TDT4240 Software Architecture

Architectural Description Document

Stigespillet
A Simple Norwegian Game

Group 26
Kristoffer Ravik Andresen
Vegard Bjerkli Bugge
Steinar Bækkedal
Mathieu Remaut Lund
Christoffer Evjen Ottesen
Stian Torjussen

Primary quality attribute: Modifiability
Secondary quality attributes: Usability, Testability

Chosen COTS: Android SDK, LibGDX

March 7, 2016
Spring 2016

Contents

Introduction

Our assignment in this project was to create a multiplayer game. We chose the Android platform as we are more familiar with this than iOS, and we also decided to make the multiplayer functionality local to avoid network complexity. To design the game we will make use of the game-development application framework LibGDX, which has an informative wiki at its own GitHub page [LibGDX 2016]. An important requirement for the game is that it must fulfill quality requirements on modifiability specified in the requirements specification. This document discusses the requirements part of the project, and it will therefore define functional and quality requirements. It will also contain an introduction to the concept of the game we are implementing.

Concept

The game is a 2D board-game for minimum 2 players, inspired by the classical Snakes and Ladders game. Our MVP (minimum viable product) will be very similar to the original game, where all players begin at the start field and move towards the finish field based on the outcome of the dice. Whenever a player lands on a field with a ladder or a snake, they follow it to the end.The first player to reach the finish field is the winner.
For now, we have the following ideas for extensions:
Adding a chance field (like in Monopoly) that may give the player either an advantage or a disadvantage.
Colour coded ladders, that limits which ladders a player can climb (for instance a player can be restricted to only climb red ladders).

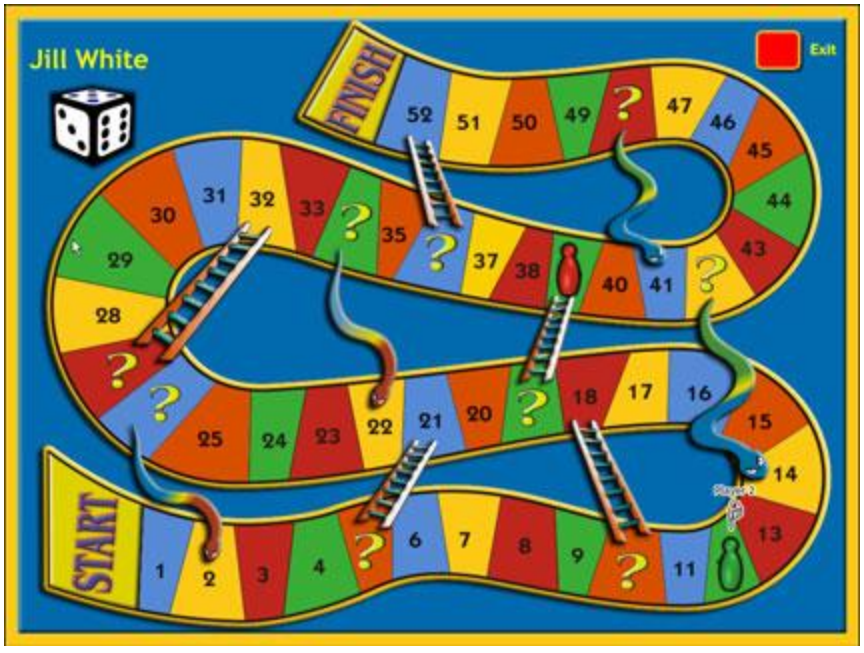


Figure 1: An example board of the game "Ladders And Snakes".

Architectural Drivers

We believe that the functional and quality requirements will be more important than the business requirements for this project, since games with a similar concept already exist on Google Play store. We are nevertheless going to work hard to achieve an architecture of high quality, with all the quality attributes that we have chosen. We will produce a playable game for the Android platform. Expectations from stakeholders have influenced our choice of architecture, here especially the requirement on modifiability for future developers. We also want to limit the workload on the project as many groups fail to estimate the time they spend, so we have a target workload for each member of 30 hours in the development phase. As this game is not developed for a customer, most budget constraints other than workload and constraints related to time may be ignored.

None in the development team have ever made a game for Android using the LibGDX framework before. We plan to make a base system using the package Scene2D [LibGDX 2016] to make the UI, and attempt to expand functionality in the game from there.

