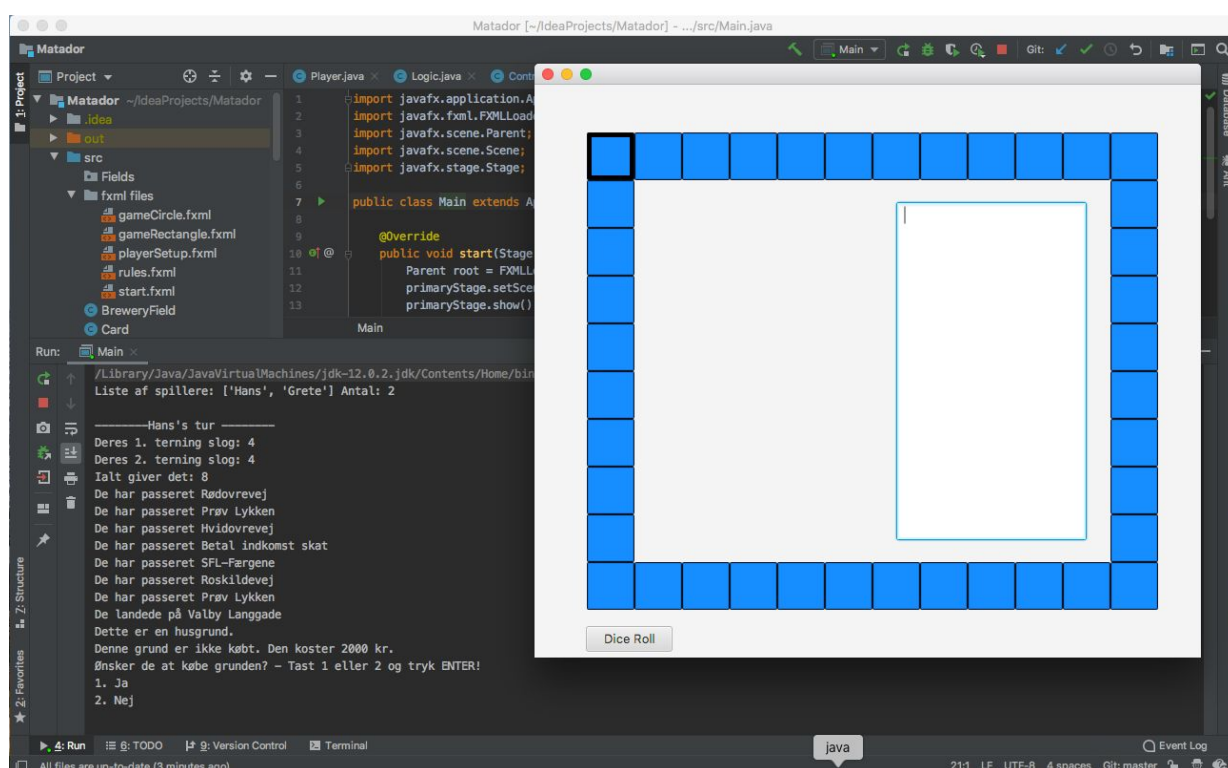


MATADOR

Digitize a Board Game



Subject Module Project in Computer-Science

Roskilde University

Group S2026631-5 (4.semester):

Mads Køie Nielsen #65199

Mads Marker Gelardi Jönsson #66384

Stefan Sverud #66858

Stig Anders Fage-Pedersen #66706

Project Supervisor:

Sune T.B. Nielsen

Abstract

This report seeks to discover what it means to incorporate Object Oriented Programming (OOP), Inheritance and data structures, such as Model View Controller (MVC), into a coding project. The project will describe the process of creating a digital version of the Danish board game Matador, with the above mentioned methods used. The programming language used is Java and JavaFX for creating the Graphical User Interface. To showcase our developed system the rapport will contain different diagrams to visualize our game systems behavior and structure by using Unified Modeling Language (UML). This also gave us several insights into our own project. We have furthermore gone into details describing the tools of Kanban board and Github and our use of these tools. The project will also explore our use of testing methods and explain bugs found after 'finishing' our code. Lastly, we will discuss how we might have tackled the project better with the lessons we have learned.

Links

Github program code: <https://github.com/jonsson0/Matador.git>

User guide: [Program setup](#)

Use-case diagram:

<https://app.lucidchart.com/invitations/accept/e960170e-f54d-4d78-901c-c0f4c213a6d9>

Class diagram:

<https://app.lucidchart.com/invitations/accept/80bd0e03-5258-4e04-9f26-51bdf46be89>

<https://app.lucidchart.com/invitations/accept/f8af90e4-a229-465f-99a4-9c78dfbcc358>

Activity diagram:

<https://app.lucidchart.com/invitations/accept/7be4f6df-fb52-4121-a85a-231bf12ac4a2>

Table of Contents

Abstract	2
Links	3
1. Motivation	7
2. Introduction	7
3. Problem area	8
3.1 Problem formulation	8
3.2 Research Area	9
4. Workflow	10
4.1 Project management	10
4.2 Kanban Board	11
4.2.1 Use of Kanban	12
4.3 VCS - Version Control System	14
4.3.1 Git	14
4.3.2 Github	14
4.3.3 Problems we encountered with VCS	15
5. UML	17
5.1 Use-Case Diagram	18
5.1.1 Our Use-Case Diagram	19
5.2 Class Diagram	21
5.2.1 Our Class Diagram	22
5.3 Activity Diagram	26
5.3.1 Our Activity Diagram	27
5.4 Acquired knowledge	29
6. Object-Oriented Programming	30
6.1 Inheritance	30
6.1.1 Our swap to inheritance	31
6.2 Our use of OOP	32
6.2.1 Use of objects	32
6.2.2 Data Structures	32
6.2.3 Why use OOP?	33
6.2.4 OOP's influence on the design	33
7. MVC Structure & System Requirements	34
7.1 MVC - Model-View-Controller	34
7.2 System Specifications	35
7.2.1 User Friendliness	36
8. Description of the Program	37

8.1 Class overview	37
8.2 Class Description	38
8.3 Bugs	43
8.3.1 Bugs we have encountered	44
8.3.2 Current bugs in our game	46
8.4 Building the game logic	46
8.5 Creating a GUI for the game	46
8.6 Adding the GUI to the game logic	50
9. User Guide	53
9.1 JavaFX	53
9.2 VM options	53
9.3 Running the game	54
9.3.1 Start window	54
9.3.2 Player setup window	54
10. Testing	55
10.1 Unit testing	55
10.2 Integration testing	57
10.3 System Testing	57
10.4 Acceptance Testing	57
11. Discussion	58
11.1 User friendliness	58
11.2 Different Design Decisions	59
11.3 What would we have done differently	60
12. Conclusion	62
13. Perspectivation	63
13.1 Writing the code in Danish	63
13.2 Access to the game rules	63
13.3 Matador as a Mobile Application	63
14. Literature	64
15. Appendix	65
15.1 Appendix 1 - Rules of the game Matador	65
15.2 Appendix 2 - Use case diagram (early version)	69
15.3 Appendix 3 - Class diagram (early version)	70
15.4 Appendix 4 - Class diagram (early version)	70
15.5 Appendix 5 - Activity diagram (early version)	71
15.6 Appendix 6 - Kanban board (early version)	71
15.7 Appendix 7 - Bugs	72

1. Motivation

Our motivation for choosing this project was on the basis of getting better at understanding and writing java code. We wanted to take on a project or problem in which the rules were already set. This allowed our attention to be focused on translation of the physical into digital instead of designing something new. By that concept we could have chosen any game in which the rules were already made, but we picked Matador because we believe it has a complexity to it, with money and banking aspects, buying and pledging property deeds, the cards of luck, which we all felt would be a good challenge for us. All these different elements to the game also seemed handy when practicing the skill of defining and creating distinguished classes in our object-oriented programming structure.

2. Introduction

We will in this report describe our program and explain our reasoning for our different choices during the development of our program. We will start by defining our problem area followed by our problem formulation. We will then take the reader through our thought- and learning process we had during the project and end our project in a perspectivation where we will present ways we could have done our project differently.

What would we do with more time? What did we learn during development that would have helped us if we knew from the start? How could we have implemented it into our practice/routine.

This project is centered around how the boardgame Matador can be created in a digital version. The Danish game Matador, a game with similarities to Monopoly, is a multiplayer game where the goal is to buy, rent and sell properties to the best of your ability to become the most dominant and richest businessman in the game. The game starts with everyone starting on the first field called "START" and you take turns to move by rolling the two dice and move the number of eyes you roll.

When the player lands on a field that is not already bought by another player, the player can then buy the property. If another player lands on it, that player has to pay rent to the owner of the property. You can buy extensions to all properties which makes the rent go up, with the exception of ferries and breweries, that makes the rent go up with each additional ownership. There are fields called "Try Your Luck" where the player draws a random card, and whatever the card declares has to be done.

A prison is a part of the game where a player can either be stuck up to 3 turns or pay to get out. As described the game is about outsmarting the other players and having a bit of luck on your side. The aim is to outlast all other players and thereby win the game.

3. Problem area

Playing the physical board game of Matador has several obstacles related to it. First of all it takes time and planning to set up the entire game before a player can even start playing. Time in the sense that it takes time to set up the game before a group of players actually gets to play the game. The planning aspect is the fact that it takes up physical space and might affect other activities if a game is conducted over several days. Planning also spans the planning of who is in charge of what when conducting a game, such as banking related features, illegal game actions as per the rules. Lastly there is the social aspect of the game where as a player, you are dependent on finding additional players to play with, and arrange a meet time and place.

These problem areas are what can be improved upon by automating the board game. But by automating it we might just introduce new problems. A huge problem for us in order to automate such a game of complexity would be to be able to keep track and have a continuous overview of our project. Especially of what we need to code, what has already been done and how do we make sure we can collaborate on the code throughout the whole process.

This leads to the difficulty of organizing our project and code in such a way as to minimize time used in constantly having to check up on previous developed code and its functions.

Translating the physical into digital quickly accumulates a lot of data. This has to be organized in such a way that it can communicate between various parts in appropriate ways. From the onset of this project Java is the chosen programming language intended to solve our objectives. Java is an object-oriented programming language and a lot of our focus will go into organizing our code in an object oriented manner. We want to create a good game, where "good" means a well structured program using principles from OOP and good coding practices, which brings us to this project's problem formulation.

3.1 Problem formulation

- How can Object-Oriented Programming (OOP) and Unified Modelling Language (UML) help to develop a good digital version of a board game?

3.2 Research Area

Our problem formulation puts an emphasis on the architecture of a program. We have chosen to code in java and since java is an object oriented language, we have decided to have our focus on object-oriented programming as a structure. We want to research the principles of OOP and understand the benefits it brings to a project to be able to create a good product. Creating a good program for us means using and understanding OOP principles and making good use of it in our development. We also want to be able to communicate our program and process. We have therefore decided to put emphasis on how to develop a GUI and how Unified Modeling Language and Kanban boards can communicate information and assist in the development process.

4. Workflow

We decided to use different tools to enhance our code and work process. The tools we primarily decided on were the Kanban board for our project management, Github for our shared version control, and object-oriented programming as our overall coding approach. We developed several UML diagrams for better clarity and overview. We will explain their methodology in this section.

4.1 Project management

Our intention for managing this project was to meet regularly so we could have coding sessions together. Firstly, because several of us felt a collective coding project was a huge undertaking if having to code on our own right from the beginning. Secondly, in order to get a mutual understanding of what needed to be coded. And thirdly, to get to know and learn from each other regarding coding. Finally, there was the aspect of also interpreting the game and rules of Matador together.

At the beginning of the project, we primarily focused on the planning of the project, and figuring out what was needed in terms of coding, version control, and general workflow.

During the project, our work process was heavily interrupted by the circumstances of the Corona situation at which point we were forced to adapt to a new workflow. The circumstances made it impossible for us to meet in person, but we made it work by setting up meetings through Microsoft Teams application. The situation had both its advantages and disadvantages. The online team meetings made it possible for us to spend more time together in calls compared to what we would have done in person.

Microsoft Teams had a screen sharing function which was a help for solving problems each member had, but not as good as in person discussion about a problem. The disadvantage was that we had to cancel our planned game research where we wanted to play Matador. Our intention was to play the physical version of Matador and thereby extract an activity diagram to use as a guide for our code structure.

We started off with the sessions being very time consuming since four people were essentially doing the work of one. One person was doing all of the coding at a time, while the rest were watching and giving feedback as we went along. This was incredibly inefficient, since one person was doing all the typing, and 3 people giving feedback simultaneously (often with differing opinions on what needed to be done, and how it should be done). As time progressed the project responsibility was divided out through the members so as to fasten the pace. Eventually, we ended meeting 3-5 times during the week, for 3-6 hours depending on what we felt needed to be done that day. We would start out the day by discussing what we had done the day before, and what we would be working on individually on the day. Often, we would meet again with 1-2 hour intervals to keep each other up to date on our assignments. We would stay in the call (or at the very least online in Teams), so if anyone were stuck on a problem, the others could help out.

4.2 Kanban Board

As a tool for project management we decided to use a kanban board provided via the trello website.

The essential part of the Kanban board tool is its method of identifying, defining, managing and optimizing a project in order for it to be completed. The Kanban process is characterized by its "Start where you are"-approach. Any team or project management can draw advantages of applying the principles of the kanban method process in any stage of development. It has become a tool for software developers. It helps by having visual cues and signals, and thereby avoiding project teams spreading their attention into completely different directions. It also helps to catch issues early before they become big and can cause a lot of problems by clearly being able to see the workflow and where it breaks.

The process of the kanban is an involvement of parties, and all parties are to be considered vital to the completion of the project and thus the process is dependent on the value of respect. Respecting the team, the work, and the people. But the kanban values also include eight other values: transparency, balance, collaboration, customer focus, flow, leadership, understanding & agreement.

All these values make up the motivation of the kanban process seeking to improve and optimize services delivered by a team.

A typical kanban project process is considered to be driven by one of three narratives. The first one being called the sustainability agenda, which mainly focuses on finding a good rhythm and work pace for the participating project members, while also focusing on the needed elements for the project.

The second narrative is driven by its focus on customer satisfaction and performance hereof. It is called the service-orientation agenda.

The third is the survivability agenda which is predominantly fixated on the internal structures of the 'organisation' in this case a project, so as to keep being adaptive to customer needs and wants, while also being competitive compared to their competitors.

(Anderson & Carmichael, 2016)

4.2.1 Use of Kanban

Our method for the kanban board was first to identify which column we felt we needed for our project in order to get a visual overview we felt satisfied with. We decided to make a backlog column and four others: breakup, working-on, verifying and completed. So in total we set up 5 columns for our project management. We mainly used our kanban as a representation of our needed work for our project to be completed and we followed it up with additional cards as our code and project went along. The process of reflecting on our code and what was especially needed for it to work was one of our main strengths that carried our project forward. Thus we had no problem with adding work to our backlog and later breaking it up into pieces. On the downside we really did not utilise the visual kanban board to its optimal potential. We lacked discipline for updating our visual board with what was currently being worked on, what needed verification and what cards were to be considered completed.

Our failure in fully using the kanban process to our advantage probably stems from the fact we never had a team conversation of which values each of us felt we needed to focus on, and what narrative agenda was to be our main focus throughout.

Looking back at the project process, we can see the values we had as our primary driven factor was: transparency, collaboration, understanding and agreement.

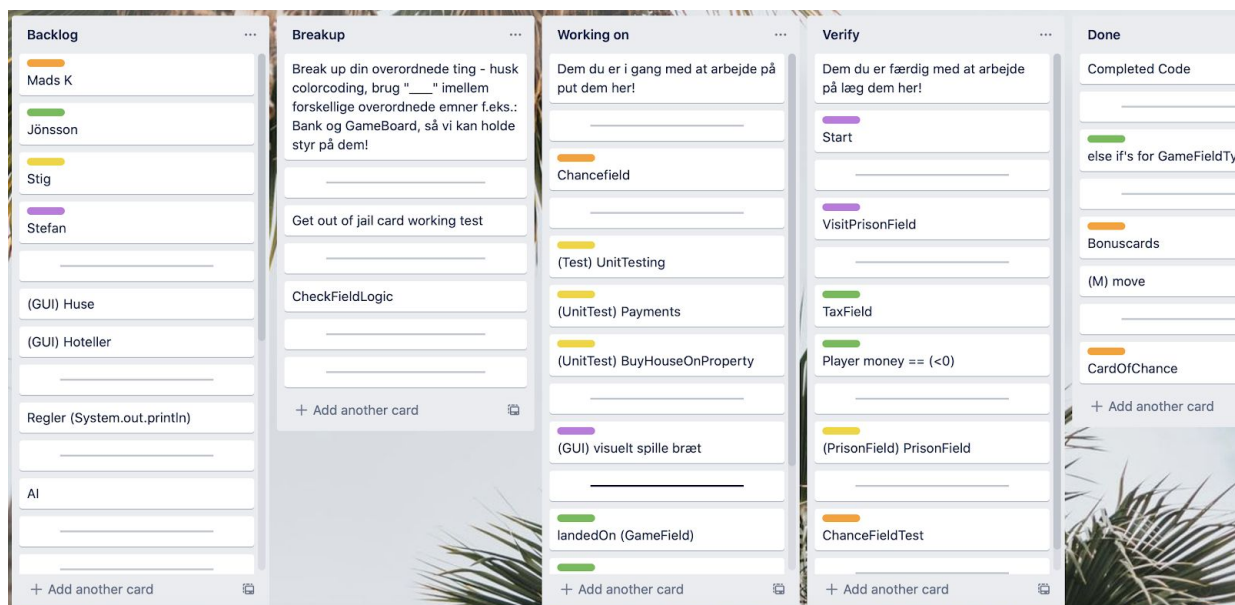
We had a clear understanding throughout the project that we needed to be sharing our thoughts and problems so as to evaluate our progress and solve problems that would arise, especially during coding our system. This open transparency with one another made our collaboration much better throughout the project and our respect for each other and our differences. The value of understanding was our foundation for the collective coding sessions. We wanted to make sure everyone could follow along with the source code that was added and why it was added. So although our pace was not as fast as it could have been, if we had chosen to divide the responsibilities of the different coding cards from the beginning of the project. Our way of collaborating with each other made it so our overall understanding of the system we have made is particularly high. Furthermore the commitment of having these regular meeting sessions, and making sure the process was transparent and understood by everyone also made our agreement of improving our game and its operation that much easier.

