Projects fall 2016 – jan 2017

02312-14 Indledende programmering, 02313 Udviklingsmetoder til IT-Systemer and 02315 Versions Styring og test-metoder.

Project name: *CDIO del3*

Group number: *14*

Due date: *Fredag, 25/11 2016 at. 23:59*

Conceive Develop Implement Operate project (CDIO) Assignment 3: Monopoly game

Group 14

Rasmus Blichfeldt	s165205
Casper Bodskov	s165211
Lasse Dyrsted	s165240
Michael Klan	s144865
Mathias Larsen	s137055
Timothy Rasmussen	s144146

Danmarks Tekniske Universitet DTU

Supervisors:

Agner Fog

Henrik Tange

Stig Høgh

Time Table & Tasks

Task	Sub total (in hours)		Name	Sub total (in hours)
Requirements	15		Timothy	52
Analysis	36		Lasse	63.5
Design	16.5		Rasmus	52
Implementation	100.5		Mathias	62.5
Test	49.5		Michael	39.5
Documentation	109.5		Casper	58.5
Total	325			325

For detailed time table see appendix 3

Name	**Number of tasks**
Casper	**24**
Lasse	**22**
Mathias	**32**
Michael	**15**
Timothy	**24**
Rasmus	**22**

For details see appendix 4

Abstract

This rapport will serve as documentation of our work and how we have used models and tables to visualize our ideas and work schedule. Our product would come to include a boardgame with 2 dice, 2-6 players, and 21 fields that said players can land upon. The players can then buy territories. Other than that there will be fields with preset effects meaning they will either add or subtract points from the player's account. Some of these include choices between a percentile taxation or a flat amount. Throughout the entire report we've abided by the FURPS+, UP, and GRASP principles. We also attempted to follow the assignment's required formalities. In conclusion we finished our product, tested the various parts of the program as well as meeting the requirements set from the assignment but didn't reach our personal second iteration.

Table of Content

1. Introduction

The purpose of this project is to satisfy an assignment in which the objective is to make a board game. The board game should consist of a set of 21 fields each with a unique effect on the game, and the game should have an interface that the user can interact with by pressing buttons or selecting options.

To accomplish this we received a GUI and researched possible ways to create code pieces that could store the different fields' information as well as their position. Other than this we needed the dice from our last game and a way to store the position of the player token.

The purpose of the assignment is to learn the 4 phases of CDIO as well as utilize the models and knowledge from lectures to fulfill a hypothetical customer's requested system.

2. Analysis

2.1. Requirements specification

1. **Players**
    1.1. There shall be 2-6 players.
    1.2. The player shall continue from the field they land on the following turn.
    1.3. The player shall have access to an account.
2. **Gameboard**
    2.1. The gameboard shall be continuous in nature.
    2.2. The gameboard shall contain the following types of fields:
        2.2.1. Territory
        2.2.2. Labor camp
        2.2.3. Fleet
        2.2.4. Tax
        2.2.5. Refuge
3. **Field**
    3.1. The fields shall have the attributes described in appendix 1.
    3.2. The Field class and all subclasses shall contain a landOnField method.
4. **Shaker**
    4.1. The shaker cup shall hold 2 dice.
    4.2. The sum of two dice shall be within the statistical probability.
5. **Dice**
    5.1. The dice shall be 6-sided.
    5.2. The probability of virtually throwing a die should be equal to that of a real one.
6. **Account**
    6.1. The account shall keep the balance of each active players.
        6.1.1. The game shall be able to withdraw and add from/to the players' designated accounts.
    6.2. The accounts shall start with a value of 30.000.
7. **Win condition**

7.1. The game shall be won when all but one player is bankrupt.

8. Non-functional requirements:

8.1. The game shall contain a GUI.

8.2. The gameboard shall contain all the Fields in an array.

8.3. The gameboard shall contain a toString method to print all the fields on the board.

8.4. The program shall run on Windows operating system.

8.5. The program shall run on the computers available in the databars on DTU campus Lyngby.

8.6. Inheritance shall be used according to the project leader's remarks.

8.7. The rapport shall contain:

8.7.1. A requirements specification artifact

8.7.2. A use-case diagram artifact

8.7.3. A fully dressed use-case

8.7.4. A domain model

8.7.5. A robustness diagram

8.7.6. A design sequence diagram

8.7.7. An explanation of the concept of inheritance

8.7.8. An explanation of the concept of an abstract class

8.7.9. An explanation of the concept of polymorphism

8.7.10. Documentation of tests with screenshots

8.7.11. Documentation of use of the GRASP

8.8. The program shall be playable by anyone with basic knowledge of computers and the capability to push a button.

8.9. Everything shall be written in English.

8.10. The program shall be written in UTF-8.

8.11. The project shall be named 14_del3.

8.12. The test classes shall be in another package than the game:

8.12.1. One package shall include the game files

8.12.2. The second package shall include the test files

8.13. All work on the code shall be done in a Github repository and “commits” shall be done:

8.13.1. Once every hour

8.13.2. Every time a smaller objective or piece of code is written

8.14. The program shall be importable to eclipse.

8.15. Test

8.15.1. 3 test scripts shall be made with a corresponding test rapport.

8.15.2. A jUnitest shall be made for all classes.

8.15.3. A jUnitest shall be made for all “landOnField” methods.

8.16. There shall be a declaration which describes the minimal computer specifications required to run the program and what programs need to be installed prior to running the program.

2.2. System requirements

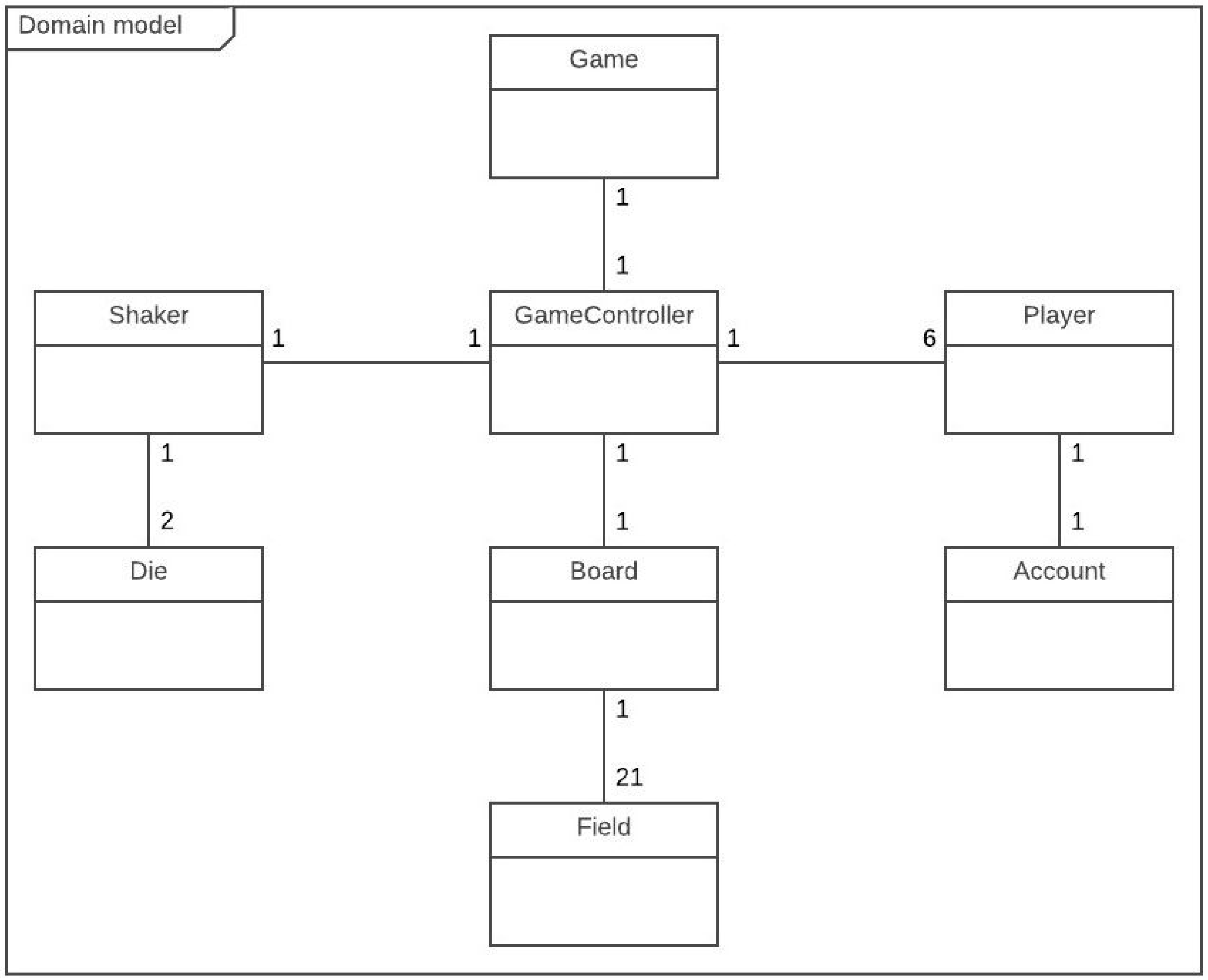
Because of our program’s simple nature, the program only requires that the systems specifications matches up with the specified system requirements for java 8, Java JRE (Java Runtime Environment), and the supplementing files. Java 8 and Java JRE must be installed.

Version of windows	Windows vista SP1 or newer.
RAM	128 MB
Disk Space	126 MB
Processor	Pentium 2 266 MHz or newer.

2.3. Word analysis resumé

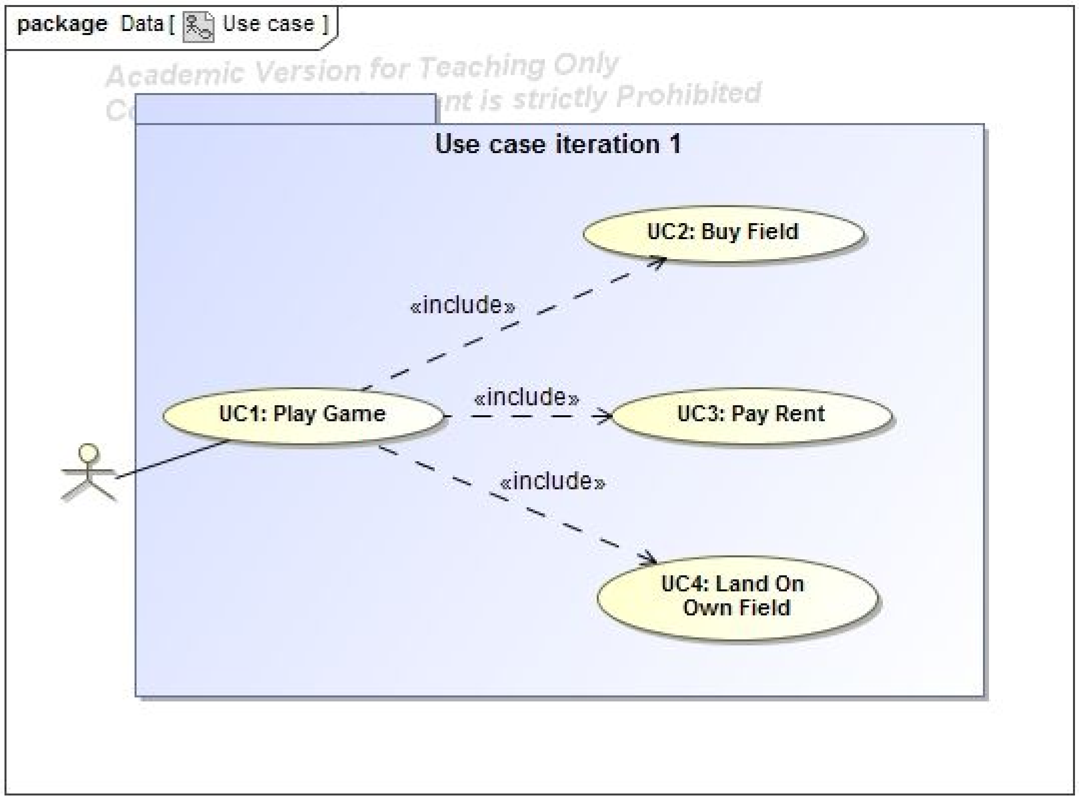
From the assignment text we could conclude that we needed a game for 2-6 players who each start with 30.000 points. The game should include a “GameBoard” class that takes the “Field” class’ objects and puts them in an array which would make it possible to pick a field from the value of a random dice throw. The field has the subclass “Ownable” which has further subclasses named “Territory”, “Labour camp” and “Fleet”. These subclasses inherit the “price” and “owner” attributes from the “Ownable” class and further have “rent” attributes with different names. Other fields that inherit from the “Field” class but not the “Ownable” class are “Tax” and “Refugee”. These have attributes which either makes the player pay tax or get a bonus. The game ends once all but one player are bankrupt.

