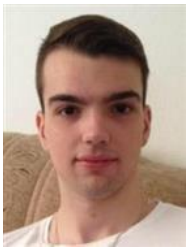
Rapport CDIO final

Mikkel Leth s153073

Sebastian Hjorth s153255



Senad Begovic s153349

Thomas Madsen s154174

Rasmus Olsen s153953

Nicki Rasmussen s153448

Dato for aflevering: 18.01.2015

Institut: Danmarks Tekniske Universitet

Opgavetype: CDIO projekt

Fag/Retning: 02312+02313+02315

Vejledere: Ronnie Dalsgaard, Stig Høgh, Jacob Nordfalk og Henrik Tange

Timetable

Timetable	Ver. 2015-17-09						
Date:	Member:	Design:	Impl.:	Test:	Doc.:	Other:	Total:
2015-17-09	Senad	20	16	4	8		48
2015-17-09	Rasmus	10	22	3	8		43
2015-17-09	Mikkel	12	28	2	8		50
2015-17-09	Thomas	12	24	2	8		46
2015-17-09	Sebastian	10	32	2	8		52
2015-17-09	Nicki	10	25	4	8		47

Index

Summery

The description of this task was very vague. We had to make a Monopoly game. It was up to ourselves to decide which features were important to make our game as good as possible with the limited time. This was the first thing we did in the project, brainstorm for ideas and features and afterwards prioritizing what was important to get the game playable. Once we had a stable build, we could later add new features to the game. We made models and diagrams to visualize what the program would look like and we came to realize that we had to redesign some of our old code. We based the game off our third CDIO assignment, but we wanted a strong foundation for the game and had to rethink some of our previous work. An example of this are our slots classes that had to be split up because they broke “The Three Layer structure”. We then started to redesign our project to follow it by making models that followed the rule. When we finished our analysis and design models and diagrams, we started to code. As we accomplished more and more requirements, the project had some changes undergoing the coding, to suit the new requirements. Development followed our MoSCoW lists, prioritizing the most important things first and then later on we started adding the optional features. We made sure that everything worked by testing the code we made, and made sure that it all worked together in the end by making a lot of JUnit tests and running the program with fixed dice to achieve the correct test results.

Introduction

Course goal

We are a group of students at the Technical University of Denmark. Who have been given the task to make a new game for IOOuterActive using our knowledge gained from the courses: “Introductory Programming”, “Developing methods for IT-systems” and “Version control”, as well as the previous CDIO assignments. With this at mind, we are going to implement the four terms (CDIO) to this assignment in the best way possible to satisfy our costumer. We will use our skills to analyze, design, implement and test our system, so it will fit the demands and requirements set down by the customer, which will be specified in the section below.

Theory

Why java

Writing the program in Java was not only a written requirement, but a good way to fulfill the need for the program to run on DTU various data bars. Java is a nice fit for the task since we don’t have to worry about memory management and its object oriented structure makes it fit for a task of this size.

XML

XML is a tree-like markup language, which easily can be set up and parsed by Java. We chose this because the parsing already is supported, so all we had to do was to design the xml structure and write the code for which nodes to get. XML is also very human readable which also were a plus.

Term definitions

CDIO

The 4 design principles: Conceive — Design — Implement — Operate

XML

Extensible markup language

Main section

Requirements and priority

The customer asked us to design and create a danish monopoly game, and make it easy to change the language of the game. The customer prefers to have a working game with less features over a game with a lot of features but doesn't work properly.

The customer also told us that we could make changes in the rules of the danish monopoly game by little, if it suited our program better, we just had to write our changes down and mention them in in the assignment, which will be shown down below.

When starting on the project we knew we had to get control of the requirements. We started to take a look into the rules of the game. From those we took heavy inspiration and it gave us a great idea of what we would need in the game. The rules almost gave us a list of requirements and we just had to add a few of our own to finish the requirements. "The Table of Requirements", shows the requirements that we found to be fundamental for the game.

Functional requirements	**Non-functional requirements**
Game for 2-6 player	Can be played at "DTU's data bars"
The roll with two dice	The game can run without any significant delays
The players move on a board	The system is written in java
40 different fields	Should be easy to translate (Localization)
A text that describes a certain fields effect has to be shown	The dice should easily be changeable
The players start with 30.000 points/money	
The last player standing is the winner	
Different effects on each fields	
Be able to roll a dice and continue from the field on your next turn	
You are moving in a circle around the board	
If you rolled a double, the player should get an extra turn	
If a player rolls 3 doubles in a row, then he should go to jail	
If a plyer lands on "go to jail"-field, the player goes to jail	
The ability to purchase 1-4 houses and upgrade to a hotel	
When a player is in jail, he should have choice of either pay X amount of money or roll 3 times to get a double to get out. If he obtained a "get out of jail"-card, he should be able use that as well	
Chance cards: - Gain X amount of money - Move to X - Get out of jail free card - Pay X amount of money - All other players give X amount of money to one player	
All money pay towards bills and the other negative effects should be placed on the field "helle", and the player should receive the cache when landing there.	
The player is awarded with X amount of money	

when he/she passes the starting field	
Players should be able to pawn and unpawn their fields	
Players should be able to trade properties with each other	

The requirements helped us organize and prioritize to find out which features should be implemented. We've also added non-functional requirements from previous CDIO projects, since they are the same as the previous ones.

As a game of many features we understood, that it would be important to prioritize the different aspects of the game. We decided to make this priority list, which includes the different rules and elements of the monopoly game - in order to finish the program within the time limit that was set by our customers. This way we can see how much we might be able to fit into the program without it claiming a cost on the functionality of the program, since we want a fully functional program with least amount of bugs and errors, and with as many of the rules and elements of the original game as possible.

Priority list:

- 40 fields with correct names and prices
- 2-6 players that can move around
- 2 dice
- It is possible to buy a field
- You can buy houses and hotels(requires 4 houses) on your fields
- Chance cards
- You get an extra roll if you roll two of a kind(You get to have the "field effect" when you land on a field after rolling two of kind before your next roll. For an example if you land on chance card field after rolling two of a kind, you get to take chance card before moving onwards).
- You go to prison, if you roll two of a kind 3 times in a row or land on "go to prison"
- You can get out of prison if you roll two of a kind (you get to try 3 times) or you pay a fine of X amount of money or you can use the chance card "Getoutofprisoncard".
- All taxes and fines goes to "Helle" also known as parking lot, if a player lands on this field, he will get the amount of money belonging to "Helle"
- Pawn
- Ability to buy fields from other players
- Extend the building of houses and hotels to only be possible if all the properties within the same field group is owned
- If any territories are not bought when a player lands on it, it is put on auction for any player to buy

Some of the requirements are pretty equal in priority, but we just had to make a number list in order to get a better overview. For an example, the first 4 priorities on the list is all a very big part of the game, and can't be avoided to not get implemented, but still we listed them in an order of numbers to get a better overview, even though some of the priorities are equal to each other.

Later on we used MoSCoW, which is a prioritization technique that basically stands for Must have, Should have, Could have and Would like to have, but won't get. We thought it would be easier to place our requirements in these four categories instead of making a numeric list. Our MoSCoW list is following:

Must have:

40 fields with correct names and prices
2-6 players that can move around
2 dice
It is possible to buy a field
You can buy houses and hotels(requires 4 houses) on your fields.
Chance cards
You get an extra roll if you roll two of a kind
You go to prison, if you roll two of a kind 3 times in a row or land on “go to prison”
You can get out of prison if you roll two of a kind (you get to try 3 times) or you can use your Getoutofprisoncard
All taxes and fines goes to “Helle”, if a player lands on this field, he will get the amount of money belonging to “Helle”
Pawn

Should have:

Ability to buy fields from other players
Extend the building of houses and hotels to only be possible if all the properties within the same field group is owned
When a player bankruptcy the houses / hotels should be removed from his fields.

Nice to have:

If any territories are not bought when a player lands on it, it is put on auction for any player to buy.
Show the amount of money on the parking lot field, and keep it updated all the times
Players can choose their own car color
Players should be able to buy themselves out of the jail

As mentioned in the beginning of the requirement section we had to eliminate some of the rules, in order to manage our program to be a fully functional monopoly game. The only thing we did not manage to implement is the last rule on our list of priorities; The rule is that there should be an auction on the territories that wouldn’t be bought by a player who landed on a territory field. We decided not to go for this one since it was on the bottom of the list, and we didn’t feet we had enough time to implement it and make it work properly. Besides that we managed to implement all other requirements, and made sure they worked by testing them, but it will be mentioned in the test section later on.

A slight change in a couple of rules has also been made. When you have gotten a chance card and used it, it does not go to the bottom of the chance card pile to become reusable, but instead disappears from the game. You therefore would not be able to get it again unless you get the same card again, since there are some cards that are in the pile multiple times. If a player gets a chance card he/she cannot use, it is put back in the pile, so it can be used again later. Besides that, we have changed the game to be for 2-6 players instead of 3-6 players in the original game. A player building houses or hotels does not have to have one house on each field in the group before he/she can build the second house. If the player wishes to do so, he/she can build a hotel on one territory before building the first house in another territory in the same group.

Configuration

To be able to run the program itself you need to have certain other programs installed. Java in version 1.7 or newer and either MAC OS X mountain lion or newer, or have Windows 7, 8 or 10 installed. The game is

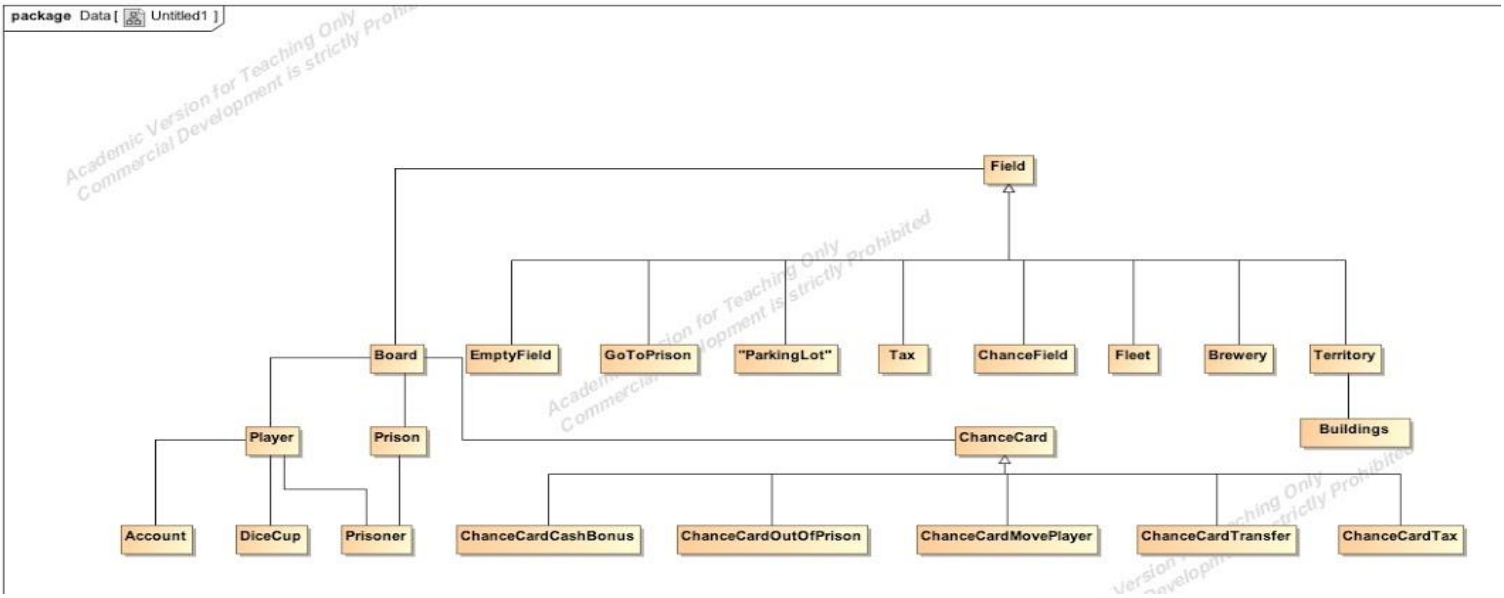
fully tested on these operating systems, furthermore it has been tested to run on one of DTU's computers, which it did without any problems. The program requires java 1.7 because this version of Java introduced the ability to “switch a string” which we utilize in several times in our program.