The agenda our project has been following has mainly been that of the sustainability agenda. It can be seen throughout the way of how we first decided to have one person coding at a time, while screen sharing so everyone could follow along and come with corrections and reflections to better our source code. Since this was our starting point for our project it meant we had to have a steady pace of meeting sessions otherwise we would stagnate and never have any real progress.

This also meant that the project's progress has been very slow in general but especially during its first phase, since four people were doing the work of one. Eventually we picked up pace by delegating work assignments to each member and optimising our work progress.

Although in hindsight we made great use of the sustainability agenda and that it surely has been our main focus, there have been several other key factors from the service-orientation agenda and survivability agenda. The factor of performance has been a constant topic we have talked about in our team meeting session, since our ambition was to have a game that ran as smoothly as possible. The way we utilised this was by relying on object-oriented programming by simplifying our methods and classes as much as possible. This also helped us with having a transparent and cleaner code as well as a readable code. This was also a major topic in our team sessions. The survivability element of adaptability has certainly been a key point of our project. We started off with having our work meeting session arranged as physical meetups, but since the situation with coronavirus pandemic, we were forced to adapt our workflow so as to still be progressive towards our end goal of completing the game.

Where we would have liked more focus and greater attention towards is the aspect of customer satisfaction from the agenda of service-orientation. The basis of the kanban process is to focus on the parts which gives the greatest immediate benefits to the customer so as to get as much feedback and thereby iterate on the product before handing it over completely. Throughout the project we had our focus on maintaining progress and completing pieces of code so we could move on to the next part and so on and so forth. What we should have done instead is seeing our progress from the perspective of a customer or user, since it would have enabled us to make a simplistic game with steady progress towards a functional interactive game with more and more complexity added to it as progress went along. This would surely have meant an attention aspect towards a progressing GUI and game logic that followed the determining game factors of a user.



4.3 VCS - Version Control System

A Version Control System keeps track of all your different versions and development of the program, and enables you to always be able to revert back to a previous version.

4.3.1 Git

Git is a DVCS - Distributed Version Control System, which means that each user that is in contact with the project also works as a repository because when checking out the project, it gets the full history of the project as well. This means that if your server that stores your project experiences some kind of corruption, then each user also has a full history of the project and the amount of data that has been lost is minimal.

With the use of Git it also means that developers are able to work on the same project without causing any issues for each other. Git keeps all your committed changes to the project on your local computer until you decide to push it up into the cloud where other collaborators on the project then can pull your changes to the project. Storing committed changes locally also means it allows everyone to work on the project while not being connected to the main server or internet, and then when you are connected to the internet you can push all your changes and at the same time and get everyone else's new changes. This enables a unique workflow because you are able to work on the project basically whenever you want and have the time, since there is no need to connect to a server to have access to the data.

Problems of course do occur when using VCS but are easily resolved. Problems may occur when two or more people are working in the same file and are both planning on pushing. Then whoever pushes second will have to go through the code to make sure that the new code that is not in the local version won't get lost during this push.

Git also has a function that allows the user to not sync selected files or file types. This functionality can be used in situations when a developer team doesn't want certain information about the project to get out to the public. This could be user data or just sensitive data in general that should be kept hidden and adding those files to `.gitignore` means that those files would not get affected by git's version control. (git --local-branching-on-the-cheap, ¶ Getting Started - About Version Control)

4.3.2 Github

Using github as the tool for sharing your completed coding, makes it easy for a team of developers to work on the same project although with occasional hurdles. Github gives a developer team the option to work on a project from anywhere. Keeping the Github repository clean, meaning only having running and working code in the main branch. It is very important for a developer team to be able to keep an overview of the project. This becomes progressively more important the bigger the project becomes.

The access point to the code for each developer is through the use of Github. Using Git a team is able to push their code onto the Github repository, and it is also through Github that the team will be able to pull the newest data from the project. IntelliJ has a built in functionality for VCS and developers are able to select that they are using Git as their VCS and select the specific Github repository, meaning developers are only a few clicks away from updating their local project as well as the main version on the Github repository.

4.3.3 Problems we encountered with VCS

VCS makes us able to keep track of all our data during a project. We decided to use Git as our VCS as we wanted the security and advantages that came with using Git.

One of the problems we ran into was when we were setting up our Git and Github we had by mistake also pushed .iml files onto our repository. These files hold onto local settings in IntelliJ and local variables like where you are on a file with the cursor. This meant that pushing and pulling became very troublesome because .iml files would change constantly and not wanting to be committed or pulled. So we had to delete our local project and reclone it from Github many times. We knew those files were the issue, but we couldn't get gitignore to ignore the files, because the files were already on the github. We experimented with a lot of things to get it to work like deleting the local files before pushing or all copying each other's iml files to sync them somehow. The solution that ended up resolving the issue was deleting the files from github. They were causing issues because everytime we pulled the project from github we also pulled those files, and what we should only pull was the source code and each of us kept our local settings of IntelliJ. After deleting the files we had no issues with VCS other than the normal merge conflicts that could sometimes take a long time to make sure we merged it correctly.

During our project we wanted to create a test branch for our project because we felt we had gone in a wrong direction, and we wanted to redo a lot of our code. We wanted to move away from only working in one class "GameField" and within that class having a variable type for each of the instances of GameField. We wanted to start using inheritance where we used GameField as an abstract class and created subclasses like PropertyField, PrisonField ect. This way we could encapsulate each different field so all fields did not have a lot of useless code that was relevant to the specific field and it would also make it alot easier to read and understand. We also with this change wanted to get rid of HousePlots, which was the property on a field. We had initially created fields which could have a HousePlot, prison or a ferry, but we were having issues with this way of making the game, and decided to make dedicated fields to each type of field to more easily create specific methods for each type of field. During this process we ended up just working in this new branch for much longer than we should have and basically just used it as our new master branch. This was obviously wrong and we should have merged the new branch into our master branch much sooner, but we didn't know how to do it correctly, so we therefore just kept pushing it further into the future. It ended up being much easier than we expected, but because we had waited a lot longer than we should have we had a lot of merge conflicts, which resulted in a lot of time used to merge it correctly together.

jonsson0 / Matador Watch 3 Star 0 Fork 0

[Code](#) [Issues 0](#) [Pull requests 0](#) [Actions](#) [Projects 0](#) [Wiki](#) [Security 0](#) [Insights](#)

No description, website, or topics provided.

[270 commits](#) [6 branches](#) [0 packages](#) [0 releases](#) [4 contributors](#)

Branch: master [New pull request](#) [Create new file](#) [Upload files](#) [Find file](#) [Clone or download](#)

Stigafp Merge remote-tracking branch 'origin/master' Latest commit 92e513a 28 minutes ago

Test	Færdiggjort ChanceField, mangler test	2 days ago
src	Merge remote-tracking branch 'origin/master'	28 minutes ago
.gitignore	Tilføjede følgende filer til git igen	2 months ago
Matador.iml	resize window and press the "Test" button	8 days ago

Help people interested in this repository understand your project by adding a README. [Add a README](#)

<https://github.com/jonsson0/Matador.git>

5. UML

We will in this segment describe our illustrated diagrams of respectively the Use-Case, Class and Activity diagram. The description and diagrams have to aim to feature the program as it is. To avoid spending more than necessary time continuously having to try and explain a piece of software and the workings of it, Unified Modelling Languages (henceforth UML) is used. UML is a tool for visualizing programming code using diagrams, and thus is a clarifying tool for the involved parties with different backgrounds. The diagrams make it easier to understand the coding architecture and the overall plan for the structure of the system, by creating the system in UML as object-oriented models.

There are two distinct proposed model theories for how to produce a good model. Respectively proposed by Herbert Stachowiak and Bran Selic.

Herbert Stachowiaks model is characterized by 3 simple keywords. 1. *Mapping* a model should always be an image of representation of something. 2. *Reduction* encapsulates the understanding that the model should not be flooded with information, but only the necessary attributes that are relevant to the modeler or user. 3. The model should be *pragmatic* in the means of usefulness, such that it is preferable to use the model compared to the original system.

The second proposed theory of good modelling by Bran Selic is distinguished by 5 terms: abstraction, understandability, accuracy, predictiveness & cost-effectiveness.

The concept of *Abstraction* and *Understandability* lines up with Stachowiak's term of *Reduction*. A model should reduce the amount of attributes to a model that does not pass on relevant information, and at the same time the remaining elements should be as intuitive as possible e.g. by giving attributes and elements clear cut naming.

The term *Accuracy* contains the understanding that the model must pay attention to the original system, and reflect it as close to reality as possible. The *Predictiveness* of the model summarizes that the model must be enabling predictions of the modelled system. The final term that Bran Selic advocates is a model's *Cost-effectiveness*. Cost-effectiveness explains that it must be cheaper to develop a model than create the system being modeled.

To get an overview of our project we started off with developing UML diagrams. Both to clarify what our project entails but also so the development group understood the fundamental idea underlying the programming of Matador. As the project went along we revised our diagrams such that they encapsulated the current state of our devised game system.

The overall understanding of diagram models is that there are two categories of said diagrams: Structural diagram models and Behavioral diagram models. The Structure diagrams visualize the architecture of the software system, while the Behavior diagram explains the consequences of actions when running the system.

Since there are several different diagrams to choose from, we have chosen to use the Class-diagram as our structural model, while using Use-Case and an Activity diagram as our behavior model. (Seidl, Scholz, Huemer & Kappel, 2012)

5.1 Use-Case Diagram

The Use-Case diagram focuses on the user's interaction with the System, called an Actor, without detailing it with too much information, but mainly about specifying the necessities of the interaction for a clear understanding of the process. It outlines what the Actor can get out of interaction with said system, and therefore what would be the motivation for the Actor to use the system. The strength of this diagram is it showcases which Actors use which functionalities of the system and thus thereby gives an indication of system requirements, since it basically represents customer needs. A use-case can also be a model of an existing system, such that it gives a visualization of the system. A good use-case diagram will typically answer the three questions of:

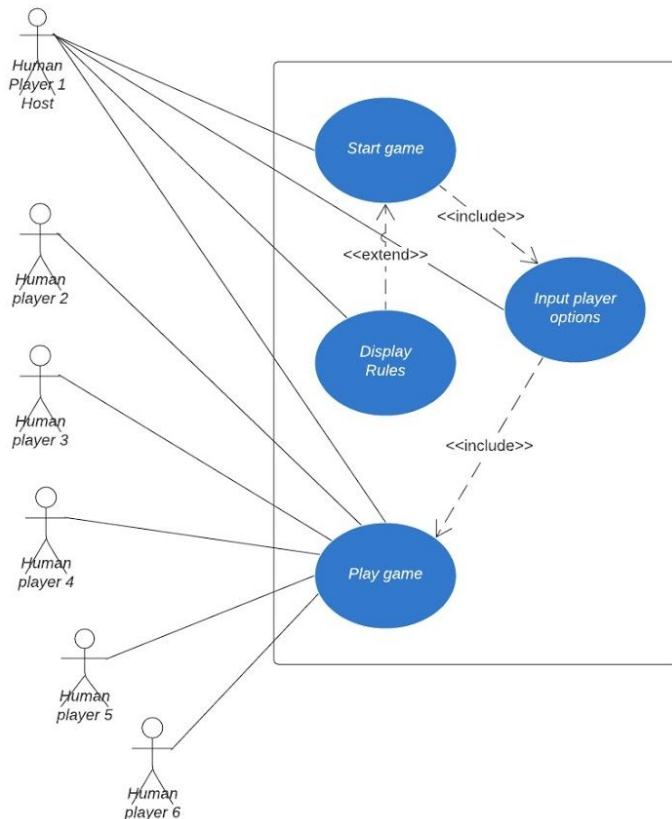
1. What is the use-case describing? (The system.)
2. Who interacts with the system? (The actors.)
3. What can the actors do while in the system? (The use cases.)

The use-case is visualized by having a rectangle that illustrates the entirety of the system with individual circles inside the rectangle that represents the use case functionalities the actor(s) can or have to interact with. The actors are illustrated by stick figures outside the edges of said rectangle with lines going from the actors to the necessary functionalities of the system that the actors are associated with.

(Lucidchart, ¶ UML Use Case Diagram Tutorial)

(Seidl, Scholz, Huemer & Kappel (2012))

5.1.1 Our Use-Case Diagram



We chose the Use-Case diagram in order to visualize the direct flow from the Actor to our system, so as to better understand the functionalities needed. It helped us in visualizing how our GUI might be set up for best use.

In drawing our diagram we drew the actor Human Player 1 Host, as our main actor. The reason being, that the way we have built our program is so only one person at a time can interact with the system, since there are difficulties in being several actors on one keyboard and mouse. Thus human player 1 has been named host. The purpose of categorizing our actors as human is to not confuse between human and non-human use-cases, and because our game has intended physical input from a human.

Of course the main purpose of wanting to use our system, and the primary use-case functionality is that a potential actor wants to play the game, and therefore we drew a bubble "Play game" with that in mind. All the actors connect to this bubble as to illustrate that this is the main purpose and this is where the main fulfillment of wanting to play Matador takes place. But as mentioned before, the one to set up the game is human player 1 host. This actor needs to go several steps before playing the game actually takes place. This is shown using included operations.

To begin the entire sequence of playing the game the host first has to click through an option called "Start game" which both has an incoming extension and an outgoing inclusion.

The incoming extension is shown as "Displaying rules", for which the actor can choose whether to read up on the rules of the game or not. The outgoing inclusion "Input player options" leads on to the most important part as the host can input how many players the actor wants to be playing. At total a max of six players can be playing the game, also shown as the six actors in the diagram. When "Start game" and "Input player options" are fulfilled the actor(s) can go on and "Play game".

To contrast our Use-Case diagram to the proposed theories of Herbert Stachowiak and Bran Selic, we used the Stachowiak theory in terms of presenting the game system to a potential player of Matador. In order to reduce our diagram we decided against specifying further details going on in the game, when the step of "Play game" has been reached. We also left out the potential factor of a development team testing and updating the game system. Considering we have upheld the first two conditions it can be argued that the third condition of the diagram being pragmatic is thereby upheld. Since a model being pragmatic is dependent on being a visual representation of a certain something with it only having the most useful information showing for greater interpretation of the system.

From Bran Selic theory we will go on from the step of Accuracy since Abstraction and Understability leans heavily on Stachowiaks three steps which we just went through. Without Accuracy our diagram would not be pragmatic since it would not follow reality, but we used our system as the basis and as such we acutely followed our system as it is for our model, thus Accuracy has been reached. With Predictiveness comes the ability to follow the Use-Case and be able to see through what the different steps encompass without it being detailed. We accomplished this by having "Start game" and "Play game", thus the user of the diagram knows the model is of a game, and in a typical game there must eventually emerge a winner, and thus there have been set up different winning conditions within the system. Although we have not spent any money on developing our system nor diagram then in terms of Cost-Effectiveness it has been cheap either way and definitely in terms of time consumption it has been much faster to develop our Use-Case diagram than model and coding our entire game system.

5.2 Class Diagram

A Class diagram is a structural visualization of classes associated with the coding project. It describes the attributes and operations of the class and the relationship between the classes.

The classes themselves are drawn using rectangles with lines splitting it and thereby creating three rectangles inside a whole. The header rectangle contains the name of the class, with the middle section containing the attribute elements, and the final bottom section having the operations of the class. Operations being the methods.

The attributes refers to the variables, which is described by its name in the code followed by its type e.g. String or integer. In front of each of the attribute and operation elements there will typically be either a minus sign (-) for private variable, or a plus sign (+) for a public assignment, then there also is protected (#) and package (~) assignments. These assignments are also named as visibility markers and indicates which functionalities of which classes have access to it. E.g. an attribute with a private labeled variable will only be visible to its operations and subclasses, whereas a public attribute will be available to all classes.

The bottom rectangle of operations is defined with a visibility marker followed by the name of the operation, which should be a clear name such as it indicates what the operation does. The operation name is then followed by the parentheses as an indicator for methods.

The relations between the classes is done by adding lines of association between them and can be ended with specific arrow symbols to show what sort of relations there is. The relation arrow symbols there is to be distinguished by, is: inheritance, aggregation, composition and association.