2.4. Domain model



The domain model gives us an overview of the classes and its relations to each other. Through a word analysis we have deducted that the classes presented in the diagram is enough to run the program. The diagram is simple in nature and leaves out details to give a better overview of the system.

2.5. Use case diagram



The use case diagram shows how interactive the program is and how the user affects the program. This particular diagram is relatively simple since there is only a single actor. The use cases will explain more in-depth as to what happens during the interaction where the diagram is more simple and only serves to give an overview.

2.6. Use case

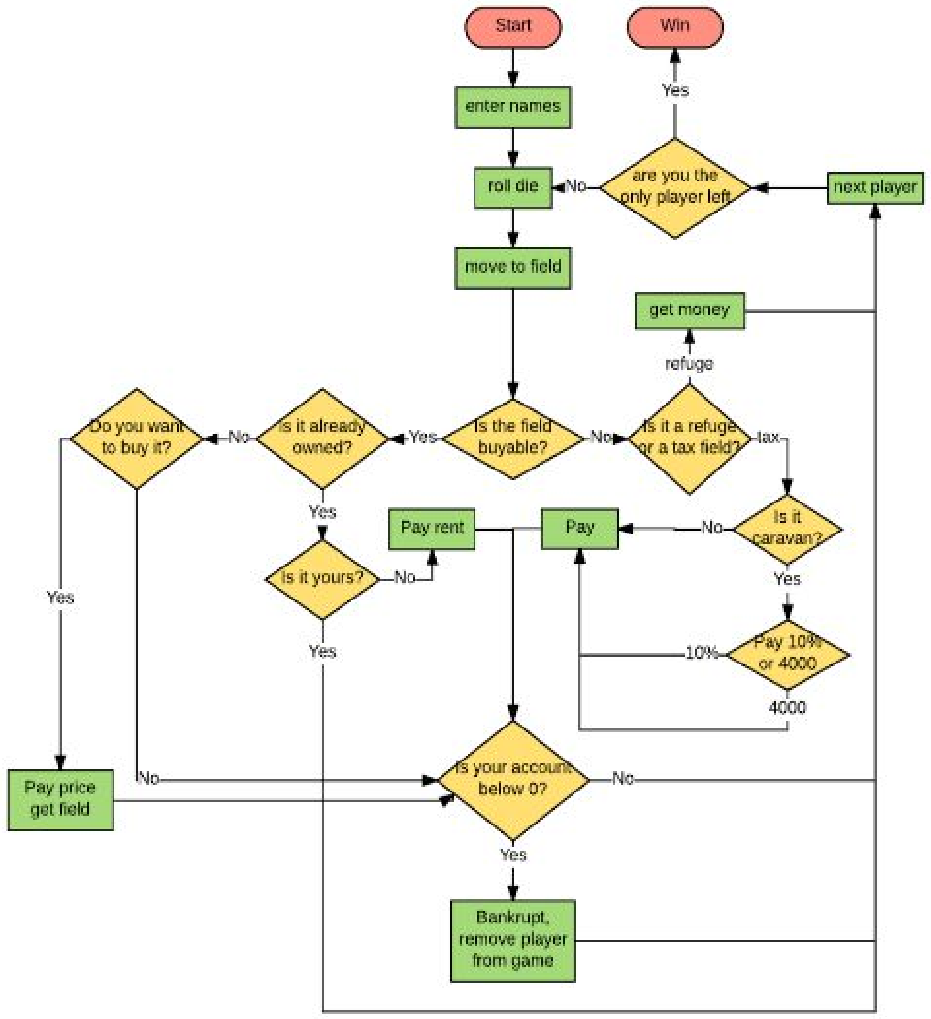
Name	Play Game.
Identifier	UC1.
Description	The flow of the game.
Primary actors	Player 1-6.
Secondary actors	none.
Preconditions	A player executes an instance of the Monopoly Game.
Main flow	1. System prints “How many players in this game?”. 2. A player enters the amount of players. 3. System prints the players name 4. A player enters his/her name. 5. Repeat step (3.) and (4.) for all players. 6. While there is more than 1 player. 6.1. System prints “player x’s turn” 6.2. Current player presses the “OK” button. 6.3. System rolls dice, and the current player moves the same amount of fields clockwise as the sum of dice. 6.4. UC3. 7. System prints “congratulations current player you won”. 8. Current player presses the OK button.
Postconditions	The game closes.
Alternative flow	At 6.4 UC2 At 6.4 UC4

Name	Buy Field.
Identifier	UC2.
Description	How a player buys a field.
Primary actors	Current player.
Secondary actors	None.
Preconditions	A player lands on a field that is ownable and not owned.
Main flow	1. The system asks the current player if she/he want to buy the field she/he has just landed on. 2. The current player presses the buy button. 3. System transfers the amount the field cost from current players account. 4. The field stores that it is now owned by the current player. 5. The system prints the current players points and owned fields. 6. Current player presses the OK button.
Postconditions	The next player's turn start.
Alternative flow	1. Instead of 2. a. The current player presses the pass button. 2. Contenues at 4.

Name	Pay Rent
Identifier	UC3
Description	How a player transfers points to another player, when the player lands on an owned field.
Primary actors	Current player.
Secondary actors	None.
Preconditions	The current player lands on an owned field.
Main flow	1. System transfers the rent amount to the owner of the field. 2. System prints the rent amount, owner of the field and balance before and after the transfer. 3. If current player's account is less than 0 3.1. Current player is removed from game. 4. Current player presses the OK button.
Postconditions	The next player's turn starts.
Alternative flow	None.

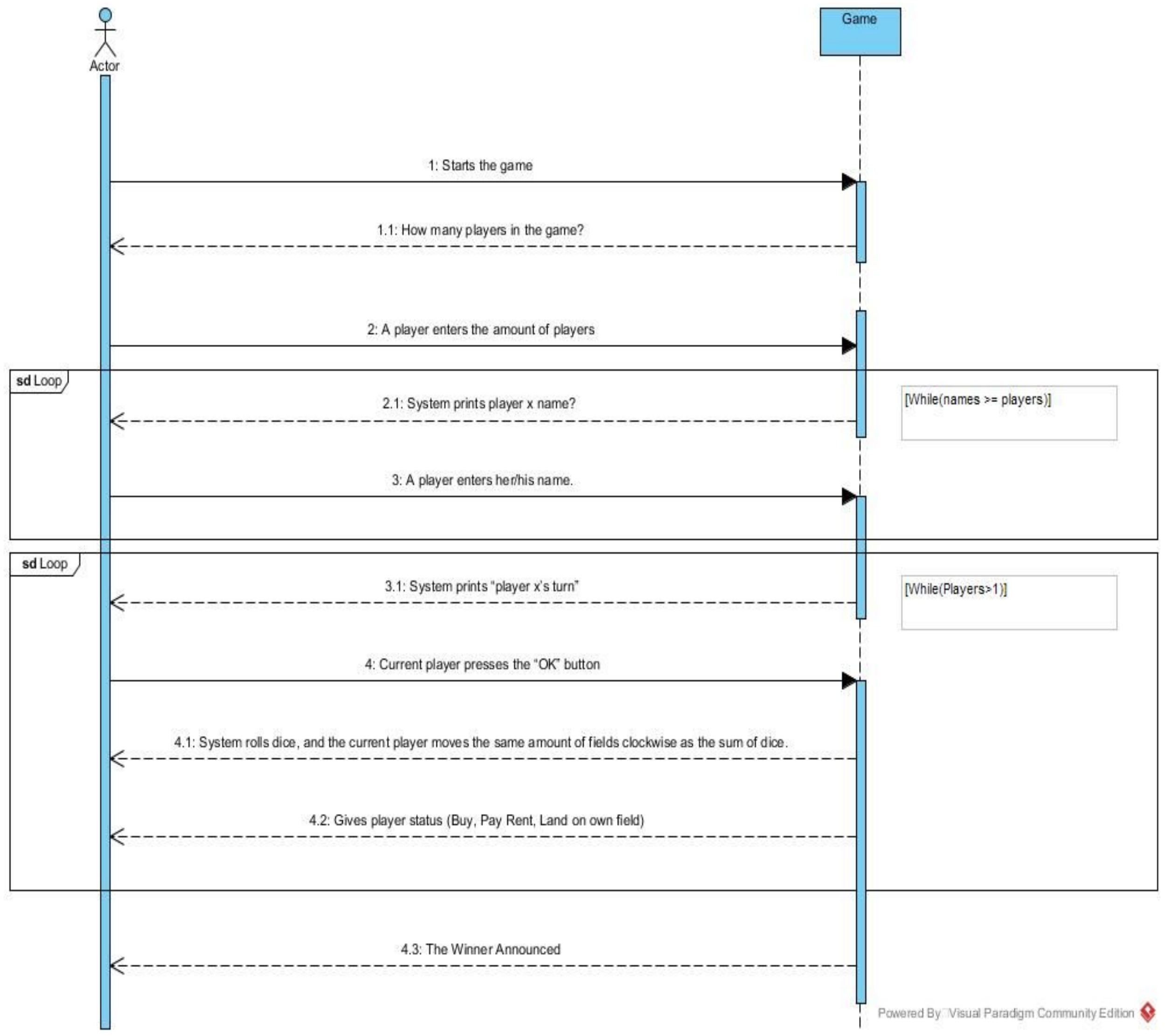
Name	Land On Own Field.
Identifier	UC4.
Description	What happens when a player lands on a field they own.
Primary actors	Current player.
Secondary actors	None.
Preconditions	Current player lands on a field s/he owns.
Main flow	1. System prints “you own this field nothing happens”. 2. Current player presses the OK button.
Postconditions	The next player's turn starts.
Alternative flow	None.

