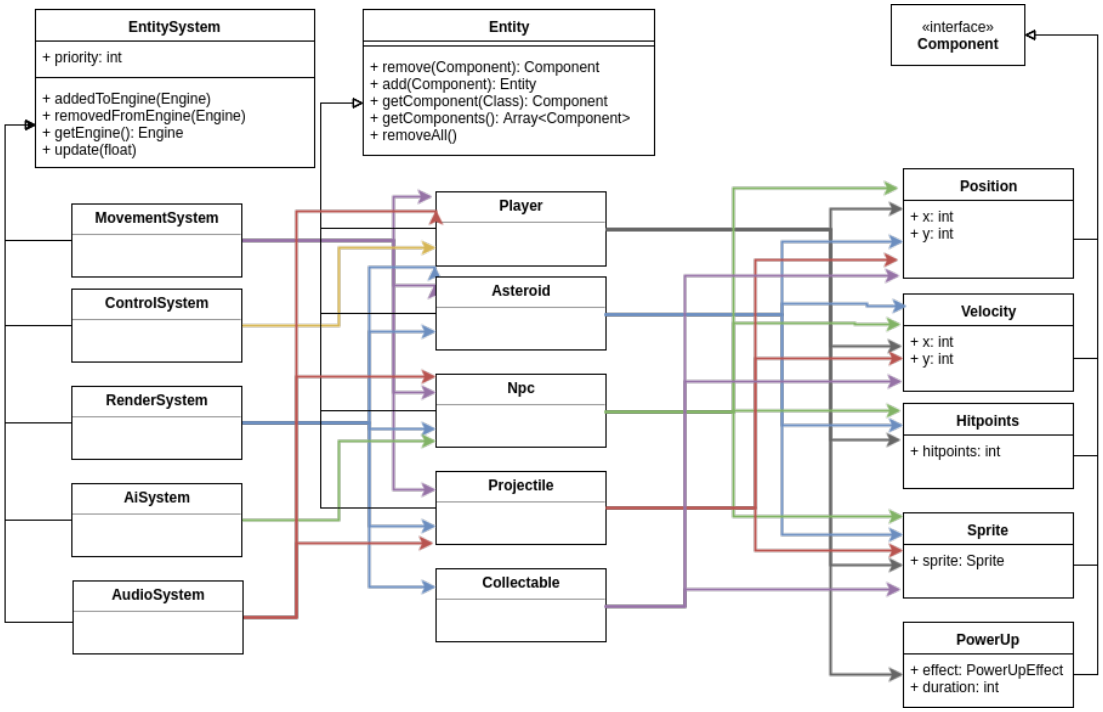

Figure 5: ECS classes

8.2 Process view

8.2.1 View/activity state machine

The state machine of figure 6 describes the flow of the application in terms of processes run, giving an indication of how the application switches between it's views.
It starts of when the application is first booted up. This initially puts the user in the MainView. From here the user is free to navigate around the application, using the in-game menu. Here the user can access the HighscoreActivity or AchievementsActivity (by pressing "Highscore" or Achiementes"), from both the user is able to return to the MainView using the top-left back arrow. The same goes for SettingsView and TutorialView (accesses through "Settings" and "Tutorial"), the user is able to navigate to these and can get back by either pressing "Back" in the TutorialView or "Save" in the SettingsView.
Pressing "Play" will send the user straight into singleplayer in a GameView. If "Multiplayer" is pressed, the user has the option to either start a quick game that sends the user directly into a multiplayer game, or invite their own players by first taking a detour into the SelectOpponentsActivity from which they all proceed to the GameView on the onMultiplayerGameStarting-call.
The GameView offers a Pause-button that displays an in-game menu. From this the user is able to "Quit to menu", taking the user back to MainView, or "Quit" to end the application. "Resume" updates the GameView to allow the user to keep playing. Hitting

