Ein Bild, das Text, drinnen, grün enthält.

Automatisch generierte BeschreibungDrecksau

DS.PM1, DS21t, Storm Hamsters

Anja Edelmann, Ricky Raths, Lars Schneckenburger, Salah Xaaji

# Einleitung

## Projektziel

Ziel des vorliegenden Projektes ist es, eine Textversion des Kartenspiels «Drecksau» von Kosmos Spiele zu programmieren. Das Spiel soll mit bzw. von 2 bis 4 Spielenden gespielt werden können. In einem zweiten Schritt ist die Spiellogik mit einer weiteren, neu ausgedachten Karte, zu erweitern und im Programm zu implementieren.

## Programm Features

Das Programm soll den geltenden Spielregeln folgen. Die Beschreibung des Spiels, sowie die aktuellsten Regeln sind unter nachfolgendem Link einsehbar: [https://de.wikipedia.org/wiki/Drecksau](https://de.wikipedia.org/wiki/Drecksau(Spiel)).

Folgende Spielkarten und deren Effekte sind im Programm einzubauen, inkl. der neu ausgedachten Spielkarte «Sturm»:

|  |  |
| --- | --- |
| Kartentyp | Effekt |
| Schweinekarte | Kein Effekt – ist zu Beginn ein «Sauberschwein» und soll in eine «Drecksau» umgewandelt werden. |
| Matschkarte | Eine Schweinekarte kann zu einer «Drecksau» umgewandelt werden. |
| Regenkarte | Alle dreckigen Schweine, die keinen Stall aufweisen, werden zu «Sauberschweinen» umgewandelt. |
| Stallkarte | Rüstet ein Schwein mit einem Stall aus, welcher vor Regen schützt. |
| Blitzkarte | Entzündet einen beliebigen Stall und fackelt ihn ab. |
| Blitzableiterkarte | Wird an einem Stall angelegt und schützt vor Blitzschlägen. |
| Sturmkarte | Sämtliche Ställe und Blitzableiter auf dem Tisch werden zerstört und auf den Ablagestapel gelegt. |
| «Bauer-schrubbt-die-Sau»-Karte | Der Bauer wäscht eine «Drecksau» ohne Stall, oder in einem unvernagelten Stall und wandelt es in einen «Sauberschwein» um. |
| «Bauer-ärgere-dich»-Karte | Nagelt einen Stall zu und schützt das Schwein vor dem Bauer. |

Tabelle : Spielkarten und ihre Effekte im Spiel

Neben den Effekten der Spielkarten sind die gängigen Spielfeatures, wie das Ziehen einer Karte, das Tauschen einer oder aller Karten von der Hand sowie das Spielen einer spielbaren Karte, im Programm einzubauen. Den Spielenden soll es möglich sein, durch beliebiges Einsetzen ihrer Handspielkarten, ihre Schweinekarten zu «Drecksauen» umzuwandeln und das Spiel zu gewinnen. Gewonnen hat, wer alle eigenen Schweine in «Drecksauen» umwandeln konnte.

## Anforderungen

* Ein unbekannter Input durch die Spielenden soll einen Fehler generieren.
* Die relevanten Punkte zu Clean-Code sollen angewendet werden.
  + kurz sein
  + nur eine Aufgabe erledigen
  + nur Code auf der gleichen Abstraktionsebene enthalten
  + keinen doppelten Code enthalten
  + beschreibende Namen verwenden
  + nur gewollte, auf das Minimum reduzierte, Seiteneffekte enthalten
  + maximal 4 Argumente enthalten

# User-Story und definition of Done

## User-Story

Ich als Kundschaft brauche eine Textversion des Kartenspiels «Drecksau», damit ich mit 2 bis 4 Spielenden spielen kann.

## Akzeptanzkriterien

* Die Spielenden erhalten Spielkarten und können diese ausspielen.
* Spielkarten mit speziellen Effekten können auf andere Karten angewendet werden.