Guidance - How to compile, install and run the program

1. Open your browser
2. Go to this website: https://github.com/G16CDIO/CDIO4
3. Copy the clone url on the right hand side(etc: https://github.com/G16CDIO/CDIO4.git)
4. Open “Eclipse”
5. File -> Import
6. Press next
7. Git -> Projects from Git
8. Press next
9. Clone URI
10. Press next
11. Paste the URI in the field
12. Press next
13. Press next
14. Choose your directory to be your workspace
15. Press next
16. Press finish
17. Now open the project in eclipse
18. Open the package called “Game”
19. Open the class called “Board”
20. Press “Run” (The green play button in the top)
21. The game now opens in a new window

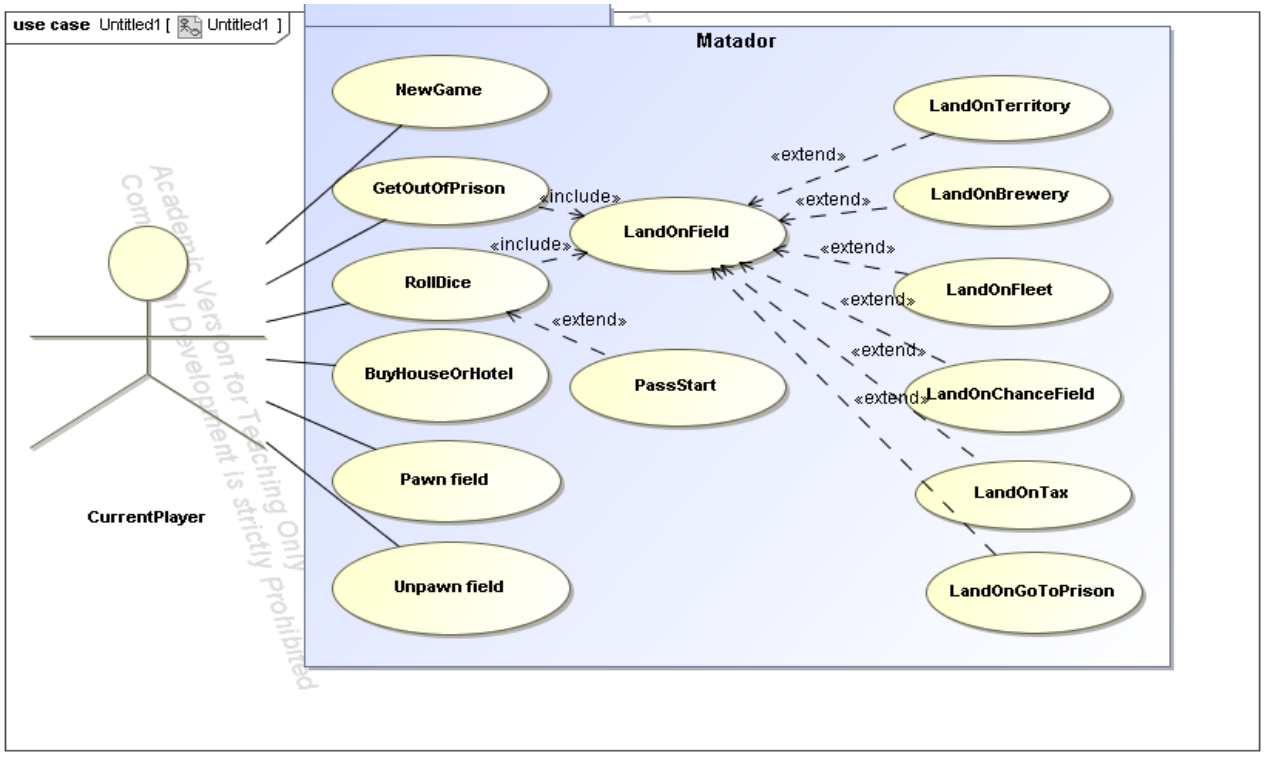
Analyze and design

After getting the requirements and prioritizing done we had to analyze the game. We had to find out, which physical object the game had, so we decided to make a domain model to make it easier to manage the actual game, it's object and their mutually connections. We ended up with the following model:



The purpose of this model was to get the actual things of the game, the features you can see or touch. We found out that the board was the essence of the game, all the interaction happens through it. The board had different kind of fields, with different duties. One of the fields was the territories, one that a player could own and buy buildings to. In addition we had some chance cards, and as it was with the fields, the chance card could also be split up into smaller groups. In this model we have chosen to show inheritance. Because all the fields have some identical attributes. They can therefore inherit from field, and have their own distinctive attributes and operation in their own class. The same is applicable for the chance cards. Besides the fields and the cards there was a player with an account, a prison, with possible prisoners and a dice cup. And with that we got control of the core structure of the game.

After we found the physical objects of the game and their relation, we started to take a look at another central object of the game. It was the role of the player/user. We asked ourselves what a user should be able to do. First of all, the user should be able to start the game, by entering number of players playing the game and their names. After that the user should have a couple of choices of how to start his turn. With that in mind he should be able to either roll the dice, buy a house (not necessarily in the first turn of course), pawn a field as well as unpawn it and even bail out a prison if the player is in prison. All these options should be available in the beginning of each turn, but only if the user aren't in prison, in which case the bail out of prison option and roll (to get out of prison) will present themselves. We also needed to think of what will happen when the user actually rolls the dice, and what fields he could land on. By rolling the dice he would land on a field, which would give the player different options or outcomes depending on the field he landed on. He might pass the start by rolling the dice, and we had to keep that in mind too. By having all this figured out, we made a use case diagram, to get a better overview of how the things would go:

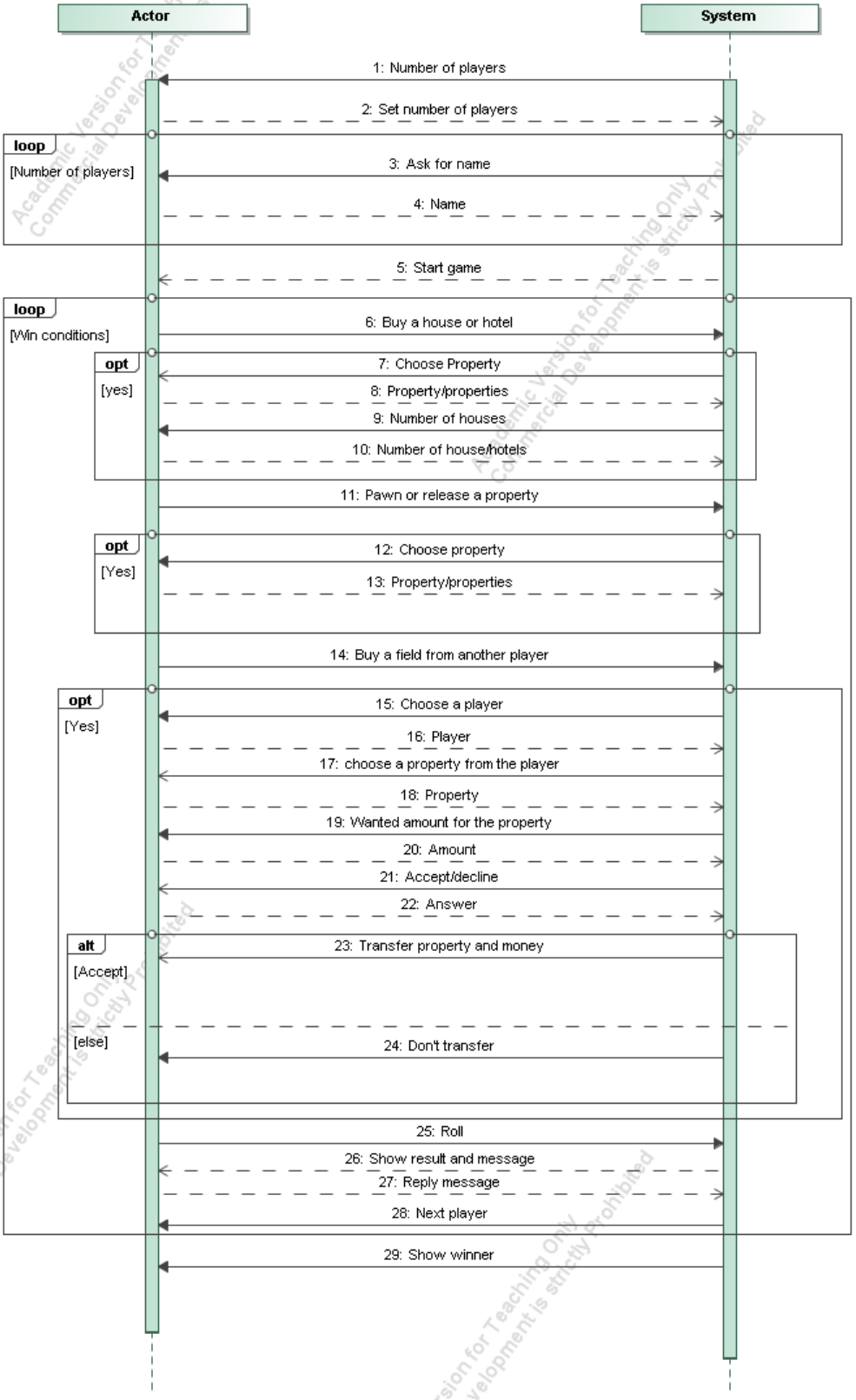


By making the use case diagram we got a better overview of how the user would interact with the program. Following that we could go into more details with our use case descriptions, making a more detailed way of describing the user's actions and what he needs to be able to do in each use case.

Down below you see one of our use cases we've worked with, that describes what happens when a user rolls the dice. With the use case description we can easily describe each use case in more detail, like; who is the primary actors, when the use cases are happening and what going to happen when the use case are initialized.

Use case description: "Roll Dice"
ID: X
Brief description: The player rolls the dice. If he gets a pair he gets another roll. If he gets 3 pairs in a row he goes to prison
Primary actors: Player
Secondary actors: None
Preconditions: It's the players turn and the player is not in prison
Main flow: 1. Player is asked to roll dice 2. Player rolls dice 3. Move player 4. Use case: LandOnField 5. If diceroll is a pair a. Give player another roll b. Player rolls dice c. Move player d. Use case: LandOnField i. If diceroll is a pair ii. Move player to prison
Post conditions: Players position has been updated
Alternative flows: None

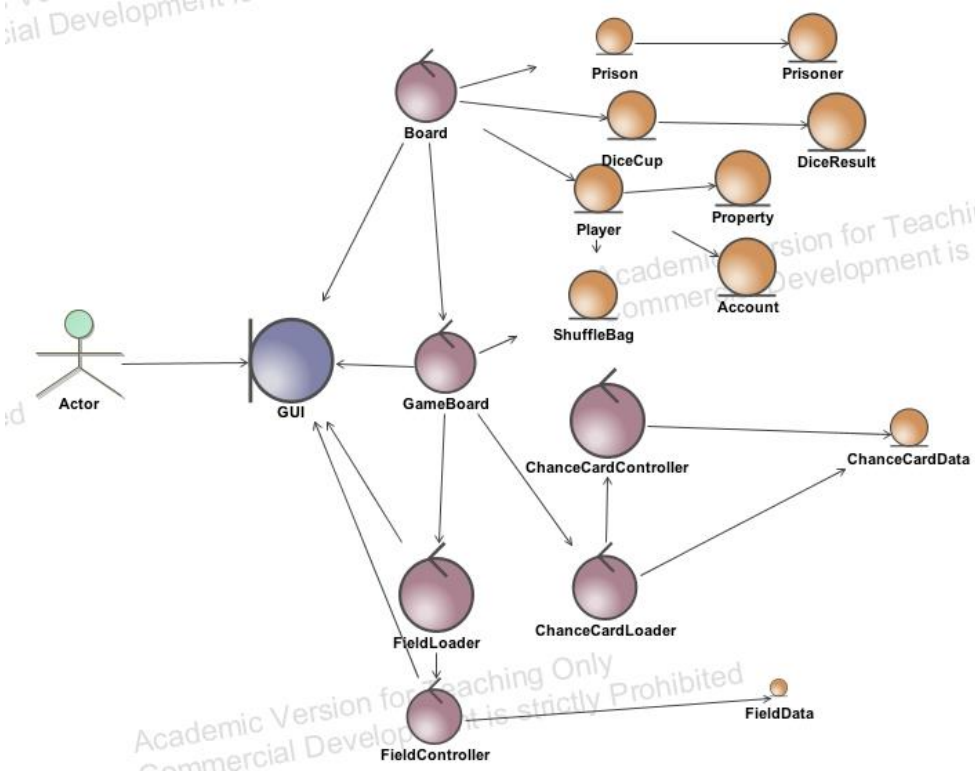
After completing the use cases and their descriptions, we could now easily make a system sequence diagram (SSD), which shows the interaction between the user and the system. In our SSD model, we are showing how the beginning of the game progresses; by first entering number of players, then a name for each player, after that the game starts, and you have those different options to begin with as we mentioned before. When you finally roll and finishes your turn, then it's the next player's turn and he goes through the same progress, except the player names and setup players. This continues until one of the players win the game, by being the last one with money. The SSD also helps describing a bit more of how the program is going to work, so we get a better understanding for how to design our program and later on it would be useful for our design sequence diagrams. To the side you can see our SSD diagram:



Having the interaction between the system and the player in place, we proceeded on to the actual classes in the program and how they connect with each other. We started to think about which classes should be boundaries, controllers and entities. The boundary of course is the GUI we have been given. It takes care of

the interaction with the user. We thought of the controller classes as definitely being the Board class. We now started to think of how we wanted to write the code in the smartest possible way. We figured out, that it would be smart to have a controller class called GameBoard. This should have the responsibility of loading the board with the correct fields and chance cards. To load the fields and the cards we wanted to use a FieldLoader class and a ChanceCardLoader class both being controllers. These should load all the required information from a XML file we would write later.
Besides that we wanted to fulfill the three layer rule. We therefore decided to split all the field classes and all the chance card classes into a controller class and a data class.
The controller classes will of course be controllers and the data classes are entities. The other entities are player, account, prison, inmate, shufflebag, property, dicecup and diceresult.