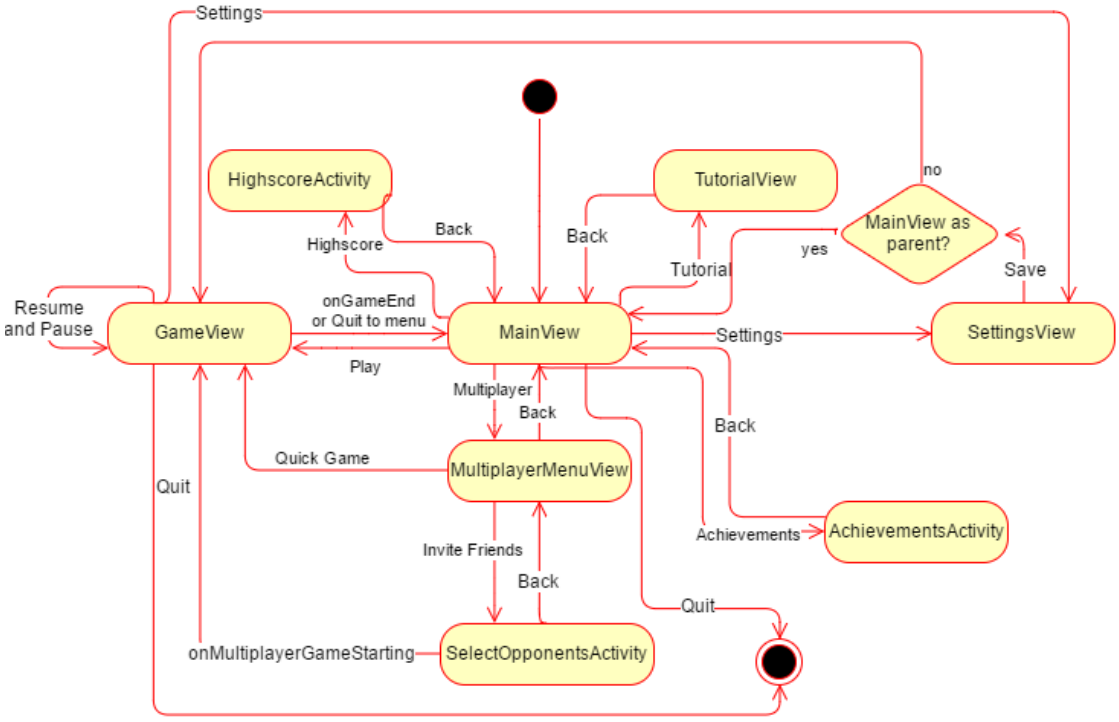

Figure 6: Process state machine, UML Activity Diagram notation

settings takes the user to the SettingsView, where ”Save” navigates back to GameView. The onGameEnd-call will take the user back to MainView upon the game being over. Finally, the user can quit the application from the MainView as well by hitting ”Quit”.