In total, we have six – 6 – weeks, to not only implement the base architecture modelled through a viewpoint consisting of the three views given based off [Kruchten 1995], but to test, increment, polish, modify and finish the product. The base implementation process must be done quickly.

Stakeholders

A stakeholder is anyone who either 1 - is involved with acquiring the right to use the system, 2 - is related to economical, security - related and/or other aspects of the system and/or 3 - has similar relations to subsystems of the system. A vast array of different people with different knowledges and different perspective may be stakeholders of a system. In Real life (TM), many of the interests of these stakeholders may collide and the software architect will have to make compromises of these colliding interests, or even sacrifice the most unrealistic ones. In this project, we in the group have not spotted any such collision of interests. To satisfy all relevant stakeholders, one must be aware of all the different stakeholders within a particular project. We have defined four different stakeholder roles of our project in the following table::

Table 1: Definition of stakeholder roles referred to in this document.

Name	Comment
Architect	The architect is creating the architecture.
Developer	The developers have a need for a project on which they can realize an architecture.
Tester	The tester is interested in testing the usability of the game and testing the architecture
User	The end users play the game. They are interested in our game because they'd like to play a mobile version of Stigespillet.
Evaluator	Those who actually read this document. Happy reading.

Once the relevant stakeholders have been identified, we can move on to the choice of architecture. It is important to select an architecture which helps to achieve the needs and expectations of the involved people and their steaks.

Table 2: Stakeholders and their concerns for the architecture.

Role	People	Concerns for the architecture
Architect	Everyone in the group	Making compromises between stakeholders' requirements. Must prove that the proposed architecture meets requirement specification as well as possible.
Developer	Everyone in the group	Understanding the software architecture and implementing it in the software project, being able to extend and modify the system.
Tester	Everyone in the group, TA	Creating tests that measure how well the implemented software meets its requirements.
User	Everyone playing the game, primarily kids in the age 8 - 15	Checking whether the promised functionality has been delivered to them.
Evaluator	TA, Evaluation team in ATAM exercise.	Checking whether the architecture has been properly implemented, reviewing our competence.

Architectural Viewpoint Selection

Table 3: Viewpoint selection.

View	**Primary related stakeholders**	**Used notation**	**Goals**
Logical	Users	UML Class Diagram	Give an overview of the logical design, mainly in relation to the functional requirements.
Development	Developers	Layered Diagram	Give an overview of the different parts that must be implemented, and make it easier to allocate tasks that can be performed in parallel.
Process	Evaluators	State Diagram	Give an overview of the design and its relation to the non-functional (quality) requirements.
Scenarios	Testers	Use Case Diagram	Validate current architecture, and drive architectural increment.