## Definition of Done

* Die Funktionalitätstest wurden erfolgreich durchgeführt und es wurden keine kritischen Bugs oder Probleme festgestellt.
* Die Usability ist gegeben durch eine klare Programmführung: User wissen zu jedem Zeitpunkt, wo sie sich befinden und welche Menüoptionen Ihnen zur Verfügung stehen.
* Die Usability wurde von der verantwortlichen Person überprüft und bestätigt.
* Die technische Dokumentation des Codes wurde erstellt und vom Product-Owner freigegeben.
* Sämtliche Dateien wurden auf das GitHub-Repository hochgeladen und gesichert.

# Organisation

Nachfolgendes Organigramm zeigt die Projektorganisation des Projektteams. Die darunterliegende Tabelle soll die Rollen und die dazugehörigen Aufgaben genauer erläutern.

Abbildung Organigramm des Projektteams

Abbildung 1: Rollen und Aufgabenverteilung

# Implementation

Für die Implementation wurde in einem ersten Schritt ein UML-Diagramm erstellt, welches im Laufe des Projekts angepasst und ergänzt wurde. Eine grössere Version ist im Anhang beigelegt.

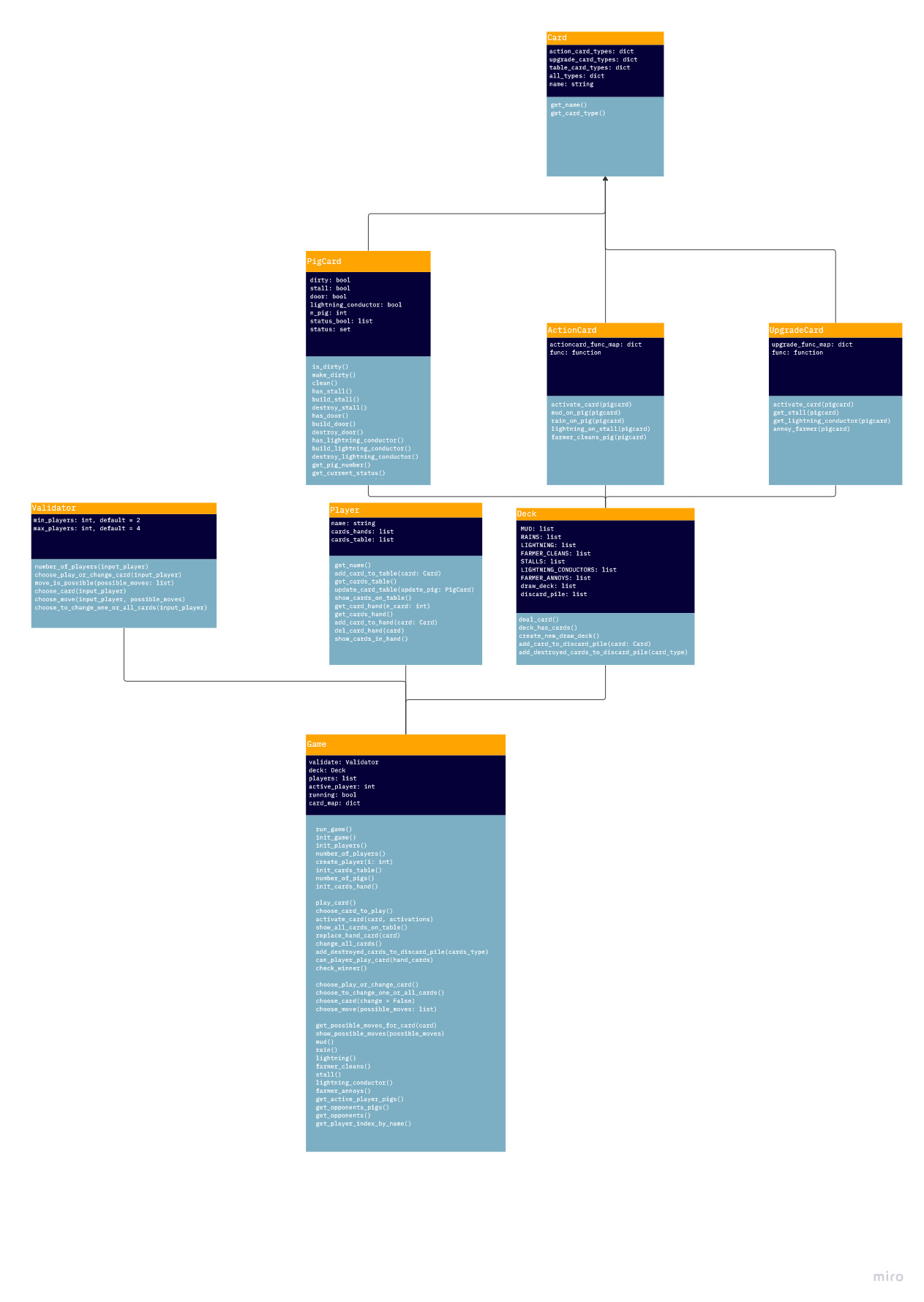


Abbildung 2: UML-Diagramm des Spiels «Drecksau»

Für die Implementierung des Spiels wurde ein objektorientierter Ansatz gewählt. Um vom Spiel auf die einzelnen Klassen schliessen zu können, werden nachfolgend die Spielelemente kurz erläutert und erklärt wie diese im Programm vorgesehen sind.

## Spielelemente

### Spielkarten

Das Spiel «Drecksau» ist ein Kartenspiel, welches die Spielkarten zum zentralen Spielelement macht, siehe auch Tabelle 1. Es werden grundsätzlich nachfolgende Spielkartentypen unterschieden:

* Typ 1 «Schweinekarten»: Die Schweinekarten sind jene Karten, welche gleichmässig unter den Spielern verteilt werden und auf dem Tisch offen gespielt werden.