8.2.2 Sequence diagram

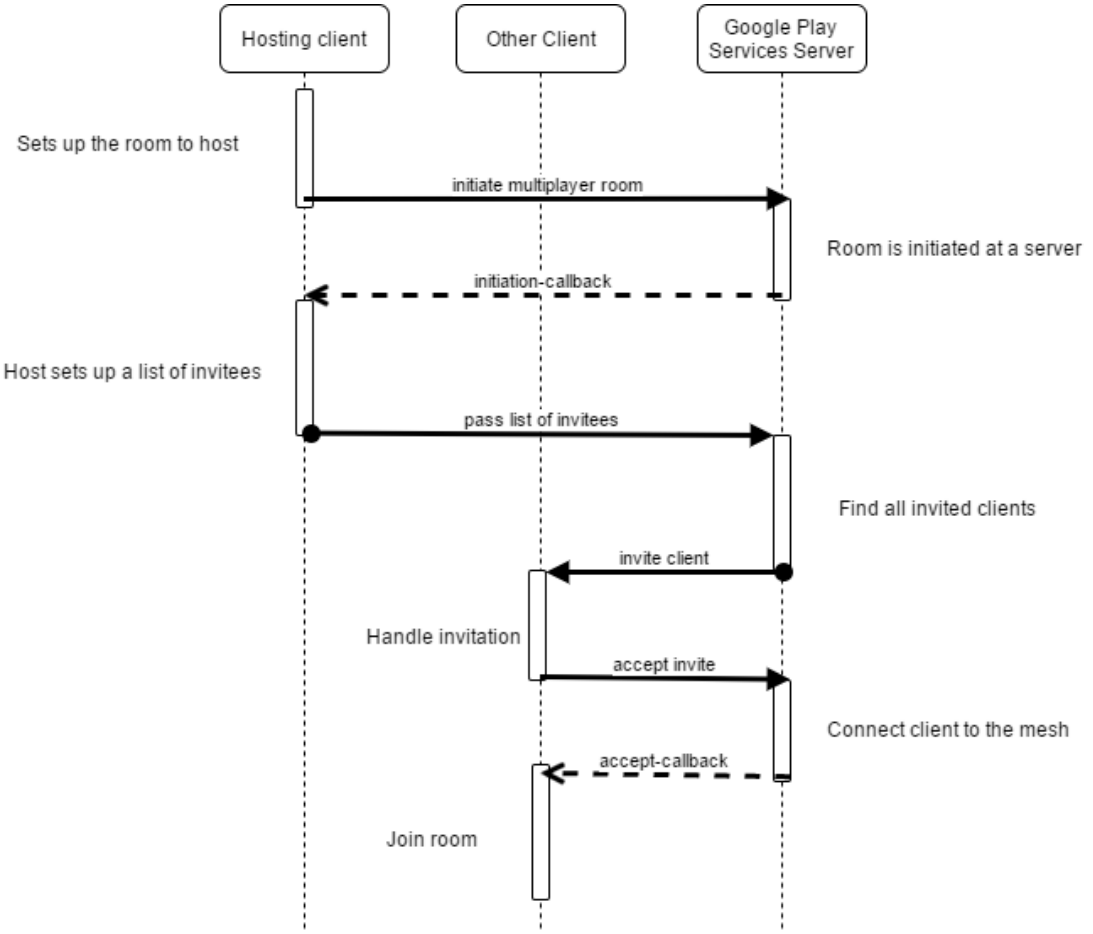

Figure 7: Sequence diagram of setting up a multiplayer room and inviting other clients

Figure 7 displays the sequence of setting up a multiplayer room with the Google Play Services multiplyer API. Initially one client decides to invite some friends to play with. A request to initiate a Room is sent to the Google Play Servers, and once a Room has been set up a callback is received. The hosting client now invites a list of other clients that is passed to the servers again. Now the servers locate the clients of the list and send an invitation to them. Upon accepting this invitation, the clients are connected to the P2P-mesh of the game and the clients are ready to communicate with each other and play a game.

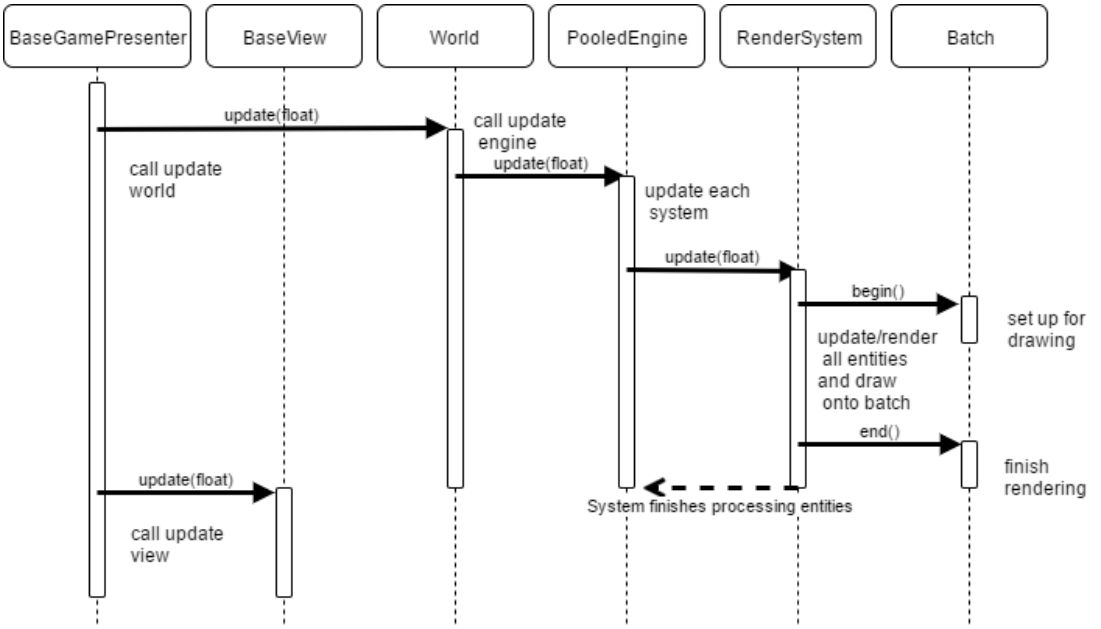

Figure 8: Sequence view of the game-loop continiously being run to update the state

