Entwicklungslog

22.07.2020

-Idee ein Entwicklungslog zu erstellen um Prozess zu beschreiben und vor allem Umsetzungs/Feature-Ideen festzuhalten

-Die ersten 2 Klassen Game und MyConsole sind implementiert. User kann mit Game interagieren. Main methode funktioniert gut.

-Früherer Gedanke (im Klassendiagramm bereits vorhanden): Es soll eine Fight Klasse geben, die einen Kampf zwischen dem Player und einem Enemy kontrolliert.

-Das Objekt dieser Klasse soll innerhalb der main von Game erzeugt (und auch wieder zerstört) werden, wenn der Player die „attack <Enemy>“ eingibt
-> Fight hat selbst auch eine main, die ähnlich ist zu der main methode von game. Diese wird innerhalb der main von Game, gleich nach der Instanziierung aufgerufen und kriegt das MyConsole Objekt als Parameter übergeben. Dann gibt es ein Attribut „isFightEnd“ (wieder analog zu „isGameEnd“ von Game), dass sobald einer der Charaktere stirbt (oder sonstige abbruchbedingungen) auf true gesetzt und somit der main-Methodenaufruf in der „zweiten Ebene“ verlassen wird. Falls der Spieler stirbt, kann das abgefangen werden (entweder über den player object verweis in main oder durch einen observer) und darauf reagiert werden (isGameEnd = true)

<u>Der nächste Schritt</u>

Allerdings möchte ich mich noch nicht der Fight Klasse zuwenden, da ich gerne erstmal „chronologisch“ weitermachen möchte.
Beim Erstellen des Game-Objekts sollen aus einer JSON-Datei Informationen zu dem Anfangszustand der verschiedenen Räume und den darin befindlichen Charakteren geholt werden, und mit diesen dann die benötigten Objekte instanziiert und verknüpft werden.
Deshalb beginne ich mit der UML Planung der dafür benötigten Methoden von der Game Klasse.

24.07.2020

Bevor ich die Planung des JSON-Lade Prozesses beschreibe, lege ich mich darauf fest, dass ich die Anwendung über http-server starte und teste, somit ist fetch möglich.
Desweiteren ist mir aufgefallen, dass die Klasse Fight und Game viele Gemeinsamkeiten haben und die Natur der main-Abläufe einer Sequenz ähneln -> deshalb neue abstrakte Klasse Sequence mit der abstrakten Methode main().

So lassen sich auch weitere Sequenzen beschreiben, und „ineinander verschachteln“. Z.b. Im Fight kann der Player sein inventory öffnen -> InventorySequence Objekt wird erzeugt und dessen main aufgerufen bis User irgendetwas benutzt oder es wieder schließt.

Interfaces müssen/sollten eingebaut werden für das geladene JSON Objekt, um die typsicherheit zu garantieren und nicht variablen von den Klassentypen zu haben, da die JSONObjekte (Room properties) sonst die Funktionen auch implementieren sollten.

Bei der Planung der JSON Laden Funktion und der Instanziierung aller geladeten Räume etc. fällt mir auf, dass man vielleicht garkeine gameMap braucht, vielleicht reicht eine statische Variable

innerhalb der Room Klasse aus. Aber dies entscheide ich zu einem späteren Zeitpunkt, wenn mehr Logik reinkommt.

-Planung der LoadJSON Aktivität fertiggestellt.
-Hinzufügen von einigen neuen Klassen und Interfaces (vor allem JSON-spezifische)

Mir fällt auf, dass die Planung der Charaktere und die Vererbung ein wenig komisch und wahrsch. noch nicht gut genug ist, dies muss noch angepasst werden in Zukunft.

Nun gilt es das Konzept umzusetzen.