- *Inheritance* has the arrowhead of an **arrow**, which is used from subclass to its superclass.
- *Aggregation* has the arrowhead of a **black diamond**, and indicates the class as being part of a set of classes but can also exist on its own, if said sets of classes did not exist.
- *Composition* has the arrowhead of a **blank diamond**, which describes a certain class and its dependency on the existence of another, otherwise it would not work.
- *Association* is illustrated by having just a **solid line** without any particular endings between classes. This connection is used for describing links between instances of classes that communicate with each other.

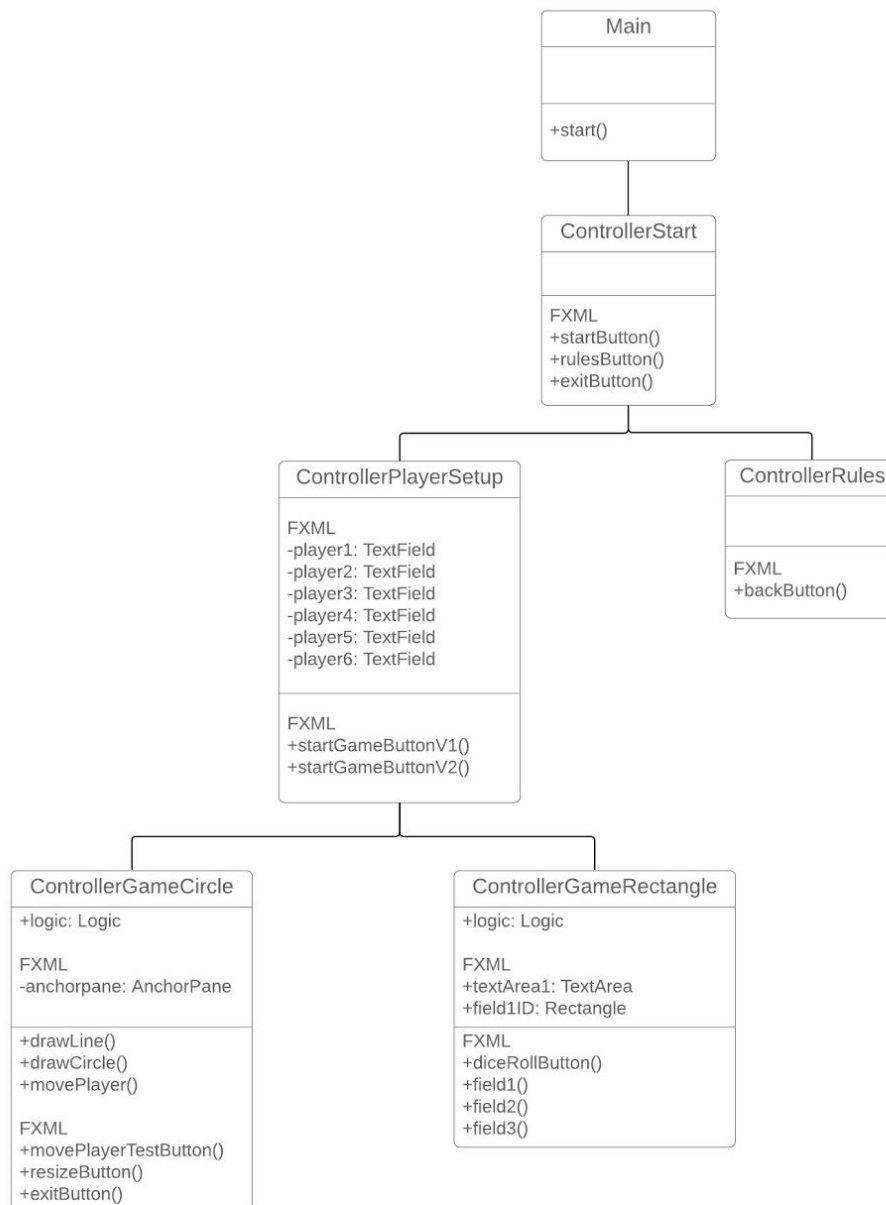
Finally there is the aspect of adding multiplicity to the relations of the classes. Multiplicity accounts for the number of objects that are associated with just one object on the other end of the relation. The multiplicity elements are either written as 'zero to one' (e.g: 0..1), a specific number (e.g: n), zero to many (e.g: 0..*), one to many (e.g: 1..*) or just a specific range of numbers (e.g: m..n).

(Lucidchart, ¶ UML Class Diagram Tutorial)

(Seidl, Scholz, Huemer & Kappel (2012))

5.2.1 Our Class Diagram

The advantages of a Class diagram is that it helps with thinking-out the structure of the programs internal architecture and how the classes might interact. It gives a good fundamental overview of the classes without having to dig into a big coding project without prior knowledge. Our first step in getting an overview of necessary game components for Matador was to write and draw on a blackboard the different classes and their operations (see [Appendix 3, 15.3](#)). The class diagram on the following pages is divided into two parts and is our last iteration of said diagram. First part visualises the GUI controller set up and the second illustrates the game logic.



The controller setup image above showcases the different controllers needed for our GUI screens, thus encompassing the first keyword Mapping from Stachowiaks theory. Each controller is a separate user screen, which has interactive elements and objects to pull out data from our system also called event driven programming. In total our GUI program has 5 screens represented by the 5 controller classes. The first class ControllerStart is the start screen menu, thus its name. it has a linked association to the controller classes ControllerPlayerSetup and ControllerRules. The association here functions as a link between them but they are not dependent on each other, as such they can operate independently and present different information to the user.

The same is also true for the rest of the controller classes. If a user is viewing the start screen, which is linked to the ControllerStart, then the user will either be able to click on an object leading to either a new screen, with the rules of game being shown, or an object that links to the game setup of how many players will be participating in a Matador match.

The controller classes of ControllerGameRectangle and ControllerGameCircle are two different iterations of game GUIs layout, but we chose to have them both showing as demonstration for our development process.

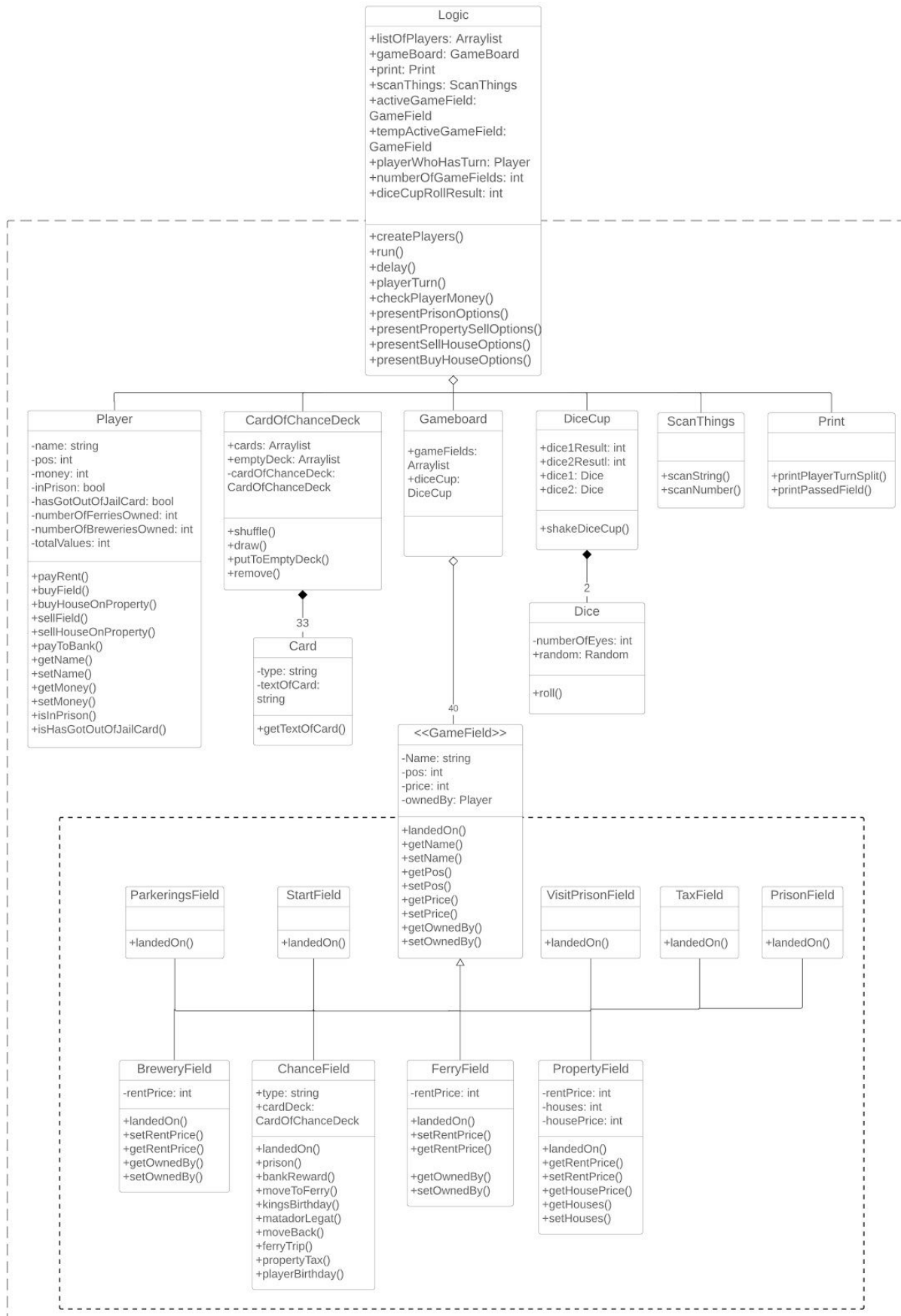
Just like our use-case diagram we aimed at making our controller class diagram as pragmatic as possible, meaning the operations and methods showing are of highest value to the interpreter. Thus we reduced unnecessary data from the diagram. In order to make an image of our system we also developed a class diagram of the game logic (see next page).

Both of these class diagrams were made by listing all the classes and their operations to begin with, from there onwards we began to reduce information in them that we did not deem necessary. This approach allowed us to go over all the information each class contained and could thereby better see the overall dynamic of each and how they interlinked. E.g. the logic class is where all the game data returns to, and thus is the gathering class for all game data. Therefore we have encompassed all the classes that interlock with the logic class by a striped rectangle, and connect to the logic class with an aggregation arrow. An aggregation arrow indicates that the class can function on it's own but is part of a group as a whole. It can be further seen in the GameField class where we have used a striped rectangle around the objects, but in this circumstance it is because of inheritance. All the classes being extended from the GameField are inheriting a set of values so as to signify a game field on the Matador game board. The GameField class is then connected to the game board by a composition line, which means without the game board there would be no game fields, also showcased by the fact that the game board is also the class that creates the arrayList of game fields. Lastly the line has a multiplicity element showing a total of 40 game fields that are making up the setup of the game board.

Between the classes Card and CardOfChanceDeck can an aggregation line be seen as well as with Dice and DiceCup. With Card and CardOfChanceDeck it can be seen that a multiplicity of 33 is returned, respectively. This means a total of 33 cards are making up the chancedeck.

Furthermore by listing up all the information of each class as the first step to make our class diagram, and then eliminate data that is not necessary. In doing this, we discovered several instances of old lines of code that were no longer relevant for our game system, and thus making a class diagram also functioned as a tool to clean up our code for unnecessary methods and lines.

In accordance with the mapping theory of Bran Selic we developed the class diagram so as to comply with the guidelines. We made sure that the class diagram first was in accordance with the Stachowiak theory developing diagram: mapping, reducing and being pragmatic. From there we reviewed our diagram to evaluate it with the theory from Selic. Accuracy was achieved by modelling our finished system. Predictiveness was accomplished by naming the classes and functions in accordance with what they do. And Cost-Effectiveness was reached by spending a limited time on this class model compared to the actual coding of the classes.



5.3 Activity Diagram

This diagram describes the flow of the procedural aspect of a system, it originates from the language of defining business processes, and thus it entails the actions of control flow and data flow in order for the system to execute a particular activity.

To illustrate an activity diagram it is common to use rectangles with rounded edges to symbolize the process stages. It is thereby followed by arrow nodes to draw attention to its further pathways, such that it makes for a “roadmap”. In order to be clear about the “begin *reading* from here” it is important to show where in the system a reader of the diagram should begin, and thus the procedural system has an added starting point symbol of a typically black-colored filled circle, called an initial node. Likewise there is a symbol for when the system is ending, which is symbolised with a similar black filled circle but with another line drawn circle around it.

A process stage can be divided out into multiple directions or be the gathering point of several earlier operations, and may thus need a merger to avoid having to put nodes everywhere. This merger is illustrated with a black line wall and is either called a synchronization node when merging incoming nodes from earlier processes, or called a parallelization node when having to diverge into several different paths.

If the system requires a decision input, or reaches a if one or the other split, it will be illustrated by having a diamond shaped rectangle from where the nodes will diverge in two to illustrate the pathway for e.g. true or false.

The activity flow of a game is typically required to be a process loop, called a game loop. This loop will run in a circle until either a winning condition is reached or a losing condition is met.

(Lucidchart, ¶ How to Make a Flowchart in 60 (ish) Seconds!)

(Seidl, Scholz, Huemer & Kappel (2012))

5.3.1 Our Activity Diagram

To illustrate our game systems internal behavioral architecture we have developed an Activity diagram. It showcases the actors' and data through the system, and the decisions needed to be made.

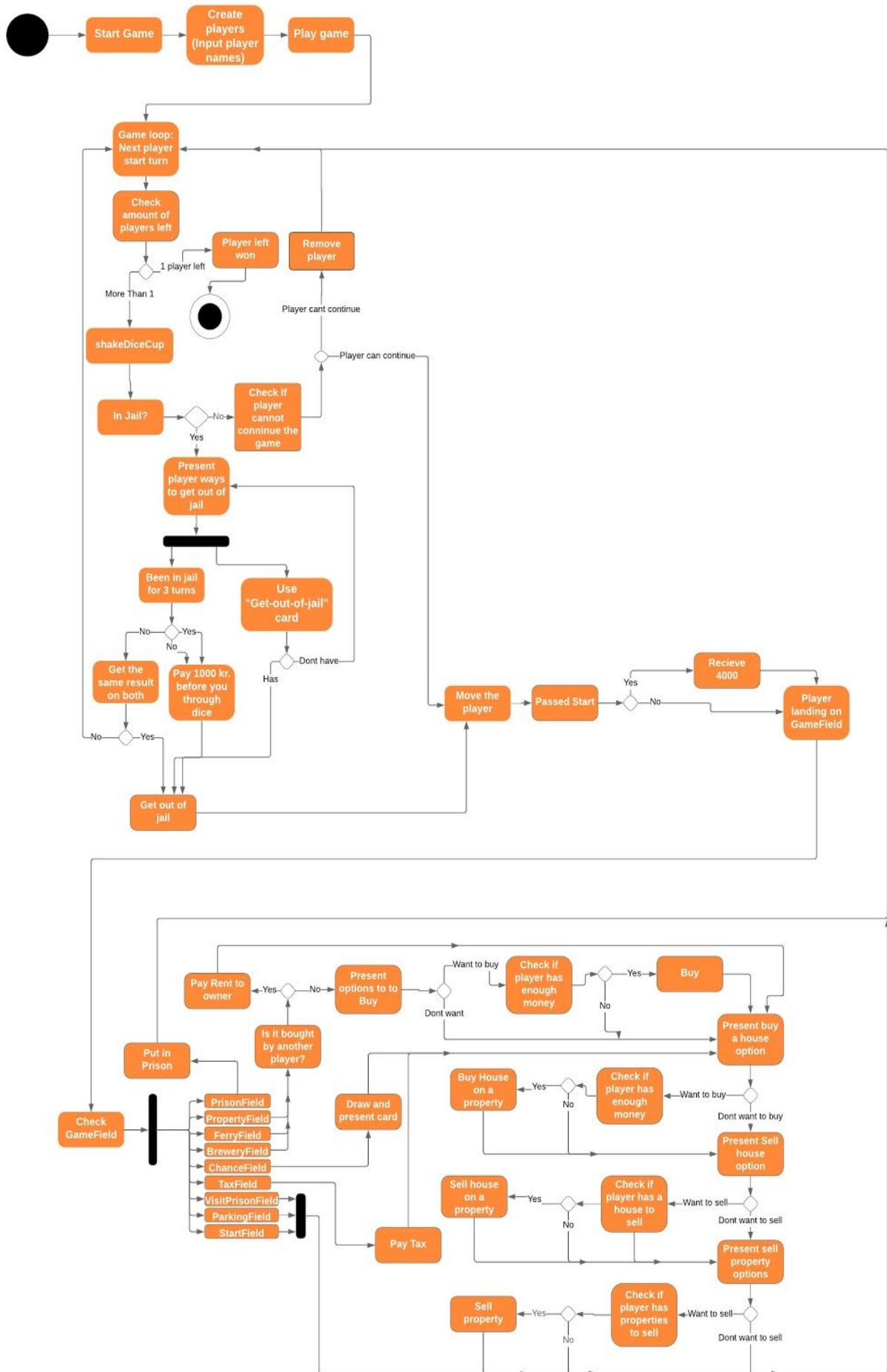
The project's activity diagram shows the systems behavioural structure and also shows the algorithm at play when a player is active on the game board making a choice towards winning the game.