* Typ 2 «Anlegekarten»: Diese Karten können direkt an die eigenen Schweinekarte angelegt werden. Dazu gehören: Stall, Blitzableiter, Bauer-ärgere-dich.
* Typ 3 «Aktionskarten»: Die Spielenden können diese Karte offen spielen, womit ein gewisser Effekt aktiviert wird. Die Karte wird anschliessend auf den Ablagestapel gelegt.

Zu diesen Kartentypen gehören: Matsch, Regen, Blitz, Bauer-schrubbt-die Sau, Sturm.

### Deck / Ablagestapel

Die Karten vom Typ 2 und 3 werden zu einem Deck gemischt und auf dem Tisch platziert. Bei Spielbeginn werden allen Spielenden jeweils drei Karten ausgegeben. Im Verlaufe des Spiels werden vom Deck Karten gezogen und gespielte Karten werden auf einem Ablagestapel neben dem Deck platziert.

### Klassenaufteilung

Die Grundidee ist, für die einzelnen Spielelemente eine Klasse zu definieren, welche obiges abbildet. Für die Spielkarten ist es naheliegend eine Oberklasse zu erstellen, und mittels Vererbung die einzelnen Spielkartentypen abzuleiten. Zudem werden für die Spielenden und auch für das Spiel jeweils eine Klasse benötigt. Letzteres bildet zudem die Hauptklasse ab, womit das Spiel gespielt werden kann. Nicht zuletzt wird eine Klasse definiert, welche die verschiedenen Validierungen durchführt. Aus den vorangegangenen Beschreibungen lassen sich nun folgende Klassen definieren:

|  |  |  |
| --- | --- | --- |
| Klasse | Spielelement | Bemerkungen |
| Card | Karte | Eine Oberklasse für die verschiedenen Spielkarten |
| PigCard | Schweinekarte | Subklasse von **Card** |
| UpgradeCard | Anlegekarten | Subklasse von **Card** |
| ActionCard | Aktionskarten | Subklasse von **Card** |
| Deck | Deck | Deck besteht aus einem Ziehstapel und einem Ablagestapel |
| Player | Spielende |  |
| Game | Spiel | Hauptklasse, worauf das Spiel läuft |
| Validator | - | Durchführung von Validierungen (z.B. User-Inputs) |