Die Initialisierung eines Spiels soll so ablaufen, dass erst die JSON geladen wird und dann über die verschiedenen Array-Properties iteriert wird, falls eine property aftritt, die ein Array ist, beispielsweise die items property innerhalb eines Raumes, (dass es geht wird über die typsicheren Interfaces der jeweiligen properties sichergestellt), wird wiederum innerhalb des Konstruktors in einem „tiefergelegenen“ (Konstruktor-) Methodenaufruf über den Array iteriert und wieder für jeden Eintrag ein Objekt erstellt und dem Konstruktor die aktuellen für ihn relevanten Daten mitgegeben.
So wird die Initialisierung bzw. Instanziierung rekursiv vollführt.

25.07.2020

Da die JSON Datei nun richtig geladen wird und das Spiel auch erfolgreich initialisiert wird, gilt es nun die verschiedenen Eingabe- Optionen abzufangen und je nach Eingabe das gewünschte Verhalten zu triggern.
Überlegung:
Sequenzen sollen eine main haben (wie schon im Klassendiagram angedeutet), sollen die aktion vom Spieler holen (stehen also in Verbindung zur Konsole) und sollen sich selbst „vorstellen“

Beispiel: Raum wird gewechselt -> Raumbeschreibung wird von der Sequenz selber ausgegeben
oder: Kampf beginnt: Kampfsequenz stellt beteiligte vor
In Jeder Sequenzvorstellung sollen die in der jetzigen Sequenz erlaubten Optionen dargestellt werden.
Jede Sequenzinstanz ist selbst verantwortlich den Input vom User über die Konsole zu holen.

Die Verifizierung der Eingabe, das Ausfindig machen eventueller Ziele (bsp: Kampf initiieren) und generell spezifische Methoden erfolgen über einen Controller.
Auch das erstellen einer „Sub“-Sequenz soll über die Controller geschehen, die selber noch auf ihre parent Sequenz verweisen und so Informationen darüber haben, was für Objekte sich gerade in der Sequenz befinden.
Bsp: GameSequence besitzt currentRoom, User möchte Ziel angreifen „attack Gnom“, Controller übernimmt die Verifizierung und muss dann herausfinden, ob in dem characters-Array des currentRoom ein Enemy Objekt existiert, dessen Name „Gnom“ ist, erst dann soll eine neue Sub-Fight-Sequenz instanziiert werden.

Die Überprüfung der Eingabe kann mit mehreren Ebenen von try/catch Blöcken realisiert werden, was es möglich macht dem User spezifischere Fehlermeldungen mitzuteilen (die – was ich später auch vor hab – in einer enum gespeichert werden sollen)

Planung in Form von Aktivitätsdiagrammen beginnt (mit kleinen tests nebenbei).

Bei der Planung fällt auf, dass die Überprüfung nicht ganz trivial ist:

-User soll a oder attack eingeben können
-Wenn user attack eingibt, muss er ein ziel mit angeben und attack und das ziel sollen mit leerzeichen getrennt sein -> Problem: Namen sollen bzw können auch leerzeichen enthalten

-> Ein Controller muss her, der die Logik und Eingabeüberprüfung übernimmt

Diese und weitere Schwierigkeiten gibt es, allerdings plane ich es erstmal für den einfachsten Fall, dass der Name keine Leerzeichen enthält.

Eine Unterscheidung zwischen „einfacher Aktion“ und „Aktion mit Ziel“ soll trotzdem vorgenommen werden, allerdings auch für einen „einfachen“ Fall:

[AKTION] [LEERTASTE] [ZIELNAME]

Wobei dann in der nächsten Ebene unterschieden wird, um welche Aktion es sich genau handelt.

Hier denke ich sind Regular Expressions sehr hilfreich.

26.07.2020

Während der Konzeption und Planung der Controller, stellt sich mir die Frage ob ich es nicht hinbekomme nur eine Controller Klasse zu machen und bei der Erstellung innerhalb der main methoden der jeweiligen Sequenz, die commands in den Konstruktor zu geben, um so nicht für jede Sequenz einen eigenen Controller zu haben, da ein Controller eigentlich immer das selbe tun soll, sich nur von den commands unterscheidet, die allerdings immer die selbe logik verfolgen.

Vielleicht ist eine Command Klasse die sich in SimpleCommand (inventory, quit, commands) und TargetCommand(fight with <target>, look at <target>, take the <item>, drop the <item>) aufteilt sinnvoll.