By making this diagram it made us more aware of the playerTurn algorithm and how it was implemented. Although we first truly set our attention onto the activity diagram after we had developed our system, we realized during making the diagram, that if we had made it just as early as our initial class diagram, we would have had a huge advantage in knowing what methods and operations should be where in the process of the player turn and would thereby know how to set it up.

The first thing we set up in our turn algorithm was to initiate the dice, although not yet showing the result to the player. From there the algorithm will check whether a player was located in jail, if so, the player would have to get out of jail before any other actions on the board can be taken, as being in jail is a punishment. To get out of jail there are several options that are represented by the block called "Present player ways to get out of jail", these options are: pay to get out, roll the same eyes of dice, or use a get out of jail card that a player can get from the Chance deck. Although if a player has been in jail for 3 turns that player will be forced to pay to get out.

If the player was not located in jail, the algorithm would go on to check if the player can continue playing by checking whether the player is bankrupt. If the player is *not* in jail and *can* continue playing, the algorithm will update the players position and check whether player passes start, because if so the player gets additional 4000 kroner to their account.

From there the field that the player has landed on gets checked for its type, as there are nine different types. If the field type is a property, ferry or brewery the system will go on and check if the game field is bought by an opponent and if it happens to be, the player has to pay rent to the opponent for landing on it. Otherwise if it is not bought the player has the option of buying the game field. Afterwards the player gets presented with the options of buying houses on their properties and then to sell houses on properties. Respectively the player gets checked if he has properties to buy on and houses to sell. But from there the loop goes onto presenting the player for options of selling owned properties before passing the turn onto the next player.



If the player does not happen to land on the previously mentioned fields but instead lands on a prisonfield, the player will go to jail, or if the player lands on chancefield a card will be drawn and presented to the player with instructions.

As with the previous two diagrams we made the activity diagram with the theories of Herbert Stachowiak and Bran Selic in mind. Although the diagram does have a bit more details put into it than it otherwise needed too, we felt the details added a better understanding of the player turn and how the complexity of the algorithm works.

5.4 Acquired knowledge

The diagram that taught us the most was the activity diagram. It gave us insight into the order of the loop algorithm. Our method of finding the right order for pieces of code in the algorithm were done by playtesting how the game felt to play. But if we had used an activity diagram from the start we could have avoided this and thereby allocated some of that spent time.

The diagram that we used the most throughout the project was the class diagram. The class diagram was particularly useful for setting up the project at the beginning.

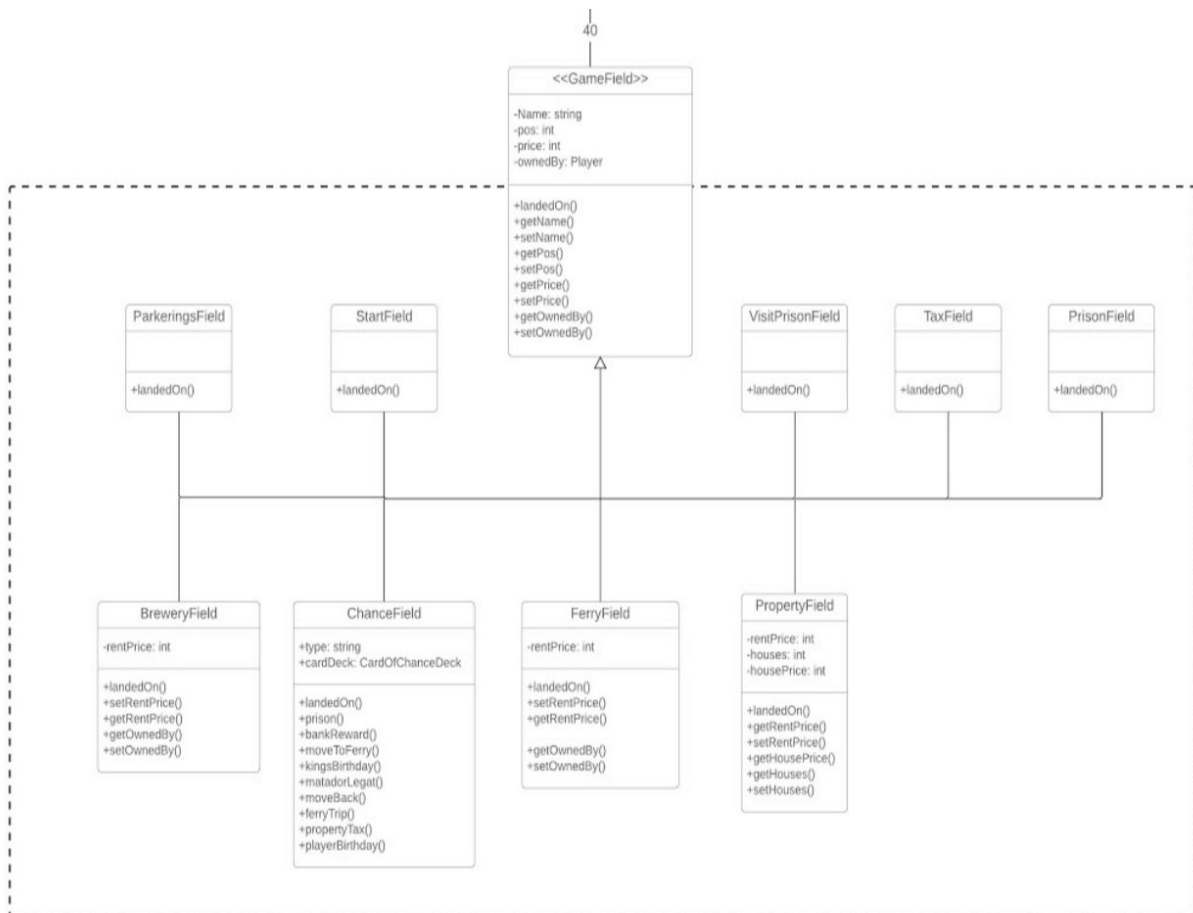
We could have enriched our understanding with a sequence diagram as it would have forced us to consider the aspects of the data structure and design patterns more in depth and it would contribute with a visual representation of such.

6. Object-Oriented Programming

For this project we have decided to use the principles of object-oriented programming. In OOP, code is written in different classes, which are used to encapsulate and delegate responsibility. These classes are used as a blueprint for objects, which are instances of these classes. By dividing functionality into separate, clearly defined classes, it is possible to write clear, easily readable code.

It is possible to create one or more objects of a class depending on what's needed for the program. These objects are then used and handled individually, meaning it is possible to change one object of the same class without changing another. When constructing a new object of a class, it is also possible to set the values the object needs.

6.1 Inheritance



Inheritance is used to derive classes from a “super class”. This super class is used to define some generic methods, which can then be implemented fully or changed in the subclasses. The main advantage of using inheritance is that many classes which are similar in structure, but function differently do not have to be written from the ground up, but will inherit all the base elements, and can overwrite anything if needed. In our program we have created a super class GameField. This takes on the role of all the game fields it is possible to land on. All subclasses of GameField inherit all the variables such as Name, Position etc., and GameField is kept abstract. Then, the subclasses can use these in their constructors and methods. The generic landedOn() method is then implemented with the methods, positions and names of that subclass. By using inheritance for this, we ensure that the landedOn() method from the specific field is executed when a player lands on a field. This means if it is a PropertyField the landedOn() in PropertyField will execute the necessary methods for that field when landed on. The landedOn() is overwriting the landedOn() in GameField.

In the PropertyField subclass, there are several different outcomes when the landedOn() is called. First, the program checks whether the propertyField is already owned. If it is, the player must pay to the owner, and if it is not, they are presented with the option to buy it.

In the ChanceField class, a card is drawn when a player lands on it, and the code for the specific card is executed. By using inheritance, this is done using a single generic call, which then executes the methods depending on the subclass. (Java T Point, ¶ Inheritance).

6.1.1 Our swap to inheritance

We have designed our game board to have game fields, and we use inheritance to have different kinds of game fields, but this was not the solution we started with. We started just having one game field class, and each game field then had a different type variable, so a gamefield could be a propertyfield-type or a ferryfield-type. This meant that when we wanted to do something with each game field, we first had to know which game field type it was. To do this we needed to create a long list of if-else statements which ended up being very cluttered and confusing to understand. We therefore wanted to redesign the way we did our different types, and we decided to try out inheritance, so we had a class for each type. Our game field types were what we were planning on using to keep track of what field the player was on to present the correct options, but as the development of the program continued we made a method called landedOn() in our GameField class which would be overwritten in each of our sub-classes of GameField. This meant that we did not need the game field types for each field, since they were already different kinds of elements with different kinds of methods. So if a player landed on a chancefield the landedOn() method from chancefield would run, making the check of what game field type it was obsolete and only caused more confusion than anything else. We did not realise this until we had already used the now obsolete game field types for checking which of a player's owned fields were property fields. This way we did get the property fields, but it would have been simpler to get the property fields to check a boolean or a specific variable that would be able to identify the propertyfield from other fields.

6.2 Our use of OOP

This section goes into our use of objects, describes our usages of ArrayList, gives arguments for using OOP and rounds off with giving examples of how OOP have influenced our design.

6.2.1 Use of objects

We use objects all throughout the program. A class can be thought of as a blueprint for objects. For instance, the Player class describes what the abstract idea of a player is. Every player has a name, an account balance and so forth. When we start the program, the user is asked to input the names of the players. This takes the abstract idea of what a player is, and then assigns real information to the variables and makes an actual object out of it. This player has a given name, and their account balance etc. can be updated as the game progresses. This player object holds all its own information.

The CardOfChanceDeck is also an example of object-oriented programming. It consists of an ArrayList of numerous Cards. The Card class defines what a card must have, namely a String called "type" and another String called "textOfCard". In the class, these only exist to show what is needed in order to create a new card. In the CardOfChanceDeck, there is a constructor which fills the ArrayList with Card objects. These cards are given a type and a text, and are instantiated as objects.

6.2.2 Data Structures

Our most widely used data structure is ArrayLists. An ArrayList is a class that implements the list interface, which unlike a regular array, doesn't have a set number of entries. This is very useful for the player list for example, as the ArrayList allows us to add players as needed, instead of having to hardcode every number of players we could potentially see. This also makes it very simple to remove a player when they go bankrupt. This could prove more tedious with a standard array, as it isn't possible to remove elements. One way to mitigate this could be running a for-loop and adding the remaining players into a new array, which does not include the bankrupt player. This is needlessly complicated, since ArrayLists has a built in function to remove elements.

Another advantage of the ArrayList is scalability. If the need arose to expand the program in some shape or form (More players, more fields, more Cards etc.), these new elements can simply be added with the ".add()" method. If we had used arrays, we would need to manually expand every array that needed to be bigger, and then add the new elements.

(tutorialspoint, ¶ Java - Inheritance).

6.2.3 Why use OOP?

An advantage with OOP is the ability to split code up in different classes. This helps keep methods and information separate to the objects themselves. For instance, there is no need for the Logic class, which is already a relatively large and complicated class, to know how a player should pay rent. Instead, if the player object itself can do that, we both get a more clear design, and we avoid bloating the Logic class too much. This also means that a given functionality only needs to be coded once, and can then be called whenever/wherever needed. Object-oriented programming is a great tool for keeping track of variables. Instead of having global values for instance, every object holds all it's own variables. By then creating getter and setter methods, which makes it possible for other parts of the program to gain access to this information. So instead of assigning names to the players in global values, they are set using the setter method. When needing to access the name again you use the getter method.

6.2.4 OOP's influence on the design

Since Java is an Object oriented language, it has had a large influence on the entire design of the program. The functionality is spread over many classes, all of which are designed to represent a single part of the physical game board. The GameBoard class consists of an ArrayList of game field objects, all with their own variables and methods. These all inherit from the GameField class, but have different functionalities and variables. For instance, the ChanceField doesn't have a price, since it can't be bought, which most other field types can. By using inheritance for this, we essentially reused the parts of the GameField superclass we needed for each subclass, while simultaneously overwriting everything that needed to be class specific, and ignored everything that wasn't required for each individual subclass.

7. MVC Structure & System Requirements

The following section describes the MVC architecture and game requirements for what we would consider a successful development implementation.

7.1 MVC - Model-View-Controller

MVC is a concept for data structure and is a design pattern. MVC is the short term for Model View Controller. Model meaning the code executing the functionalities of the system. View being the part the user is presented for and the Controller is the part that interprets the users requests and pulls the desired data forward.

The benefits by using the MVC concept is that the system's architecture is distinguished between its separate sections of code objects. This makes it easier for the developers to navigate and make changes where they need to, and also makes it easier to identify bugs. More important is the fact that it encapsulates functionalities and the responsibility is divided between components to reduce errors.

Our starting point when formulating this game project was to make a complete separation between the different layers so as to make it easier for ourselves to navigate in our code, but also to practice the principles of object-oriented programming in its fullest. Thus we started the project by only focusing on the aspect of the game logic, since we were convinced that adding a viewer and a controller would then be simple. As we progressed and had to begin working on the viewer and the controller part of the data structure, we were dismayed that it was not as straightforward as we had hoped. We had started off with the logic part as a terminal game. This meant that we were working towards something that would require a lot of restructuring to make use of our methods in a GUI.

We made our application by using the Scene builder to generate our fxml files representing the view part of the GUI. The controllers contain the variables generated in the Scene builder, the buttons executing our code and most of the code being executed. In this sense we have placed some of our code belonging to the model section in the controllers themselves. This would have been better placed in our Logic class since this would be a cleaner setup and more in line with the MVC design.

7.2 System Specifications

What makes the game Matador feel like Matador? What we wanted was to transform the physical aspect of the game to a digital one and maintain the same gamification and playfulness the original board game has. To do this we knew we had to have certain characteristics implemented such as being able to trade, buy properties and buy houses. The table below explains our thoughts on the game requirements for a digital version although some with a higher priority queue than others. The requirements are made both with our thoughts on what makes Matador what it is and going through the rule book of Matador.

System specifications (need to have) (done):	System deficiencies (to be implemented):
<ul style="list-style-type: none"> - buyable deeds - buyable houses - pay rent - active gameboard - chance field 	<ul style="list-style-type: none"> - property pledging (nice to have) - hotels (nice to have) - house rent payment increase (nice to have) - player trade interactivity (need to have) - chat function (nice to have) - ai + difficulty level (nice to have) - game setup of fast pace game (nice to have) - game GUI (need to have)

Throughout this project we made sure all positions in the Gameboard had an active role outputting different information to the player, depending on the type of field landed on.

The player's ability to buy property deeds and houses is a fundamental mechanic of the game and thus we saw it as the highest priority after having made a game board. To make sure that the gamification of the game survived the transition to being digital we knew we had to likewise prioritize the players being able to pay rent to each other, such as a victor could be found.

7.2.1 User Friendliness

User friendliness was a high priority for us from the start, since we wanted to play an enjoyable game without the user getting the feeling of being stuck on a screen or not knowing where to press to do certain things. During our development process we were aware that we needed to later on connect our logic to a GUI. This meant that we did not put in much thought into which order the user was asked during our primitive stages of our game, since we would make bottoms and alert windows instead of text in the terminal. We ended up not completing our GUI which meant that the way we had built our text based program now mattered alot. We chose to prioritise getting the game completed over making the game more user friendly. The things we did to make the game more user friendly are things like clearly marking when it is a player's turn and giving the player information about the player's money after each purchase, which makes the player aware of the amount of money the player has left.

```
-----Mads's tur -----
Deres 1. terning slog: 1
Deres 2. terning slog: 5
Ialt giver det: 6
De har passeret Rødovrevej
De har passeret Prøv Lykken
De har passeret Hvidovrevej
De har passeret Betal indkomst skat
De har passeret SFL-Færgene
De landede på Roskildevej
Dette er en husgrund.
Denne grund er ikke købt. Den koster 2000 kr.
Ønsker de at købe grunden? - Tast 1 eller 2 og tryk ENTER!
1. Ja
2. Nej
j
Deres valuta er 30000
Denne grund er nu ejet af: 'Mads'
Deres samlede værdier af valuta og grunde er 30000
Deres valuta er 28000
```

Picture taking from the terminal by running the game

8. Description of the Program

This section will focus on describing the classes and the GUI operations in detail. We will also go into more detail about our program and its classes. We want to make a detailed description of each of our classes and how it fits into the overall design of our program. We want this description of our program to make it easier to understand our code and program by explaining important details, core methods in each class and how it interacts with other classes.

8.1 Class overview

The code for our game has 25 classes and 2 test classes which are the following:
See also section [5.2.1](#) for our visualization of our class diagram

Classes:

- Main
- Logic
- game board
- Player
- GameField
 - BreweryField
 - ChanceField
 - FerryField
 - ParkeringsField
 - PrisonField
 - PropertyField
 - StartField
 - TaxField
 - VisitPrisonField
- CardOfChanceDeck
- Card
- DiceCup
- Dice
- Print
- ScanThings
- ControllerGameCircle
- ControllerGameRectangle
- ControllerPlayerSetup
- ControllerRules
- Controller Start

TestClasses:

- ChanceFieldTest
- TestingDice

8.2 Class Description

We will now be going through each of the classes and describe how the code works in more detail and why we decided on the architecture we did. We will be grouping some classes together as they may have very similar architecture or may be connected in a way that makes it better to explain them together.