The game continuously updates it's state, which is illustrated in figure 8. This is done through LibGDX, which makes the BaseGamePresenter start it's update-method with a given duration since last update. The presenter then updates the world, which in turn starts updating the pooled engine that updates all of the systems of ECS. Once the PooledEngine has updated all of it's systems, meaning that all entities have all their components processed. The view is also updated to reflect changes.
This view only shows one of the many systems in the Ashley-framework, but the logic is the same for all of them. The RenderSystem is unique in the fact that it also draws on the Batch, updating the graphics. Each system processes all the entities that have a given set of components. Each time an entity has one of it's components processed, they are updated based on their underlying logic. For instance the EffectComponents have their duration reduced by the amount of time since last update by the EffectSystem. Or as illustrated in the sequence diagram, the render system will update and transform then render all the relevant entities based on their drawable and transformation components.

8.3 Development view

The development view in figure 9 shows a simplified view of the main packages in the application. Many of the packages which interact with each other through interfaces, can be developed in parallel. The specifics of the interfaces has been left out of the diagram for simplicity, but the specifications can easily be viewed in the code. The dependencies of the packages has been greatly simplified to make the diagram more readable, and some packages have been left out. Eg. the presenter package virtually depends on every

other package, but makes it impossible to read the diagram if adding them. The ECS systems and components are tightly coupled, but can be developed independently from the rest of the application. IEffect and IShotHAndler define clear interfaces so they can be developed independently. Some services such as INetworkService and ISettingsService are defined by interfaces as they need to implemented specifically for the platform. In general it is very difficult and time consuming to define interfaces in advance, especially when the team is new to the platform, libraries and everything else used in the project.