Logical view

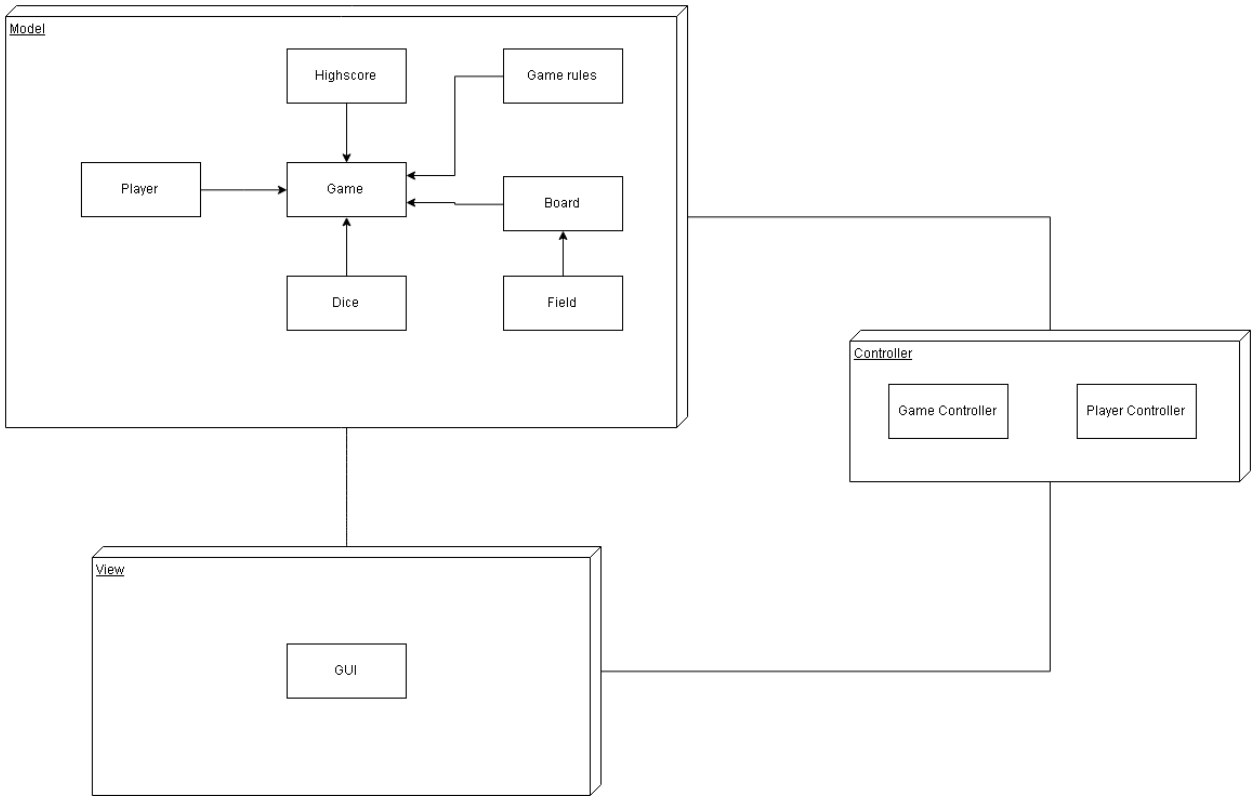
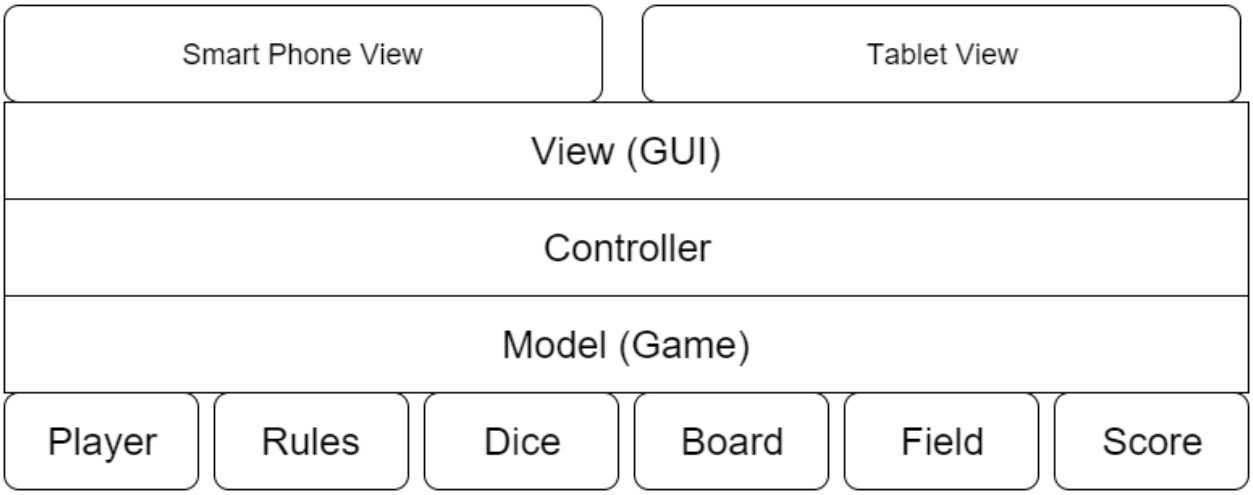


Figure 2: A class diagram with an overview of the classes in relation to the MVC-architecture.

The development team has also made note by looking into a few example projects that use LibGDX that view elements are gathered in a package called “Assets”, which we for now simply call “GUI”.

Development view



Again, LibGDX uses the term “Assets” when speaking about the package of classes that concentrate on view logic.