2.7. Flowchart



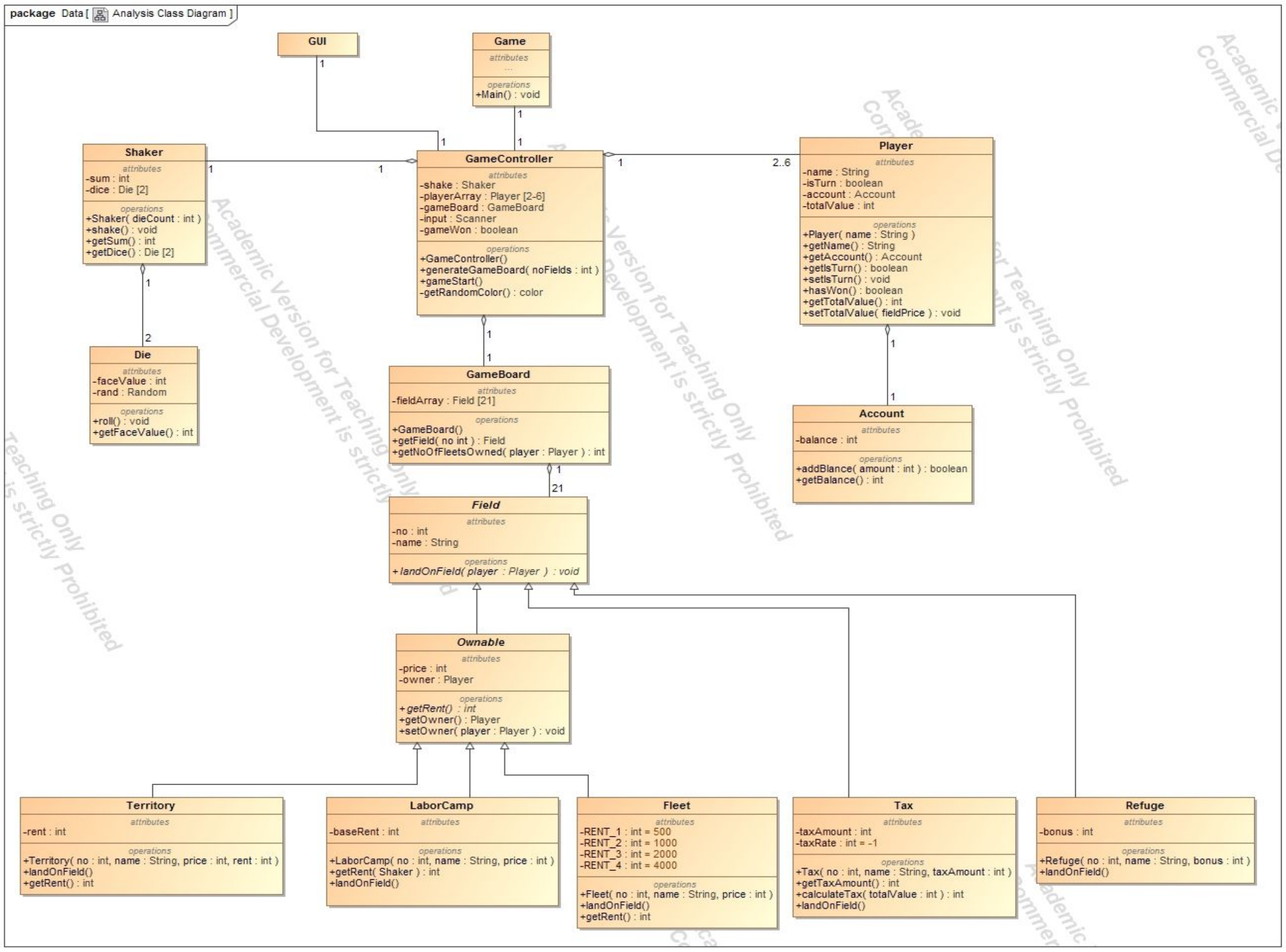
The flowchart shows the flow of all possible branches in the program. The flowchart thereby outlines the logic of the program in a clear manner. The game start is presented as a problem and reaching the end (In our case "winning the game") is the solution. The red squares are the "terminators" and are where the flowchart begin and ends. The green square are the actions that affect the player in the game performed by the system. The yellow squares represents the if-statements in our program.

2.8. SSD


Actor
Game
1: Starts the game
1.1: How many players in the game?
2: A player enters the amount of players
sd Loop
2.1: System prints player x name?
[While(names >= players)]
3: A player enters her/his name.
sd Loop
3.1: System prints "player x's turn"
[While(Players>1)]
4: Current player presses the "OK" button
4.1: System rolls dice, and the current player moves the same amount of fields clockwise as the sum of dice.
4.2: Gives player status (Buy, Pay Rent, Land on own field)
4.3: The Winner Announced
Powered By Visual Paradigm Community Edition

The SSD serves to show the messages invoked by an actor. In our case it's the player(s) of the game that acts upon the program. It can be interpreted as a visual representation of the use case diagram. It also shows which parts loop. Our system loops when giving the players their names and while there are more than one player active. Meaning until there's only one player left, the game continues.

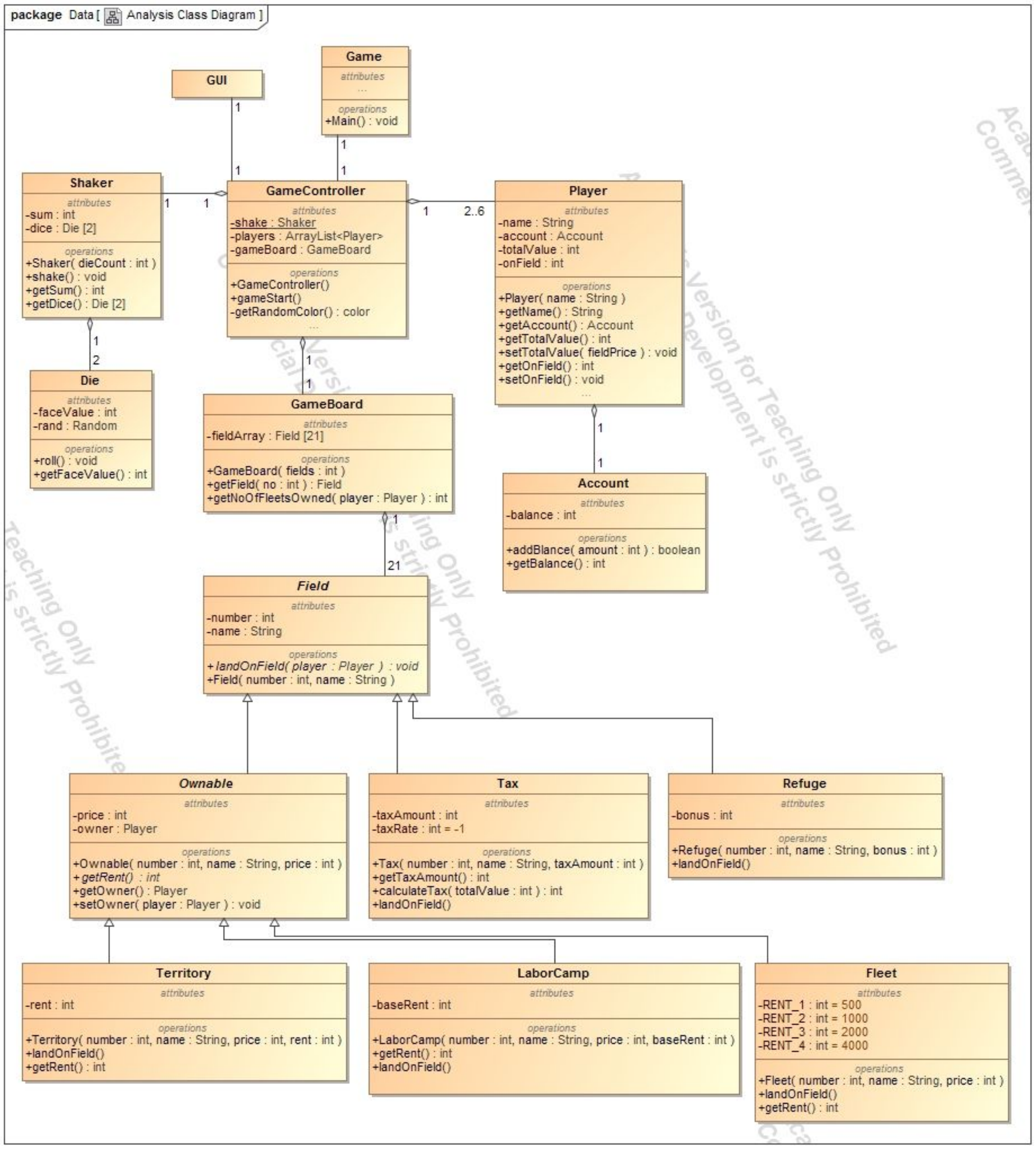
2.9. Analysis Class diagram



The analysis class diagram is fundamental to coding the program as clean and as structurally sound as possible. It outlines how we want to begin writing the code, not how the final code code was done. It looks in many ways similar to the domain model, only with more detail.

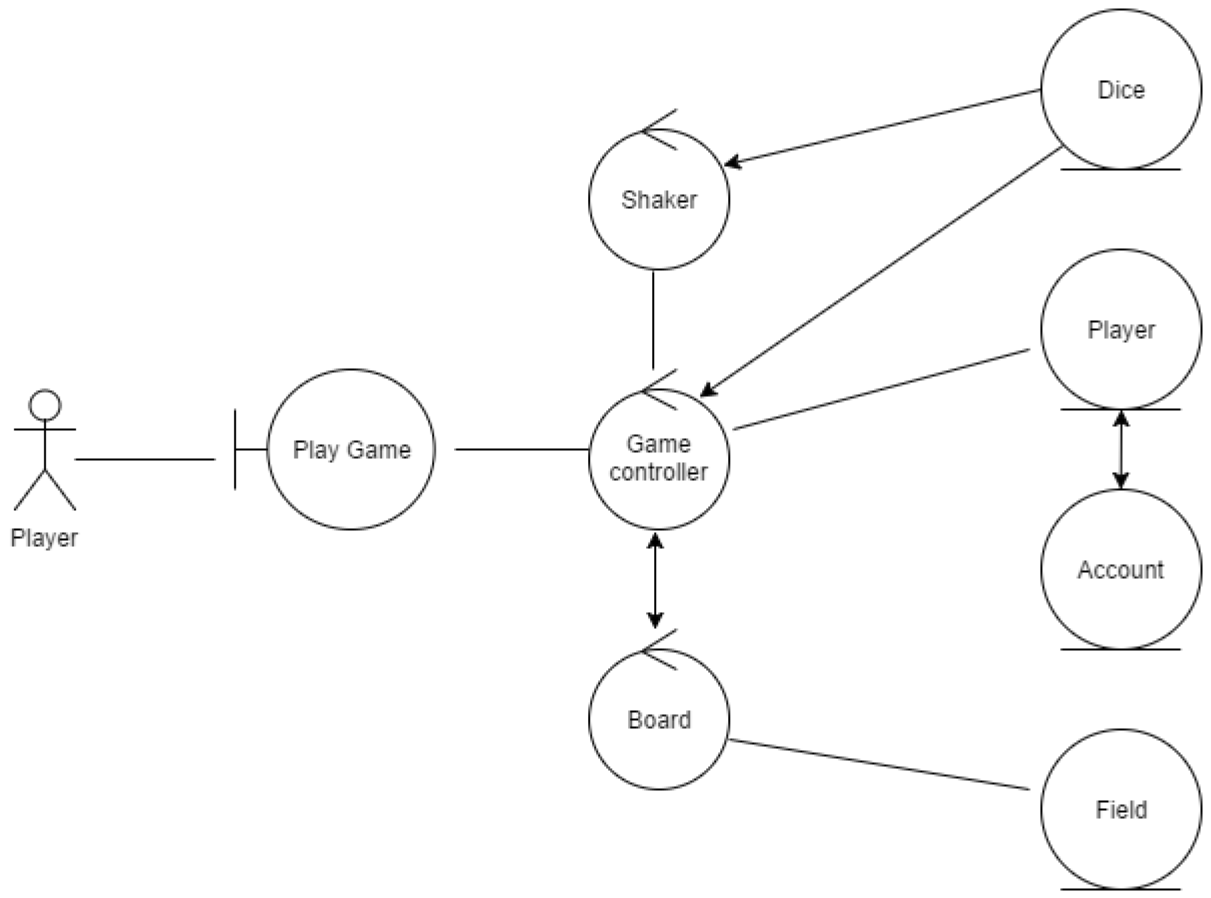
The methods and variables are described as well as their visibility.

2.10. Analysis Class Diagram - version 2.0



After discovering that our code had to be a bit different from the class diagram, we drew a new one so we can precisely show the code with minimal errors. This is a second version of the diagram and not representative of the second iteration of the program.