Tabelle : Definition der Klassen

In den nachfolgenden Abschnitten werden die Grundkonzepte der einzelnen Klassen erläutert sowie spezielle Lösungen aufgezeigt. Für die genauen Erklärungen der Methoden und ihre Funktionsweisen ist die Dokumentation im jeweiligen Code zu konsultieren.

## Card

Die Klasse ***Card*** bildet die Oberklasse für die Spielkarten ab. Um den Umgang mit den verschiedenen Kartentypen bei der Implementierung zu erleichtern, wurden Keywords pro Kartentyp definiert, z.B. «MUD» für die «Matschkarte». In dieser Klasse wurden anschliessend Dictionaries aufgestellt, welche die verschiedenen Keywords mit den entsprechenden Kartennamen verbindet, siehe Abbildung 3.

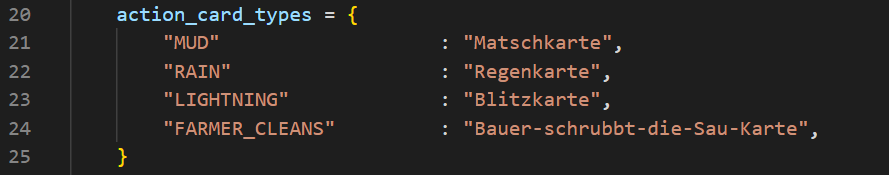


Abbildung 3: Dictionary der ActionCard (Mapping der Keywords mit den Kartennamen)

Im Dictionary ***all\_types*** sind alle Kartentypen und Keywords zusammengefasst.

Wird ein Objekt von dieser Klasse initiiert, muss als Input das Keyword der Karte mitgegeben werden. Daraufhin wird dem Objekt den entsprechenden Namen aus dem dictionary ***all\_types*** zugewiesen.

## PigCard

Die Klasse ***PigCard*** ist eine Subklasse von ***Card*** und bildet die Schweinekarten ab. Diese weisen Attribute auf, welche boolesche Werte annehmen und geändert werden, wenn Aktions- oder Anlegekarten gespielt werden. Der Kerngedanke ist, dass ein Schwein «sauber» oder «dreckig» sein kann (***self.dirty***), einen Stall (***self.stall***), eine vernagelte Türe (***self.door***) sowie einen Blitzableiter (***self.lightning\_conductor***) aufweisen kann. Mit dieser Funktionsweise interagieren die Aktions- und Anlegekarten nur mit der jeweiligen Schweinekarte und es werden keine weiteren Klassen benötigt. Um den jeweiligen Status einer Schweinekarte abfragen zu können oder anzeigen zu lassen, wird das Attribut ***self.status\_bool*** eingeführt, welche die vorangegangenen Attribute zusammenfasst. Mit der Methode ***get\_current\_status*** kann eine Liste zurückgegeben werden, welche diejenigen Statuseigenschaften aufweisen, bei denen das dazugehörige Attribut den Wert ***True*** aufweist, z.B.