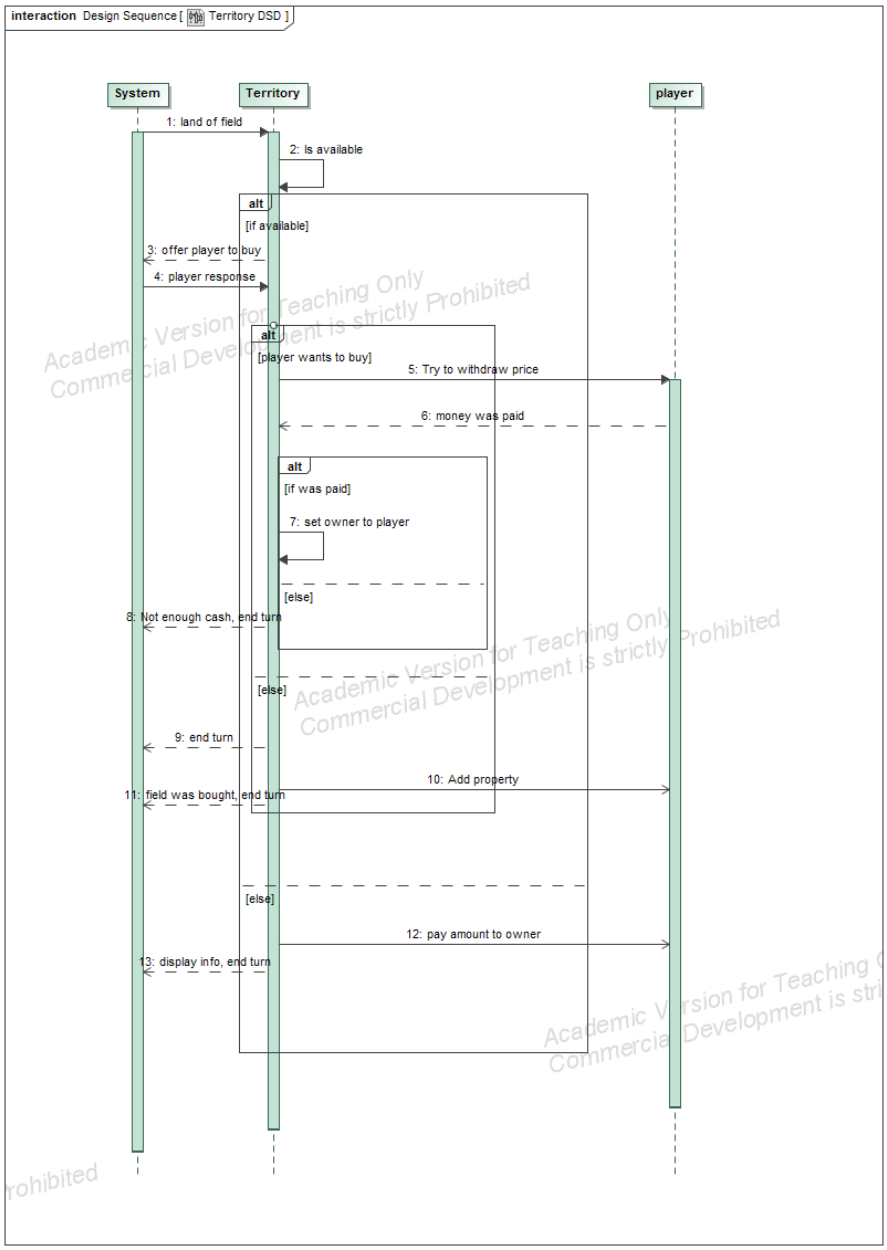
The BCE-model can be seen here:



We are aware that we've since this added an additional controller class: PlayerCreater. This was a late implementation and edit of the program, since we discovered that Board simply had too much responsibility, so we decided to take some of the logic regarding player creation and give it a class of its own.

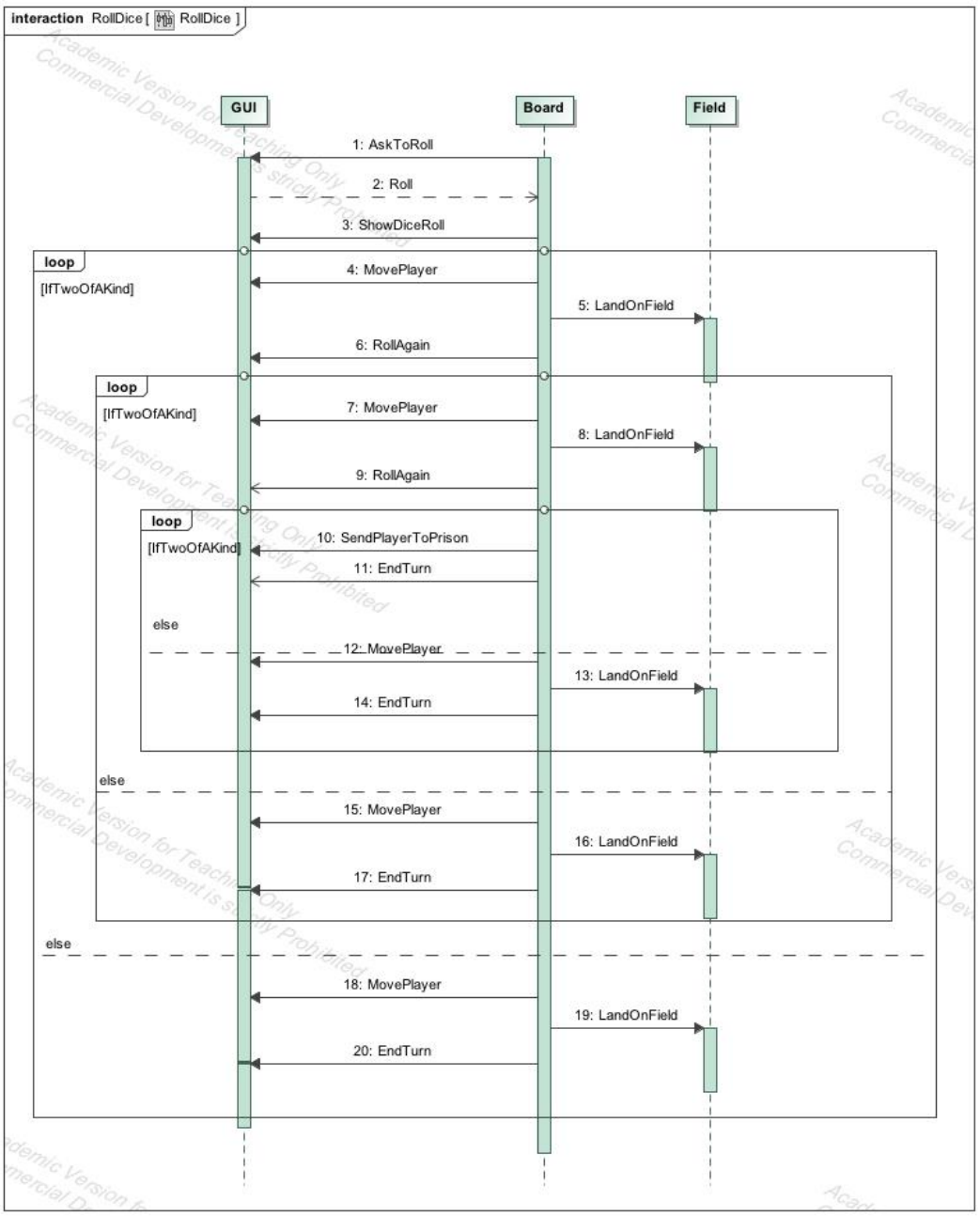
What we wanted to do with the different field controller classes was to make them inherit from FieldController. It is not shown in this model, but they way it works is that all the different controllers are dependent of the FieldController and have influence on the FieldLoader. There are 3 controllers that doesn't inherit directly from the FieldController, and therefore does not depend on those. It is the Territory-, Brewery- and FleetController, they instead inherits from the OwnableController, which inherits from FieldController. They still have influence on the FieldLoader. All the data classes give data to the corresponding controller classes and inherits from the FieldData. The exact same thing is replicable for the chance card classes. They just use a ChanceCardLoader instead. From here we moved on to our design sequence diagrams. We did this to get a clear idea and make a sketch of what happens in the system when a player interacts with the program.

DSD LandOnTerritory



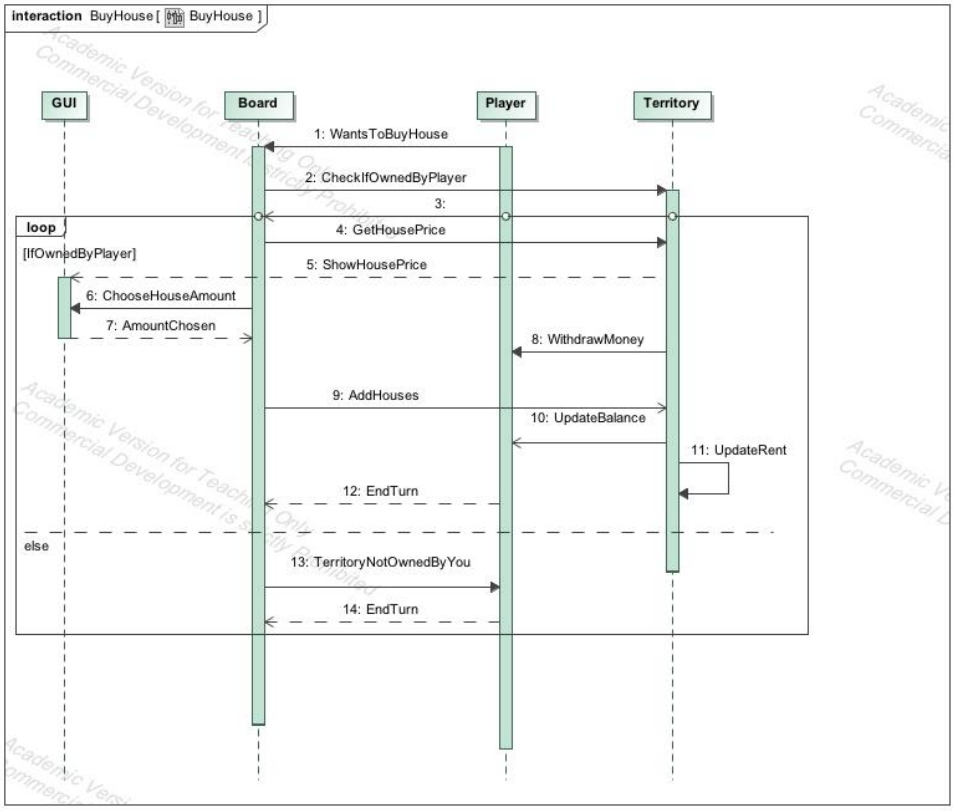
LandOnTerritory describes what happens when the player lands on a territory. The field first checks if anyone owns it. If not, the player will be offered to buy the territory. If he refuses the turn will end. If he accepts, money will be attempted to be transferred. Territory will then verify that money was transferred. If it failed the turn will end. Otherwise the player will be given the territory and the turn ends. If the territory was already owned the player will have to pay the desired amount to the owner.

DSD “RollDice”



The sequence begins with the player being prompted to roll the dice. The player will be moved to the next field and the turn will end after the LandOnField method is finished. If the result was a pair the player will be given another roll and the same thing will happen. If a pair is rolled three times the player will be moved to prison and the turn will end.

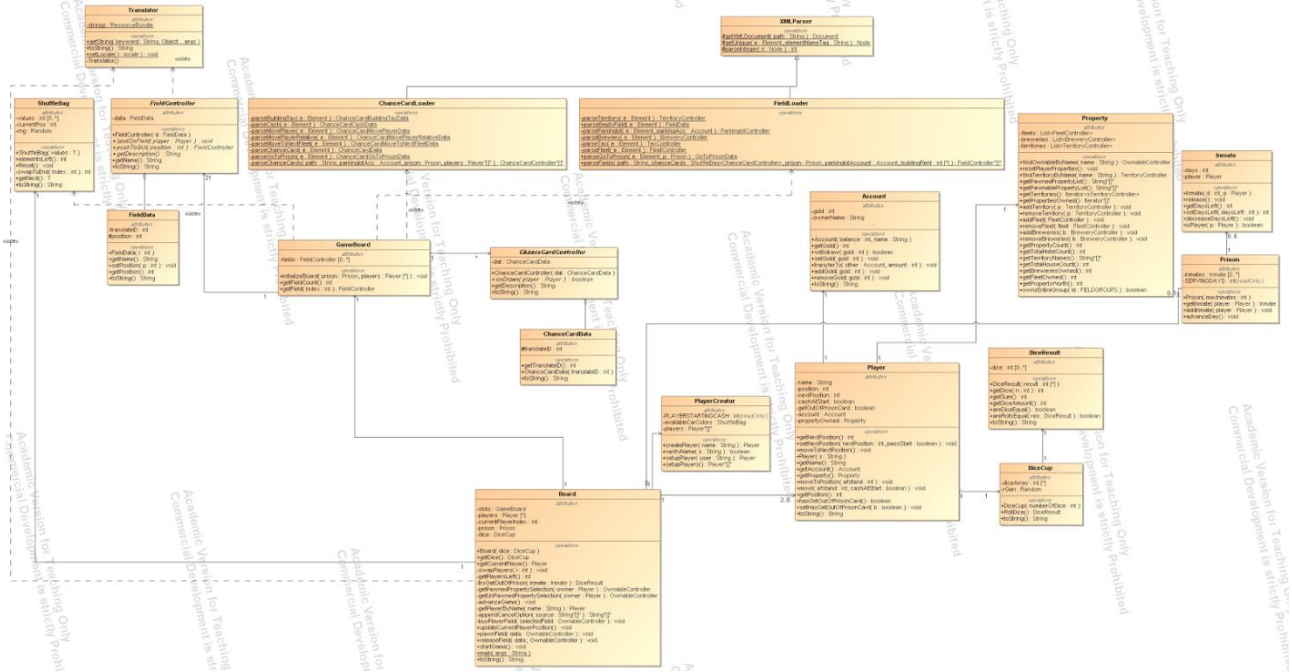
DSD “BuyHouse”



The sequence begins when the player chooses to buy a house through the dropdown menu on the gameboard. The board will confirm that the player is the owner of the territory. If not, the player will not be able to continue his purchase. If he is the owner, the price of a house will be fetched from the territory and displayed to the player. Money will be drawn from the players account and a house will be added to the player’s territory. The rent on the territory will be updated and the turn ends. This diagram deviates a bit from the actual code. Late in the process we found time to implement that a players has to own all territories in one group before he/her can buy a house or hotel. This has not been documented in this model.

The design sequence diagrams for “NewGame”, “LandOnTax”, “LandOnBrewery”, “GetOutOfPrison”, “LandOnFleet” and “LandOnChanceField” can be found in the Appendix.