Process view

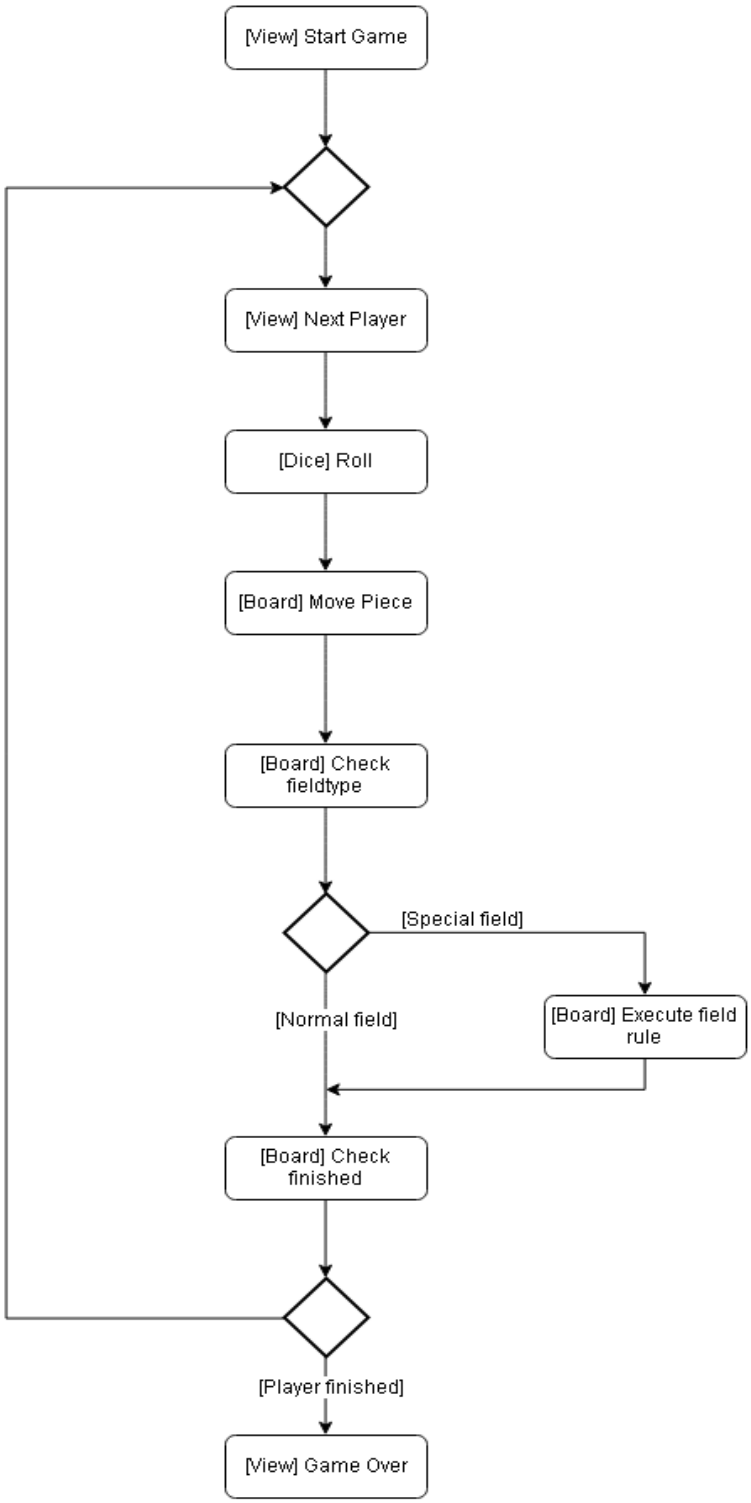


Figure 3: A state diagram with an overview of the possible states of the game.

Architectural Tactics

Tactics to achieve modifiability

There are two important measures that constitute the degree of how easy and fast it is to make changes on a software system.

We will want the responsibilities each module in the architecture has to be as strongly related as possible. Each module should not have disjunct responsibilities. This means that if a change that is to be made affects several related responsibilities, we want to have to make changes to one or more modules. We want to achieve high cohesion.

The other requirement to achieve a high degree of modifiability is to maintain loose coupling. We do not want software elements related and connected to each other be distributed among several modules, as a change in one of these elements would propagate changes in many affected modules.

We will here list our preferred architectural tactics which helps us achieving high cohesion and loose coupling within our architecture.

Reduce size of modules

Module splitting proves beneficial to the time spent on average for each change in a module. If a module contains two sets of interconnected elements it is desirable to extract one of which into a new module. Thus changes are easier to make and contain. In the case of our project we will for example make efforts to separate the logic for viewing the dice onscreen into a class by itself as well as writing the logic for validating input on the dice in its own controller class. Thirdly the dice should be represented in its own model class. Thus changes regarding the dice as a whole is contained to its own classes in each of three layers of the MVC architecture.