Main

The class Main loads the first GUI to start the game. This GUI's setup is contained in the fxml file located under "fxml files/start.fxml". A new Scene is setup calling the start.fxml file stored in the variable "root" and defining the new windows size to 600x600 pixels. After the new Scene has been defined in PrimaryStage the last step is to use the method show().

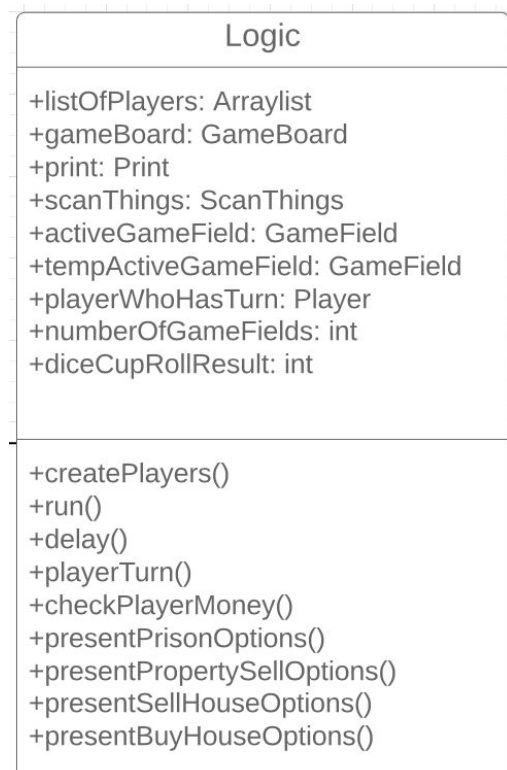
Logic

Logic is the class where the game will be running. Logic is the class that has all the relevant objects and makes use of them in our game loop.

First thing to take note of in our logic class is the importation of the ArrayList tool. We use ArrayList to create a list of players in the game, that during the game updates dynamically when a player does not have enough resources to continue the game. When there is just one player left in this array this player is the winner of the game.

We create core objects for our game at the top of our logic class such as our game board, which is what the players will be playing on, and our scanThings object, which the players will encounter every time the game requests an input from the player whose turn it is. We also create a GameField called activeGameField, which we use to keep track of which field is currently being played. By doing this we are able to better keep track of the active game field, instead of picking out a specific field from the ArrayList each time we want to do something to the field, like change ownership or run its landedOn() method.

When our game starts the first thing a player needs to do is to create the players that will be participating in the game. This is done by the method called createPlayers which takes an ArrayList of playerNames and for each player name it creates a player with that name.



We then get to our game loop, this is the loop that runs each player's turn. We have before this loop created an int $i = 0$ and for each iteration the game loop runs through 1 and is added to "i". If "i" is the same as the number of players in the game, "i" will reset back to 0. We use "i" to keep track of whose turn it is, so by "i" starting being 0, this means that the player that is on index 0 in the list of players array will go first and then index 1 until "i" is the same as the number of players and will then reset back to 0. An important method in this loop is the playerTurn method, which takes the player whose turn it is, and we have a Player variable called playerWhoHasTurn, which is set at the start of the loop. The playerTurn method is responsible for manipulating everything that has to do with each player's turn such as position on the board, money amount and owned fields. PlayerTurn has a crucial method call, which is the landedOn() method, which starts the events that happen when landing on a field of a specific type. We will go into more detail in the section about our GameField Class.

In our playerturn() we first check if the active player is in jail, and if so we make a call to presentPrisonOptions(), which will run the player through the options the player has while being in prison. If the player is not in prison the turn will continue as normal where the player has the option to buy a house, sell a house, or to sell a property, if the requirements for doing so is fulfilled.

Game board

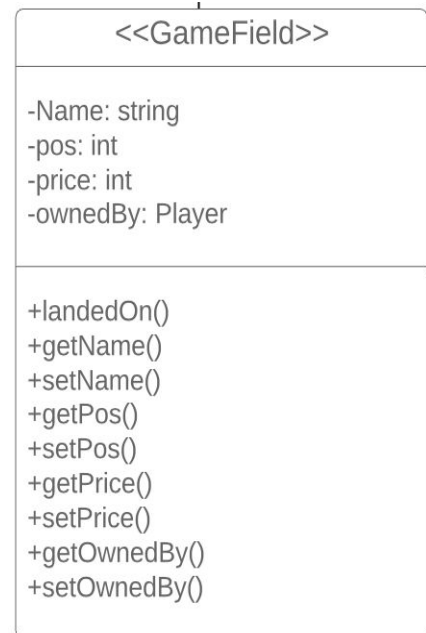
This is our class that represents our play area. It does not have a lot of functionality and does not play a core part of our game but serves to keep track of what the game board consists of such as game fields and dice cup. When we create a game board we also create our list of game fields and our dice cup for the board. When we want to manipulate data in a game field we need to access them through our game board class.

Player

Our player class consists of everything that a player has such as a name, money, owned fields and actions such as buying a property. We also have a variable called totalValue which is an integer that keep track of all kinds of values the player has including money, properties, ferries, breweries and houses on properties, this is used to calculate tax when the player lands on the tax field and has to pick between a flat tax or a procent based tax of everything the player owns.

GameField & sub-classes

Our GameField class is one of our core classes which plays a role in almost every part of the game. GameField is a super-class since we used inheritance to create our different kinds of game fields. We also gave each kind of field a type where we used enums. This way we could use the type of the field to do specific things depending on the type of the field. We also used enums to keep track of the different types of PropertyFields that we had. We needed this because it is needed to have a set of properties before you are allowed to build houses on your properties. The core method of each of our game fields is the landedOn() method that runs each time a player lands on a field. The landedOn() method in our GameField is to make sure that we did not get an error during the creation of our subclasses when we move our player around the board. The landedOn() method is meant to be overwritten in each subclass.

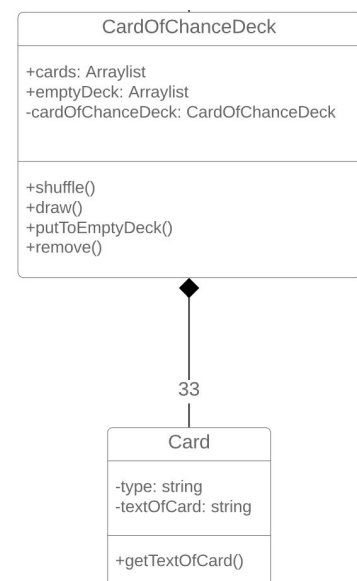


PropertyField, BreweryField and FerryField

These fields are similar in nature, because these are all fields that are purchasable by the player. This means that every time a player lands on such a field we have to check if the active game field is bought by another player and if so the active player has to pay rent to its owner. If it is not owned by any player the active player gets the option to buy it.

Card & CardOfChanceDeck

The Card class is used for the cards of chance, which are drawn when the player lands on a chancefield. It has two variables, type and textOfCard. The "type" variable is used in the ChanceField to determine what card is drawn, and the textOfCard string is printed to the player so they know what card they drew, and what it does. CardOfChanceDeck is the class that controls the cards from the ChanceField. It is created as a singleton. Singleton is a design pattern that is used to create one single instance of a class.



```
public static CardOfChanceDeck getInstance(){
    if(cardOfChanceDeck ==null){
        cardOfChanceDeck = new CardOfChanceDeck();
    }
    return cardOfChanceDeck;
}
```

This single instance is then returned in the `getInstance` method, which is called when the other classes need access to the deck. This is done to ensure that the card deck is consistent across the entire game, and that all players draw from the same deck. It holds an `ArrayList` "cards", which contains all the chance cards for the game. It has a `shuffle` method, which uses Java Collections to shuffle the deck. It also contains an `ArrayList` named `emptyDeck`, which the drawn cards are put into. If the cards list is empty (all cards have been drawn) we call upon the `changeDeck()` and all the cards in the empty deck are then added back to the active deck and shuffled. The drawn card list is then cleared. This was to be used in order to be able to reshuffle the deck if all the cards had been drawn. The class also contains a "remove" method, which removes the specified card from the deck. This is both used when putting the card to the empty deck, but also when a "Get out of Jail" card is drawn, since the player keeps this on their hand.

ChanceField

The `ChanceField` is a subclass of `GameField`. It contains all the functionality of the cards from the `cardOfChanceDeck`. In `landedOn()`, which is inherited from `GameField`, a card is drawn from the deck, and an action is taken depending on the card. All the methods are called from within `landedOn()`, in a large switch statement, which takes the String "Type" from the card. Several different types (for example "aktier", "dyrtiden", "præmie" and "tipning") have different texts, but have the same effect on the game. These are called in one block, instead of having to copy and paste the code for every single case.

In line 8-11 we use a static block to get the instance of `CardOfChanceDeck`. Using a static block means that the code inside is run one single time in the execution of the program. This specific block is used to get the `CardOfChanceDeck` instance and shuffle the cards. By using a static block, we make sure the program doesn't call the `CardOfChanceDeck` into `Chancefield` repeatedly, and that it is only shuffled once.


```
{
    cardDeck = CardOfChanceDeck.getInstance();
    cardDeck.shuffle();
}
```

In `propertyTax()`, we make use of casting. When adding elements, these are casted as property fields. This is done because the `ownedFields` from the player consists of game fields, as it is possible to own Property Fields, Brewery Fields and Ferry Fields, but the only game field type the player can build houses on is Property Fields. Since `ownedFields` consists of game fields, and the `getHouses()` method can only be used on property fields, these fields are casted as property fields for the new `sumList`. By checking if the field is a property field before adding it, we ensure we don't cast fields of any other types than property fields.

```
for (int i = 0; i < player.ownedFields.size() - 1; i++) {
    GameField temp = player.ownedFields.get(i);
    if (temp.getGameFieldType() == GameFieldType.PROPERTYFIELD) {
        sumList.add((PropertyField) player.ownedFields.get(i));
    }
}
```

Dice & DiceCup

Our Dice class has imported the tool `Random`, which we use to make dice that returns a random number between 1 and 6. The Dice class has a `roll` method which returns the result of the roll with the dice. We used two dice in our `DiceCup` class and have created a `shakeCup` method which calls the two dice `roll` method and returns the result of each of the dice roll returns added together. We decided to create it like this because it best represented how it would be done in the real world, but also because each of the dice roll results would be needed to check if a player is able to leave prison. This could be done with just a random number between 2-12, where the player in prison would get out if he rolled 2, 4, 6, 8, 10 or 12, but we wanted it to be as close to the real world as possible.

Print & ScanThings

Our `Print` and `ScanThings` classes are classes that make our code more readable and clean. We have in each of the classes created methods we would often call upon to make our coding process easier and more clean to read and understand. Such as when we want an answer from the player, the player is not able to give us an unacceptable answer, since the only acceptable answers are the ones presented. Our `Print` class was more of a help during our testing process when testing our game. This meant we were quickly able to get a lot of information by just typing one line instead of typing multiple print lines.

Controllers & fxml

The controller and fxml classes make out our GUI. The fxml classes hold the design of the GUI layout and the controller classes hold the functionality of buttons and text areas. The game board is initiated from the controller ControllerPlayerSetup and so is the Logic class which runs the game. In principle the five different Controller classes could have been written in one class but to make the code easier to navigate we decided to split it up so every GUI window has its own Controller class.

8.3 Bugs

Since the first day we started creating our digital version of Matador bugs have been a part of our development and have been taking up a lot of time. Bugs are a behavior that is not intended or wanted in your program. Bugs can take a lot of time to sort out, because problems that may seem simple can come from deep within the program, and searching for where the bug is hiding can be a grueling task.

Using IntelliJ IDEA as our IDE of choice has been a great help in bug finding. Even if we do not get warned about a bug before we run the program, or we do get a runtime error (an error while the program is running) IntelliJ will give us a lot of relevant information. Such as which line, method and class the error is in, and we are able to decipher where the error is around that. There are of course times where IntelliJ is pointing to a place with an error, but actually that piece of code works as it should, but information that it was given was not fit for the method where the error occurred. This means that the bug was where we did something to that information, but getting a starting point is a great help in fixing the bugs in your program.

The most common way we found bugs in our game was to play it and to keep track of our dynamically changing variables where we would keep track of the real value and would compare it to the value in the game. An example of this could be when we wanted to test if each player's position was updated correctly we would keep track of each player's position in the game fields array and a bug we encountered here was that the player position would go further out than array indexes and would therefore return an out of bounds error.

Using tests is also a great way to catch a lot of bugs and more specifically JUnit can be used to test specific things, but that is also the problem with this kind of testing. It tests very specific things, and if the developer team does not know where there specifically could be bugs, it is hard to test for it this way. We therefore tested big parts of our program by playing our game.

We also made use of a few lines of code to test specific scenarios like if we wanted to check when a player lands on a specific field we changed the result from the dice to make sure the player landed on the field we wanted to test, which can be seen on the picture below in line 104, that is where we change the result of of the roll that happened in line 91.

```
91         diceCupRollResult = gameBoard.diceCup.shakeDiceCup();
92
93         if (playerWhoHasTurn.getTurnsInPrison() == 3) {
94             System.out.println("Da De har været i fængsel i 3 omgange skal De betale 1000 kr. for at blive sat fri");
95             playerWhoHasTurn.outOfPrisonWithMoney();
96         }
97
98         if (playerWhoHasTurn.isInPrison() == true) {
99             presentPrisonOptions(playerWhoHasTurn);
100         }
101     }
102     if (playerWhoHasTurn.isInPrison() == false) {
103         diceCupRollResult = 30;
104
105         // Loop for player move (Move 1 field per iteration)
106         for (int i = 0; i < diceCupRollResult; i++) {
107             playerWhoHasTurnPos = playerWhoHasTurn.getPos();
108             playerWhoHasTurnMoney = playerWhoHasTurn.getMoney();
109         }
110     }
111 }
```

Picture taking from our Logic class

We did the same thing with testing cards, we just made sure that the card drawn was the card we wanted to test.

8.3.1 Bugs we have encountered

When a player rolls the dice in the program and the player moves from game field to game field, we made it so the player moved more like the player would in the real game, like 1 field at a time, meaning the player would pass a field, pass a field and then land on a field.

We had first implemented it so the player both passed and landed on the last field, what we had not implemented was a check if it is not the field the player is going to land on, and if it is not, the player will pass it. We implemented it as seen on the picture below.

```
108     for (int i = 0; i < diceCupRollResult; i++) {
109
110         playerWhoHasTurnPos = playerWhoHasTurn.getPos();
111         playerWhoHasTurnMoney = playerWhoHasTurn.getMoney();
112
113         playerWhoHasTurnPos = playerWhoHasTurnPos + 1;
114         playerWhoHasTurn.setPos(playerWhoHasTurnPos);
115
116         // Checking if player is out of bounce, if so go back to start
117         if (playerWhoHasTurnPos > numberofGameFields - 1) {
118             playerWhoHasTurnPos = 0;
119             // System.out.println("we set playerwhohasturnpos = 0");
120
121             playerWhoHasTurnMoney = playerWhoHasTurnMoney + 4000;
122             System.out.println("4000 kr. er blevet tilføjet til spillerens valutabeholdning ved passering af 'Start'");
123         }
124
125         tempActiveGameField = gameBoard.gameFields.get(playerWhoHasTurnPos);
126         checkPlayerMoney(playerWhoHasTurn);
127
128         if (i < diceCupRollResult - 1) {
129             print.printPassedField(playerWhoHasTurn, tempActiveGameField);
130         }
131         playerWhoHasTurn.setPos(playerWhoHasTurnPos);
132         playerWhoHasTurn.setMoney(playerWhoHasTurnMoney);
133     }
134     activeGameField = gameBoard.gameFields.get(playerWhoHasTurnPos);
135
136     System.out.println("De landede på " + activeGameField.getName());
137
```

Picture taken from our Logic class

In line 108 is where the player will move 1 field with each eye on the dice then we on line 117 check if the player is on the last field in the array we will place the player to the first field so the player moves in a circle. We then save the game field the player is on in tempActiveGameField. In line 128 is where we had our bug. We were not checking if the field was the last field, so we then implemented this if-statement, which checks if "i" is less than the amount of eyes on the dice -1 since we only wanted to print that the player passed a field if it was anything before the last field. This implementation makes it so it only prints out that the player passed a field if it is not the end destination. We then set the active game field after the movement of the player is done.

When we had implemented our game board and our game fields array, and we were able to play the game on a core level. We quickly figured out that we were not able to play the game for very long. This was because as soon as the player would be close to the end of the players first round about the board and he would pass the start field, the game would break, because we had not implemented a way to keep the player going around in a circle. Our implementation of this can be seen on the picture above on line 117. We made it so if the player position was on a field that was further than the second to last field, we update the players position to be on the start field. Since the player is moving in increments of one field, we knew that if that statement was true, the player should be on the startfield so we set the players position to be 0. This is a crucial step, because we in like 134 use the players pos to decide which field is the active game field for the player during the turn. If we did not do this, we would try to set the active game field to a field that does not exist and we would get an error.

8.3.2 Current bugs in our game

We have a few bugs where variables are not updated correctly. For instance we have a variable in the player class that keeps track of the total value of each player, which is used to calculate how much tax the player needs to pay. If this variable is not updated correctly a player will end up paying a wrong amount of tax in the process. We have implemented it in such a way that the method needs to be called to update the player value each time a change in the player value has been completed, and we have not implemented such an update after each change, and the variable will sometimes not be correct.