Design part "The whole picture"



Implementation

Three-layer-architecture

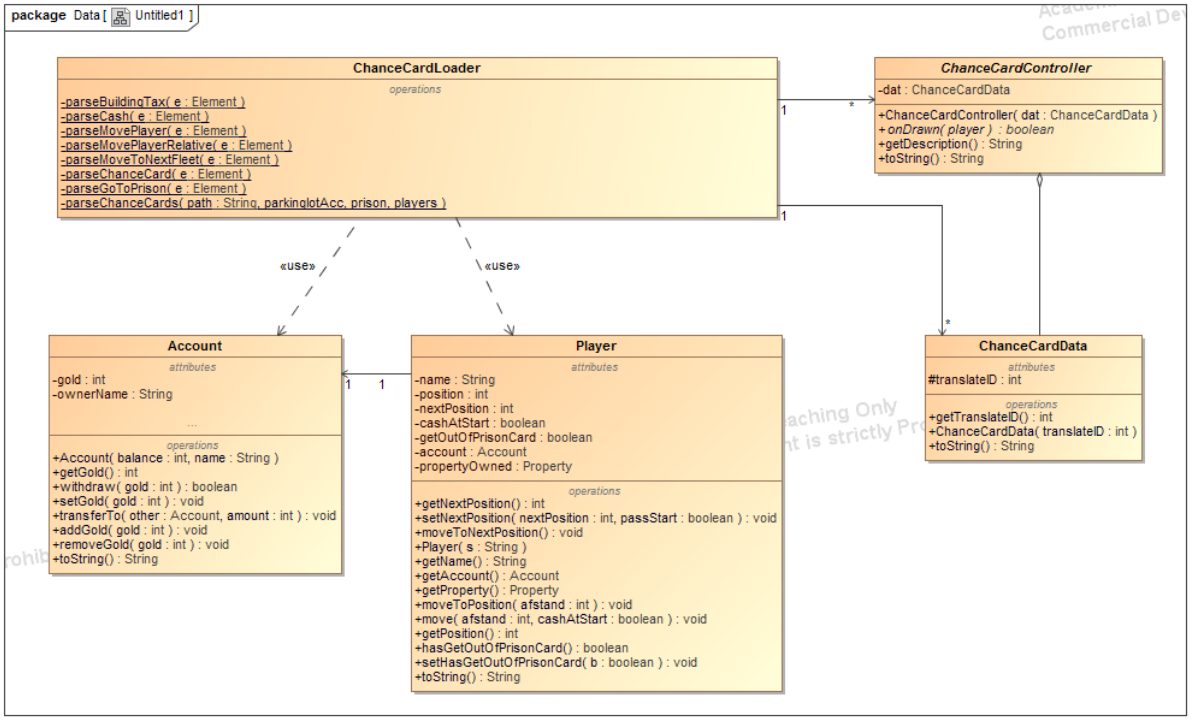
We chose to structure our code after the three layer model. We did this to simplify the process of making the game saveable, should this ever be needed. It will also simplify the process of switching to another GUI, should we ever need to. The extra time it took us to split the code into entities and controllers are easily justified by the time saved, should the requirements ever change. Because of our choice of structure most of the classes are split up in an entity and a controller part where the entity contains data for the controller to work upon.

This structure also helped us develop the xml files as we could just check the entity classes to quickly check what kind of data each field was using.

GRASP:

GRASP (General Responsibility Assignment Software Patterns) describes how best to divide the responsibility between the parts of the program. In our program we utilize quite a lot of small controller classes which acts as our information experts and creators. Dividing the responsibility out on a lot of smaller classes gives us a lower coupling in the program overall, which in turn makes changing a feature or removing it completely a lot easier. Low coupling insures that we can take parts of the program and change according to demand without causing major problems in other parts of the program itself, this also maintains the high cohesion in the program. So by following the principles of GRASP we get a more manageable program that is easier to interact with and change to the customers liking.

Even though we have strived for low coupling for the program in general, there are still a place we could not achieve such a thing, namely the chance cards. This is because they influence quite a lot of different



elements to function, e.g a list of players is needed for the cards where it's a player's birthday, since everyone needs to pay him a certain amount. Also all taxes are transferred to the parking lot, hence does it need to share an account object with the field. The high coupling can be seen in the following diagram:

Board

Board has the responsibility of making the game progress. It controls which turn it is and it's responsible for turn progression, such as asking what the player wants to do at the start of his turn. The class contains a startGame() method which starts the process of setting up the game. One could say that Board has too much responsibility, but really all it does is to ask the player what he wants to do within his turn and then makes the appropriate calls to make it happen. This is also the reason as to why Board is the class which contains startGame(). It needs to know the most about the other classes in order to make the right calls. The Board class takes a DicePair object for its constructor, which can be used for testing purposes. We also tried to make it take in a player array, but due to restrictions from the GUI(the fields has to be registered before the players). It turned out to be too much work as the GameBoard class(which loads the fields) need to share a prison object and an array of players with Board, where the latter would be the biggest problem to workaround. Because of the restrictions from the GUI on not being able to enter player names before the board is created, we had to make a workaround to pass the player's to the ChanceCardLoader:

```
Player[] chanceCardPlayers = new Player[6];
slots.initializeBoard(prison,  chanceCardPlayers);
players = playerFactory.setupPlayers();
//Workaround for players needed before the board is created, but player names can only be gotten after.
for(int i=0;i<players.length;++i)
{
    chanceCardPlayers[i] = players[i];
}
```

What's done here is that we pass a reference to an array of player's to initializeBoard, which then passes it on to the chance cards, which makes use of the player array when someone draws the card. Since it doesn't need the player's right away the array is first populated when the board has been properly set up and the player names has been entered. We can't just pass our player object, as the array reference is first assigned

after the call to setupPlayers(); Our approach, however, works by giving a known reference that points to an array full of null-references and then setting the references to actual objects later on.

Board contains the method: advanceGame(). It might seem confusing at first, but really all it does is to ask the player if they want to roll their dice, upgrade, pawn or pay of their property at the beginning of their turn, if they're not in prison. To do this it has to set up a lot of lists and keep track of selections(which is a bit tricky since the GUI only returns the selected string, which then has to be found within an array in order to manipulate correct field). When handling selections, then the game is run in an infinite while(true) loop, until the player chose to roll the dice, which then calls break and the execution moves onto the next loop. The next loop makes the player move around the board and ends his turn if he runs out of rolls or goes to prison. This loop also checks if the player goes bankrupt and removes him from the game if that happens.

```
private void swapPlayers()
{

    do
    {
        if(++currentPlayerIndex==players.length)
        {
            prison.advanceDay();
            currentPlayerIndex = 0;
        }
    }while(players[currentPlayerIndex]==null);
}
```

The way we keep track of the player's turn is handled by the method on the left. We keep track of the current player by using an index. Once said player has ended his turn we then check if the next index exceeds the last element of the array, if it does then we decrease the time in prison for all prisoners by one and starts over from the first player. Notice how it is embedded in a do while loop. This is because, when a player leaves the game we set his index to null. Therefore we have to skip the indexes at which there no longer are a player object. A way to avoid this would have been to copy the players into a new, shorter array every time a player lost the game, but we chose the simple solution and just used the null approach. The last thing to notice is that we use Thread.sleep when a player is trying to bail out of prison by rolling the dice. This is done to halt the execution for 200 ms for a dramatic effect where the dice are rolled 3 times for you.

Prison

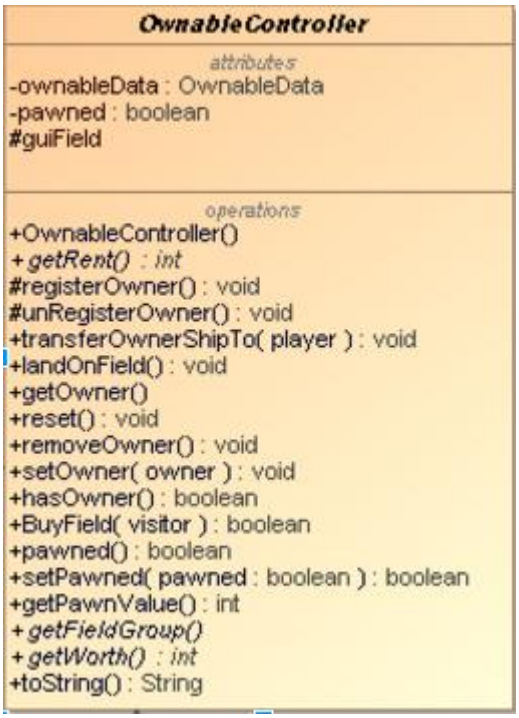
We decided to make a prison class instead of giving the player class a variable that tracks the player's state of imprisonment. The reason for making a new class was that otherwise the player would also have needed a method to keep track of how many days the player had left in prison. If the player was to have the prison methods too we thought that it would give too much responsibility to the player and therefore decided to make a Prison class and an Inmate class instead. Prison contains an array of Inmates and every Inmate object represents a player in prison. In the beginning of a player's turn Prison checks if the player is imprisoned and if so it will return the Inmate object that represents the player. The Board class will then use this return statement to set the player's days left to zero, the player will then leave prison upon his next turn.This way makes it easier to reduce the amount of days left for all inmates since we just have to call the method on Prison which takes care of the logic compared to having to loop over every player.

FieldController

We created the abstract class FieldController in order to collect all the fields into one array without having to deal with what type of field it was. This was done by calling the landOnField(Player player) method which were implemented in the specialized subclasses. The method would be called on the element corresponding to the players position. The class also contains a pushToGUI(int position), since each FieldController has to create an object which contains information for the GUI, so that it knows what to display. We can't do this in the constructor because the elements has to be gathered in an array and pushed to the GUI together. pushToGUI also takes an integer because the position the field is given depends on where the field is in the array being pushed to the GUI and therefore the position is only known when the GUI fields are added to an array. We could have chosen to keep track of the position by having a

static variable declared in the FieldController array and then just increment it as the pushToGUI method was called, but that just seemed like a mess, as any method in the class would have access to it.

OwnableController



The OwnableController inherits from FieldController and specializes in the fields that the player can buy. It handles the purchasing, ownership, paying rent, pawning and unpawning of fields. It's an abstract class, reason being that the OwnableController needs to be instantiated when the game starts even though no one owns anything, and therefore subclasses are forced to implement the following methods(unless the class marked is abstract as well): getRent(), chargeRent(Player player), registerOwner() and unRegisterOwner(). getRent() is needed because the way the rent is calculated varies between the fields. We couldn't just use getRent() and then make the OwnableController charge the rent, as the user interaction differs from each subclass. We could have moved the logic from chargeRent to getRent, but that wouldn't work because getRent is also used to display the rent(without prompting the user first). register and unRegisterOwner() are used to register the field to the right array in the Property class(more on this in a bit) we did this to avoid an incrementally amount of instanceof calls to check which list the object belonged to. registerOwner basically adds the current object to the current owners property register and is called whenever the owner of the field is changed.

Ownable also handles FIELDGROUPS. Since one of the rules in the game states that you must own all territories in a group to build on them, we needed something to classify in the different groups. Fields like breweries and fleets are also dependent on the grouping system since they get a bonus when you own several of them, FIELDGROUPS made this easy too.

Property

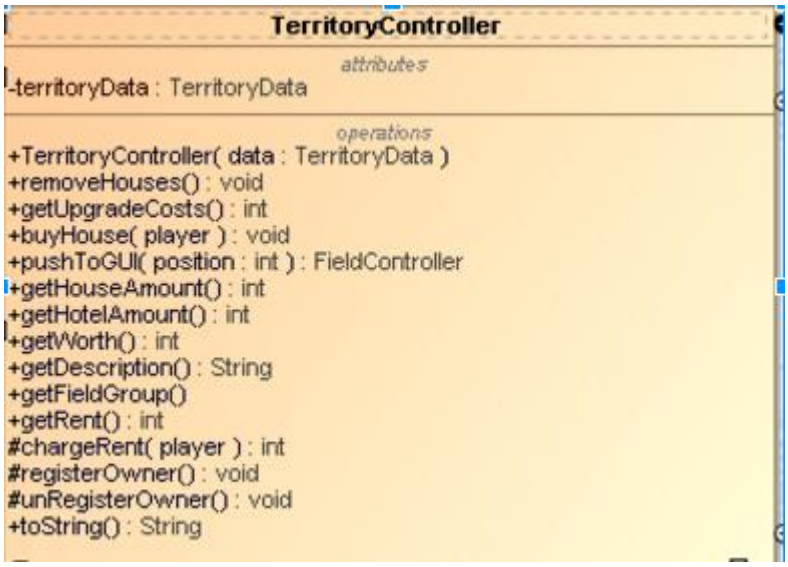
The property class functions as a register for all the properties owned by the player. To begin with we had designed it to store all the properties as OwnableControllers, but we quickly realized that we had to cast forth and back from Ownable as only the TerritoryController were able to have houses added. We also had to perform runtime checks to check the amount of each type in the array, which is used to calculate the rent of fleet and brewery. Therefore we decided to keep track of each type of property in their own list. We could now get the TerritoryControllers and counts without performing run-time checks, while still being able to obtain a list of OwnableControllers by merging the 3 list together. This is used when getting a list of pawned / unpawned properties.

The reason why we use List in this case over a conventional array is that the player can end up with anywhere between none and all the fields on the board. We don't know the exact amount of fields, which the player is going to own. Therefore we went with Java's built-in List class because it extends itself when there are no more spaces in the array and a new element is added to the list. This way we don't have to deal with the problems that might occur if we rolled with

our own implementation. The List also have the benefit of the remove() method, which allows us to remove elements from anywhere on the list. This is very useful when we remove the property from one player's properties to another's when trading properties.