If two methods do not serve the same abstracted purpose we will separate these two into two separate classes, either through defining one, two or no new classes.

Increase cohesion

The curriculum [BCK 2012] proposes the following tactic to increase cohesion in the architecture's modules.

The name of each module must reflect its designated areas of responsibilities. Implementation - wise this means that each class has a name that clearly communicates what the class is supposed to do. This allows us to easily pinpoint classes that requires editing when changes are imminent. With this tactic we will maintain a one-to-one relation between each class and the number of purposes they serve, and each class will do no more than what it is supposed to do.

Reduce coupling

One good strategy to avoid tight coupling is to route all communication between the modules via so-called intermediaries. In MVC the controller classes play the role as intermediaries between the view logic on the data model classes. The purpose of the controller classes is to prohibit the user from always changing the application data whenever they interact with the user interface, in case they input illegal variables. Other benefits with this separation of logic include more cohesive modules.

If several classes contain duplicate code then a change in the code in one class would strictly necessitate equal changes in all classes with these duplicate lines. An effectivisation of this editing process would be achieved through making an abstract superclass for all the nearly equal classes. Differences between the subclasses may be implemented through overriding functionality inherited from the superclass. This tactic is called "abstract common services" [BCK 2012, p.124]. By changing the code in the abstract class, we instantly change the behaviour of all its subclasses. This strategy also contributes to introduce polymorphism to the architecture, a binding deferring tactic that binds values at runtime [BCK 2012, p.125].

Another good tactic to loosen the coupling between the modules is to restrict dependencies between them. In Java this means to have classes implement an interface to communicate with certain other classes through. For the MVC pattern, interfaces ensure that the view classes are blissfully unaware of their equivalent classes in the model layer. In most layered architectures, where modules in each layer may only communicate with modules in neighbouring layers, dependency restriction is an often employed tactic.

Our final strategy to achieve loose coupling is to hide information in the modules, i.e. encapsulation, the tactic in which a module's data variables are hidden behind an interface that everyone must call for when accessing the protected variables. In Java this is realized with visibility modifiers (normally private) and public getter and setter methods. A setter method changes one variable - usually indicated by its name, and the getter method returns the value of a variable.

Defer binding

The final category of tactics stresses flexibility in the design artifacts. The more changehandling that can be automatized, the better. The more general functions are coded, the better. The more parametrization in the methods, the later values are binded to the modules. However, implementing mechanisms that facilitate late value bindings come at great performace costs. Of all the different binding deferring tactics the curriculum introduces to us, we will go with binding values at deployment through introducing game configuration to the software. Naturally, before a game starts the players might want to define how many that are playing, the graphical design of the board they are playing on, how large the board is, et cetera. This ensures that there is a whole range of different setups a game can be instantiated in, and giving the players as many choices as possible benefits usability as well as the modifiability. To achieve our binding goals, parametrization is key.

Tactics to achieve testability

The main reason testability is an important quality attribute for us is that we are developing the game incrementally, which means that we add one feature atop another one module at the time. There are mainly two categories of testability tactics. One focuses on making the software system states observable and controllable. The other focuses on limiting the system's complexity, both behavioral and structural. We intend to test modules by themselves (unit testing) as well as running the entire game through emulation of an Android device, a feature supported by the Android SDK and comes bundled with the IDE Android Studio. [BCK 2012, p. 167] also states that a high degree of modifiability positively affects testability, because of the ease of adding testing functionality to the system.

Controlling and observing the system state

The emulator provides a sandbox for us to play the game as we like, without jeopardizing or breaking a real Android device. Hardware related worries such as broken screen and depleting battery life are non-existent. Once the emulated device is turned off, the device is reset to factory zero, aside from the user data image file in which player scores are persisted. The stored user data may be used to verify that models are behaving nicely.

The **Executable Assertions** tactic [BCK 2012, p.166] is a testability tactic in which assertion flags are coded, usually in separate testing classes or a testing interface, against which components are run. If our asserted value test returns failure, then something is wrong with the software unit's behaviour. The assertions are functions that compare the output of an operation/method with certain arguments against an expected, specified value constraint. This tactic may be realized through the use of a testing framework such as JUnit. The purpose of the Executable Assertions tactic is to increase the system's observability.