Attribute: ***self.dirty*** = True, ***self.stall*** = True, ***self.door*** = False und ***self.lightning\_conductor*** = False

Statusliste: ***self.status\_bool*** = (True, True, False, False), ***current\_status*** = (Dreckssau, Stall)

Die Klasse weist zudem Methoden auf, welche die dazugehörigen Attribute ändern (z.B. ***build\_door***) oder diese abfragen (z.B. ***is\_dirty***)

## ActionCard

Die Klasse ***ActionCard*** ist eine weitere Subklasse von ***Card*** und bildet die Karten vom Typ 3 ab (siehe Kap. ‎4.1.1). Bei den Methoden dieser Klasse handelt es sich um die Karteneffekte, welche die Aktionskarten haben können. Diese sind folgende:

* ***mud\_on\_pig*** 🡪 Matschkarte: nimmt eine Schweinkarte entgegen und ändert dessen Attribut ***self.dirty*** zu True.
* ***rain\_on\_pig*** 🡪 Regenkarte: nimmt eine Schweinekarte entgegen und ändert dessen Attribut ***self.dirty*** zu False. Es ist anzumerken, dass die Regenkarte im Spiel alle dreckigen Schweine sauber macht. Für die Methode in der Klasse ***ActionCard*** wird jedoch nur eine Schweinkarte übergeben. Die Implementation, dass alle dreckigen Schweine gesäubert werden, wird in der Klasse ***Game*** durchgeführt.
* ***lightning\_on\_stall*** 🡪 Blitzkarte: nimmt eine Schweinekarte entgegen und zerstört dessen Stall, falls es einen besitzt.
* ***farmer\_cleans\_pig*** 🡪 Bauer-schrubbt-die-Sau-Karte: nimmt eine Schweinekarte entgegen und macht das Schwein sauber (***self.dirty*** = False)

Diese genannten Methoden geben nicht nur die geänderte Schweinkarte zurück, sondern auch die Upgradekarten, die an den Schweinen angelegt waren, z.B. ein Stall. Diese werden als ***destroyed\_cards*** bezeichnet und beinhalten die Keywords der zerstörten Upgradekarten. Das ist notwendig, um im Spiel diese Upgradekarten auf den Ablagestapel zurücklegen zu können.

Um im ***Game*** zu wissen, welchen Effekt resp. welche Funktion eine Aktionskarte besitzt und durchführt, wird zum einen ein dictionary ***self.actioncard\_func\_map*** und zum anderen das Attribut ***self.func*** eingeführt. Der Dictionary mappt den Kartentyp einer Aktionskarte, z.B. «MUD», mit ihrer entsprechenden Funktion, in unserem Beispiel mit der Funktion ***self.mud\_on\_pig***, und weist diese dem Attribut ***self.func*** zu.

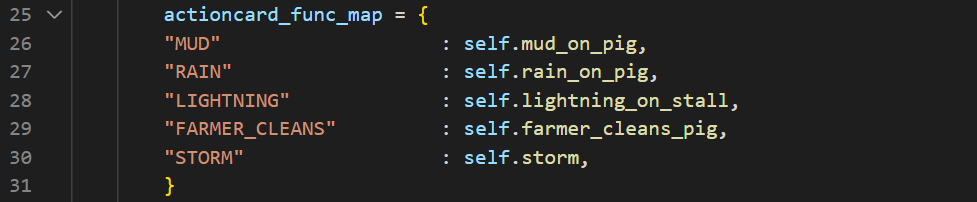


Abbildung 4: Mapping zwischen Kartentyp und Karteneffekt/Methode bei Aktionskarten

Somit muss bei der Aktivierung einer Karte nur die Methode im ***self.func*** aufgerufen werden. Hierfür ist die Methode ***self.activate\_card*** vorgesehen.

## UpgradeCard

Die Klasse ***UpgradeCard*** ist eine weitere Subklasse von ***Card*** und bildet die Karten vom Typ 2 ab (siehe Kap. ‎4.1.1). Der Aufbau und die Funktionsweise dieser Klasse ist an der Klasse ***ActionCard*** angelehnt (siehe Kap. ‎4.4). Auch hier wird ein Dictionary ***upgrade\_func\_map*** für das Mapping von Kartentyp und Karteneffekt eingeführt. Die Funktion einer Upgradecard wird wiederum in ***self.func*** hinterlegt und mit der Methode ***activate\_card*** aufgerufen. Hier ist noch anzumerken, dass ***activate\_card*** neben der geänderten Schweinkarte auch den booleschen Wert False zurückgibt, was darauf hindeutet, dass bei den UpgradeCards keine Anlegekarten eines Schweines zerstört werden und somit keine Karten zurück auf den Ablagestapel gelegt werden müssen.