TerritoryController

The TerritoryController class inherits from the OwnableController class. The main responsibility of the TerritoryController is handling the purchases of houses in addition to keeping track of the amount of houses and who the owner is. getUpgradeCost(), is a method that returns the cost of building an additional house on that a specific territory. The information needed is retrieved from TerritoryData class and returned by the method getUpgradeCost(). The buyHouse() method is used for buying a house on a territory. The method checks if you are the owner of the selected territory you wanted to buy a house on, if you are the owner, the method will check if there are less than 5 houses (since 5 houses are equal to a hotel), after which it will check if the player can afford the house by using the getUpgradeCost() method, at last it will check if it's your 5th house, if so, it will convert it to a hotel. If any of these checks fail, the player will receive an error message. To keep track on the houses/hotel, we've added two methods, the first one is getHouseAmount(), which will return a number between 0 and 4, depending on how many houses are built on the territory. It will return 0 even if you have 5 houses, because the 5th house will count as a hotel, and therefore you have to use the second method getHouseAmount(). Those two methods are actually accessing the same variable, as the 5th house being a hotel is just a visual thing, therefore the first method checks if the house count is within 0 and 5, whereas the latter returns one if the house count value is 5. Next we have getWorth() method, which returns a sum of the following things: price of the territory, amount of houses multiplied with the house cost. We only use this method to when "matador legatet" gets drawn and asks how much a player has in total, and also use it when a player has won the game. To sum it up with the houses, we've the last method removeHouses(), which removes all houses and hotels on the territory. This function is only used when a player is out of the game and has to remove his owned territories.



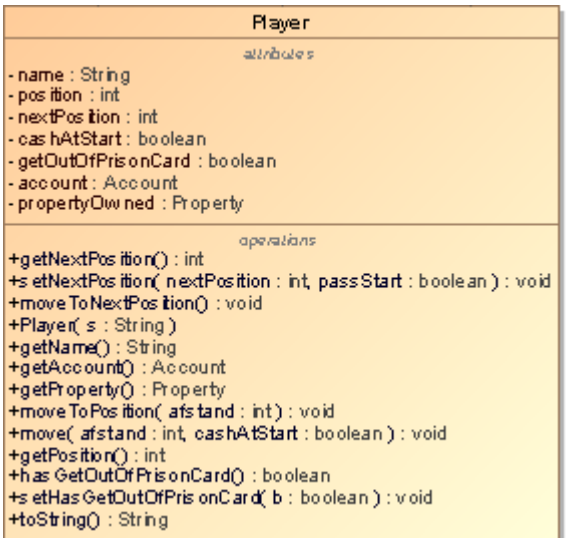
PlayerCreater

To start off the game, there has to be created a certain amount of players, giving each a car as well as making sure that each player object is unique. Firstly the user is asked for the amount of player objects that has to be created in the form of players participating. setupPlayers() are making sure that the number of players are within the limits between 2 and 6. With the chosen amount, a loop is runned for each player to enter their wanted names. To make sure the players are unique, setupPlayer() is called to makes sure (through verifyName()) that the players are using correct names, which aren't containing spaces nor are empty, as well as their names won't be the same as those of the other previously entered names or longer than 15 characters. If the names are fine, then each player will be put into the array of players, which are used to keep track of whose turn it is as well as who is still in the game.

PlayerCreater runs the method createPlayer() which function is to give players their own unique car-color. This method are basically going to setup the player on the GUI, with starting cash, the chosen name and random colored car.

Player

When a game is started the player is asked to give a number of players and names of each player. What happens here in the code is the player objects being created. The player constructor takes a name only, but will also create an account for the player that keeps track of his economic balance. One of the great responsibilities of this class is keeping track of the players position on the board, this also includes updating the position when the player is supposed to move. The class contains four different methods for moving the player to a new position: setNextPosition(), moveToNextPosition(), moveToPosition() and move().



moveToPosition() is a method that moves the player the given distance. It also checks if the player passes start, in which case the player receives 4000 they deserve.

moveToNextPosition() has a variable, nextPosition, that shows where the player is moving next. The moveToPosition() method is then called and the nextPosition is updated to the new position. The reason that we have moveToNextPosition() is we needed a way of updating the players coordinates on the gameboard itself. This means we use setNextPosition() to set the variable nextPosition to a new value. It also updates the cashAtStart variable which tells whether or not the player has passed start.

move() takes the values that are set by setNextPosition() and uses them to move the player there. The reason why the player class keeps track of two positions is that everytime Board has called landOnField for the field on the current player's location, then the player's location can be altered. By keeping track of the current location and the location at which the player can move to, then the board class can detect that the player should move to a new location when landOnField returns and then the board class knows to call the new location's landOnField as well.

The player object also remembers if the player has a getOutOfPrisonCard which it receives from a chance card. Each player also has a property counter which keeps track of what fields the player owns and if there are any houses or hotels built there.

Translator

The translator class is being used for easily translate the game into other languages. It uses Java's MessageBundle to turn a file into a hashtable, which we then can use keys to identify the different strings. We chose not to add field names and descriptions to our fields.xml file as it would make the program more difficult to translate, as we would be mixing the data. It's a lot cleaner just to have one file with all the strings needed by the program. Instead the field.xml file contains a translateID which corresponds to a key in the MessageBundle of that type. This is done by appending translateID to the string token, such like: SLOTDSC+translateID.

The strings returned by the translator is rarely stored (only when pushed to the board), this was done to be able to change language at real-time rather than having to restart in order to change the language.

The ChanceCard classes

The chance card classes were a new implementation compared to the previous CDIO assignment. With our new knowledge about the three layer rule we had to come up with an solution, that fulfilled the rule and simultaneously would make the code effective. We designed it like we did with our fields. First we came up with the different types of chance cards. We had to make the classes as general as possible in order to keep the number of classes down. We ended up with the following types:

- "Bonus", when a player need to get money from the chance card.
- "Tax", when a player needs to pay a tax when drawn a chance card.
- "CashTransfer", for when a player needs to receive cash from the other players.
- "BuildingTax", when the player needs to pay a certain amount for every house and hotel they own.
- "MovePlayer", when the player moves to a given field.
- "MovePlayerRelative", when the player needs to move a certain number of fields forwards or backwards.
- "MoveToNextFleet", when a player needs to move to the next fleet.
- "GoToPrison", when a player has to be moved to jail.
- "OutOfPrison". The queen's birthday, saying the player can get out of jail, using the card.
- "MonopolyGrant". A player would be granted 40.000 under the condition that the total values the player owns would not be more than 15.000.

All these different classes that represents the different types of chance card, inherits from the superclass with the name "ChanceCard". The ChanceCard class contains all the variables and methods, that the other chance card classes have incommon.

We still had to fulfill the three layer rule and in order to that we chose to split the classes up, in a controller class and a data class. Also the super class "ChanceCard" was split up into a controller and a data class. Now data classes of the children classes inherits from the "ChanceCardData", and the controller classes of the children classes inherits from the "ChanceCardController".

The Data classes contains all the variables the program needs, a constructor and proper getters for the variables. The ChanceCardData have an integer called translateID, which is common for all the other data classes. The other data classes contains other and different variables that suits the functionality of that type of card. But we do not have all the data here, actually we only have the names for the variables. The actual data, the actual numbers we need in our program is saved in a XML file. We will write more about that later on.

The data classes is made to deliver information the controller classes. Having this in our mind we have reduced the number of classes by having one data class deliver information to a numerous of controller classes. The one who does that is the "CashData" class. It delivers data to the classes: "BonusController", "TaxController", "CashTransferController" and "MatadorLegatController". These controllers need the same kind of data, but have different methods. Therefore we could melt the data classes together to one. This data class contains an integer called "money". And together with the inherited translateID from the super class, it delivers the right information to the 4 different controllers.

The controller classes have one method besides the constructor. This is the "onDrawn" method that tells the program what to do, when a player draws this type of chance card.

We have made this method abstract in the "ChanceCardController", because none of the classes have anything in common in this method.

The method returns a boolean. We have done it like this, because our rules says, that if the card can be used, it needs to be thrown away from the pile. If it cannot be used, it needs to be put back in the bottom

of the pile. So the method returns true if the card has to be put back in the pile and false if the card can be used, and therefore has to be discarded.

The argument is a player object because every chance card has some influence on the player's situation.

All of the controllers have an object of their corresponding data class, in order to get the information needed. A few of the controller classes also have other variables. This is when the onDrawn method needs an object or an arraylist. An example is the "TaxController", that has an account object called "parkinglotAccount". The onDrawn method needs an account object to transfer the money to. The reason it is not called in the "TaxData" class is because it is split between classes. It does not influence the ChanceCardTax alone, but also the field "Parkinglot", therefore it needs to be put in the controller. The data classes only contains data that influences the chance card.

Use of XML

While designing the game we realized that we would need to import "large amounts" of data for the fields. Instead of making long, unreadable arrays we decided to take use of a xml data system instead. The xml files consists of nodes. Some nodes will be thought of as branches, while others can be thought of as leaves. To find a specific thing in the xml file you tell the program to walk down a path along the branches until you reach the data you need. This way we could make a branch for each field and give it the values that it would needs Each branch in fields.xml contains a type, price and a pawn value. Each branch also contains additional data, depending on its type, such as rental value variations depending on the house count of a territory. This way we could write the data that each field needed, while having a quick overview of which tag each value belonged to.

We found this to be a much nicer approach, rather than having long unreadable arrays in-code.

XML was also used for the chance cards. As written in the chapter about the chance cards, we put all information about the different chance cards in an xml file, which the parser then later could load and grant to the respective chance card. According to the type of chance card the branch would contain different node values for each type. As opposed to our territory.xml layout, chancecards.xml contains two attributes: One for the chancecard type and another for the amount of chancecards of this type there should be added to the pile. This attribute is optional as only some chancecards are identical and appears multiple times. Therefore we decided to reuse the branch instead of having duplicates in our xml file.

Shufflebag

The chancecards had to come out in a random order, we also needed to assign the players a color without any of the players getting the same color. To do this we made use of our shufflebag from CDIO3. Since we had two uses for the shufflebag, one that needed integers and one that needed colors, we had to make the shufflebag generic, meaning that it can work with multiple types of input instead of just one. Marking a class as generic is actually just telling the compiler to create a version of the class, which supports whatever types it encounters it being used with. Had we not marked the shufflebag generic, then another, but discouraged approach, would be to duplicate the code in two files, each supporting different types. Following is a quote from the CDIO3 assignment explaining how it works:

"ShuffleBag works as a randomized queue in the way that you add variables to it and then they come out in a random order. It has a getNext() method, which works by picking a random index number from 0 to the index of the last available element. When the index is found, then the element is swapped with the last available element in the array and the index for available elements are decreased by one. This ensures that the elements which has already been selected falls outside of range of indexes and hence will never be

selected again, however if desired one can call the reset() method which sets the index to the last element in the array and the ShuffleBag can then be used once again."[1]

There are a few differences in the new ShuffleBag. The main() method has been removed since it was just an oddly placed test of the ShuffleBag. A new method has been added, pushBackLastCard() puts a card back in the pile. This is used when the player draws a card that for some reason cannot be used.

DiceCup and DiceResult

The DiceCup class is all about rolling dice. The class has a constructor which takes an integer, an array meant for dice rolls is then made with the length of the given integer. In this game this array would of course only be two spaces long since only two dice are used at any point. This is a feature from an earlier CDIO project that required several dice. We saw no need to change it since it still got the job done. The class also contains a rollDice() method. This method rolls two six sided dice for every spot in the array and inserts the result.

DiceResult is a storage class with some functionality related to the dice. DiceResult stores the arrays produced by DiceCup and is then utilized by other classes through getDice(), which returns the stored dice array. getSum(), used for adding up the dice, this is used when moving the number as far as the dice show. areDiceEqual() compares two dice and returns a boolean stating whether they are equal or not. This is used when determining if a player should be given another roll.

Tests

Throughout the progress of making the monopoly game, we made sure to test our code, and to see if everything worked as they should. We tested the code with multiple methods. Mostly JUnit testing and by playing the game multiple times to see how the whole program worked together.

We made a JUnit test for each class, to make sure they all worked properly, and made sure that all their methods were correctly done. We've tested all the field controller classes, to make sure that they had the correct rent, correct description and functions when you landed on the corresponding field. Some of the JUnit tests required us to make "annotation type Before", which is a method that causes the test to run some objects before the tests began. We only did it to those test which required the test methods to be needing the same objects, so it would made it easier to test objects, instead of making us create new ones in every test method.

We made sure to test out the player class, to make sure that his properties, account and dices are working as well as having the proper movement on the board.

We played the game a lot once we had it at a playable state. Even though playing it would hardly be an effective way of finding potential errors it still gave us a clear view of what was missing in the game and what needed more work.

We wanted to make sure that the game worked too in reality, outside the JUnit tests. Just playing the game and seeing that everything would take too long and there were too many random factors to test out specific parts of the game. Our solution to this was making a new pair of dice , testdice. These dice were manipulatable so that we could choose the result that we wanted when rolling the dice. This proved effective when we were testing out the prison.

To conclude the tests we tested which version of Java the program could run on and we tested the program on the most common operating systems in use. We found out that the program requires atleast java 1.7, since 1.7 introduced the ability to switch a string and two of our classes use this function, namely the two

[1] Gruppe 16 CDIO3 rapport, page 5 line 30

dedicated loader-classes. Otherwise we concluded that the program could run on the 3 newest mac OS X versions, Windows 7, 8 and 10. Finally we tested if it would run on one of the computers on DTU Campus and it did.