Die Controller Klasse soll doch nur eine Ausprägung haben und einen Commands-Array besitzen. Bei der Instanziierung von einem Controller, sollen aus dem String Array, der von der spezifischen Sequence kommt und alle legalen Commands aus seiner spezifischen Enumeration holt, die Instanzen der Commands erstellt und in den Array gepusht werden.
Dann findet die Validierung, wie sie bis jetzt schon geplant ist (per regex) statt und der erste Entry des zurückkommenden String Arrays (durch .match(regex)) entspricht dann dem commandName.

In meiner ersten Überlegung wollte ich gleich mit der Regex nur die Inputs abfangen, die auch den commandName’s der erlaubten Commands entsprechen, allerdings müsste ich dann wenn ich nicht

für jede Sequenz einen eigenen Controller haben wollte, anhand der viableCommands die als String von der Sequenz zur Verfügung gestellt werden würden eine relativ aufwendige und fehleranfällige Methode schreiben müssen, die eine regex, erstellt, die für die spezifischen commands angepasst ist, um den Controller dynamisch zu halten.

Dies ist mit der neuen Methodik nicht mehr notwendig. Das Regex wird nurnoch genutzt um zu unterscheiden ob es sich um ein SimpleCommand oder ein TargetCommand handelt.

Dann kann folgend überprüft werden:

Falls der matchArray an der Stelle [2] undefined ist, heisst das, dass es sich um ein SimpleCommand handeln muss, also wird kein target gesucht und einfach der viableCommands Array des Controllers durchsucht nach einem Objekt, dessen commandName dem matchArray index 1 entspricht, und - falls eins gefunden wurde - dessen executeCommand-Methode aufgerufen (ansonsten throw(„Input is invalid“).

Falls der matchArray an der Stelle [2] nicht undefined ist, heisst das, der User hat wahrscheinlich irgendwas in der Form „[wort] [abstand] [wort] ([abstand] [wort] [abstand]…)“ eingegeben.
Minimum anforderung für ein match ist „[buchstabe] [abstand] [buchstabe]“. Wobei die erste Match gruppe (also der eitnrag index 1 im match array) „[buchstabe/wort] [abstand]“ beinhält.
Dann wird erstmal nach dem commandName gesucht (siehe oben).
Falls ein treffer erzielt wird, darf die executeCommand-Methode noch nicht aufgerufen werden, da zuerst noch das CommandTarget ausfindig gemacht werden muss.

Per .find() wird das Objekt aus dem Entities-Array geholt, welcher einfach lokal mit dem spread operator aus dem characters, items und adjacentRooms array zusammengestellt wird.
Falls nichts gefunden wird -> throw(„That is not a legal target!“)

Auch wenn diese Gedanken bis hierhin recht gut enstanden sind und ich für die Probleme die sich mir gestellt haben, relativ schnell Lösungsansätzen gefunden hab, bin ich schon seit 20 Minuten am überlegen, wie ich das Problem löse, dass ich während der Entwicklungszeit nicht weiss, welchen Typ im Endeffekt das Target hat.
Da ich den Controller sehr allgemein halten möchte, und ihm nicht die Logik zumuten will, dass er wissen muss, was er bei welchem input zu tun hat, gibt es ein Problem.
Wenn der Controller die executeCommand Funktionen implementieren würde, dann könnte man erst das Command ausfindig machen und dann gleich den Typ des Targets nach der Fallunterscheidung wissen und somit nurnoch den relevanten Array durchsuchen.
Die Schwierigkeit bei meiner Herangehensweise ist, dass ein Raum und ein Character beides legale targets sind und somit beide Arrays (characters und rooms) der parentSequence durchsucht werden müssen.
Nur weil die beiden Typen im allgemeinen legale targets sind, heisst es jedoch nicht, dass sie für jeden TargetCommand legale Ziele sind – so soll zum Beispiel ein Room nicht ein legales Ziel von „fight with „ sein.
Eine Lösung wäre, den Commands bei der instanziierung den jeweiligen relevanten Array mitzugeben und dann innerhalb des Command Objekts nochmal das Target in dem Array zu suchen, was allerdings nicht sehr performant wäre.