2.11. Robustness diagram



The purpose of the robustness diagram or BCE (boundary, control, entity) is to analyze the steps of the use case and thereby validate the business logic. It also tests the robustness of the whole system by ensuring that the terminology is consistent across all the use cases and that the usage requirements are fulfilled. The terminology determined in the use cases and robustness diagrams is then used in other UML diagrams.

Actors are the same as in other UML diagrams. People or things that interact with the system.

Boundary elements represents interfaces such as GUI's and TUI's.

Control elements connects boundary elements and entity elements. They also implement the logic required to manage elements and their interactions.

Entity elements are often the entities found in the domain model. In Objektorienteret analyse og design (OOAD)they will be classes. However some classes can be control-elements.

2.12. Theoretical probability calculation

2.12.1. Theoretical probability calculation for a die

The frequency of a certain value being rolled was calculated by taking the number of possible combinations that could result in that value being rolled divided by the total number of combinations multiplied by 100%.

$$\frac{Tally}{observations} * 100\%$$

Example

The total number of observations (total tally) was 6

So the Frequency of rolling any value were: $\frac{1}{6}$*100%=16,667%

The mean value was calculated by the formula: $\overline{x} = \sum_{i=1}^{k} f_i * x_i$

$$\overline{x} = 0,1667 * 1 + 0,1667 * 2 + 0,1667 * 3 + 0,1667 * 4 + 0,1667 * 5 + 0,1667 * 5 = 3,5$$

The variance was calculated by the formula: $var(x) = \sum_{i=1}^{k} f_i(x_i - \overline{x})^2$

$$var(x) = 0,167 * (1-3,5)^2 + 0,167 * (2-3,5)^2 + 0,167 * (3-3,5)^2 + 0,167 * (4-3,5)^2 +$$
$$0,167 * (5-3,5)^2 + 0,167 * (6-3,5)^2 = 2,922$$

To get the spread over 60000 rolls the variance was multiplied by 60000.

$$v = 2,922 * 60000 = 175320$$

Then the spread was calculated by the formula: $\sigma(x) = \sqrt{v}$

$$\sigma(x) = \sqrt{175320} = 418,71 \approx 419$$

Our calculations was done using statistics. The decision to test over 60000 rolls is due to the nature of statistic as the spread becomes relatively smaller as the number of rollers increase. This in turn increases the precision of the test. The probability of rolling each value was calculated to 16,67% with a spread of 419 over 60000 rolls.

2.12.2. Theoretical probability calculation for 2 dice

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

The frequency of a certain value being rolled was calculated by the formula: $\frac{Tally}{observations} * 100\%$

Example

The total number of observations (total tally) was 6*6= 36

Combinations that gave the value 4 (tally for the value 4) was 3

So the Frequency of rolling the value of 4 was: $\frac{3}{36}$*100%=8,334%

The mean value was calculated by the formula: $\overline{x} = \sum_{i=1}^{k} f_i * x_i$

$$\overline{x} = 0,028 * 2 + 0,056 * 3 + 0,083 * 4 + 0,111 * 5 + 0,139 * 6 + 0,167 * 7 + 0,139 * 8 + 0,111 * 9 +$$
$$0,083 + 0,056 + 0,028 * 12 = 6,91 \approx 7$$

The variance was calculated by the formula: $var(x) = \sum_{i=1}^{k} f_i(x_i - \overline{x})^2$

$$var(x) = 0,028 * (2-7)^2 + 0,056 * (3-7)^2 + 0,083 * (4-7)^2 + 0,111 * (5-7)^2 + 0,139 * (6-7)^2 +$$
$$0,167 * (7-7)^2 + 0,139 * (8-7)^2 + 0,111 * (9-7)^2 + 0,083 * (10-7)^2 + 0,056 * (11-7)^2 + 0,028 * (12-7)^2$$

$= 5,852$

Then the spread was calculated by the formula: $\sigma(x) = \sqrt{v}$

Over 60000 rolls the spread was.

$\sigma(x)\sqrt{5,852 * 60000} = 592,55 \approx 593$

Value	Tally	Frequency	$\overline{x}$	var(x)	$\sigma(x)$ over 60000 rolls	Expected observations
2	1	2,778%				1667
3	2	5,556%				3334
4	3	8,333%				5000
5	4	11,110%				6666
6	5	13,889%				8333
7	6	16,667%				10000
8	5	13,889%				8333
9	4	11,110%				6666
10	3	8,333%				5000
11	2	5,556%				3334
12	1	2,778%				1667
Total	36	100%	$6,91 \approx 7$	$5,852 \approx 6$	593	60000

Due to the nature of statistics, it was chosen that the test of the probability of rolling a certain value with two dice would be done over 60000 rolls to achieve a precise result. The probability of rolling each value calculated and presented in the table above in the frequency cullom. From those results it was possible to calculate how many observations was expected, the results were presented in the observations cullom in the table above. Finally the spread was calculated to 593 over 60000 rolls.

2.13. Explanation of concepts

2.13.1. Inheritance

Inheritance is a concept that makes it possible to reuse methods and protected variables in classes. In the Java language we use the word “extends” to specify inheritance. For example: “Class A **extends** Class B”. Though most modern OO-programming languages supports inheritance, and some, inheritance from multiple classes, Java can only inherit from one class at a time. Though multiple classes can inherit from the same class. The class that inherits another class is called it’s sub- or child class. The class inherited from is called the super- or parent class.

The general idea of inheritance is to specify and generalize classes. For example, we could say that a specified class Horse “is-a” generalised class Animal. This idea of specifying and generalising is essential for understanding and effectively using class-inheritance.

2.13.2. Abstraction

When we work with inheritance, we generally use it because we want multiple classes to be able to use the same methods. Abstract classes assembles the methods and variables that is shared with all subclasses of the abstract class. Everything that is specific for a subclass is implemented differently in each subclass. If some method or variable is true for all subclasses it can be defined in the abstract class.

For example: A Bicycle class and a Motorcycle class have their own “wheelDiameter” variable with two different values. But a “numberOfWheels” variable can be implemented in the abstract class, because the variable value is shared between the two subclasses.

Abstract classes cannot be instantiated and as such cannot create an object. You can only use an abstract class by inheritance. If a subclass has their own “version” of a method that is declared in the abstract class it overrides the abstract class.

2.13.3. Polymorphism

The general concept of polymorphism in programming can be described as reusing the names of variables, methods, and fields. There exists multiple types of polymorphism:

- **Static polymorphism:** An example of static polymorphism would be method overloading. It's possible to declare multiple methods in the same class, or between a superclass and its subclass, with the same name but with different parameters. Java knows that the methods are different, because of this, even though they have the same name. Static polymorphism is done in compile time when running the program. This is also called method overloading.
- **Dynamic polymorphism:** Overriding a method declared in a superclass with a method in a subclass. Dynamic polymorphism is done in runtime when running the program.
- **Upcasting:** Declaring a new object of a subclass as the type of a superclass. For example: Superclass sup = new Subclass()

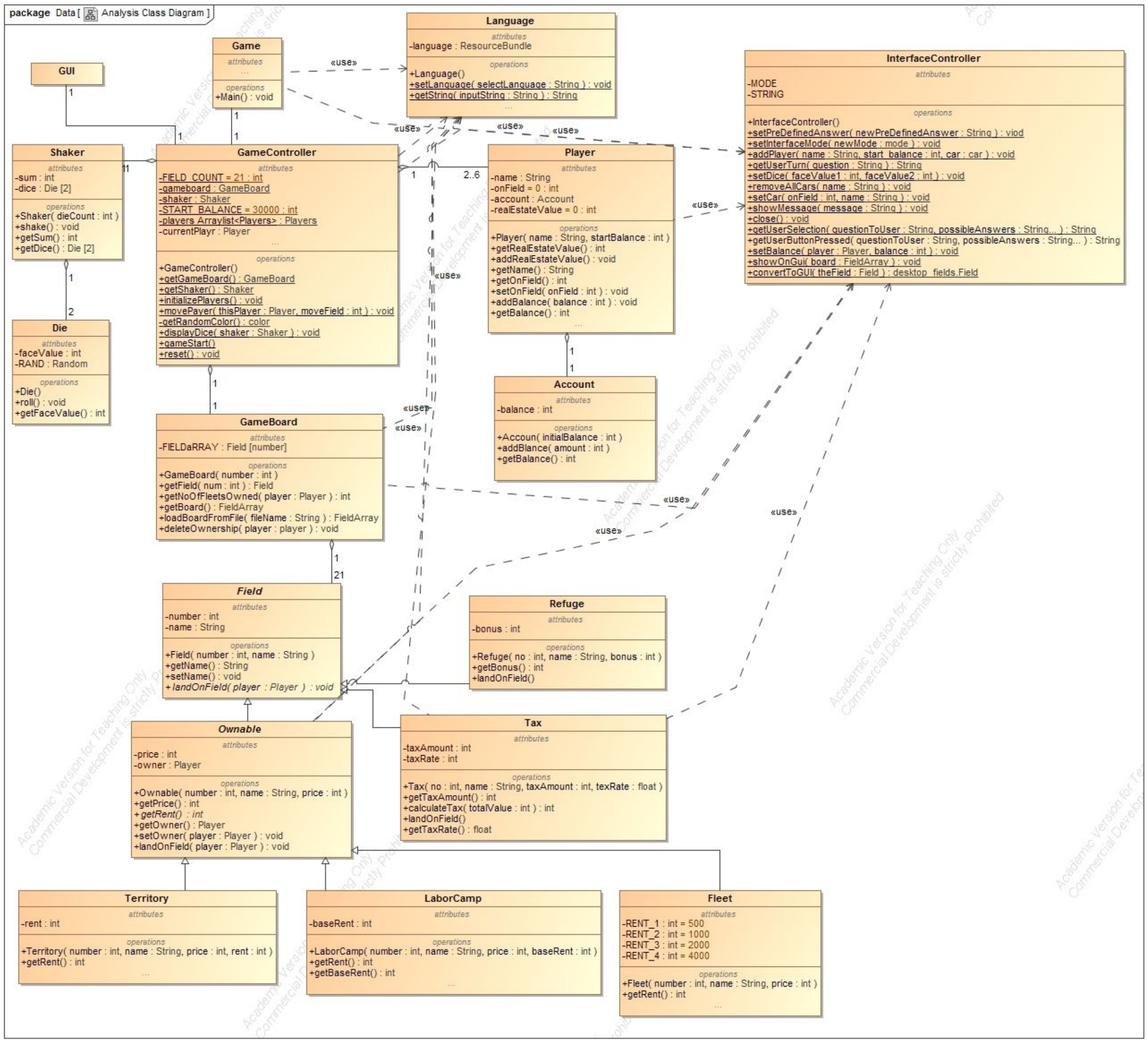
  We can also upcast an object that has already been created.

3. Design

3.1. Design Sequence Diagram

The design sequence diagram shows how the classes interact with each other. Given its size, we chose to add the diagram in separate pdf named “14_del3_DSD”, located in the root folder. The messages between them are displayed as arrows indicating which classes call methods and which returns values. It also shows the flow of time in the program, so which method is used first and last when executing the program. The vertical lines are called life lines and represents the classes in our program. The system sequence allows for graphic representation of the program’s runtime unlike the design sequence diagram which is used to describe the program’s runtime in detail.