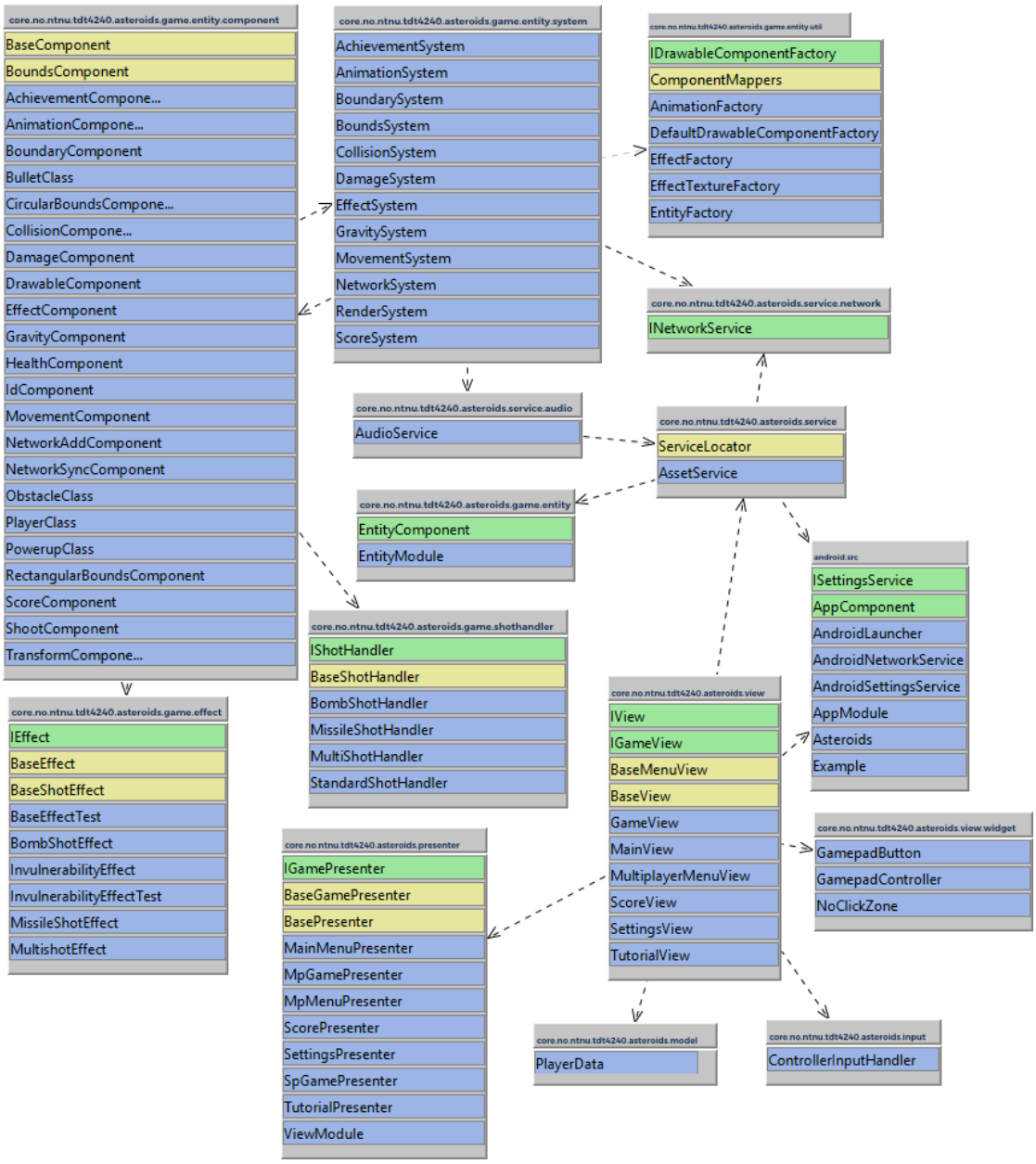

Figure 9: System packages

8.4 Physical view

Figure 10 - copied from [1] - describes the deployment of the application to android devices. The deployment is fairly straight-forward, and is the same regardless of the channel used to deploy it.