-Regex angepasst für den dynamischen Fall. (vorher waren die command Namen innerhalb des regex wortwörtlich drin, aber siehe o. warum das ein Problem war).

27.07.2020

Erweiterungen der spezifischen TargetCommand Klassen um das „Problem“ der Target suche zu lösen:

Reminder: Problem war, dass der InputValidator (vorher: Controller) nur geschaut hat, ob die eingabe einem Muster entspricht, allerdings nicht geguckt hat, ob das Ziel für das command erlaubt ist (Bsp: fight with [Raumname] wäre nicht erlaubt, da man keinen raum angreifen können soll, ganz geschweige von einer non-sense eingabe (was allerdings einfacher zum überprüfen wäre)).

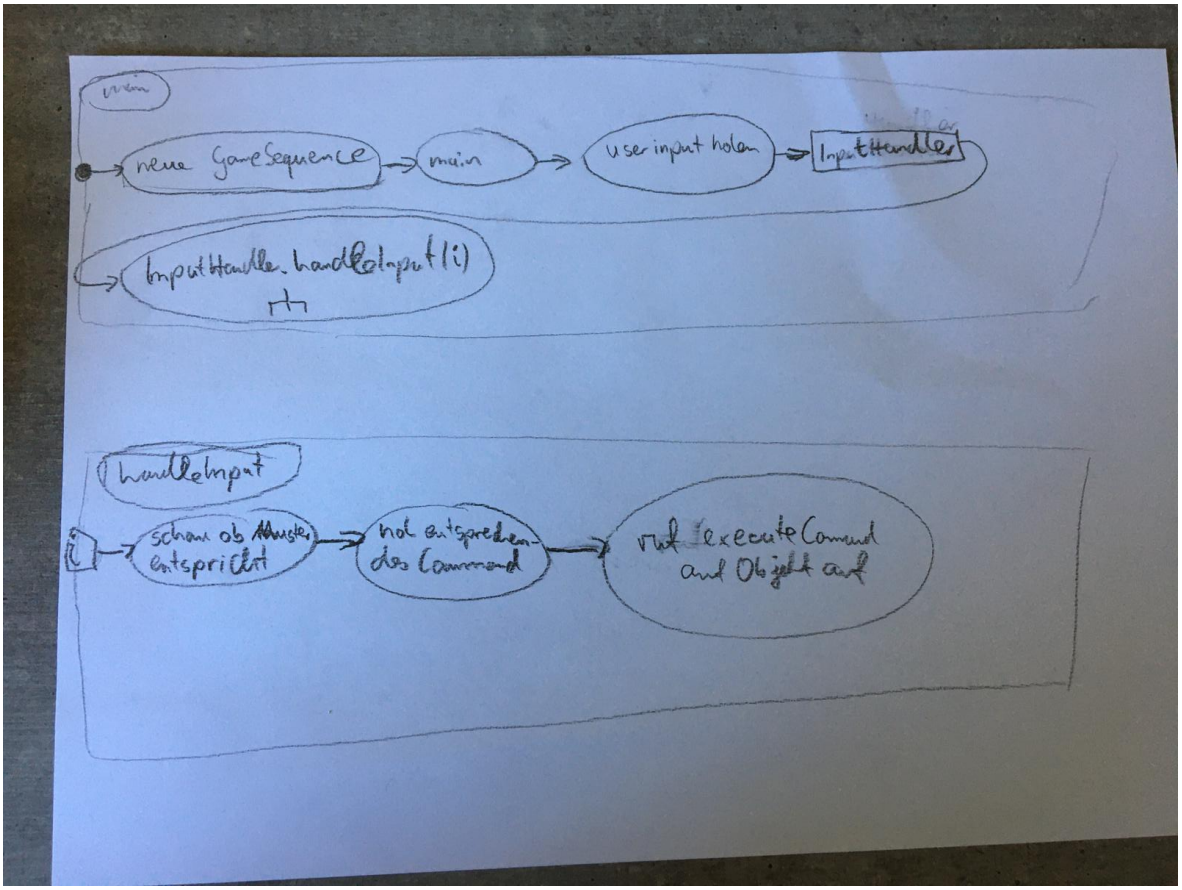
Die Superklasse TargetCommand wird nun dahingehend erweitert, dass ein TargetCommand immer weiss, welchen Array er durchsuchen muss, das heisst, dass ein Objekt des Typs StartFightSequenceCommand den Characters-Array (später Enemies) des currentRoom bei der Instanziierung erhält, die Klasse LookAtTargetCommand einen per spread operator zusammengesetzten array bestehend aus characters items und dem namen des currentRoom (damit dieser auch angeschaut werden kann).
So kann das Programm sehr schnell mit neuen Commands versehen werden, indem man diese unterteilt ob sie ein Ziel benötigen oder nicht, und ob der Nutzer die Freiheit hat das Ziel selbst zu wählen.