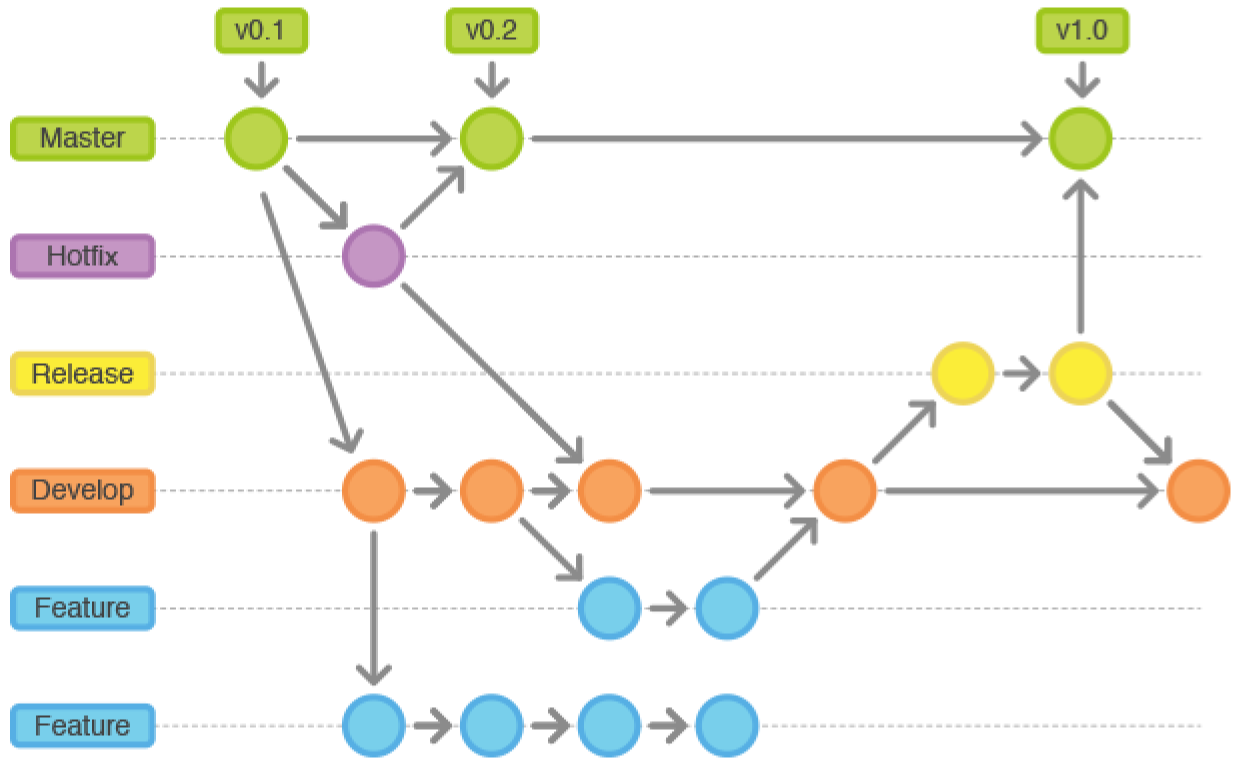
3.2. Design Class Diagram



package Data[Analysis Class Diagram]
GUI
Game
+Main() : void
Language
-language : ResourceBundle
+Language()
+setLanguage(selectLanguage : String) : void
+getString(inputString : String) : String
InterfaceController
-MODE
-STRING
+InterfaceController()
+setPreDefinedAnswer(newPreDefinedAnswer : String) : viod
+setInterfaceMode(newMode : mode) : void
+addPlayer(name : String, start_balance : int, car : car) : void
+getUserTurn(question : String) : String
+setDice(faceValue1 : int, faceValue2 : int) : void
+removeAllCars(name : String) : void
+setCar(onField : int, name : String) : void
+showMessage(message : String) : void
+close() : void
+getUserSelection(questionToUser : String, possibleAnswers : String...) : String
+getUserButtonPressed(questionToUser : String, possibleAnswers : String...) : String
+setBalance(player : Player, balance : int) : void
+showOnGui(board : FieldArray) : void
+convertToGUI(theField : Field) : desktop_fields.Field
Shaker
-sum : int
-dice : Die [2]
+Shaker(dieCount : int)
+shake() : void
+getSum() : int
+getDice() : Die [2]
GameController
-FIELD_COUNT = 21 : int
-gameboard : GameBoard
-shaker : Shaker
-START_BALANCE = 30000 : int
-players Arraylist<Players> : Players
-currentPlayr : Player
+GameController()
+getGameBoard() : GameBoard
+getShaker() : Shaker
+initializePlayers() : void
+movePayer(thisPlayer : Player, moveField : int) : void
-getRandomColor() : color
+displayDice(shaker : Shaker) : void
+gameStart()
+reset() : void
Player
-name : String
-onField = 0 : int
-account : Account
-realEstateValue = 0 : int
+Player(name : String, startBalance : int)
+getRealEstateValue() : int
+addRealEstateValue() : void
+getName() : String
+getOnField() : int
+setOnField(onField : int) : void
+addBalance(balance : int) : void
+getBalance() : int
Die
-faceValue : int
-RAND : Random
+Die()
+roll() : void
+getFaceValue() : int
Account
-balance : int
+Accoun(initialBalance : int)
+addBlance(amount : int)
+getBalance() : int
GameBoard
-FIELDaRRAY : Field [number]
+GameBoard(number : int)
+getField(num : int) : Field
+getNoOfFleetsOwned(player : Player) : int
+getBoard() : FieldArray
+loadBoardFromFile(fileName : String) : FieldArray
+deleteOwnership(player : player) : void
Field
-number : int
-name : String
+Field(number : int, name : String)
+getName() : String
+setName() : void
+landOnField(player : Player) : void
Refuge
-bonus : int
+Refuge(no : int, name : String, bonus : int)
+getBonus() : int
+landOnField()
Ownable
-price : int
-owner : Player
+Ownable(number : int, name : String, price : int)
+getPrice() : int
+getRent() : int
+getOwner() : Player
+setOwner(player : Player) : void
+landOnField(player : Player) : void
Tax
-taxAmount : int
-taxRate : int
+Tax(no : int, name : String, taxAmount : int, texRate : float)
+getTaxAmount() : int
+calculateTax(totalValue : int) : int
+landOnField()
+getTaxRate() : float
Territory
-rent : int
+Territory(number : int, name : String, price : int, rent : int)
+getRent() : int
LaborCamp
-baseRent : int
+LaborCamp(number : int, name : String, price : int, baseRent : int)
+getRent() : int
+getBaseRent() : int
Fleet
-RENT_1 : int = 500
-RENT_2 : int = 1000
-RENT_3 : int = 2000
-RENT_4 : int = 4000
+Fleet(number : int, name : String, price : int)
+getRent() : int
«use»
1
2..6
21
2

The design class diagram is an exact representation of the program. It is more detailed than the domain model since it keeps all information needed to code the program from scratch. It it used to explain the program more visually than looking in the source code without losing information about the code.

3.3. Version control and git-strategy



[Source: https://leanpub.com/site_images/git-flow/git-workflow-release-cycle-4maintenance.png]

The strategy depicted above has been chosen for our use of Github. The following types of branches were used.

Master was the first and final branch. All commits to the Master branch were done when a version of the program was ready to be tested thoroughly. The only branch that was allowed to be merged with the Master branch was the Release branch. So the Master branch only contained versions that was "ready to ship".

Hotfix was used to fix bugs found in the tests done in the Master branch. The one point in the Hotfix branch shown above covers over several back and forth movements between the Release and Hotfix branches. When all known bugs were fixed the changes were committed to the Master branch.

Release marks a feature freeze. So when this branch is created, no new features can be added to this version. All work that was done on the Release branch is to make sure the program works as intended and contained no known bugs.

Develop branch was where the main workflow was collected. Every time a feature was done it was added to this branch. It was also the only branch that could be split into a Release branch.

Feature branches were where all the code's features were made. It held the classes that were created and all the later changes done to said classes. All programmers made their own branch when they started working on a new part of the program. Once a class or feature was done the branch was merged with the Develop branch.

Versions: Two numbers separated by a punctuation mark were used ex. v.2.5. The first number marks the version and increments every time a new Release branch is created. The second number marks the amount of fixes done and increment every time a Hotfix branch is merged with the Release branch.

Versions in classes are marked in the header of the class. The version is incremented every time changes were made.

3.4. Class responsibility

3.4.1. Game folder

Class	Responsibility	Connections
Die	Generate random value in the range 1-6.	Shaker.

Class	Responsibility	Connections
Shaker	Create two die. Roll the die. Calculate the sum of the dice.	Die. GameController.

Class	Responsibility	Connections
Account	Keeping track of the players points. Changing the player’s points.	Player.

Class	Responsibility	Connections
Player	Storing the players name. Determining if it is the player's turn. Creating an account object.	Account. GameController.

Class	Responsibility	Connections
Tax	Specialises a field, so it contains a price the player must pay.	Field

Class	Responsibility	Connections
Refuge	To transfer an amount of money to a player.	Field

Class	Responsibility	Connections
Territory	Specialises ownable with a specific rent	Ownable

Class	Responsibility	Connections
Fleet	Specialises ownable with a specific rent	Ownable

Class	Responsibility	Connections
Labour camp	Specialises ownable with a specific rent	Ownable

Class	Responsibility	Connections
Ownable	Gives a Field the ability to store the owner of the Field	Field Fleet Labour camp Territory

Class	Responsibility	Connections
Board	Encapsulates the fields	Field. GameController.

Class	Responsibility	Connections
GUI	Handling input from the actors.	GameController.

Class	Responsibility	Connections
GameController	Creates the gameboard. Creates the players. Handles input from GUI. Handles win condition. Controls the flow of the game	GUI. Game. Player. Shaker.

Class	Responsibility	Connections
Language	1. Changing the language of the GUI.	

Class	Responsibility	Connections
InterfaceController	1. Controls the mode of the program “test or play”. 2. Works as a wall between the game model and the GUI.	

3.4.2. Test folder

Class	Responsibility	Passed
DieTest	1. Tests if the probability of rolling each value is the same.	Yes

Class	Responsibility	Passed
ShakerTest	1. Tests if the probability of rolling each value is the same. 2. Tests if the sum is calculated correctly.	Yes

Class	Responsibility	Passed
AccountTest	1. Tests if a value can be add to the account, including negative value. 2. Tests if the balance can be returned.	Yes

Class	Responsibility	Passed
PlayerTest	1. Tests it if the real estate value can be set and returned. 2. Tests if the player's account can be returned. 3. Tests if the balance of the account can be changed and returned.	Yes

	4. Tests if the player's name can be returned. 5. Tests if the player’s placement on the gameboard can be returned. 6. Tests if the player is able to store the value of the houses owned. 7. Tests if we can get the name of the player. 8. Tests if we can get the index of the field the player is currently on.	

Class	Responsibility	Passed
GameTest	1. Tests how long it takes for the game to be ready after launching it.	Yes

Class	Responsibility	Passed
FleetTest	1. Tests if the entities are created. 2. Tests if we can get/set an owner. 3. Tests if we can get/set a name. 4. Tests if we can get the rent. 5. Tests the land on field and when the player buys a field.	Yes

Class	Responsibility	Passed
LaborCampTest	1. Tests if the entities are created. 2. Tests if we can get/set an owner.	Yes

	3. Tests if we can get/set a name. 4. Tests if we can get the rent. 5. Tests the land on field method.	

Class	Responsibility	Passed
LaungageTest	1. Checks the ability to set a language and what happens when you forget to set a language.	Yes

Class	Responsibility	Passed
TerritoryTest	1. Tests if the entities are created. 2. Tests if we can get/set an owner. 3. Tests if we can get/set a name. 4. Tests if we can get the rent. 5. Tests the land on field method.	Yes

Class	Responsibility	Passed
RefugeTest	1. Tests if the entities are created. 2. Tests if the player get the actual bonus, when landing on the field. 3.	Yes

Class	Responsibility	Passed
TaxTest	1. Tests if the tax amount can be returned. 2. Tests if we can get/set a name.	Yes

	3. Tests if the tax rate can be returned. 4. Tests if “points” are transferred, from the player who landed on the fields account.	