## Deck

In der Klasse ***Deck*** werden die verschiedenen Karten als Objekte ihrer Klasse erzeugt (die Anzahl der Karten wird durch die Spielregeln bestimmt) und in einem ersten Schritt in einzelne Listen gespeichert. Im Attribut ***draw\_deck*** werden diese Listen bei der Initiierung des Decks zusammengeführt und gemischt. Gleichzeitig wird eine leere Liste für das Attribut ***discard\_pile*** erzeugt.

Neben den Methoden für das Ziehen einer Karte (***deal\_card***) wurde eine Methode entworfen, welche den Ablagestapel mischt und als neues Deck ausgibt (***create\_new\_draw\_deck***). Dies ist für den Fall, wenn das Deck aufgebraucht wurde. Die Methode ***deck\_has\_card*** überprüft, ob das Deck noch Karten hat (return True) oder aufgebraucht wurde (return False).

Zudem wurde die Methode ***add\_destroyed\_cards\_to\_discard\_pile*** entwickelt, welche die zerstörten Upgradekarten zurück in den Ablagestapel legt. Im Unterschied zur Methode ***add\_destroyed\_card\_to\_pile*** wird dieser Methode keine Karten übergeben, sondern Keywords der zerstörten Karte(n), um sie in einem ersten Schritt als neue Objekte zu erzeugen, und anschliessend dem ***discard\_pile*** hinzuzufügen.

## Player

Die Klasse ***Player*** enthält neben dem Namen, auch die Listen ***cards\_table*** und ***cards\_hands*** welche die Spielkarten einer/-s Spielenden auf dem Tisch resp. auf der Hand beinhalten.

Die Methoden dieser Klasse dienen dazu die Karten einer/eines Spielenden, sowohl auf dem Tisch als auch auf der Hand, zu verändern, updaten, zu zeigen oder zu löschen.

Besonders hervorzuheben ist die Methode ***update\_card\_table***, welches eine veränderte Schweinkarte ***updated\_pig*** einer/eines Spielenden entgegennimmt. Die Methode prüft daraufhin, welches der Schweinkarten der/des Spielenden dem ***updated\_pig*** entspricht (if pig == updated\_pig) und ersetzt es.

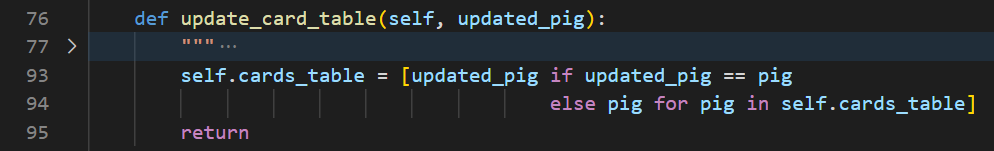


Abbildung 5: Methode zur Aktualisierung einer Schweinkarte

## Game

Die Klasse ***Game*** bildet das Kernstück des Programms, bei der die restlichen Klassen zusammengetragen werden. Wie bereits erwähnt werden hier nicht alle Methoden aufgezeigt, sondern lediglich die Grundkonzepte und Ideen erläutert.