Test cases

Double rent

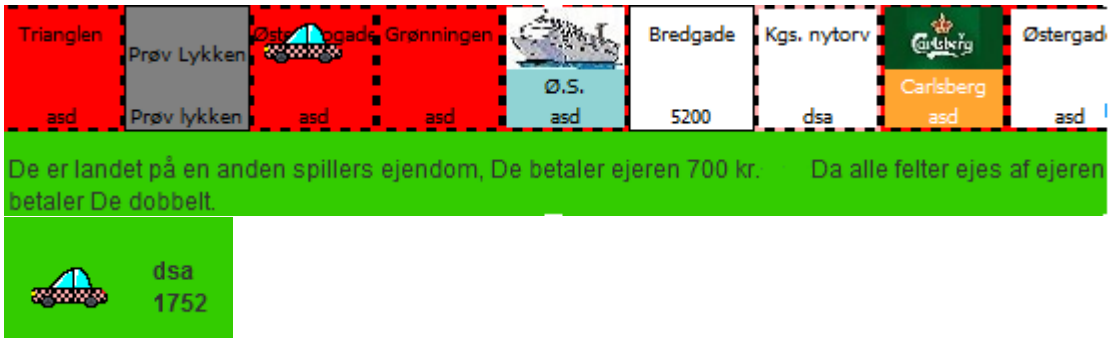
Preconditions:

All fields of a type is owned and no houses is built on the field landed on.

Expected output: A message is displayed declaring why you have to pay double rent and double rent will be charged.

Preconditions reached:

Test outcome:



Test result:

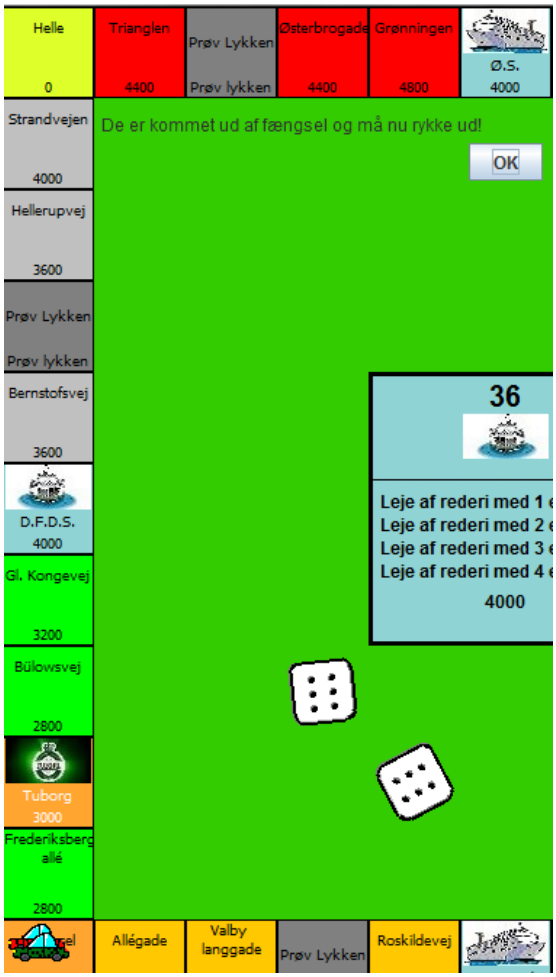
Succeeded

Slå et par for at komme ud af fængsel:

Preconditions:

The player is in prison and has chosen not to pay, nor use his card to get out of prison.

Test outcome:



Test result:

Succeeded

Conclusion

Overall Conclusion

To sum it all up, we've been very satisfied with our final product, especially since we changed a lot of things, including a new code structure in this project compared to our previous ones, which have made us almost work from scratch instead of continuing our work from CDIO3. We've done very well overall as group, by deciding to start early in the project, make some long days to have more time in the end to fix all the bugs and make sure everything works. We are pleased with the overall job, since we managed to complete almost every single one of the requirements for a real monopoly game by the danish rules. Down below you would find our product oriented conclusion, that goes a little more in details of with what we've made.

Product Oriented Conclusion

We've made a fully functional monopoly game, that have the 13 out of 14 requirements completed, which is listed in the section "Requirements and Demand". We have been able to implement almost every aspect of the real monopoly game and still have the game run stable without any apparent bugs in the code. The code is clean and understandable, and making additional features or changing the game would be easy.

Literature- and Source Directory

Gruppe 16 CDIO 3 rapport(2015)

Gruppe 16 CDIO 2 rapport(2015)

Appendix

Use case descriptions:

Use case description: "New game"

ID:
I

Brief description:
The pregame phase where the players enter their names

Primary actors:
Player

Secondary actors:
None

Preconditions:
Game has launched

Main flow:

1. Enter the amount of players (x) participating in the game (2-6 players)
2. Player 1 asked to enter his name
3. Player 1 enters their name
4. Player 2 asked to enter his name
5. Player 2 enters their name
6. if there's more than 2 players

a. Ask for each additional player's name
i. each additional player enters their names

Post conditions:
The game starts with the x players

Alternative flows:
None

Use case description "Land on fleet"

ID:
II

Brief Description:
When a player lands on one of the fleet fields

Primary actors:
Player

Secondary actors:
None

Preconditions:
Player has landed on a Fleet field.

Main flow:

1. The system has to show a message to the player, that describes the fields effect on the player
2. If the field is

a. not owned by another player
i. if player wants to buy the current field
    1. if enough money
a. buy the field
    2. else end turn
ii. else end turn
b. owned by another player
. the player that lands on the field pays the owner of the field a certain amount of money
    1. If the owner owns 1 fleet: 500
    2. If the owner owns 2 fleet: 1000
    3. If the owner owns 3 fleet: 2000
    4. if the owner owns 4 fleet: 4000

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Land on territory"

ID:
III

Brief Description:
When a player lands on one of the territory fields

Primary actors:
Player

Secondary actors:
None

Preconditions:
Player has landed on a Territory.

Main flow:

1. The system has to show a message to the player, that describes the fields effect on the player
2. If the field is
    c. not owned by another player
i. if player wants to buy the current field
        1. if enough money
a. buy the field
        2. else end turn
ii. else end turn
d. owned by another player
. the player who landed on the field, pays the owner of the field, an amount according to the field's rent

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "passStart"

ID:
IV

Brief Description:
When a player lands on or pass the refuge field.

Primary actors:
Player

Secondary actors:
None

Preconditions:
Player has landed on or passed the start field.
It is not the first turn the player has in the game.

Main flow:
1. The system has to show a message to the player, that describes the field effect on the player
2. Give the player a certain amount of money

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Land on tax"

ID:
V

Brief Description:
When a player lands on one of the tax fields

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's a player's turn

Main flow:

1. The system has to show a message to the player, that describes the fields effect on the player
2. If the field is

a. Tax1
i. pay 2000 money to the game
b. Tax2
. pay 4000 money to the game
i. pay 10% of the player's total assets to the game

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Land on brewery"

ID:
VI

Brief Description:
When a player lands on one of the brewery fields

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's a player's turn

Main flow:

1. The system has to show a message to the player, that describes the fields effect on the player
2. If the field is
    e. not owned by another player
i. if player wants to buy the current field
        1. if enough money
a. buy the field
        2. else end turn
ii. else end turn
f. owned by another player
. roll dice again
        1. They player that landed on the field pays the amount according to dice multiplied with 100 and multiplied with how many labor camps the owner, you are paying to, have.

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Land on Go to Prison"

ID:
VII

Brief Description:
When a player lands on the "Go to prison" field

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's a player's turn

Main flow:

1. The system has to show a message to the player, that describes the fields effect on the player
2. The player that landed on "Go to prison" field, moves to "Prison" field

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Get out of Prison"

ID:
VIII

Brief Description:
When a player attempts to get out of prison through rolling dice or paying a certain amount of gold to the bank.

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's a player's turn, and the player is in prison.

Main flow:

1. Player chooses to either roll, pay or use a valid Chance card which the player owns.

a. Roll

i. The system does 1-3 auto generated rolls each show with a slight delay

1. If one of the rolls is two of a kind

a. the player moves the amount which was rolled.

2. If there's no "two of a kind"-rolls

. end turn.

b. Pay a certain amount of gold.

. Players gets to roll and move accordingly to the roll.

c. Chance card.

. If the player owns the special chance-card to get out of prison this opportunity is available, and the player can choose this option.

Postconditions:
Next player's turn

Alternative flows:
None

Use case description: "Roll Dice"

ID:
IX

Brief description:
The player rolls the dice. If he gets a pair he gets another roll. If he gets 3 pairs in a row he goes to prison

Primary actors:
Player

Secondary actors:
None

Preconditions:
It's the players turn and the player is not in prison

Main flow:
1. Player is asked to roll dice
2. Player rolls dice
3. Move player
4. Use case: LandOnField
5. If diceroll is a pair
1. Give player another roll
2. Player rolls dice
3. Move player
4. Use case: LandOnField
5. If diceroll is a pair
1. Move player to prison

Post conditions:
Players position has been updated

Alternative flows:
None

Use case description: "Land on Chance field"

ID:
X

Brief description:
When a player lands on one of the chance fields

Primary actors:
Player

Secondary actors:
None

Preconditions:
Player has landed on ChanceField

Main flow:

1. The system has to show a message to the player, that describes the fields effect on the player
2. Depending on the cards description the player can either do the following:

a. Move to X
i. Move the player to the field described on the card
    1. If the player passes start field on the way, he receives his bonus money

b. Gain X amount of money
. Give the player X amount of money

c. Get out of jail free card
. Give the player the card, that he will be able to use it for later to get out of jail

d. Pay X amount of money
. Make the player pay X amount of money to the parking lot

e. All other players give X amount of money to one player
. The player that gets the card, get paid by all other players by X amount

Post conditions:
Chance card has been used

Alternative flows:
None

Use case description "Land on parking"

ID:
XI

Brief Description:
When a player lands on Parking Lot

Primary actors:
Player

Secondary actors:
None

Preconditions:
Player has landed on parking lot.

Main flow:
3. The system has to show a message to the player, that describes the field effect on the player
4. Give the player the flat amount of money + the collected amount of money which has been given to the field through taX and chance cards.

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Land on brewery"

ID:
XII

Brief Description:
When a player lands on one of the brewery fields

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's a player's turn

Main flow:

3. The system has to show a message to the player, that describes the fields effect on the player
4. If the field is
   g. not owned by another player
      i. if player wants to buy the current field
         1. if enough money
            a. buy the field
         2. else end turn
      ii. else end turn
   h. owned by another player
      . roll dice again
         1. They player that landed on the field pays the amount according to dice multiplied with 100 and multiplied with how many labor camps the owner, you are paying to, have.

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Buy house"

ID:
XIII

Brief Description:
When a player wishes to buy a house on one of his owned fields

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's the beginning of a player's turn

Main flow:

1. The player clicks on a field he owns and wants to buy a house on
2. Verify that the field is in fact owned by him.
3. Check if the player owns all properties from that group
4. Inform the player about the price of an upgrade and tell him/her if they cannot afford an upgrade
5. Once bought, update the fields with the new bought house.
6. Ask the player if he wishes to buy another house.

Postconditions:
None

Alternative flows:
None

Use case description "Start turn"

ID:
XIV

Brief Description:
When a player start his turn

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's a player's turn

Main flow:
The Player is asked to do one of the following
1. Roll the dice
    1. Use case: Roll dice
    2. Buy a house/Hotel
        1. Use case: Buy house/hotel
    3. Pawn a field
        1. Use case: Pawn a field
    4. Release/unpawn a field
        1. Use case: Unpawn a field
    5. Buy a field from another player
        1. Use cases: buy field from another player

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Pawn a field"

ID:
XV

Brief Description:
When a player choose to pawn a field

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's a player's turn

Main flow:

1. Player selects to pawn a field
2. Player gets a list of available fields to pawn
3. Player selects a field to pawn
4. Player received the money for pawning a field
5. Player gets new options to do

a. Use case: Start turn

Postconditions:
Next player's turn

Alternative flows:
None

Use case description "Unpawn a field"

ID:
XVI

Brief Description:
When a player choose to unpawn a field

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it's a player's turn

Main flow:

1. Player selects to unpawn a field
2. Player gets a list of available fields to unpawn
3. Player selects a field to unpawn
4. The game checks if the player have enough money to pay back for unpawning a field

a. If yes, the player unpawns the selected field
b. Else the player receives a message which says not enough money

5. Player gets new options to do

. Use case: Start turn

Postconditions:
Next player's turn

Alternative flows:
None

Use case description “Buy field from another player”

ID:
XVII

Brief Description:
When a player choose to buy a field from another player

Primary actors:
Player

Secondary actors:
None

Preconditions:
When it’s a player’s turn

Main flow:

1. Player selects to buy a field from another player
2. The buyer receives a list of players to select from
3. The buyer selects a player
4. The buyer now gets a list of fields the selected player have
5. The buyer chooses a field from the selected player
6. The selected player now enters a price for the buyer to buy the field
7. The buyer can now choose to accept or decline the amount

a. If accepted, the money transfers between the two players and the buy recieves the field

b. If decline the buyer can try again

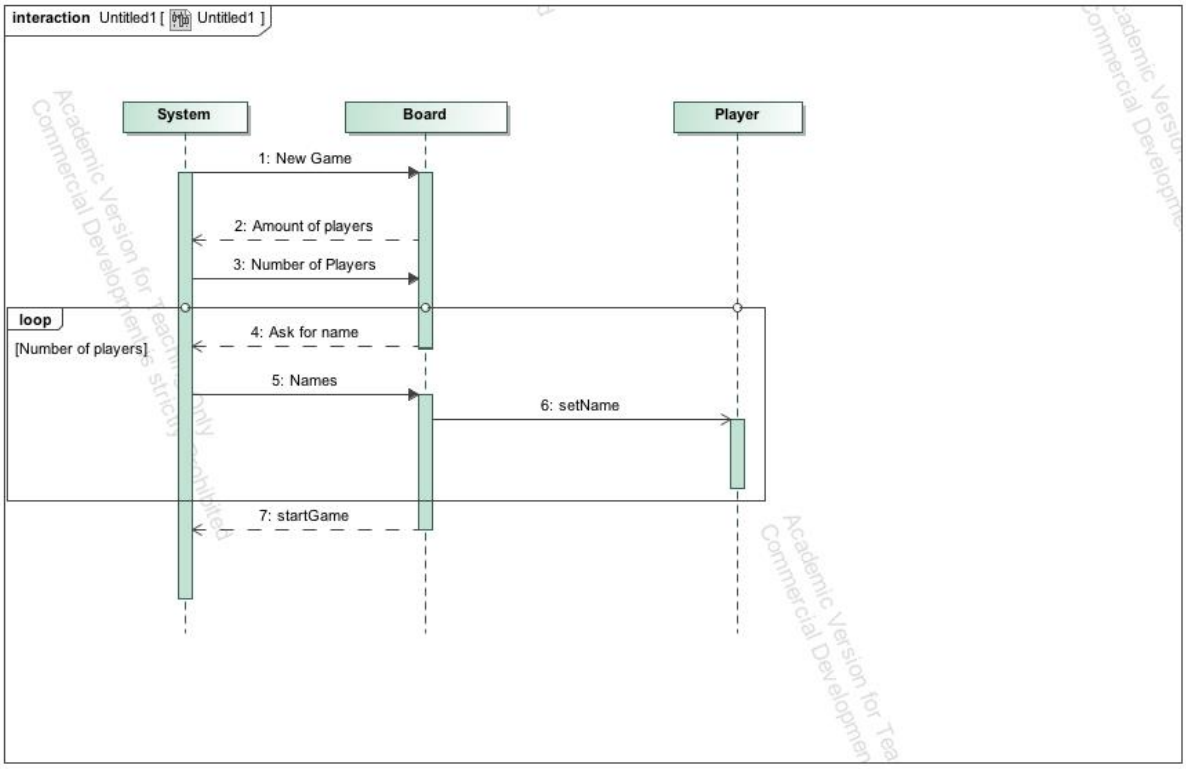
8. Player gets new options:

. Use Case: Start turn

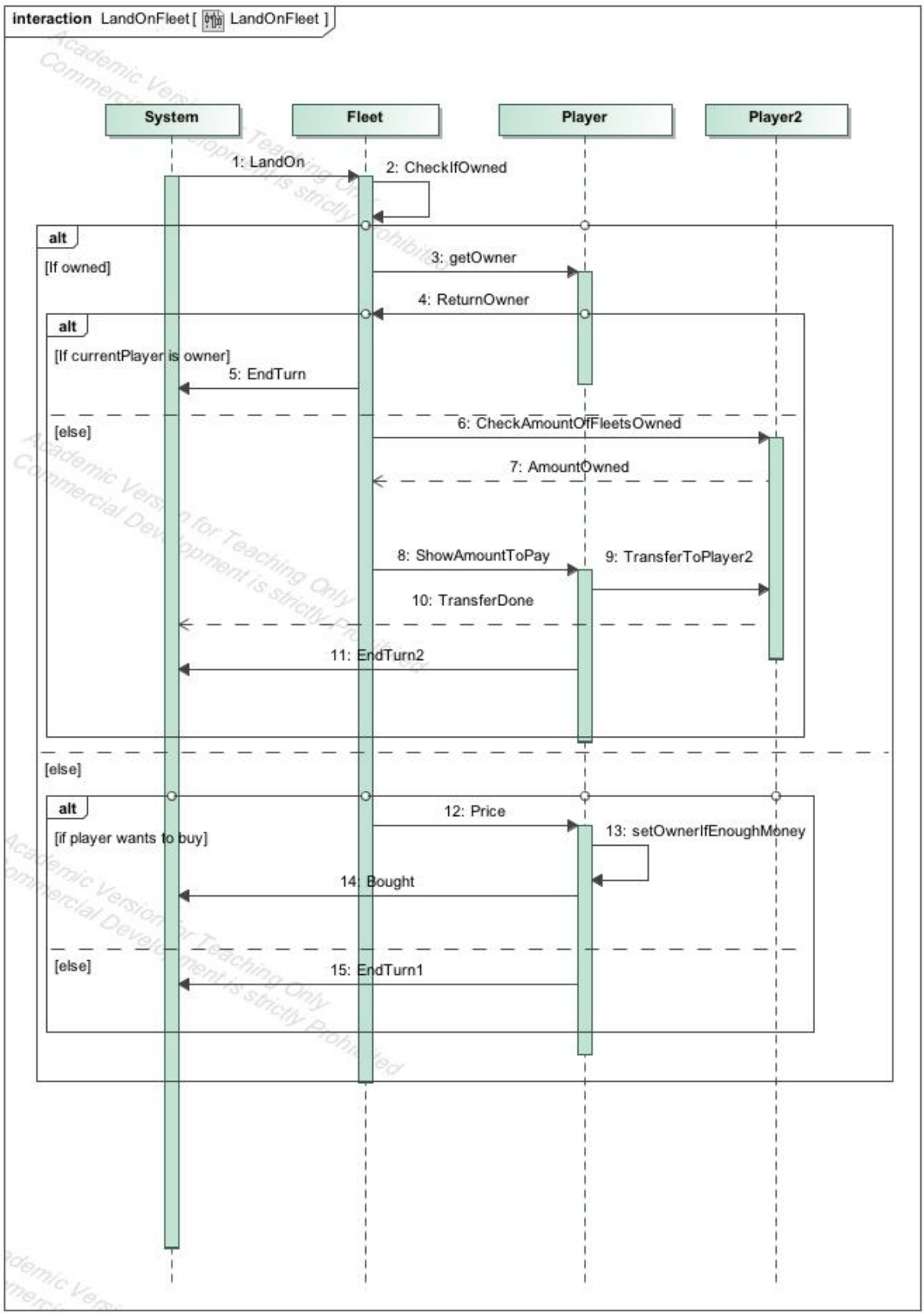
Postconditions:
Next player’s turn

Alternative flows:
None

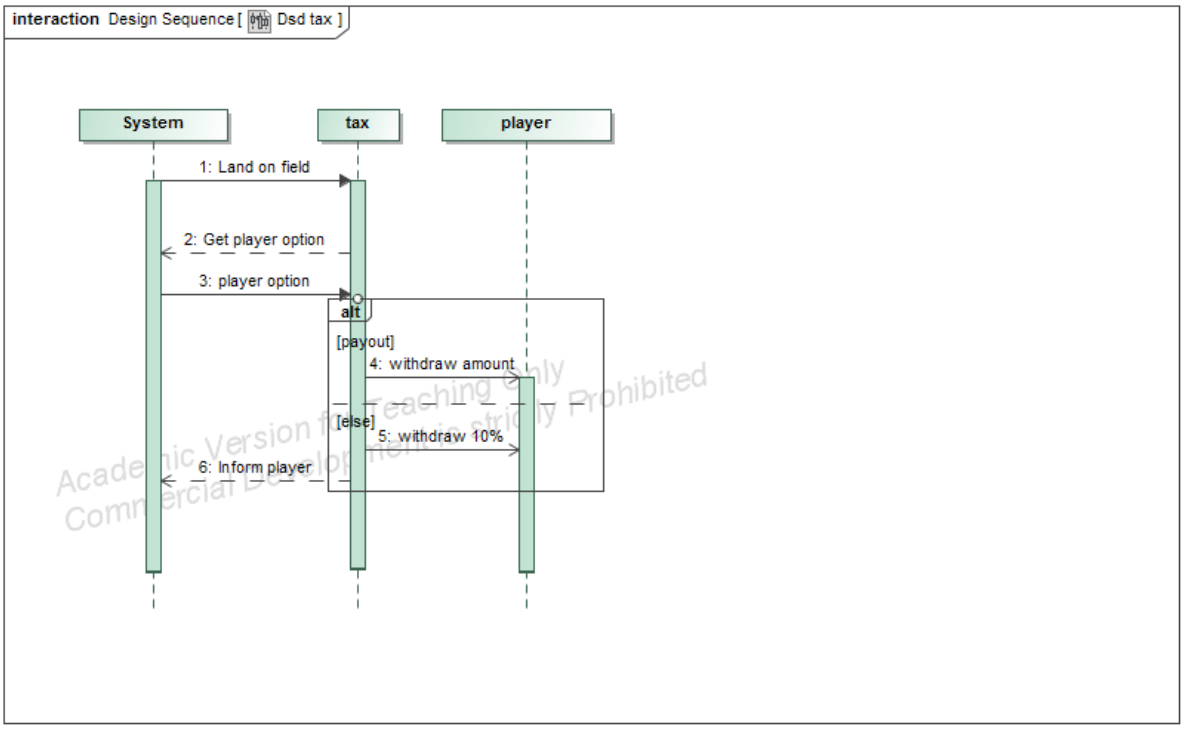
DSD "NewGame"


interaction Untitled1 [Untitled1]
System
Board
Player
1: New Game
2: Amount of players
3: Number of Players
loop
[Number of players]
4: Ask for name
5: Names
6: setName
7: startGame

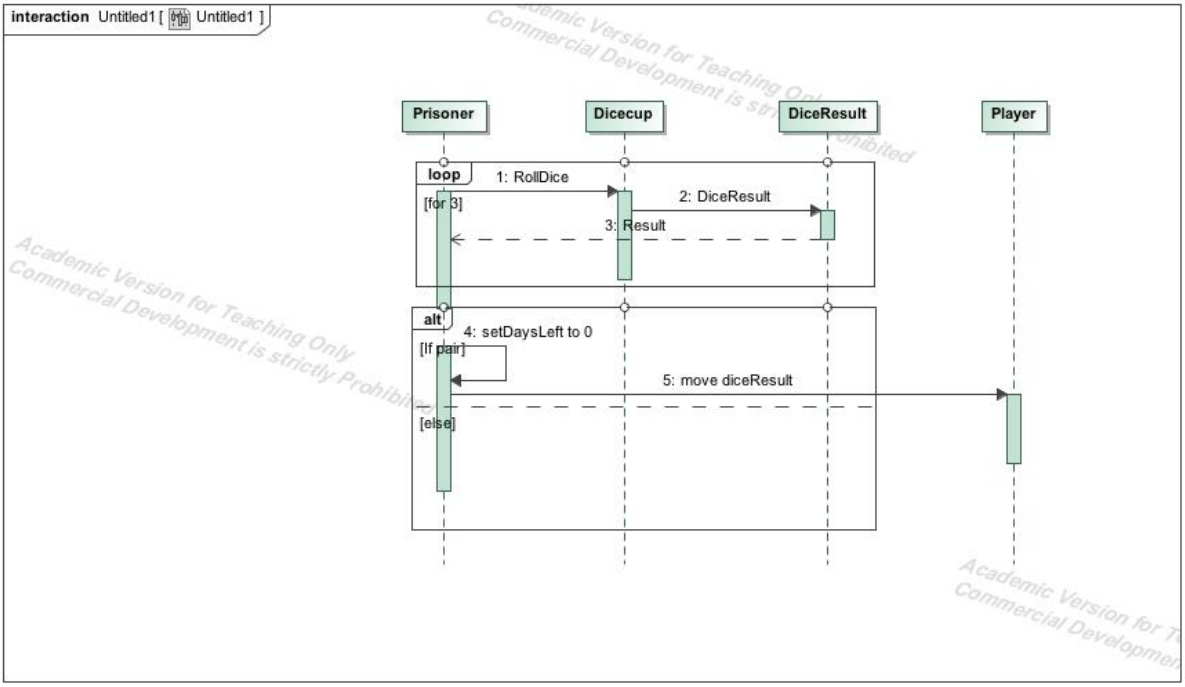
DSD "LandOnFleet"



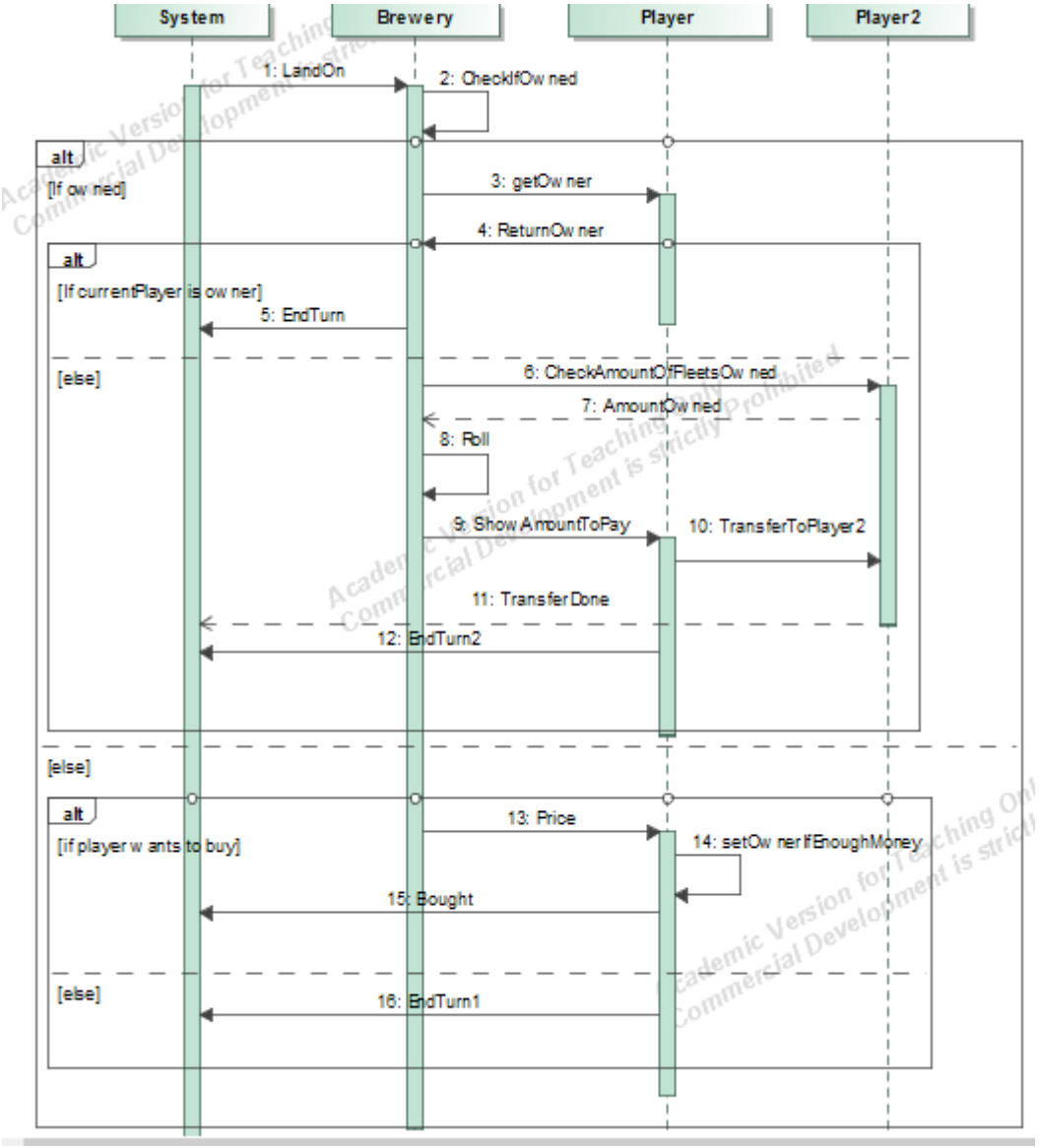
DSD "LandOnTax"