3.5. Documentation for use of GRASP

GRASP is an abbreviation of General Responsibility Assignment Software Patterns (or Principles). We abide by these principles by dividing the system into specialized classes, thereby keeping their cohesion high. By making a controller class we assign the responsibility of dealing with system events to a non-UI class. There is minimal coupling as the only class with more than two connections is the GameController class. The Player creates an Account and only draws information from Account and gives it to the Controller. Most of the classes does not have much responsibility outside their main purpose/methods. This can be seen on the design class diagram, class responsibility chapter, as well as the code itself.
The patterns:

1. The controller pattern can be directly correlated to our GameController class. It deals with system events separating the UI from the model.
2. The creator pattern describes that the entity that contains or has the most interactions with an object or instance should create it. This pattern was used when creating the Shaker/Die classes and Player/Account classes.
3. High cohesion is a pattern that keeps objects understandable and specified. This means that all related tasks should be collected in one class or one class with subclasses thereby making the code easy to maintain and understand.
4. The indirection pattern is used in our program with the model-view-controller pattern. The actor sees the View then uses the Controller to manipulate the Model. In our case the actor uses the GameController, which manipulates the other classes then shows the results in the GUI.
5. Information expert is a principle where you assign responsibilities to methods. An example of how we used this, was to set the getField in the GameBoard class instead of the Field class since it is the class with the most information, required to complete the method.

6. Low coupling lowers the dependency between classes and reduces the impact on other classes when changing one class. This was done by designing the program in a way so any class had as little aggregation associations as possible and having individual people make the different classes meaning the programmers were forced to make their class as closed as possible.
7. The polymorphism principle means using polymorphic operations. This was used in the Field class and all the subclasses that inherit from it. Thereby reducing the complexity and repetition of the code. Polymorphism was explained in detail in chapter 2.13.3.
8. The protected variations pattern protects elements by using polymorphism to create various implementations of this interface. Our GUI does not change when some variations occur in objects or subsystems behind the logic of the GameController.
9. A pure fabrication is a class that does not belong to the domain model and is made to achieve low coupling, high cohesion and reuse potential. This pattern was used in the language class, which serves to control the the language of the GUI. The same result could be achieved by implementing the function i several classes, but by separating this function out, greater cohesion and lower coupling was achieved.

3.6. Gson

GSON is a Java library that can convert a JSON string into Java objects. It's an open-source project by google. JSON is a data-interchange format. It's a filetype which allows people to read and edit the strings in the code without much knowledge of the actual code and it's easy to parse and generate for the computer. The GSON library basically translates the text file and we can then call it from the program. In our project we use GSON to allow all flavour text and descriptions to be easily translated. GSON can read from all datatypes so all we had to do was create a text file and have GSON read it.

3.7. Internationalizing the game

By using properties files our project can then be correctly translated by someone who has no knowledge about the code or how to code and simply has to run through the individual lines. Also it makes the code more compact since all the characters from the toStrings have been moved to a different file and is simply called from within the program.

3.8. Version management

This program was produced on Windows 10 x64 OS with the IntelliJ IDEA version 2016.3 and GitKraken version 1.9.1 using the Java SE development kit 8u112. We included the libraries gson version 2.8.0, hamcrest-core version 1.3, junit version 4.12. The implementation also involved the GUI that we received from DTU.

3.9. The toString-method

We disregarded the use of the toString method, as we deemed it more time consuming and less intuitive code-wise. We could have returned a string that describes what happens in every turn, through a toString method. Instead we deemed it more convenient to return such strings directly in the methods that need them.

4. Test

4.1. Test strategy

We have in this project given each of the group members individual classes to create however since we also have to make test classes of each class we made, every member tests a class that they didn't make. This way the programmers have little partiality for their classes and therefore can make more critic and in-depth tests. By reducing the bias for our code pieces we make better and more thorough tests. Other than making the tests objectively better we also force the members to enhance their understanding of the code and the specific individual classes before they eventually review them in the classes' absolute final stage. We will continue this practise in future projects.

4.2. Acceptance test

The acceptance test was chosen to document if the program fulfilled the functional requirements. The acceptance test that can be found in the source code's test folder is not complete. The structure of the code prevents an example of a play through of the game, from being made. This design flaw was found so late in the process that a correction could not be completed before the due date. The level of detail is also less than a fully dressed acceptances test for instance: When the permutations of how the fleets could be owned was tested, only the different ways one player could own them was actually tested. Here, all the combinations of ownership should have been tested. Due to time restriction it was decided that some of the requirements that were tested in the different classes JUnit tests would not be replicated in the acceptance test.

4.3. Manual tests

4.3.1. Positive test

The programme only have one input field that accepts a string, the functionality of which work as expected, with the exceptions of the situations described in the negative test chapter.

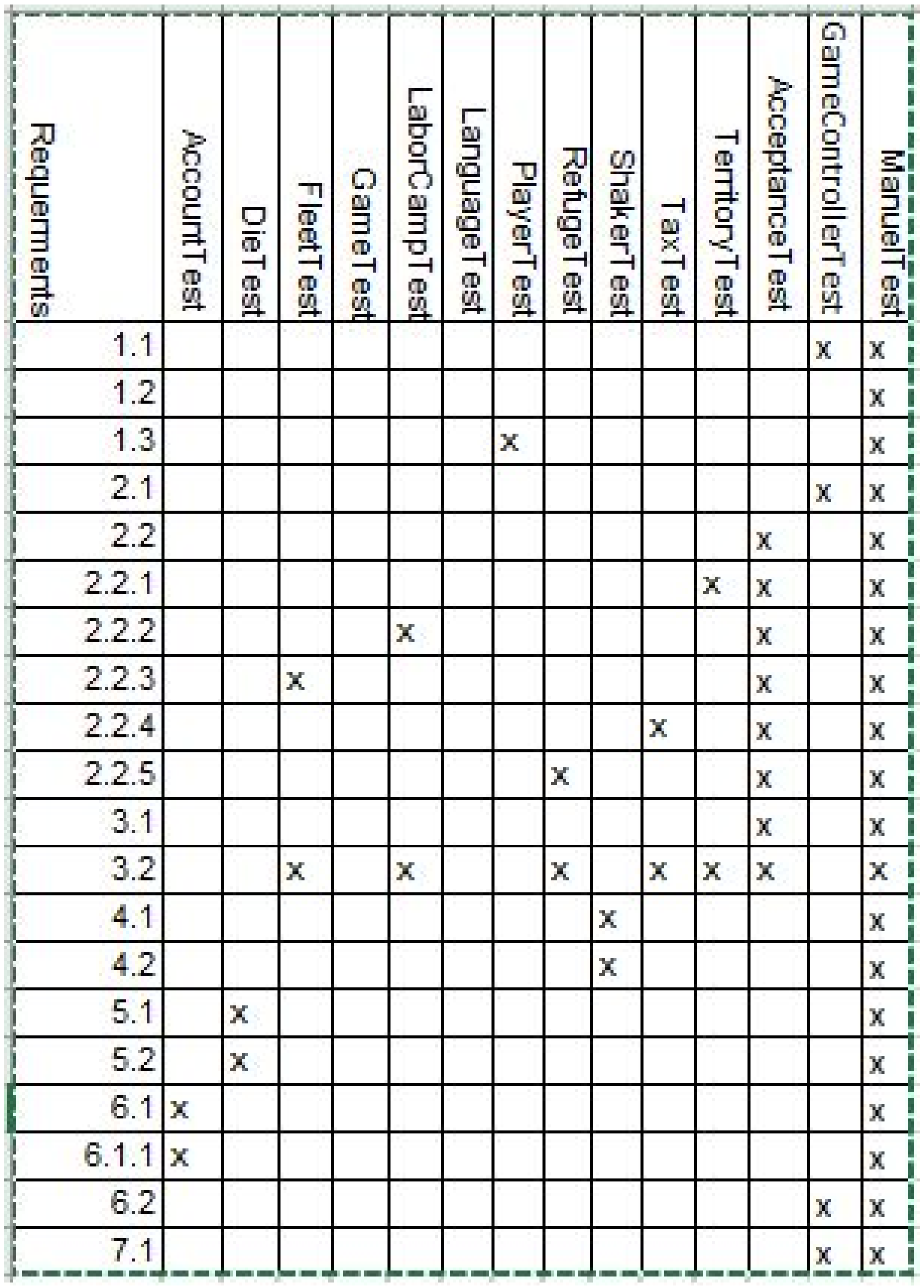
4.3.2. Negative test

1. If the player's name is longer than 17 characters the full name will not be shown in the GUI.
2. If more than one player has the same name only one player will be created and the creation process of the duplicate will be skipped.

4.3.3. Boundary test

Since there's no integer or float values that the player can manipulate the boundary test will be shortened to the String value that holds the player's inputted name. The negative test already covers these inputs. Other than that there's the possibility of an overflow error in the balance. This is statistically impossible and would take years considering that the integer in question would have to reach 2147483647. Since there's virtually no chance of this happening we have not taken any precautions or countermeasures.

4.3.4. Test coverage analysis



Requerments	AccountTest	DieTest	FleetTest	GameTest	LaborCampTest	LanguageTest	PlayerTest	RefugeTest	ShakerTest	TaxTest	TerritoryTest	AcceptanceTest	GameControllerTest	ManuelTest
1.1													x	x
1.2														x
1.3							x							x
2.1													x	x
2.2												x		x
2.2.1											x	x		x
2.2.2					x							x		x
2.2.3			x									x		x
2.2.4										x		x		x
2.2.5								x				x		x
3.1												x		x
3.2			x		x			x		x	x	x		x
4.1									x					x
4.2									x					x
5.1		x												x
5.2		x												x
6.1	x													x
6.1.1	x													x
6.2													x	x
7.1													x	x

The table above documents where and how the functional requirements have been tested. It clearly shows that all requirements except 1.2 have been tested at least twice. First automatically by the black box JUnit tests and then manually. The exceptions to this is the acceptance test which is both white and black box test. There is no JUnit test of the GameBoard class as it was tested through the acceptance test and field test classes.

4.4. Import to eclipse

To Import:

1. Extract the folder to a location of preference
2. Open eclipse
3. Goto File > Import > General and choose Projects from Folder or archive
4. Click next
5. Press the Directory button and choose the location you extracted the project folder to.

To run:

6. Right-click on the imported project and choose Run as and then choose Java Application.

To test:

7. Right-click on the imported project and choose Run as and then choose JUnit Test

4.5. Run the game

1. Doubleclick the “start game.bat” file in the root of the extracted archive.

4.6. Results

As seen on the screenprint below, all the automated tests (test classes) has passed. The manual tests run throughout the process of this assignment has shown that our program functions as intended. The manual test also followed the main flow of the use case and all alternate flows were run without errors showing that the program doesn’t have any game breaking bugs or errors.

4.7. Requirements fulfillment

Requirements	Documentation for fulfillment
8.1.	The provided GUI was used as documented in the design class diagram and the source code.
8.2.	How the Fields are arranged can be seen in the GameController line 21 to 22.
8.3.	Not fulfilled. (See the toString section 3.9.)
8.4.	The program was run on the Windows operating system throughout all tests. This is documented in the different Test paragraphs.
8.5.	Since the computers in the databar on DTU campus Lyngby uses Windows OS, the program can run on the databar computers.
8.6.	We have used inheritance as according to the assignment which can be seen in the class diagrams.
8.7.	-
8.7.1.	The requirements specification artifact is documented at section 2.1.
8.7.2.	The use-case diagram is documented at section 2.5.
8.7.3.	The fully dressed use-case is documented at section 2.6.
8.7.4.	The domain model is documented at section 2.4.
8.7.5.	The robustness diagram is documented at section 2.11.
8.7.6.	The design sequence diagram is documented at section 3.1.