Limiting complexity

Our specified tactics for achieve loose coupling between system modules helps a lot with limiting system complexity, which is measured by computing the average number of methods in a module plus all calls to external methods in the same module. Encapsulation and intermediaries are typical coupling strategies that contribute to good testability.

As the software we are making does not employ online multiplayer, we sneakily avoid concurrency issues such as race conditions. The game will never run in multiple parallel instances depending on each other, but on one Android device at the time. As the game is made up by clear rules it is wholly deterministic, which makes testing as easy an ordeal as possible.

Tactics to achieve usability

Most software systems used in a productivity environment that employs a user interface focus heavily on supporting user initiative. After all, if a user feels that they are unable to efficiently use a word processor or calculator program, then that software is useless and the fault is on its authors. Usability tactics are divided into two kinds of initiative support, either for the user or the system. User Initiative Support centres around functionality for cancel, undo, pausing and resuming the software. Even though the game is turn based, we believe implementing pause and resume functionality gives the players, especially young players, a way to temporarily stop the gameplay when being called for dinner. "But Mom, we have to finish playing Snake and Ladders first!" We will look into ways to make the application eat less battery life when paused.

The support system initiative tactics are usability tactics centred around functionality that intends to assist the user in doing things on the software [BCK 2012, p. 180]. We will make the running game maintain a model of itself to explain how the game is going, for when the dice has rolled, it would be nice if the player could see at once where they were going. As the rules for Snakes And Ladders do not permit the users to go in any direction they please, the amount of initiative needed aside when rolling the dice is next to none. It is important for us to make it easy for the user to roll the dice through a simple touch on the screen.

Upon configuring the game, the user must be aware of the consequences of their choices. We will attempt to enlighten them through showing this on an image onscreen that varies with whatever is selected on the configuration menu. If the user opts to increase the size of the board, the preview window will show a larger board along with an estimated playtime displayed beneath the picture. This is a concrete strategy for a task model.

Architectural pattern - Model View Controller

We are going to base the architecture of our game on the MVC architectural pattern. This Component-and-Connector pattern divides the software into three layers - the model, the view and the controller. This division allows the developers to separate how information is represented internally in the system (the model), and externally to the user (the view).

The model is the core component of the system, and manages the data in the system. It can be thought of as the component that rules the system, and defines the logic. But the information that is represented internally in the model will be very confusing for the user. Thus, one needs a view, that is, any output representation of the data, like a diagram or a person form to fill out. So the view is basically a nice presentation of the internal data that is easy for the user to understand. It is also common to use different views on the same information, depending on the users IT level. The third component, the controller, handles the the inputs from the user, and converts it to actions to manipulate the underlying model. The model then updates the view with the new information, so the user always sees how things have changed [BCK 2012, p. 212 - 213].

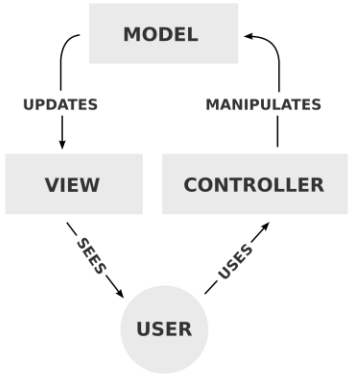


Figure 4: MVC, simply explained.

Rationale

The main benefit for us using the MVC pattern, is that it allows us to separate the functionality in the three different parts. This implies that as long as we have well defined interfaces for how they react, the different parts can be developed and changed independent of each other. First of all, this is an advantage in the development phase, allowing us to split up the work neatly between the group members. Secondly, it is very good for modifiability, as one can modify the different parts independently if new technology or techniques should be introduced in the evolution phase. The modifiability part is particularly interesting for us, as it is very likely that developers in the evolution phase would like to introduce new levels or new sets of rules for the game. By using the MVC pattern, this can be done in a very neat manner.

Furthermore, MVC is well suited for testability, as it is easier to isolate and test the individual parts. It is also well documented, and works great with both the programming language Java and the COTS frameworks which we intend to use.

Design patterns suitable to work with on the project

Singleton

We are going to use the Singleton pattern, to avoid instantiating more than one board object for a game. A lot of information will be stored in the board object, so to avoid unnecessary resource consumption the Singleton pattern will work greatly.

The main idea of the Singleton pattern is that you are not allowed to call a constructor publicly. You rather instantiate the class by calling a getObject() method that checks if there already exists an instance of the class. If so, the function returns a pointer to this instance. If not, it creates a new instance and then returns a pointer to this instance.