Der Aufbau dieser Klasse orientiert sich insbesondere an den Spielzügen und den Möglichkeiten, die den Spielenden zur Verfügung stehen. Die Methoden lassen sich generell in folgende Gruppen unterteilen:

|  |  |  |
| --- | --- | --- |
| Methodengruppe | Methoden | Beschreibung |
| Initalisierung | init\_game()  number\_players()  create\_player(i)  init\_players()  number\_of\_pigs()  init\_cards\_table()  init\_cards\_hand() | Diese Methoden sind für die Initialisierung des Spiels notwendig. Die Anzahl Spielende wird abgefragt und anschliessend werden Spielende, Karten, Deck und Spiel erzeugt und verteilt. |
| Spielzug | play\_card()  choose\_card\_to\_play()  activate\_card(card, activations)  show\_all\_cards\_on\_table()  replace\_hand\_card(card)  change\_all\_cards()  add\_destroyed\_cards()  add\_destoryed\_cards\_to\_discard\_pile(cards\_type)  can\_player\_play\_card(hand\_cards)  check\_winner() | Diese Methoden befassen sich mit dem Spielzug der Spielenden. Es geht insbesondere um das Spielen oder Tauschen von Karten, das Ziehen oder Ablegen oder auch um Hilfsmethoden, welche z.B. eruieren, ob es eine/-n Gewinner/-in gibt. |
| Wahloptionen | choose\_play\_or\_change\_card()  choose\_to\_change\_one\_or\_all\_cards()  choose\_card(change = False)  choose\_move(possible\_moves) | Methoden, welche den Spielenden eine Wahl bieten und die Entscheidung der Spielenden aufnehmen |
| Possible Moves | get\_possible\_moves\_for\_card(card)  show\_possible\_moves(possible\_moves)  mud()  rain()  lightning()  farmer\_cleans()  stall()  lightning\_conductor()  farmer\_annoys()  get\_active\_player\_pigs()  get\_opponents\_pigs()  get\_opponents() | Diese Methoden dienen dazu zu ermitteln, welche Spielzüge («possible\_moves») für eine beliebige Spielkarte möglich sind. |

Tabelle : Methodengruppen für die Klasse **Game**

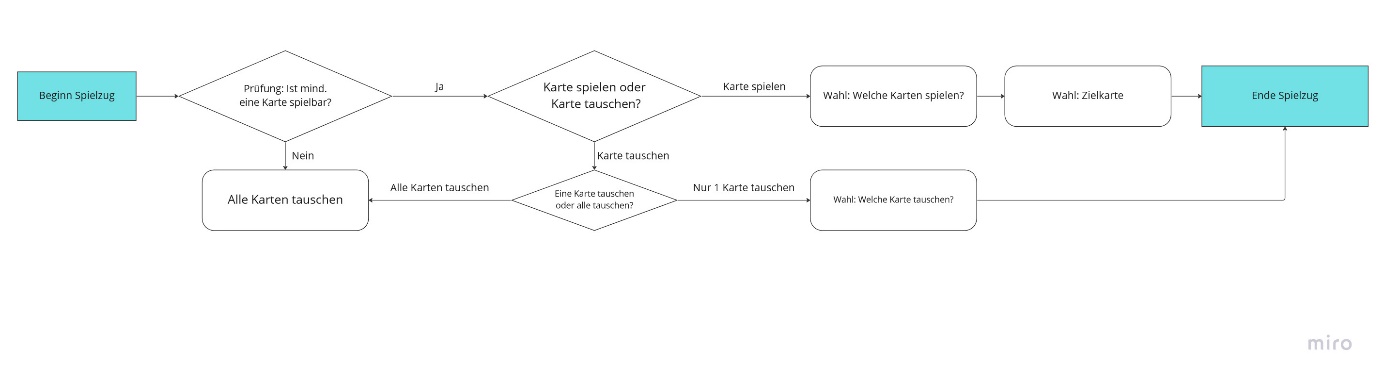
Nachfolgende Grafik soll den Ablauf eines Zuges aufzeigen:

Abbildung Ablauf eines Spielzuges als Grundlage für die Struktur der Klasse **Game**