8.7.7.	The explanation of the concept of inheritance is documented at section 2.13.1.
8.7.8.	The explanation of the concept of an abstract class is documented at section 2.13.2.
8.7.9.	The explanation of the concept of polymorphism is documented at section 2.13.3.
8.7.10.	The documentation of the tests with screenshots is documented at section 4.6.
8.7.11.	The documentation of the use of GRASP is documented at section 3.5.
8.8.	The program was run by fellow students with basic knowledge of computers and the capability to push a button.
8.9.	Everything is written in English.
8.10.	The program is written in UTF-8 as can be seen in most IDEs when loading the program.
8.11.	The project is named 14_del3 as can be seen on the front page.
8.12.	The test classes are in another package than the game as can be seen in the program folder.
8.12.1.	-
8.12.2.	-
8.13.	All work on the code has been committed regularly which can be seen on GitHub.
8.13.1.	It was done at least once an hour. Se github
8.13.2.	It was done every time a smaller objective was reached. Se github.

8.14.	The program was imported to Eclipse which is documented at section 4.4.
8.15.	All tests can be found in the source code.
8.15.1.	-
8.15.2.	-
8.15.3.	-
8.16.	There is a declaration which describes the minimal computer specifications required to run the program and what programs need to be installed prior to running the program on page section. 2.2.

5. Conclusion

The assignment was completed to the required specifications as shown by the results and requirements fulfillment chapters. The tests run on and in the program were successful and showed little to no errors while it was under development.

We mostly overcame any major hurdles with proper file management, enabling us to effectively backtrack to earlier builds that didn’t include whichever error was displayed. This time we also utilized branch management following a specific model we agreed upon early in the planning phase, but we did have some problems with updating the files to the correct versionnumber, and thought about this too late into the project.

As shown in the report we also abided to the GRASP principles and followed the Unified Process model, and applied it to the project structure & Development. We had plans of a second iteration featuring flavour text and the ability to sell fields already bought.

The program runs smoothly and the report features all the models we’ve had to use to show the process and explain the program, so overall we are really happy about our finished product.

6. Appendix

6.1. Appendix 1 CDIO del opgave 3

Indledning

IOOuterActive har fået endnu en opgave.

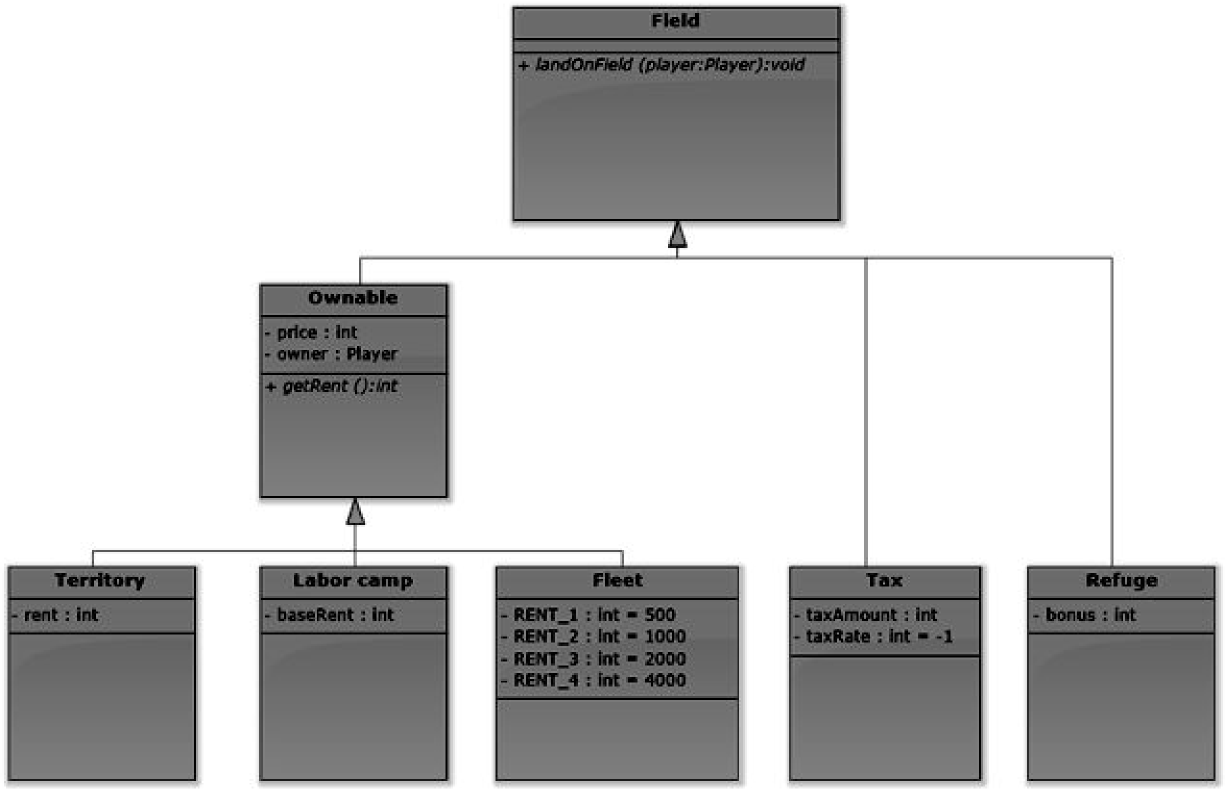
Kundens vision:

Nu har vi terninger og spillere på plads, men felterne mangler stadig en del arbejde. I dette tredje spil ønsker vi derfor at forrige del bliver udbygget med forskellige typer af felter, samt en decideret spilleplade.
Spillerne skal altså kunne lande på et felt og så fortsætte derfra på næste slag. Man går i ring på brættet.
Der skal nu være 2-6 spillere.
Man starter med 30.000.
Spillet slutter når alle, på nær én spiller, er bankerot.

I bilag kan I se en oversigt over de felter vi ønsker, samt en beskrivelse af de forskellige typer.

Projektlederens bemærkninger:

Husk FURPS+.



Forslag til klassediagram med nedarvning.
I kan selvfølgelig godt huske at når noget står i kursiv så er det abstract.

Det er fuldt ud lovligt at ændre på attributterne, men ikke på metoderne og heller ikke på forholdet imellem klasserne. Det er selvfølgeligt tilladt at tilføje både attributter og metoder. Hvis man har en meget god grund til at ville ændre på metoderne kan projektlederen måske give lov til det.

Husk GRASP.

Jeg forventer følgende artefakter:

Krav / Analyse

- Kravspecifikation
- Use-case diagram
- Use-case beskrivelser. Som minimum skal ”Land on fleet” være fully dressed. Gerne flere.
- Domæne-model/BCE

Kode

- Lav passende konstruktører.
- Lav passende get og set metoder.
- Lav passende toString metoder.
- Lav en klasse GameBoard der kan indeholde alle felterne i et array.
- Tilføj en toString metode der udskriver alle felterne i arrayet.
- Lav det spil kunden har bedt om med de klasser I nu har.
- Benyt GUI'en.

Design-dokumentation

- DSD'er (Design Sekvens Diagrammer). Som minimum vil jeg se ”Land on fleet”.
- Forklar hvad arv er.
- Forklar hvad abstract betyder.
- Fortæl hvad det hedder at alle fieldklasserne har en landOnField metode der gør noget forskelligt.
- Dokumentation for test med screenshots.
- Dokumentation for overholdt GRASP.

Test

Lav tre testcases med tilhørende testscripts og testrapporter.
Lav en Junit test til hver af felttypernes ”landOnField”-metode. Den første test er i bilag 2.

Versionsstyring

Lav et lokal Git-repository som en del af Eclipeprojektet.
Alternativt kan afleveres et link til et repository på nettet eks.

https://github.com/cbudtz/TestRepo42.git.

Rapporten skal indeholde en vejledning i hvordan man importerer Get-repository i Eclipse.

Konfigurationsstyring

Udviklingsplatformen er alt det software i bruger under udviklingen af jeres projekt.
Produktionsplatformen er alt det software der skal bruges til at køre jeres færdige program. I dette projekt er de ens. I skal dokumentere platformens dele med versionsnummer så den kan genskabes til senere brug. Jeres platform består af operativsystem, java, og Eclipse samt biblioteket GUI.jar.
Endvidere skal i beskrive hvordan man importerer jeres projekt i Eclipse fra git, og hvordan man kører jeres program.

Bilag 1

Feltliste:

1.	Tribe Encampment	Territory	Rent 100	Price 1000
2.	Crater	Territory	Rent 300	Price 1500
3.	Mountain	Territory	Rent 500	Price 2000
4.	Cold Desert	Territory	Rent 700	Price 3000
5.	Black cave	Territory	Rent 1000	Price 4000
6.	The Werewall	Territory	Rent 1300	Price 4300
7.	Mountain village	Territory	Rent 1600	Price 4750
8.	South Citadel	Territory	Rent 2000	Price 5000
9.	Palace gates	Territory	Rent 2600	Price 5500
10.	Tower	Territory	Rent 3200	Price 6000
11.	Castle	Territory	Rent 4000	Price 8000
12.	Walled city	Refuge	Receive 5000	
13.	Monastery	Refuge	Receive 500	
14.	Huts in the mountain	Labor camp	Pay 100 x dice	Price 2500
15.	The pit	Labor camp	Pay 100 x dice	Price 2500
16.	Goldmine	Tax	Pay 2000	
17.	Caravan	Tax	Pay 4000 or 10% of total assets	
18.	Second Sail	Fleet	Pay 500-4000	Price 4000
19.	Sea Grover	Fleet	Pay 500-4000	Price 4000
20.	The Buccaneers	Fleet	Pay 500-4000	Price 4000
21.	Privateer armade	Fleet	Pay 500-4000	Price 4000

Typer af felter:

- **Territory**
  - Et territory kan købes og når man lander på et Territory som er ejet af en anden spiller skal man betale en afgift til ejeren.
- **Refuge**
  - Når man lander på et Refuge får man udbetalt en bonus.
- **Tax**
  - Her fratrækkes enten et fast beløb eller 10% af spillerens formue. Spilleren vælger selv mellem disse to muligheder.
- **Labor camp**
  - Her skal man også betale en afgift til ejeren. Beløbet bestemmes ved at slå med terningerne og gange resultatet med 100. Dette tal skal så ganges med antallet af Labor camps med den samme ejer.
- **Fleet**

- Endnu et felt hvor der skal betales en afgift til ejeren. Denne gang bestemmes beløbet ud fra antallet af Fleets med den samme ejer, beløbene er fastsat således:
  1. Fleet: 500
  2. Fleet: 1000
  3. Fleet: 2000
  4. Fleet: 4000