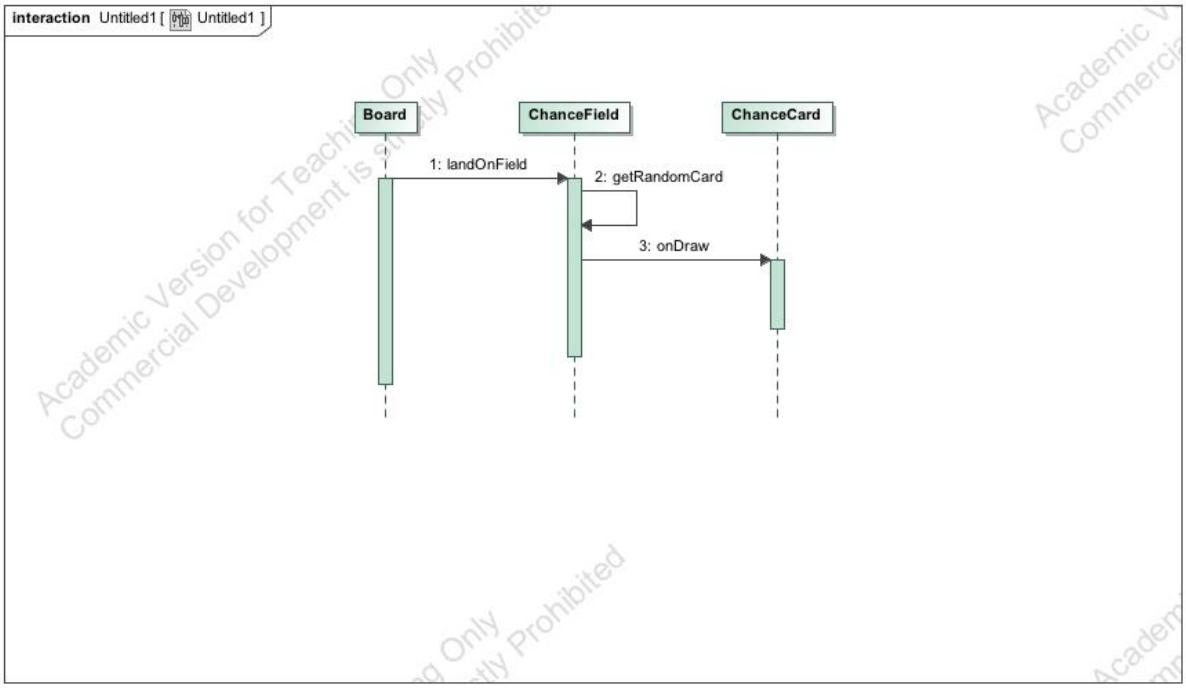
DSD "GetOutOfPrison"

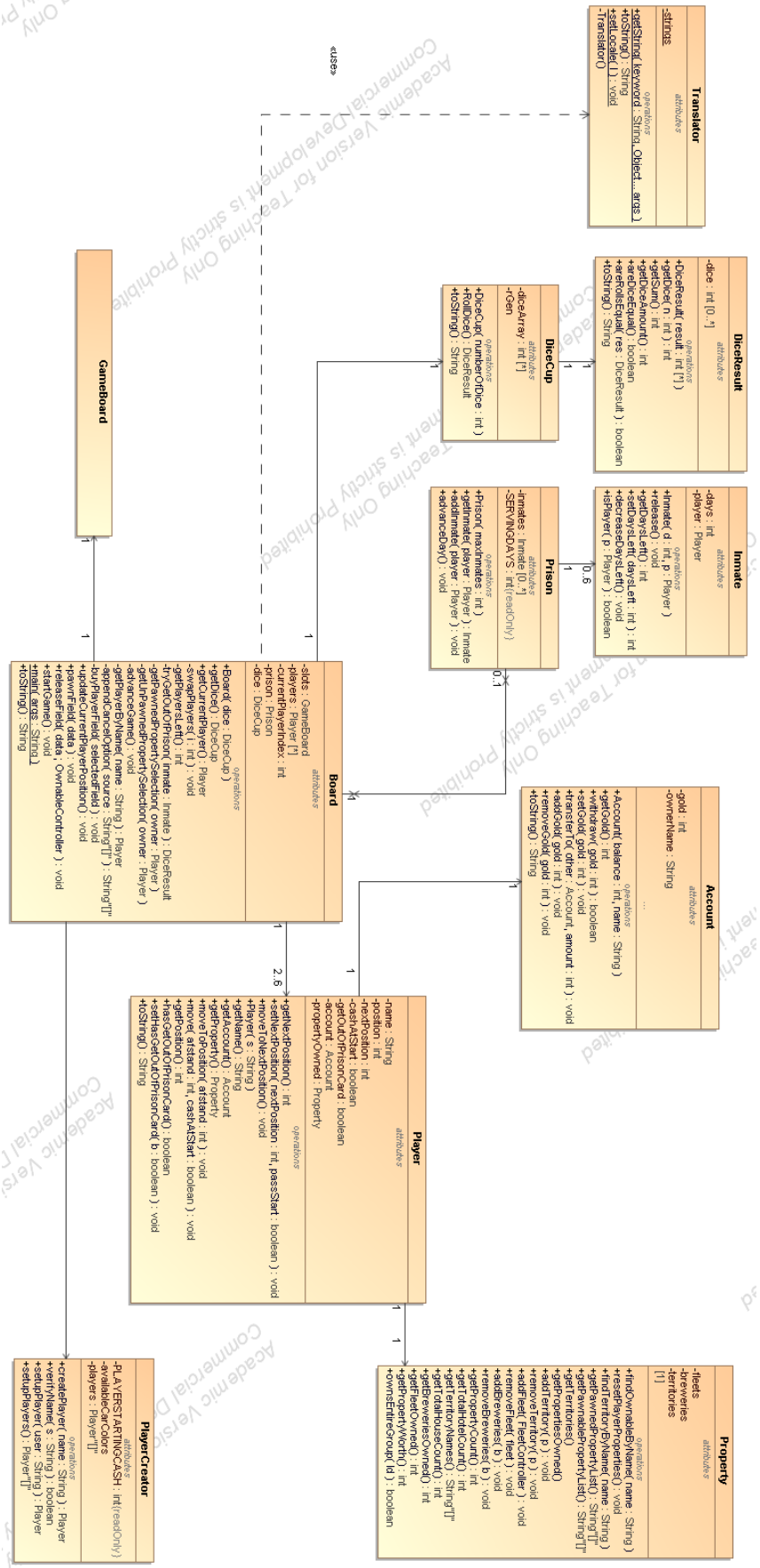


DSD "LandOnBrewery

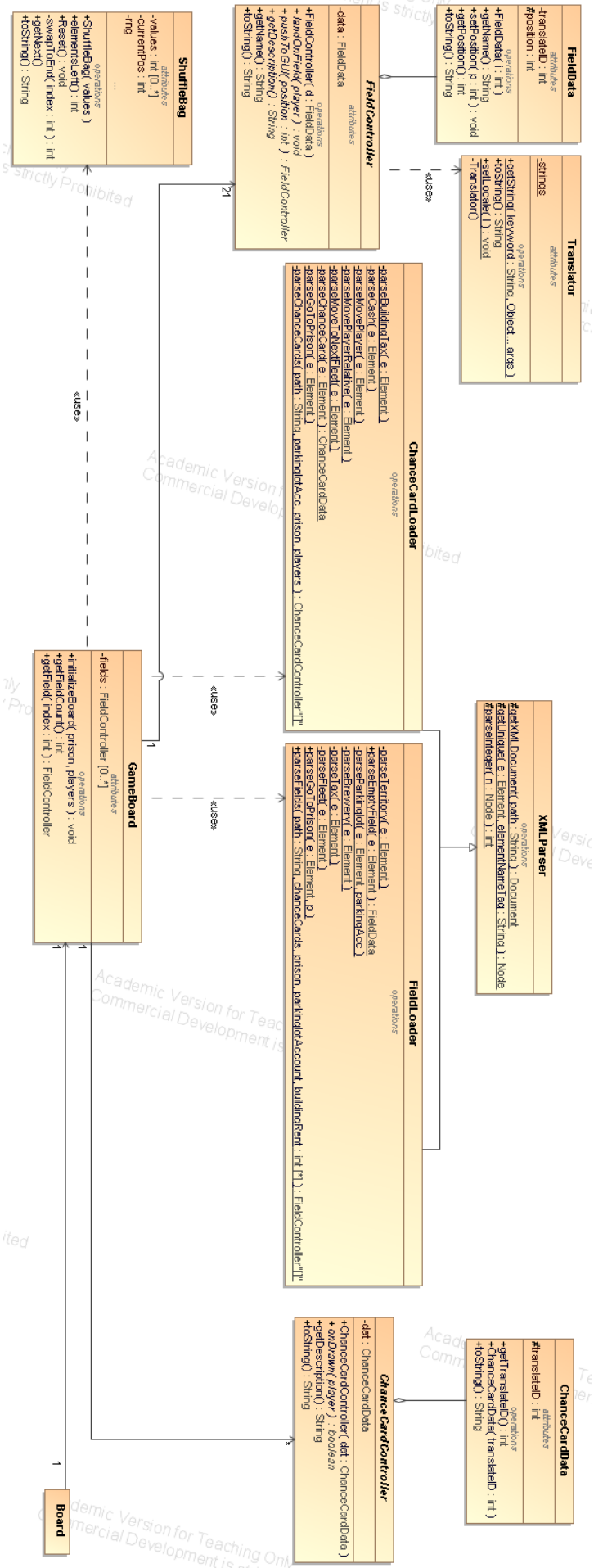


DSD "LandOnChanceField"




Translator
DiceResult
DiceCup
Inmate
Prison
GameBoard
Board
Account
Player
Property
PlayerCreator
«use»
1
2..6
0..1
0..6

Gameboard and its relations


FieldData
-translatedID : int
#position : int
+FieldData(i : int)
+getName() : String
+setPosition(p : int) : void
+getPosition() : int
+toString() : String
Translator
-strings
+getString(keyword : String, Object... args)
+toString() : String
-setLocale() : void
-Translator()
«use»
FieldController
-data : FieldData
+FieldController(d : FieldData)
+landOnField(player) : void
+pushToGUI(position : int) : FieldController
+getDescription() : String
+getName() : String
+toString() : String
21
ShuffleBag
-values : int [0..*]
-currentPos : int
-rng
+ShuffleBag(values)
+elementsLeft() : int
+Reset() : void
-swapToEnd(index : int) : int
+getNext()
+toString() : String
«use»
ChanceCardLoader
+parseBuildingTax(e : Element)
+parseCashIn(e : Element)
+parseMovePlayer(e : Element)
+parseMovePlayerRelative(e : Element)
+parseMoveToNextFleet(e : Element)
+parseChanceCard(e : Element) : ChanceCardData
+parseGoToPrison(e : Element)
+parseChanceCards(path : String, parkingAcc, prison, players) : ChanceCardController[*]
XMLParser
#getXMLDocument(path : String) : Document
#getUnique(e : Element, elementNameTag : String) : Node
#parseInteger(n : Node) : int
GameBoard
-fields : FieldController [0..*]
+initializeBoard(prison, players) : void
+getFieldCount() : int
+getField(index : int) : FieldController
1
«use»
«use»
FieldLoader
+parseTerritory(e : Element)
+parseEmptyField(e : Element) : FieldData
+parseParkingAcc(e : Element, parkingAcc)
+parseBrewery(e : Element)
+parseTax(e : Element)
+parseFleet(e : Element)
+parseGoToPrison(e : Element, p)
+parseFields(path : String, chanceCards, prison, parkingAccount, buildingRent : int [*]) : FieldController[*]
ChanceCardData
#translatedID : int
+getTranslatedID() : int
+ChanceCardData(translatedID : int)
+toString() : String
ChanceCardController
-dat : ChanceCardData
+ChanceCardController(dat : ChanceCardData)
+onDrawn(player) : boolean
+getDescription() : String
+toString() : String
1
Board