We also have an infinite loop in our method to sell a property which occurs if you do not have any properties to sell. This is because we are asking for which property the player would like to sell, tries to match what the player input to an index in the array of owned fields and if no match is found the program asks again and since there are no properties, there will never be a match.

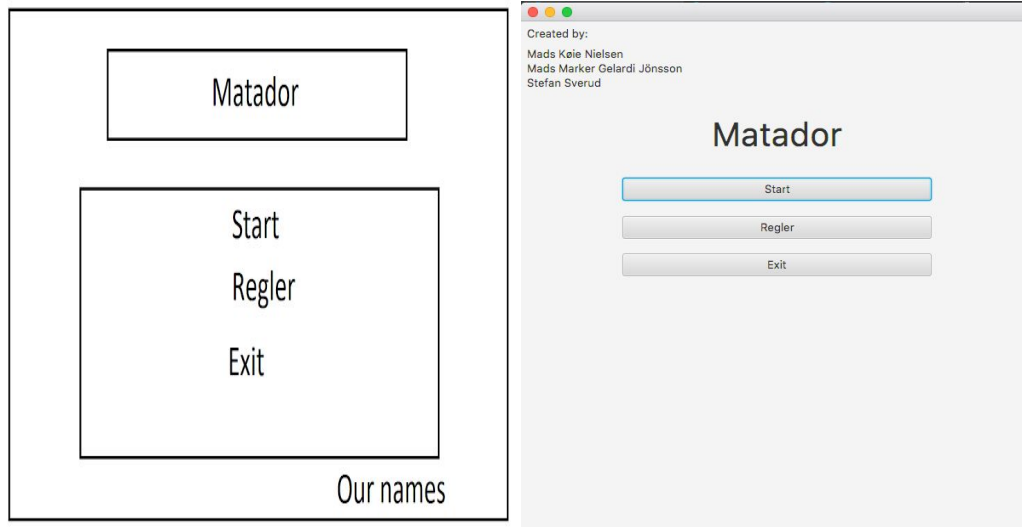
We have made a document with all our current bugs in our game, which can be seen in appendix [15.7](#).

8.4 Building the game logic

Before working with JavaFX for the GUI portion of the program, we decided to start out with a program running from the terminal. As we had limited experience with creating GUIs, we would ensure the program was functioning as intended before we started creating the visual aspects of the game. We knew how to work with the terminal and using print outs and Scanner seemed like an obvious way to get started building our game logic. Getting closer to a functional game logic in the terminal our intention was to add the GUI part of our game on top of the already functional game logic. This approach turned out to cause a lot of issues looking back

8.5 Creating a GUI for the game

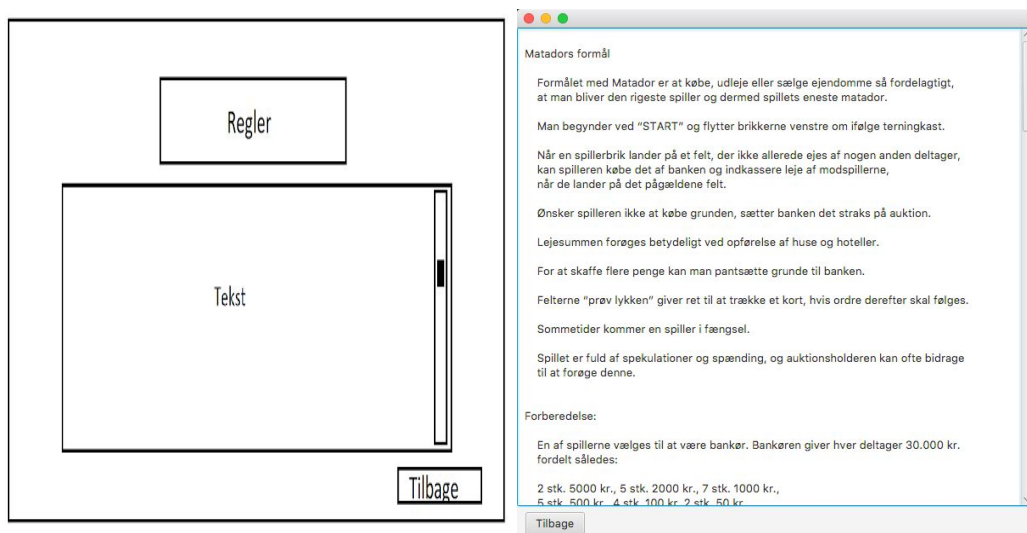
Our idea was to create GUI windows using a combination of scene builder, fxml classes and controller classes. We set out making some sketches in paint showing the content of the different GUI windows, we felt was needed for our game application. We sketched out a total of 3 screens for the different non-play activities, such as reading up on rules and deciding how many players would be playing the game. Window 1 is launched from the class Main.



window 1.

The button “Start” loads window 3, which leads to an option screen for further options of player setup. The button “Regler” loads window 2, where the user can read the rules of Matador. The “Exit” button is set up so it closes down the program.

Window 2 shows the players the rules of the game. It has a scroll down bar on the right side of the screen, to let the players display the whole text in a 600 x 600 pixels window.



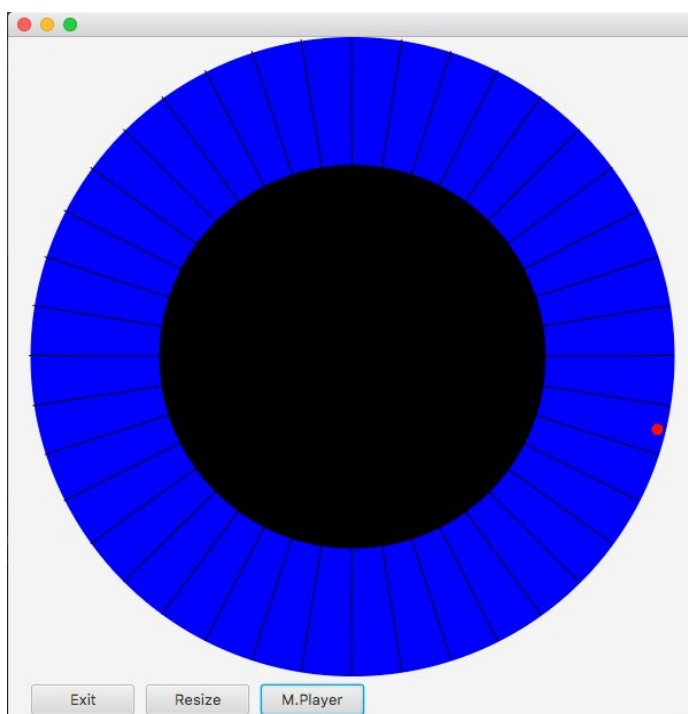
window 2.

The button “Tilbage” loads window 1 again. From there the player can continue by choosing “Start” which leads to window 3. In this window the players can write what name they want to be playing as. When the button “Start Game” is pressed the names of the different players are stored in a list, which is then passed on to the Logic class.

window 3.

In the window “Opret spillere” the players are presented with 6 textareas. The game automatically registers how many of the text areas are filled out by the players with their chosen player names and starts a game loop running a list using these names.

The buttons “Start Game V1 Circle” and “Start Game V2 Rectangle” load the game board and start the game.



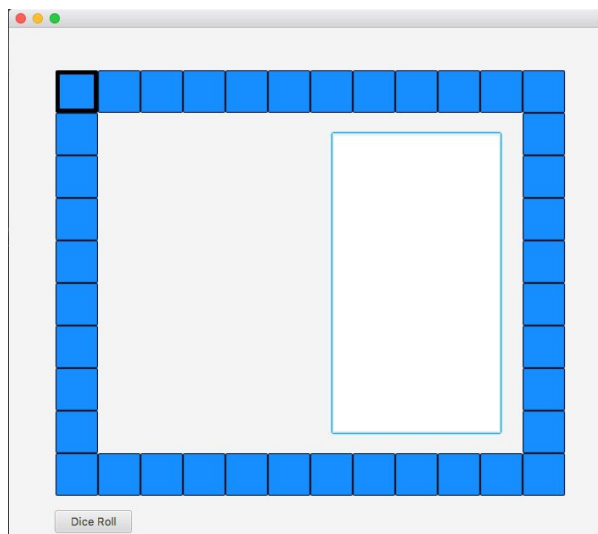
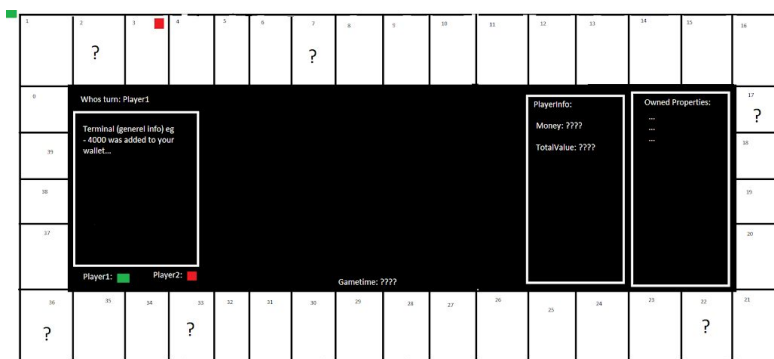
Our first design of the actual game board was heavily inspired by the physical Matador game board we had on hand. This version is a circular game board with all the information printed on the board. By trying to replicate this design we decided to display the same circular design. Our first game board design consisted of two circles and 40 lines dividing the blue area into game fields. The circles and lines are drawn in the controller class (`ControllerGameCircle`) and not in the fxml class (`GameCircle`). The intention was to be able to change the size of the game board by redrawing the circles and lines the board consists of, in relations to the

size of the window. Users are then able to drag the game board window to the desired size and thereafter hit the “Resize” button to change the size of the game board itself.

The logic that was used to calculate the end positions of the dividing lines in the game board, was then reused to calculate the position of a players position on the game board. The logic of moving the player to the game board was tested using the button "M.Player".

However there was a problem with this design. We had achieved a game board that could resize to any screen size. But we also wanted to be able to mouse click on any field on the game board and get information about the field in a Text area. We decided to go with the shape Rectangle in scene builder to represent every field since this gave us the option of a mouse clicked event.

The reason we had to redesign our game board was mainly because it would take too much time trying to mouse click a specific gamefield with the way we had set up the round game board. Instead we designed the game board using the predefined shape rectangles placed just beside each other, around the edges of the GUI window. This way we made it easier for ourselves to implement the rectangles as individual game objects that could be mouse clicked.



window 4.

8.6 Adding the GUI to the game logic

It was now time to start connecting our GUI to our game logic. Here we started running into serious problems with our design process. By this time the game logic was about 3/4 finished and running with input and output using the Scanner class in the terminal.

Our plan was to start substituting the input and output in the terminal with buttons and text areas in the GUI.

This turned out to be a much bigger task than anticipated, since the Scanner class acted as a system block that stopped the code from executing, in order to wait for input. Looking back we should have known it would cause trouble, but we thought it would be simple to just layer our code with a GUI.

The first step was to input the player names through the GUI instead of the terminal. We did this by collecting the names in a list and passing it to the Logic class.

To illustrate; below is an example of old code operating with a Scanner as input from the terminal.

```
/*
// old version of createPlayers
public void createPlayers(int numberOfPlayers) {
    // for each numberOfPlayers create scanner and create a player
    for (int i = 1; i < numberOfPlayers + 1; i++) {
        System.out.println("What is the name of the " + i + ". player?");
        // Scans name, create player,
        // add player to listOfPlayers,
        // print out the player
        String nameOfPlayer = scanThings.scanString();
        Player player = new Player(nameOfPlayer);
        listOfPlayers.add(player);
        System.out.println("Okay the name of the " + i + ". player is: " + nameOfPlayer);
    }
    System.out.println("Printing out our list of players:");
    System.out.println(listOfPlayers);
}
*/
```

Below can be seen a new version of the method createPlayers being called from the ControllerPlayerSetup class. This is a good example of how different the code looks after it has been adjusted to run from our GUI. Even though the functionality of the code in this case is the same the process of getting to this new configuration was rather complicated. Originally we thought we could build a logic with output and input in the terminal where after a GUI could be developed to run the same logic. This turned out much different in practice.

```
public void createPlayers(ArrayList<String> playerNames) {  
    for (String s : playerNames) {  
        Player player = new Player(s);  
  
        if (!s.equals("")) {  
            listOfPlayers.add(player);  
        }  
    }  
    System.out.println("Liste af spillere: " + listOfPlayers + " Antal: " + listOfPlayers.size());  
}
```

Next challenge was to convert the `welcomeToTheGame` method to GUI use. To begin with this method took integer as input via a `Scanner` to represent the number of players to play the game. A `ComboBox` was then set up in the `playerSetupController` to take this integer input and pass it into the `Logic` class. In the end the whole `welcomeToTheGame` method was passed over, and the number of players was instead derived from getting the size of the list `listOfPlayers`.

```
public void welcomeToTheGame() {  
    // Print out the gamefield list  
    //System.out.println("printing the gamefield list:");  
    //print.printGameFields(gameBoard.gameFields);  
  
    //System.out.println("How many player are going to play?");  
  
    // getNumberOfPlayersPlaying kommer fra ControllerPlayerSetup via "extendt" øverst i logic klassen  
    //System.out.println("??? " + getNumberOfPlayersPlaying());  
    //numberOfPlayers = getNumberOfPlayersPlaying();  
  
    //System.out.println("Okay you are going to be playing " + getNumberOfPlayersPlaying() + " players");  
    //System.out.println("Lets start creating your chars!");  
  
    // createPlayers(numberOfPlayers);  
    //createPlayers();  
}
```

The next difficulty was to replace the delay method in our main while loop in Logic with a button action from the game board GUI. By this time we had started to realise the immensity of our approach. Continuing like this, every method in our Logic class would have to be rewritten more or less from scratch.

We had already spent a lot of time and effort creating the terminal version. We now realise how little of the logic behind it, could actually be used in connection with our GUI. Because we were at a crucial stage of our game development we came to the conclusion that we had to prioritise on either finishing the GUI visuals or the terminal game version, or have a half and half of both. We decided to focus our effort on finishing the terminal version.

We ended up creating a very primitive GUI example where we show how we would have liked our logic to interact with the gameboard, where clicking on field 1 makes it change bordercolor, meaning we needed to add the correct functionality to the event like opening a alert window or a menu for the player to pick what the player wants to do.

9. User Guide

The following setup description is based on using the IDE IntelliJ to run the program.

The program can be cloned from the git repository <https://github.com/jonsson0/Matador.git>

Open IntelliJ and choose | Get from Version Control |

The first time running, it needs two inputs during the setup process to function.

9.1 JavaFX

When running the program for the first time all the javaFX imports in the classes will be red. This has to be taken care of before we can continue. Start by downloading the SDK file corresponding to your system <https://gluonhq.com/products/javafx/>.

Then go to the IntelliJ menu and follow this thread in the main menu:

| File | Project Structure | Libraries | --choose the + at the top of the screen-- | Java |
--find the SDK file you downloaded from the link above, open it and choose the lib folder--
| Open | Apply |

All the red javafx imports should now turn white or black depending on what theme you are using in IntelliJ.

9.2 VM options

From the main menu, select | Run | Edit Configurations| Select | Application | Main | from the menu on the left side of the popup screen. Find the location of the SDK file downloaded in the section above, copy the path to the lib folder, inside the SDK file, and replace it with the blue highlighting below.

```
--module-path /Users/Hans/Desktop/javafx-sdk/lib --add-modules=javafx.controls,javafx.fxml
```

After updating the highlighted path, copy the whole line above and paste it into the input area "VM options:"

Choose Apply and OK

You can now run the program from IntelliJ's main menu choosing | Run | Run 'Main' |.

9.3 Running the game

This section instructs how to navigate through the game GUI.

9.3.1 Start window

The first GUI window presents the player with 3 options. Start, Regler (rules) and Exit. Exit terminates the game, Regler opens a new window displaying the rules of the game and Start opens another window for player name registration. See [“8.5 Creating a GUI for the game”](#) for pictures of the GUI windows.

Choosing Regler a new window opens with only one option, a Back button, taking the player back to the first GUI.

9.3.2 Player setup window

In the player setup window the players input their player names. When this is done the game moves forward by pressing either “Start Game V1 Circle” or “Start Game V2 Rectangle”. After this the rest of the game is played in the terminal, following the print out statements and using the keyboard and enter key.

10. Testing

In this section we will describe how we have accomplished our testing of the code. We will first describe our method of unit testing thereby followed by integration testing.

10.1 Unit testing

For testing our code we decided to use Unit testing using the JUnit framework. By using JUnit, we can test different methods in the program without having to run the full program each time we make a change. For example, in the ChanceField class there are 33 cards, with 27 different cards and 6 duplicated (although some exhibit more or less the same effects). Instead of having to run the game, and playing until we hit a chancefield, the test class ensures everything is working as intended.

Testing this way has several other advantages. The “bankReward” method is used for several cards, where the player gets a monetary reward. By using a unit test on this method, we ensure that every card that uses this method not only works the same way, but also works as intended. Using the JUnit framework’s “assertTrue” method, which is a boolean comparator, we check to see if the player’s account balance is what it is supposed to be after the method we are testing is called. If we get a different value than expected, the test will tell us what the actual result was and compare it to the expected value. This makes it quite simple to figure out what is wrong with the method, and correct it.