Bilag 2 - Eksempel på JUnit test

```
import model.Player;

import org.junit.*;

public class RefugeTest {
	private Player player;
	private Field refuge200;
	private Field refuge0;
	private Field refugeNeg200;

	@Before
	public void setUp() throws Exception {
		this.player = new Player("Anders And", 1, 1000);
		this.refuge200 = new Refuge(1, "Helle +200", 200);
		this.refuge0 = new Refuge(2, "Helle 0", 0);
		this.refugeNeg200 = new Refuge(3, "Helle -200", -200);
	}

	@After
	public void tearDown() throws Exception {
		this.player = new Player("Anders And", 1, 1000);
		//The fields are unaltered
	}

	@Test
	public void testEntities() {
		Assert.assertNotNull(this.player);

		Assert.assertNotNull(this.refuge200);
		Assert.assertNotNull(this.refuge0);
		Assert.assertNotNull(this.refugeNeg200);

		Assert.assertTrue(this.refuge200 instanceof Refuge);
		Assert.assertTrue(this.refuge0 instanceof Refuge);
		Assert.assertTrue(this.refugeNeg200 instanceof Refuge);
	}

	@Test
	public void testLandOnField200() {
		int expected = 1000;
		int actual = this.player.getBalance();
		Assert.assertEquals(expected, actual);

		//Perform the action to be tested
		this.refuge200.landOnField(this.player);

		expected = 1000 + 200;
		actual = this.player.getBalance();
```

```
        Assert.assertEquals(expected, actual);
    }
    @Test
    public void testLandOnField200Twice() {
        int expected = 1000;
        int actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);

        //Perform the action to be tested
        this.refuge200.landOnField(this.player);
        this.refuge200.landOnField(this.player);

        expected = 1000 + 200 + 200;
        actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);
    }

    @Test
    public void testLandOnField0() {
        int expected = 1000;
        int actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);

        //Perform the action to be tested
        this.refuge0.landOnField(this.player);

        expected = 1000;
        actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);
    }
    @Test
    public void testLandOnField0Twice() {
        int expected = 1000;
        int actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);

        //Perform the action to be tested
        this.refuge0.landOnField(this.player);
        this.refuge0.landOnField(this.player);

        expected = 1000;
        actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);
    }

    @Test
    public void testLandOnFieldNeg200() {
        int expected = 1000;
        int actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);

        //Perform the action to be tested
        this.refugeNeg200.landOnField(this.player);

        //It is not possible to deposit a negative amount
        expected = 1000;
        actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);
    }
    @Test
    public void testLandOnFieldNeg200Twice() {
```

```
                int expected = 1000;
                int actual = this.player.getBalance();
                Assert.assertEquals(expected, actual);

                //Perform the action to be tested
                this.refugeNeg200.landOnField(this.player);
                this.refugeNeg200.landOnField(this.player);

                //It is still not possible to deposit a negative amount
                expected = 1000;
                actual = this.player.getBalance();
                Assert.assertEquals(expected, actual);
        }
}
```

6.2. Appendix 2 Word analysis (Danish)

Ord analyse

Vi er givet tre stykker tekst. Det ene er “Projektlederens bemærkninger” og det andet “Kode”. Disse to behøver knapt analyse og nærmere en slags fortolkning.
“Projektlederens bemærkninger” indeholder et klassediagram hvor vi kan se nogle klasser og nogle af deres attributter og metoder. Field klassen skal indeholde en public metode “landOnField”. “Ownable” klassen indeholder en public “getRent” metode og har “owner” (bruger Player) og “price” (bruger int) som attributter. Den arver af “Field” klassen. “Tax” og “Refugee” klasserne arver også direkte fra “Field” klassen. “Tax” indeholder attributterne taxAmount (bruger int) og TaxRate (bruger int) og “Refugee” indeholder “bonus” (bruger int).
“Territory” som arver fra “Ownable” har attributten “rent” (bruger int). “Labour camp” som også arver fra “Ownable” har attributten “baseRent” (bruger int).

I “Kode” stykket står der som følger:

Kode

• Lav passende konstruktører.
• Lav passende get og set metoder.
• Lav passende toString metoder.
• Lav en klasse GameBoard der kan indeholde alle felterne i et array.
• Tilføj en toString metode der udskriver alle felterne i arrayet.
• Lav det spil kunden har bedt om med de klasser I nu har.

Det vi kan lede herfra er, at vi skal bruge toString i koden og ikke “system.out.println” og vores felter i spillet skal skrives i et array i en klasse vi kalder “GameBoard”, som skal have en toString, der udskriver alle felter i arrayet.

Det sidste stykke tekst er kundens vision. Her får vi de informationer vi allerede har kunne udlede fra de andre tekststykker og derfor blev denne sat til sidst. “Spilleplade” kan direkte oversættes til “GameBoard” og de “forskellige typer felter” bliver så vores “Fields”. Dog får vi en information der er kritisk for spillet og ikke beskrevet tidligere. “Spillerne skal kunne lande på et felt og så fortsætte derfra på næste slag." Og at man "går i ring på brætter". Det vil sige der skal være en funktion eller metode, der smider spillerens brik tilbage til første felt og fortsætter med at tælle derfra. Yderligere kriterier i tekst stykket er start balancen skal være 30.000, spillet skal kunne spilles af 2-6 spillere og spillet slutter når alle, på nær én spiller, er bankerot.

6.3. Appendix 3 Time tables

Casper							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
7/11/2016	2					2	4
8/11/2016			2				2
9/11/2016		2				2.5	4.5
10/11/2016		0.5	0.5	5		2	8
11/11/2016				1	2	1	4
14/11/2016				3			3
16/11/2016				1	1	2	4
17/11/2016				4		4	8
18/11/2016						4	4
23/11/2016						4	4
24/11/2016						8	8
25/11/2016						5	5
total	2	2.5	2.5	14	3	34.5	58.5

Michael							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
7/11/2016	2					2	4
9/11/2016		3				1	4
10/11/2016		1	3				4
11/11/2016		1				3	4
14/11/2016		2.5					2.5
16/11/2016				0.5		3.5	4
17/11/2016				3	1	4	8
18/11/2016						4	4
21/11/2016					1	3	4
23/11/2016							0
24/11/2016							0
25/11/2016						1	1
total	2	7.5	3	3.5	2	21.5	39.5

Mathias							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
7/11/2016	2					2	4
8/11/2016		2					2
9/11/2016		3				1.5	4.5
10/11/2016			2	6			8
11/11/2016		1		3			4
14/11/2016				3			3
16/11/2016				4			4
17/11/2016				4	4		8
18/11/2016					4		4
18/11/2016					4		4
21/11/2016							0
23/11/2016					4		4
24/11/2016					8		8
25/11/2016						5	5
total	2	6	2	20	24	8.5	62.5

Rasmus							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
7/11/2016	2					2	4
9/11/2016		1				3	4
10/11/2016		1.5		4	0.5	2	8
11/11/2016		2		2			4
14/11/2016				3			3
16/11/2016				4			4
17/11/2016				4		4	8
18/11/2016				2		2	4
21/11/2016							0
23/11/2016							0
24/11/2016				3	4	1	8
25/11/2016						5	5
total	2	4.5	0	22	4.5	19	52

Lasse							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
7/11/2016	2					2	4
8/11/2016	2						2
9/11/2016		1	2	1			4
10/11/2016		1	2	6			9
11/11/2016			2	6			8
12/11/2016				2			2
14/11/2016				3			3
16/11/2016		1				3	4
17/11/2016				3	3	1	7
18/11/2016					2	2	4
23/11/2016				3	0.5	1	4.5
24/11/2016				7	1		8
25/11/2016						4	4
total	4	3	6	31	6.5	13	63.5

Timothy							
Dato	Requirements	Analysis	Design	Implementation	Test	Documentation	Total
7/11/2016	2					2	4
8/11/2016	1						1
9/11/2016		4					4
10/11/2016		4		4			8
11/11/2016							0
14/11/2016		3					3
16/11/2016		1.5					1.5
17/11/2016							0
18/11/2016			1	4	1	2	8
19/11/2016						2.5	2.5
21/11/2016							0
23/11/2016					4.5		4.5
24/11/2016				2	4	1.5	7.5
25/11/2016			5			2	7
total	3	12.5	1	10	9.5	13	49

6.4. Appendix 4 tasks

Abstract	Casper, Timothy
Abstract in java text	Rasmus
Acceptance test Text	Mathias
Account class	Mathias, Lasse
AccountTest	Mathias, Timothy, Casper
Analysis class diagram	Mathias
Analysis class diagram text	Casper, Michael
Appendix	Mathias
BCE	Rasmus
Bibliography	Casper
Boundary test	Casper
Class responsibility	Lasse
Conclusion	Casper, Timothy
Design Sequence Diagram	Timothy
Design Sequence Diagram text	Casper, Rasmus
Die class	Timothy
DieTest	Mathias, Timothy
Domain model	Timothy
Domain model text	Timothy
Field class	Timothy
Fleet class	Casper, Rasmus
FleetTest	Mathias, Lasse
Flowchart	Michael
Flowchart text	Rasmus, Michael
Frontpage	All
Game class	Rasmus, Lasse
GameBoard class	Mathias, Lasse
GameBoardTest	Mathias
GameController class	Mathias, Lasse
GameControllerTest	Lasse

GameTest	Rasmus
Git strategy	All
Grammar read-through	All
GRASP text	Mathias
Gson text	Casper
Import to Eclipse	Lasse
Inheritance text	Rasmus
InterfaceController	Lasse, Timothy
Introduction	Casper
LaborCamp Class	Casper, Rasmus
LaborCampTest	Mathias
Language class	Rasmus, Lasse
Language properties file	Rasmus, Michael
LanguageTest	Rasmus
Ownable class	Timothy, Rasmus
OwnableTest	Mathias, Lasse
Negative Test	Mathias
Player class	Timothy, Lasse, Michael
PlayerTest	Mathias
Polymorphism text	Rasmus
Positive test	Mathias
Probability calculation for a die	Mathias
Probability calculation for two dice	Mathias
Refuge class	Casper, Rasmus
RefugeTest	Mathias
Requirements fulfillment	Casper
Requirements specification	All
Results	Timothy, Casper, Rasmus
Shaker class	Timothy, Rasmus
ShakerTest	Mathias
Show rules in game	Michael, Lasse
SSD	Timothy
SSD text	Timothy
String document	Lasse

Tax class	Casper, Lasse, Rasmus
TaxTest	Mathias, Timothy
TerritoryTest	Mathias
Territory class	Timothy, Lasse, Michael
Test	All
Test class responsibility	Lasse
Test coverage analysis	Mathias
Test strategy	Michael
Time table and tasks	Mathias
UP text	Casper
Use case diagram	Mathias
Use case text	Mathias, Michael
Version management	Casper
Version strategy	All
Word analysis	Casper
Word analysis resume	Casper

7. Bibliography

1. **IEEE Recommended Practice for Software Requirements Specifications**
   Version Std 830-1998
   ISBN: 0-7381-0332-2, SH94654 (PDF)
   The Institute of Electrical and Electronics Engineers, Inc.
   345 East 47th Street, New York, NY 10017-2394, USA
   Approved by the IEEE-SA Standards Board

2. **Software engineering**
   9th edition
   Author: Ian Sommerville
   Publisher: Addison-Wesley (imprint of Pearson)
   ISBN: 0-13-703515-2 (PDF)

3. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**
   3rd edition
   Author: Craig Larman
   Publisher: Addison-Wesley (imprint of Pearson)
   ISBN: 0-13-148906-2 (PDF)

4. **Java Software Solutions**
   7th edition
   Author: John Lewis & William Loftus
   Publisher: Addison-Wesley (imprint of Pearson)
   ISBN: 0-13-214918-4 (PDF)

5. **Java for dummies**
   4th edition
   Author: Doug Lowe
   Publisher: John Wiley & Sons, Inc.
   ISBN: 978-1-118-40803-2 (pbk) (PDF)

6. **UML Distilled: A Brief Guide to the Standard Object Modeling Language**
   3rd edition
   Author: Martin Fowler
   Publisher: Addison-Wesley (imprint of Pearson)
   ISBN: 078-0321193681

8. Glossary

OO	= Object Oriented
CDIO	= Conceive Develop Implement Operate project
DTU	= Technical University of Denmark
FURPS	= Functionality, Usability, Reliability, Performance and Supportability
(FURPS)+	= Design contraints, Implementation-, Interface- and Physical requirements
UP	= Unified Process
GRASP	= General Responsibility Assignment Software Patterns (or Principles)
SSD	= System Sequence Diagram
UI	= User Interface
GUI	= Graphical User Interface
TUI	= Text-based User Interface
OS	= Operating System
UTF-8	= Universal Transformation Format 8
JRE	= Java Runtime Environment
SP1	= Service Pack 1
UC#	= Use Case (# = any number indexing the different use case)
BCE	= Entity-Control-Boundary diagram
OOAD	= Object-Oriented Analysis and Design
JSON	= JavaScript Object Notation