There are different tactics to implement this pattern, including lazy instantiation, eager initialization, static block initialization, initialization-on-demand and enum [Wikipedia 2016]. No class may extend a Singleton Class, as this would instantiate several Singleton objects, defeating the purpose of using the pattern. Another drawback with the Singleton Pattern is that it may hide instantiation dependencies

Factory

The meaning of the Factory Pattern is to break the dependency between an object (the **client**) that requires the existence of an instance of another class, called the **product**. In Java, this need is usually satisfied by calling on a public constructor, a method in the class that gives birth to a new, unique instance of the class. Calling on a constructor to create a necessary object also creates a direct association between the client and the product [Purdy 2002]. In general, the Factory pattern introduces an intermediary through which the client may instantiate different products.

In the case of this project the we are interested in having a Dice object to play with and a set of Players that play on the Board singleton. In order to avoid having to couple the Player objects to tight with other model objects, we will program in a PlayerFactory, whose sole purpose is to receive an order for how many players to create on the board, and then create them and add them to the running game instance.

State

LibGDX, the COTS framework for developing games in Android, provides support for coding with not only MVC but also the State Pattern. The rationale behind representing behavior of Game elements into States are the following system constraints: Our Game may be paused, during which it should not be actively occupying hardware resources, as Android device have little primary memory to spare, not to mention the device's CPU capabilities is not on par with that of the typical modern desktop PC. Furthermore, the game may be just about to be starting, which is when the players configure the board, register their names for the game statistics and so on. The game may be running, during which players move around on the board and player specific statistics such as number of dice throws, field moves, chance events occured et cetera, are being counted and/or aggregated. Finally, the game may be finished or cancelled, at these points of its life cycle the recorded stats are persisted to a flat, unstructured game file. We could have represented each of these game stats as one boolean variable for each state, but this would be propagate to a vast number of nested if - else if - loops and/or nested while loops, which is a very daunting task to modify and update when a change is to be implemented or the game is to be incremented.

Strategy

We would like each board to contain different kinds of fields each player may land on. A player may arrive at a Ladder field, in which they are forced to teleport to either a previous ("snake") or a later ("ladder") field in the ordered list of fields the board consists of. The field may also be a normal field that does nothing for you aside from being the field the player is standing on, or it may be a Chance field.

Issues

The members of Group 26 feel that they have accomplished the task of producing this architectural description to a certain point of satisfactory. About 80 per cent of the contents in this document was produced, reviewed and celebrated eight hours before deadline, whereas the requirements were mostly finished several days beforehand. The team is still determined to better the communication when producing these kinds of texts, and while chatting on Google Docs is already being well used, mapping the curriculum to our architectural activities has proven to be a challenge, as everyone in Group 26 has not necessarily read the book along with the lectures. To deal with this shortcoming of knowledge, we plan to arrange code nights where we review lecture slides while reviewing our own specifications, this architecture document and the requirements document, and our own views and beliefs on the implementation process.

An issue related to LibGDX is found in scene graph package Scene2D [LibGDX 2016]. When using Scene2D to build not only a UI, but an entire application, its Actor classes couple view logic (draw() - methods) with logic related to editing the model (act() - methods). We will resolve this issue by providing our own way of interacting with models, and we will make the scene graph interact with our own Controller intermediaries.

We are also fully aware that the architectural document, as well as the requirements specification are subject to change as the project progresses and new extensions and their implications to the architecture are to be made and added to each corresponding document.

References

- [BCK 2012]: Len Bass, Paul Clements, Rick Kazman. “Software Architecture in Practice – Third edition”. Addison Wesley. September 2012.
- [Purdy 2002]: Doug Purdy. Microsoft Corporation. February 2002. Available at https://msdn.microsoft.com/en-us/library/ee817667.aspx. Accessed 6.3.2016.
- [Wikipedia 2016]: Wikipedia, The Free Encyclopedia. Singleton pattern. Available at https://en.wikipedia.org/wiki/Singleton_pattern. Updated 23.2.2016. Accessed 5.3.2016.
- [LibGDX 2016]: LibGDX Wiki. https://github.com/libgdx/libgdx/wiki. Accessed 6.3.2016.
- [Kruchten 1995]: Phillipe B. Kruchten. “The 4+1 View Model Of architecture”. IEEE Software Magazine, Volume 12, Issue 6. 1995.