Other, more complex methods also gain advantages from unit testing. If a method can have several outcomes depending on the situation, these can also be tested quickly and efficiently. By using unit tests on these methods, it is possible to “artificially” set up these specific situations and test their outcomes like before. In the moveBack method for example, there is a certain condition for the 2 fields on the board. If the player had to move back 3 spaces here, they would be moved out of bounds in the game fields arraylist. By testing this specific occurrence, we can ensure the method works as intended for every possible scenario.

Originally, testing the ChanceField class proved more difficult than expected. The first version of the class didn’t have the different methods defined separately. Instead, almost all functionality was contained within the switch in the landedOn() method, which is inherited from the GameField class and called from the Logic class. However this turned out to be extremely inefficient for testing. The entire cardDeck had to be excluded, with the exception of the card we wanted to test, and we had to run the entire program every time. Furthermore we had to make small temporary changes to a lot of other classes (E.g, we had to hardcode the dice roll to land on the desired chancefield), which had to be changed every time depending on what card we were testing. Eventually, most of the functionality of the switch cases were moved to individual methods, which we then tested using JUnit testing.

```
@Test
void propertyTax() {
    setup();
    p1.buyField((logic.gameBoard.gameFields.get(1))); // costs 1200
    p1.buyField((logic.gameBoard.gameFields.get(3))); //costs 1200
    p1.buyHouseOnProperty((PropertyField) p1.ownedFields.get(0));
    cf.propertyTax(p1, amount: 500);
    assertEquals( expected: 6500, p1.getMoney());
}
```

An example of JUnit testing for the “propertyTax” method from ChanceField.

The two functions we ended up using are assertEquals and assertTrue.

“assertEquals” is used to compare two numbers, integers in our case. This is used in propertyTax, among others. By setting up the player in a way where we know what their starting money is, and then using the method we are testing, we ensure the method is actually drawing the correct amount from the players account. In the “propertyTax” example, the player is given 10000 kr. as a starting amount. Then, they buy 2 properties at 1200 kr each, and then a house on the first property. We know the player should use 2400 for the properties alone, and 600 for a house. Lastly, the amount of money owed depends on the card (oliepriser or ejendomsskat). In the test, we set the amount to 500, and expect the player money to be 6500 after the method is used. This expected value is then compared to the actual money (p1.getMoney()), to see if they are actually equal.

```
@Test
void moveToFerry() {
    setup();
    cf.moveToFerry(p1, logic);
    assertTrue( condition: logic.gameBoard.gameFields.get(p1.getPos()).getGameFieldType() == GameField.GameFieldType.FERRYFIELD);
}
```

“assertTrue” checks whether a boolean condition is true or false. For example, this is used for the methods where the player is moved to the next field of a certain type (i.e FerryField). By using assertTrue, we ensure that the field the player is moved to is actually of the required type. If the comparator is true, the test will pass.

10.2 Integration testing

Integration testing is used to ensure separate parts of the program works together. The reason we use unit testing before integration testing is to ensure the classes are fully functional by themselves, so it's easier to identify the problems when the classes need to work together. For example, in our program the `ChanceField` and `CardOfChanceDeck` are dependent on each other. In our integration testing, we tested whether the `ChanceField` could access the card deck. For this part, we made a print-out of the number of cards in the deck, then after the card had been used, we printed the number of cards again to see if the card had been removed.

10.3 System Testing

System testing is the next step after integration testing. The goal of system testing is to ensure that all the pieces of the program work together as intended.

System testing was primarily done by running the game. When a new type of game field was completed, we would hardcode the die to land on these, so we didn't have to keep rolling until we randomly landed on the field we wanted to test.

10.4 Acceptance Testing

Acceptance testing is usually done by the user who needs to use the program. As we don't have such a user, we primarily did a simulation of what an acceptance testing would be like ourselves, while we were doing the system testing.

In the end we did not have the possibility to execute an acceptance test since we did not have a functional GUI to test it on.

11. Discussion

In our discussion we will present three aspects of considerations we gained throughout this project. The first will go into how user friendly our game is for the user. It presents what efforts we have made to enhance user friendliness and ideas for making it even more enjoyable for the player. Second part dwells on how we might have organized our code in a different structure, highlighting an example from our rapport concerning the creations of our ChanceCards. Finally we will go through our entire project process with reflections on how we would change the order of tasks, if we would go through the whole process again.

11.1 User friendliness

User friendliness means that using something should be easy and intuitive. This principle can also be applied to coding projects and games. When using a product the user should not be confused as to what the user should do in any given situation. The experience should be smooth and the user should not feel stuck in a situation where the user is left having to just try things out to continue, the user should always have an idea what is next.

We have in our program used different tools to make the game experience as smooth and enjoyable for the players as possible. Things such as doing all the calculating when it comes to buying, selling, resting and passing - these things can be a pain to do mid game and can for some take out the fun of playing the game. We have in our game automated a lot of these processes. This means that when a player lands on a field that another player owns the active player automatically pays the other player rent for the stay. This will take some of the communication between the players out of the game which can also be a bad thing, since communication between the players can also be part of what makes the game so enjoyable to play, but due to the fact that we were not able to integrate a full GUI, we decided this was the best way to do it, where on a GUI there could be a button that needed to be pressed for the rent payment to actually go through, so the button is like a call for rent payment.

Another thing we did to make the game more user friendly was to make it as easy as possible for the player to respond to the game logic. Each turn we ask the player if the player wants to buy the property, house, sell property or sell house, by having the player respond with a 1 (yes) or 2 (no). The way we decided to implement the turn for each player could be even more improved by having it be more like a menu where the player could pick between 5 options:

1. Buy property
2. Sell property
3. Buy house
4. Sell house
5. End turn

This would run in a while loop so the player could do anything he wanted in whatever order he wanted and end the turn when done. This would reduce the amount the player has to respond with 2 (no) to a lot of the questions, since the player would be able to choose. It could also be cut down more to make it into sections as well:

1. Buy & sell options
2. View stats
3. End turn

Where pressing 1 would take you to the list seen above, pressing 2 you would see all your owned fields + extensions, your money and if you had a get out of jail card.

We did not work more on the player turn as we made it with the expectation that it mostly would be replaced by our GUI where the player would have buttons and information visually available on screen, so the player easily can make decisions during the turn.

11.2 Different Design Decisions

Another approach to the cards could have been to use inheritance. By creating a generic Card superclass, with a string containing the text of the card, and a method to be overwritten later. There are pros and cons to this as well. In a general sense, it could give a better overview of the specific cards, and their functionalities etc. However, in the current version there are 33 cards, but since there are some cards that are represented more than once, essentially there are 27 individual cards. These can be further subdivided. For instance there are several different cards that will have the player pay to the bank, or move to certain fields on the board. These could have been created as individual subclasses (ie. MoverCard, PaymentCard and so on). In hindsight, this approach would have been significantly more efficient. The “type” the cards are given are more of a reference to the programmer than the program itself, and because of this, only cards that have the exact same functionality are given the same type. These are then used in different cases in the switch in landedOn(). This means that there are several switch cases that pay the player, but they are separated into different cases depending on how much the player is to be paid. Instead the method would take an amount as an argument, and include either an if-else statement or a switch block depending on the amount for the specific card. The switch in the ChanceField would then check which version of the Payment card it is, and run the relevant code.

Another approach to the cards could have been to use inheritance as well. Since the “types” in the Cards are primarily used to access the cards, which can’t be done with their position since the deck has to be shuffled, giving them certain subclasses instead could have made the full deck more manageable. A Superclass “Card” would have a string for the text of the card, and enums for the different types. Then, the cards could be divided into subclasses, for instance “MoveCard”, “PaymentCard” and so on. These would then define their own texts and methods. Since there are several cards with similar functions, for instance where the player has to pay the bank, that have different amounts, the method would take the amount of the card as an argument. A switch case could be used to differentiate what version of the card was used, and execute the relevant version of the method. This would make the ChanceField class much less cluttered, while also keeping the cards clearly separated.

11.3 What would we have done differently

Our first step, if we had to remake our project, would be to do an in depth game research of Matador before doing anything coding related. We would gather and play the game while taking notes and starting the process of creating an activity diagram. The reason behind these thoughts is that all the project members would have a fundamental knowledge of the game and its rules and the mechanics driving the game. The start of an early activity diagram would give an insight into how the player turn algorithm could work and give a basis for the workings of the logic class.

Another aspect we would focus on earlier would be a system specification list for the game. This list should be organised in “need to have” and “nice to have”. The priority of sequence should ideally be organized with the functionality most valuable to the player at the top and the functionality least valuable at the bottom.

To make sure we had a mutual understanding of our envisioned finished product, we would sketch out on pen and paper the visuals for the player in terms of the GUI game windows, so that the project had a design to aim for. This approach would also have given us an idea of priorities to focus in terms of visuals, which would translate to an outline for the code of game logic.

Thereafter we would set up a class diagram and discuss the architecture of the system. This would give us an indication of the associations between the classes, and would help us get an idea of a priority list for what is important for the program.

With our class diagram and our activity diagram we now have a much better understanding of the program we want to make, and we would be able to fill in a lot of tasks into a priority list in a kanban board. We would always be keeping the kanban board upto date and would return to update it whenever an assignment or issue has been resolved.

Although the players ability to trade with each other should have been high on our priority list, we overlooked this functionality and only realized this, when also realising how poorly we had implemented our use of the kanban board tool. The Kanban tool emphasizes especially on making sure a developer team is focused on the functionalities that have the most value to the customer. In this case the players of our digital Matador game. Thus our highest priorities should have been to develop all the player functionalities first such the player interactivity was even more present. This also goes for the usability of our system. We should have made sure that we had the time to develop a satisfying GUI for a better game experience.

When we had set up all the preliminary project setup, we would start to code part of the GUI windows so every piece of logic game code could be tested in the GUI instead of having to rely on making terminal commands. This would make sure we avoided the issue concerning problems transforming an almost finished game logic to GUI.

Beginning the coding part of the GUI would also mean beginning coding the game system of Matador. During this process, we would have a bigger focus on unit testing, as it would make the process significantly faster, since we wouldn't have to run the program every time something needed testing. In doing so we would ensure that we had working code, and from there on we would be able to make a corresponding part of the GUI. We would then always have a working logic with a GUI to support it.

We changed our approach to relying heavily on inheritance for our game fields, but we went into it too quickly without having fully understood how it worked exactly.

We therefore ended up keeping more than we should from our old solution, which caused us to use old parts of our old solutions for new ideas and solutions. This made us dependent on those old solutions, and would require big changes in our code structure to untangle old parts of the solution.

Furthermore we would make sure to regularly revisit and reassess our diagrams so as to make sure they corresponded with our code and whether changes needed to be made in one or the other.

The unit testing of the ChanceField in particular proved to be more difficult than expected. By putting all functionality inside a single switch case at the start, it proved very difficult to test. After we realised this, we took all the code from the switch cases, and created them as methods instead. Then it was possible to test the individual methods in a much more controlled matter.

Setting up the tests themselves did also pose some challenges, but that was more due to our little experience with JUnit. When we learned how to set up these tests, the process was fairly simple.

12. Conclusion

This project's foundation was about utilising the principles of object-oriented programming in designing the architecture of our program. We wanted to create a good game seen from an object oriented standpoint, which we have throughout the rapport argued in favor of. We have had OOP principles in the back of our heads when writing code, and that can especially be seen with the design of using inheritance in our game field classes. This is a good example of how OOP is supposed to encapsulate objects and make sure to divide the responsibility to relevant encapsulated pieces of code. Designing our field classes we also had inheritance in mind. This meant that they were different classes, but still part of the same cluster. With our use of OOP we have learnt that having OOP principles in mind while coding is worth it and will save time in the end. Over having to revisit code to untangle dependencies that can easily occur if during the development process you do not respect the OOP principles. We have also learnt that using OOP principles makes revisiting code a lot easier and less time consuming, because the changes we make only affect the encapsulated pieces of code, and the rest of the program will adapt to the changes.

In order to communicate our program we wanted to create a GUI for the player. We wanted to make it as easy for the user to play whilst having fun as well. We did not accomplish that part of our ambition. In terms of player user friendliness our game did not reach our own expectations. Our first iteration of our GUI was of a circle with dividing lines to make up game fields, but we did not take into account the practicality of this design. The reason why this first iteration went wrong was because we didn't apply our acquired object oriented thinking. Drawing a big circle with lines drawn on top is going against the object oriented thinking, because we are not able to interact with each field as it's only visually that there are different fields, meaning it is not understood as different fields. We chose to reiterate our gameboard GUI design through sketches this time with object oriented thinking in mind, and came up with individual squares that could encapsulate the game field data from our game field class, which we were able to partially implement on our first game field on our GUI, where we have the mouse click handled and we execute a piece of code when clicked.

User friendliness was a priority and as such our GUI was on our list of 'need to have', but we did not manage to finish it, because we chose to focus our attention on the game logic, as it determined whether the game was playable or not, and thus being higher priority.

The use and modelling of our system through UML taught us the importance of applying it from the very start of a computer science project and especially creating it as a continuous process throughout our project. The benefits from having it applied from the very beginning counts as having a better understanding and overview of the data structure for what a team is about to code and how the system might behave. This approach will give a developer team a preliminary algorithmic flow of their system, and might give an outline for their class structure.

13. Perspectivation

13.1 Writing the code in Danish

Choosing danish class, variable and method names could have been an option worth considering more deeply. This could have been relevant in this case for several reasons. First the game board we wish to digitize in this project has all text and names in danish. Translating this into english creates an extra layer of complexity which considering the groups wish to improve java skills might take away attention from this objective. Our original reason for choosing english for our code section was the premise that non danish speakers should be able to understand and contribute to the code in the future. This objective however is not that relevant for this project since the group has no intention of continuing the development process after the project is finished.

13.2 Access to the game rules

Looking back on our design we see that the rule window should have been accessible to the players throughout the game. In our current design the players are only able to read the rules before the game starts and not once the game has started. One way to solve this could be to create a separate rule window that is open throughout the game.

13.3 Matador as a Mobile Application

Very briefly we discussed in the group whether or not it could be possible to launch the game as a mobile application. With all the focus on having mobile games and being on the go, it would have been fun to see if it could have been possible to go in the direction of a mobile application. How exactly we could accomplish this we never got around to talk about, but with the application game market seeming to continuously expand it would likely have been a challenge worth trying out, especially since IntelliJ has kotlin as an extension.

14. Literature

Anderson J. D & Carmichael A. (2016). Essential Kanban Condensed. Seattle, Washington: Lean Kanban University.

docs.oracle.com. Java - ArrayList. lokaliseret d. 29. maj på:

<https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>

Git --local-branching-on-the-cheap. Getting Started - About Version Control. Lokaliseret d. 25 marts på: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

Java T Point. Inheritance.

Lokaliseret d. 10. april 2020 på: <https://www.javatpoint.com/inheritance-in-java>

Lucidchart. (2019, 4. september). *How to Make a Flowchart in 60 (ish) Seconds!* [Video File]

Lokaliseret d. 18. maj 2020 på: <https://www.youtube.com/watch?v=JMuys5DyunE>

Lucidchart. (2018, 7. februar). *UML Use Case Diagram Tutorial.* [Video File]

Lokaliseret d. 19. maj 2020 på: <https://www.youtube.com/watch?v=zid-MVo7M-E>

Lucidchart. (2017, 21. juli). *UML Class Diagram Tutorial.* [Video File]

Lokaliseret d. 19. maj 2020 på: <https://www.youtube.com/watch?v=UI6lqHOVHic>

Seidl, M., Scholz, M., Huemer, G., Kappel, G. (2012). UML @ Classroom: An Introduction to Object-Oriented Modeling. Heidelberg, Germany: dpunkt verlag GmbH.

Spilregler.dk. Matador regler. Lokaliseret d. 20. april 2020 på:

<http://spilregler.dk/matador/>

Tutorialspoint. Java - Inheritance. Lokaliseret d. 10. april på:

https://www.tutorialspoint.com/java/java_inheritance.htm

15. Appendix

15.1 Appendix 1 - Rules of the game Matador

The rulset below is taken form the homepage Spilregler.dk (Spilregler.dk, ¶ Matador regler).

Målgruppe

3 – 6 deltagere fra 10 år.

Matadors formål

- Formålet med Matador er at købe, udleje eller sælge ejendomme så fordelagtigt, at man bliver den rigeste spiller og dermed spillets eneste matador.
- Man begynder ved "START" og flytter brikkerne venstre om ifølge terningkast. Når en spillerbrik lander på et felt, der ikke allerede ejes af nogen anden deltager, kan spilleren købe det af banken og indkassere leje af modspillerne, når de lander på det pågældene felt. Ønsker spilleren ikke at købe grunden, sætter banken det straks på auktion.
- Lejesummen forøges betydeligt ved opførelse af huse og hoteller.
- For at skaffe flere penge kan man pantsætte grunde til banken.
- Felterne "prøv lykken" giver ret til at trække et kort, hvis ordre derefter skal følges.
- Sommetider kommer en spiller i fængsel.
- Spillet er fuld af spekulationer og spænding, og auktionsholderen kan ofte bidrage til at forøge denne.