Die Hauptmethode im Spiel ist ***run\_game()***, welche zu Beginn das Spiel initiiert und anschliessend die Spielzüge simuliert. Zu Beginn eines Spielzuges können Spielende eine oder alle Karten tauschen oder eine spielbare Karte ausspielen und erhalten dafür eine Auswahl von möglichen Zielkarten. Diese werden als ***possible\_moves()*** gekennzeichnet. Für die Ermittlung der möglichen Spielzüge ruft die Hilfsmethode ***get\_possible\_moves\_card()*** die entsprechende Methode für den Kartentyp auf. Für jeden Kartentyp steht eine Methode zur Verfügung, die den gleichen Namen besitzt wie der Karteneffekt, und die für den Kartentyp möglichen Kartenziele ermittelt. Beispielsweise zeigt ***mud()*** auf, welche Schweinkarten als Ziele ausgewählt werden können. Auch hier wiederum wird ein Mapping zur Verfügung gestellt, welche den Kartentyp mit der Funktion mappt:

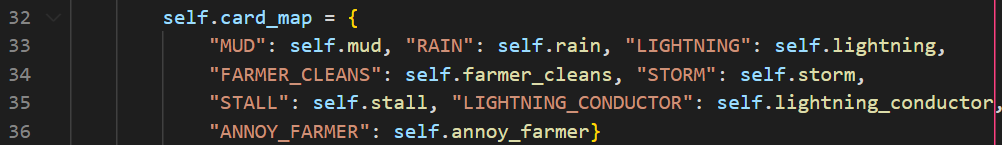


Abbildung 6: Mapping Kartentyp mit Methode zur Ermittlung der möglichen Kartenziele

Wählt ein/-e Spieler/-in ein Kartenziel (im Falle von Regen sind es alle dreckigen Schweinkarten), wird das Ziel in die Variable ***activations*** gespeichert, welche der Methode ***activate\_card(card, activations)*** übergeben wird.

Zum Schluss werden die Karten auf den Händen der Spielenden und auf dem Tisch aktualisiert, alte Karten mit neuen ersetzt und die verbrauchten oder zerstörten Karten auf den Ablagestapel gelegt.

## Validator

Die Klasse ***Validator*** ist für die diversen Validierungsschritte zuständig. Hier werden insbesondere die Anzahl Spielende geprüft, ob ein Spielzug möglich ist und ob die gewählte Karte existiert. Ein Objekt dieser Klasse wird im ***Game*** initiiert und in den Methoden aufgerufen.

## Start des Spiels

Für die Durchführung eines Spiels wurde das Skript ***start\_drecksau.py*** erstellt.

# Rückblick

In der Startphase des Projekts, war es wichtig, ein gutes Konzept zu erarbeiten. Durch die Erstellung eines UMLs, konnten wir die ***must haves*** identifizieren und untereinander aufteilen. Die Schwierigkeit im vorliegenden Projekt bestand darin, die Komplexität der Klassen in den Griff zu bekommen und zu ermitteln, welche Methoden benötigt werden und von welcher Klasse diese durchgeführt werden. Diese Abgrenzung war zu Beginn teilweise schwierig zu definieren, da z.B. gewisse Methoden sowohl vom ***Player*** als auch von einer ***Card*** durchgeführt werden können. Die frühzeitige Erstellung des UMLs hat uns jedoch diesen Prozess erleichtert und wir konnten immer vom Gleichen sprechen.

# Ausblick

Beim vorliegenden Kartenspiel handelt es sich um eine Textversion. Es wäre naheliegend dieses mit einem GUI zu erweitern, so dass es visuell einfacher ist für die Spielenden, das Spiel als solches wahrzunehmen.

Programmtechnisch konnte erkannt werden, dass sich die Methoden in der Klasse ***Game*** in diverse Gruppen unterteilen. Als mögliche Optimierung könnten weitere Klassen erstellt werden, die jeweils eine dieser Methodengruppen durchführen. Beispiel: Die Erstellung einer Klasse ***PossibleMovesGenerator***, welche die Methoden der ***possible\_moves*** besitzt und im Game initiiert und aufgerufen werden kann.