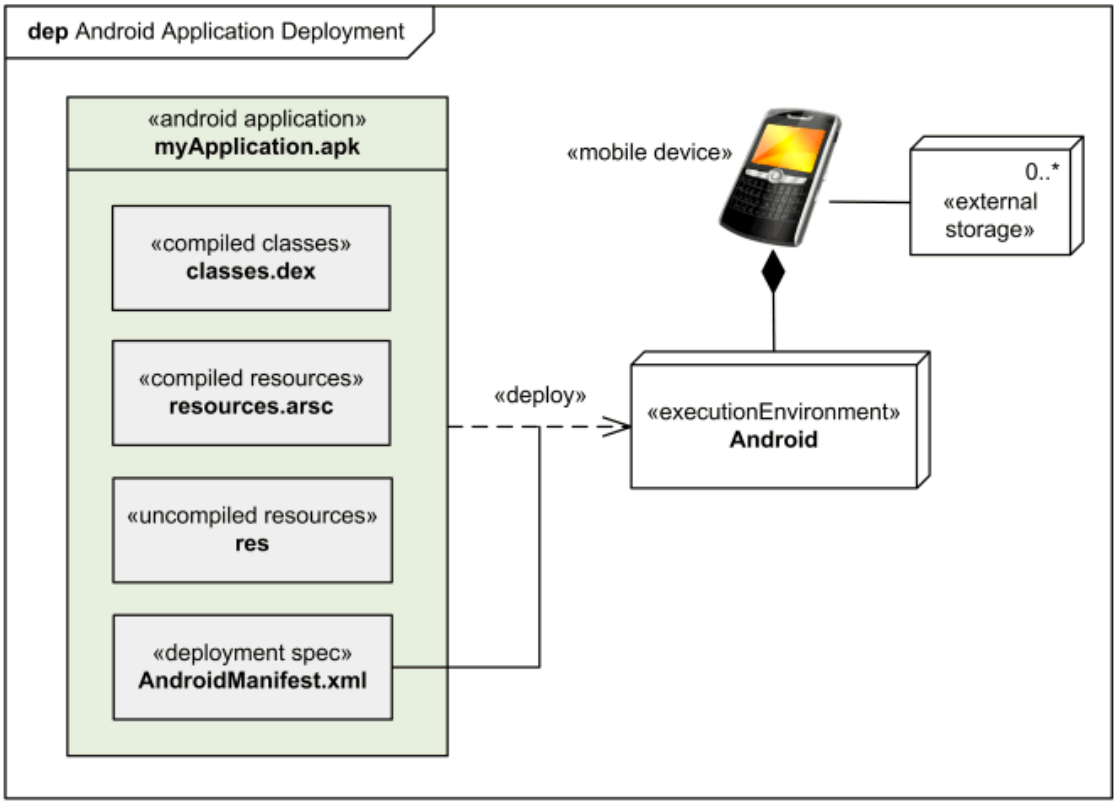

Figure 10: Android Application Deployment [1]

8.5 Inconsistencies between views

There are a few inconsistencies between our views. First of all the process view in figure 7 differs from the physical view as it does not indicate the existence of any servers.
Secondly the development view (which was made after the logical view) shows a more realistic structure of the application that the structure initially made and displayed in the logical class diagram. It also goes to greater lengths to describe the different presenters, views and components.
Also the process view of figure 8 by itself does not show any of the other systems, views etc. that are visible in the development view.

9 Architectural Rationale

The overall server-client cloud based architecture is dictated by the wish to integrate with Google Play Services. This further dictates the use of a peer-to-peer network architecture as Google Play Services sets up a peer-to-peer mesh network between all participants, where clients can communicate with each other directly rather than through the Google Play game services servers.

Model-View-Controller or one of it's many variations has become a de-facto standard for most UI driven applications. MVC style architectures are well suited for applications where the primary driver comes through the user interface in terms of user input. The MVC pattern will be used as a high-level application architecture to navigate between screens, specifically when navigating the menu screens. The MVC pattern will help achieve modifiability.

MVC will not be used for the gameplay part of the game, as it is not very well suited for games that are driven by a game loop rather than user input.

Entity-Component-System is a fairly new pattern specifically targeting at games. It follows the composition over inheritance principle and allows for a much cleaner separation than it's hierarchical counterparts. The ECS pattern will also aid in achieving modifiability.

Our main quality focus is modifiability, which is strengthened and well proven with MVC (and its variations) and ECS architectural patterns.

The secondary quality attributes performance and interoperability is addressed by Google Play Service and its peer-to-peer meshed network architecture. Google Play Service provides API towards different clients, and its meshed network architecture improves performance as data can be sent directly between players. The latter benefit can vanish when number of players increases, but it's no issue for us because right now the limitation is eight players in one playroom.

The other secondary quality attribute usability is also addressed by MVC (or its variations), which separates user interface from back end, hence user interface can be implemented simple and easy to use.

10 Issues

Setting up a complete overview of all the classes proved to be a challenging task for our project. Hence in the logical view we tried to generalize and explain further in the textual description of the view.
As the project grew, the architecture became quite complex. Maybe even a bit too complex for what this project actually required. This led to a lot of additional work, but in the end made for a pretty decent architecture, that could be easily reused for another game.

11 Changes

This section records change history for this document.

Date	Change History	Comments
February 27, 2017	First released version	None
March 17, 2017	Some corrections based feedback from teacher	Game name added on front page. Citation corrected. Chapter 'Changes' moved towards the end.
April 18, 2017	Changes were made to the views, better reflecting the structure of the application and taking feedback into account.	None.

12 Bibliography

[1] UML Diagrams. Android application uml deployment diagram. `http://www.uml-diagrams.org/android-application-uml-deployment-diagram-example.html`.

[2] Phillipe B. Kruchten. The 4+1 view model of architecture. *IEEE Software Magazine*, 12, 1995.

[3] Rick Kazman Len Bass, Paul Clements. *Software Architecture in Practice - Third Edition*. Addison-Wesley, 2012.

[4] Jacob Nielsen. 10 usability heuristics for user interface design. `https://www.nngroup.com/articles/ten-usability-heuristics/`, 1995.

[5] Donald A. Norman. *The Design of Everyday Things*. Basic Books, 1988.