Forberedelse:

En af spillerne vælges til at være bankør. Bankøren giver hver deltager 30.000 kr. fordelt således:

2 stk. 5000 kr., 5 stk. 2000 kr., 7 stk. 1000 kr.,

5 stk. 500 kr., 4 stk. 100 kr. 2 stk. 50 kr.

Banken beholder resten af pengene samt skøderne, de grønne huse og de røde hoteller. Gennem banken foregår alle spillets ud- og indbetalinger undtagen leje, der betales til ejeren, samt handel med skøder og løsladelseskort, der foregår blandt spillerne indbyrdes.

"Prøv lykken"-kortene lægges i en bunke på spillepladen med bagsiden opad.

Selve spillet

Deltagerne stiller deres biler/brik på feltet "START" og bliver enige om, hvem der begynder. Alternativt kan der slås en terning om hvem der starter. Spillet fortsætter derefter i urets retning.

Første spiller kaster begge terninger, og flytter sin bil/brik så mange felter frem, som øjnene viser. Når en spiller har benyttet retten eller opfyldt pligten, som feltet angiver, går turen videre til næste spiller. Hver gang man passerer "START", modtager man 4000 kr. fra banken.

Lander man på et "**prøv lykken**", skal man tage det øverste kort i bunken med Prøv lykken-kort og rette sig efter ordlyden på det. Når et kort er benyttet lægges det nederst i bunken.

Lander man efter et terningkast eller ifølge ordren på et af Prøv lykken-kortene på en grund eller virksomhed, der ikke ejes af nogen anden deltager, kan man købe denne af banken for den pris, der står på feltet, og man får så udleveret skødet, der lægges med forsiden opad foran spilleren. Efter de takster der står på skødet, kan man nu opkræve leje af de spillere, der lander på ens grund. Køber man ikke skødet, sætter banken det straks på auktion, og denne har alle lov til at deltage i.

Lander man på "**De Fængsles**", skal man gå direkte i fængsel og modtager ikke de 4000 kr. for at passere "START". Lander man derimod på feltet "**I Fængsel**", er man blot på besøg og fortsætter næste gang uden straf.

Indkomstskatten har man lov til at betale med 4.000 kr. Men man kan også betale 10% af sine værdier: Kontanter, bygninger og den påtrykte pris for sine grunde og virksomheder (også pantsatte). Spilleren skal vælge betalingsmåden, inden han tæller sine værdier sammen.

Man får et **ekstra kast**, hvis man kaster 2 ens (f.eks. 2 femmere), og man skal rette sig både efter forskrifterne for det felt man på efter første kast og efter ekstra kastet. Kaster man 2 ens 3 gange i træk, må man ikke flytte tredje gang, men skal gå direkte i fængsel.

Feltet med **Parkering** er et fristed, indtil man skal kaste igen.

Man kommer ud af fængslet på en af følgende måder:

1. Ved at betale en bøde på 1.000 kr., inden man kaster terningerne.
2. Ved at benytte et af løsladelses kortene fra bunken med Prøv lykken-kort.
3. Ved at kaste 2 ens. Man flytter straks det antal felter frem, som øjnene viser, og har alligevel et ekstra kast

Man kan ikke blive i fængslet i mere end tre omgange. Får man ikke 2 ens når man kaster tredje gang, må man betale bøden på 1.000 kr. og flytte, som øjnene viser. Er man i fængsel, har man stadig ret til at købe grunde (ved auktion eller handel spillerne imellem), men man kan ikke kræve leje af de andre spillere.

Huse og hoteller

Ejer man alle grundene i samme farve, får man dobbelt leje af de ubebyggede grunde og har ret til nårsomhelst at bygge huse, der købes hos banken til den pris, der står på skøderne.

Der skal bygges jævnt, det vil sige at man kan opføre det første hus på den grund man ønsker, men inden andet hus opføres på den grund, skal der være bygget et hus på hver af de andre grunde i gruppen.

Inden man opfører et hotel, skal der være opført fire huse på hver grund i gruppen.

Der må kun opføres et hotel på hver grund. Når man køber et hotel, afleverer man de fire huse tilbage til banken

Banken skal, når som helst man ønsker det, tage bygningerne tilbage til halv pris. Prisen for et hotel er fem gange prisen for et hus.

Har banken ingen bygninger, når man vil købe, må man vente til der kommer nogle tilbage. Er der flere der vil købe, og har banken ikke nok til alle, sætter banken de bygninger der er, på auktion.

Indbyrdes handel med ubebyggede grunde og virksomheder er spillerne tilladt til den pris, de kan blive enige om.

NB! Har man bygget, skal man sælge bygningerne tilbage til banken, inden man kan afhænde nogen grund i den pågældende gruppe.

Pantsætning

Man kan kun pantsætte sine ubebyggede grunde og virksomheder til banken for det beløb, der står trykt på skøderne. Har man bygninger på grundene, skal man først sælge disse tilbage til banken. Spilleren beholder skøde kortene, men vender bagsiden opad. Renten er 10% (der rundes op til nærmeste 100 kr.), og renten betales sammen med lånet, når pantsætningen hæves.

- Hvis en pantsat ejendom sælges, og køber ikke straks hæver pantsætningen, skal han alligevel betale 10%, hvis køber senere hæver pantsætningen.
- Af pantsat ejendom kan der ikke opkræves leje.
- Banken giver kun lån mod pantsætnings sikkerhed.
- Pantsætning af grunde og virksomheder samt handel med bygninger, sker kun gennem banken.
- Spillerne må ikke låne indbyrdes.

Glemmer man at opkræve leje af en medspiller, har man tabt sin ret, når spiller nr to efter vedkommende medspiller har kastet terningerne.

Fallit

Skylder en spiller mere, end han ejer, skal han overdrage alt til sin kreditor efter at have solgt eventuelle bygninger tilbage til banken, og han må derefter udgå af spillet.

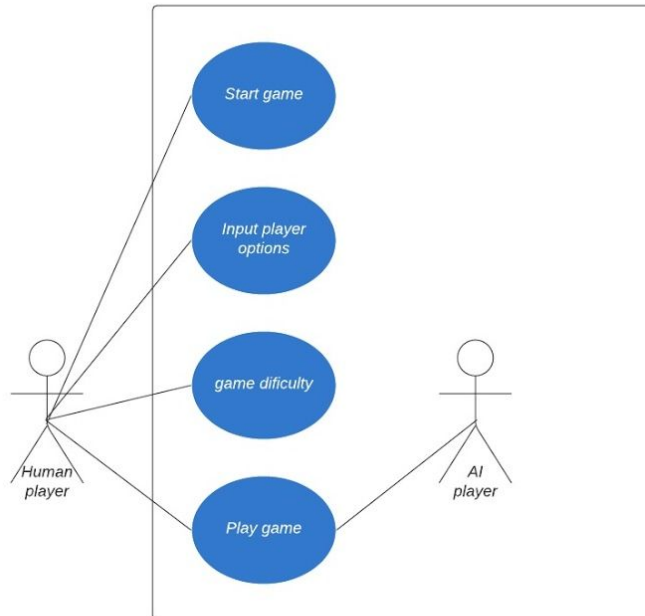
Er det banken, der er kreditor, sælger bankøren straks modtagne grunde på auktion.

Hurtigt spil

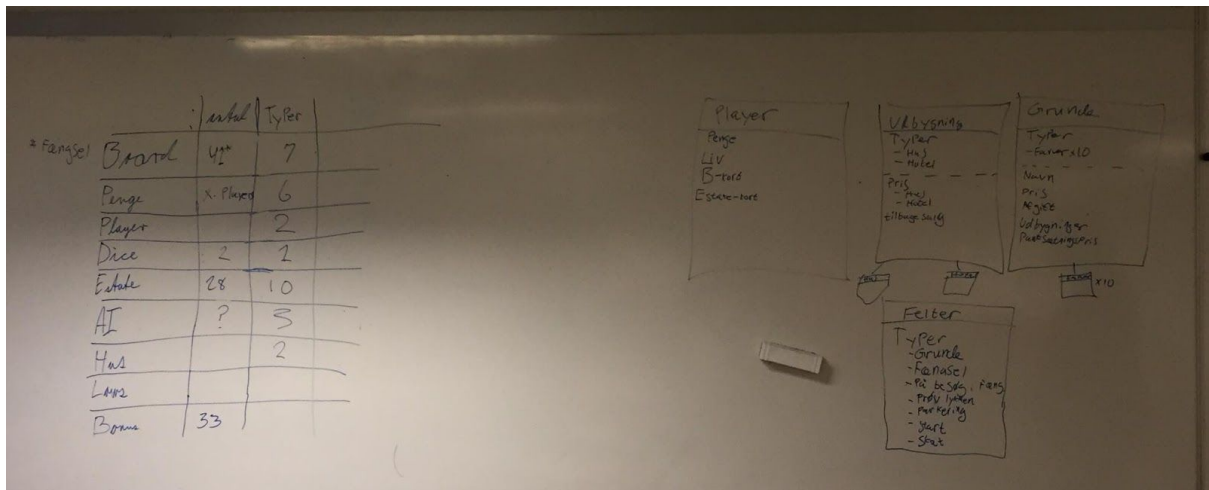
Bankøren blander skøde kortene og giver hver spiller to skøder, for hvilke bankøren modtager den trykte pris.

Der bestemmes en spilletid, og når tiden er gået, er den spiller Matador, som har størst formue.

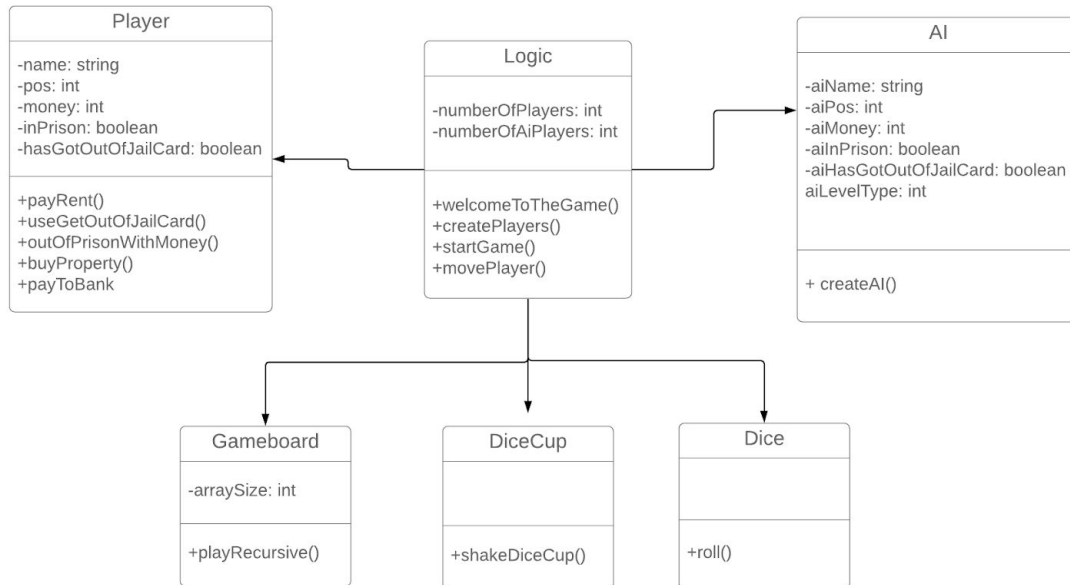
15.2 Appendix 2 - Use case diagram (early version)



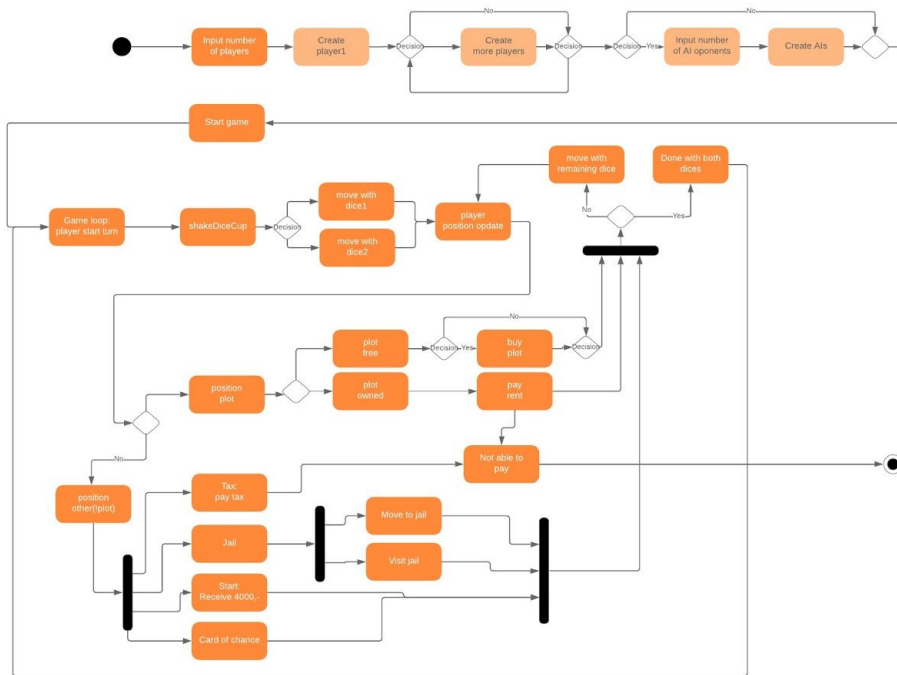
15.3 Appendix 3 - Class diagram (early version)



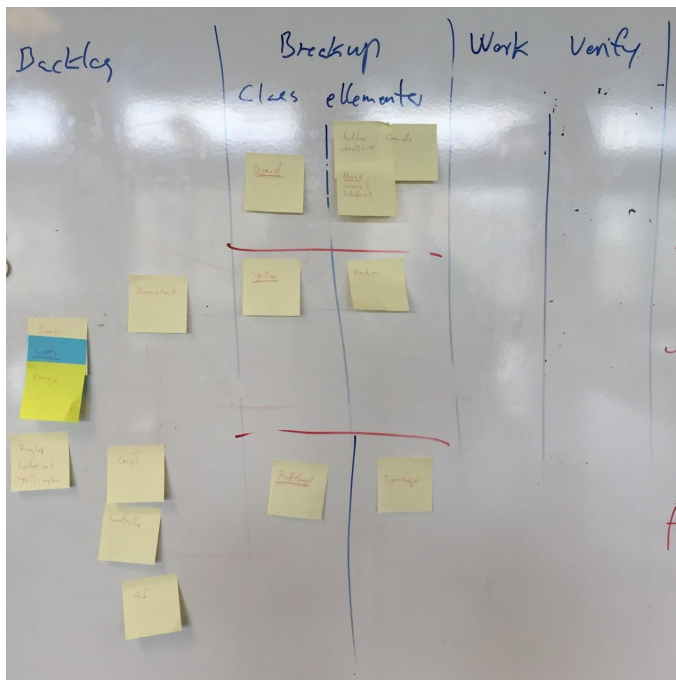
15.4 Appendix 4 - Class diagram (early version)



15.5 Appendix 5 - Activity diagram (early version)



15.6 Appendix 6 - Kanban board (early version)



15.7 Appendix 7 - Bugs

presentbuyhouseoptions presents itself all the time
even if you can't buy a house on any property

presentsellhouseoptions don't present the option again
after no match was found

when buying a property it prints out:
This property is now owned by: null
Buying a brewery works tho
Maybe it's just PropertyFields that don't work,
or maybe it's just the first buy that don't work

total value of player not updating correctly
or not being updated:
This is Mads' total value: 0
should be 30k always during first round 34k during 2nd
unless you hit tax fields or pay rent ect

When selling a property so an sout to confirm
right now nothing is being printed making it hard
to see if its actually sold and what it sold for
selling property is just bad or no sout weird

when buying something, also say the price of what
you are buying

would u like sell a property:
may have infinite loop somewhere when you don't have
any properties to sell

You can see what you rolled when you are in prison
making it easy to pick a choice to pay or roll

playermoney maybe not updating correctly sometimes

Printing amount of cards left in the deck

A pic we took from an error with a card or something of that sort



```

-----M's tur -----
Din 1. terning slog: 1
Din 2. terning slog: 1
Ialt giver det: 2
M Du har passeret Rødovrevej
Du landede på Prøv Lykken
Running chancefield landedOn()
You landed on a chance field! Draw a card.
33
Ryk brikken frem til det nærmeste rederi og betal ejeren to gange den leje, han ellers er berettiget til. Hvis selskabet ikke ejes af nogen kan De købe
32
Exception in thread "Thread-4" java.lang.NullPointerException
    at ChanceField.moveToFerry(ChanceField.java:42)
    at ChanceField.landedOn(ChanceField.java:190)
    at Logic.playerTurn(Logic.java:217)
    at Logic.run(Logic.java:101)
    at java.base/java.lang.Thread.run(Thread.java:835)

```

FerryField LandedOn() rent price cal is wrong - it always when landed on it cal the new rentprice to be *2 per ferry the person who owns the ferry got, which scales infinitely - fixes by using if numberofferrys == 1 pay this else if numberofferrys == 2 pay that.

Logic i Present Buy House Option:

I while loopet bør der være et if() statement der tjekker hvis propertyPairs == false
Hvis dette passer skal nedenstående printout ikke dukke op men hoppe ud af while loopet i stedet, med et printout "De har ikke nogen grunde De kan købe huse på"

Her er en liste af grunde De kan købe huse på:

[]