In einer FightSequence soll der User zum Beispiel (erstmal) nicht aussuchen können, welches Ziel er angreift, da in einem Kampf (erstmal) nur Zwei beiteiligte exisiteren und der User sich nicht selber angreifen können soll.
Das bedeutet, dass die neue Klasse die dann diesen Befehl beschreibt von TargetCommand erbt, aber den Konstruktor so anpasst, dass er keinen array vom Typ CommandTarget erhält, sondern direkt das Enemy Objekt (Dies geschieht dann bei der Instanziierung vom InputController, wenn alle viableCommands abgearbeitet werden und dann für jedes eine Instanz erzeugt wird.
Die Fight Sequence beinhaltet sowieso schon die Kampfbeteiligten, somit ist das dann kein Problem.

Diese neuen Ideen sind am Anfang der Klassendiagramm und Aktivitätsdiagramm Planung (Kontext des Controller/Sequence Gedanken) entstanden.

Mittlerweile bin ich mit der Planung zufrieden und habe kleine Tests (vor allem was Polymorphie/Vererbung und abstrakte Klassen angeht) gemacht.

Die Klassendiagramme sind neben diesen Gedanken entstanden.
Die Aktivitätsdiagramme gilt es nun zu planen. (**grobe** Skizze habe ich schon erstellt)



Jetzt geht es an die konkrete Aktivitätsdiagrammplanung

-„Command“ wurde generell zu „Action“ umbenannt

Bei der Planung fällt mir auf, dass zum Beispiel die LookAt.doAction die description loggen soll, und somit zugriff zur Konsole braucht, die Konsole ist in der Sequenz vorhanden, so kann man diese dann an den InputHandler und der InputHandler diese dann weiter an die Actions geben.

28.07.2020

Das Konzept ist bis hierhin erfolgreich umgesetzt.

Der Aufbau der Software und meiner Klassen erlaubt es mir jetzt eigentlich recht schnell neue Features in Sequenzen und Spielerentscheidungen in Aktionen zu beschreiben und die in der Sequenz erlaubten Aktionen dynamisch erzeugen zu lassen.

Nun muss ich mir genau überlegen, was ich als nächstes machen soll.

Um sicher zu gehen, dass ich bis zur Abgabe sicher alle Grundanforderungen umgesetzt habe, werde ich diese nochmal festhalten und diejenigen abhaken, die schon implementiert sind.

Umgesetzt:
Look, Commands

Es fehlt:
Bewegungsaktionen, inventory, take item, drop item und quit
Intelligente Charaktere

Da inventory, take item und drop item alle sehr verwandt sind und eng mit dem User also der Player Klasse in Verbindung stehen, und diese Klasse nur im Klassendiagram existiert und die Methoden noch nicht konzipiert wurden, ist das der nächste Schritt.

Bis jetzt fehlt die Player instanz noch komplett in den Sequenzen, diese sollte allerdings in jeder Sequenz vorahnden sein, da der Player im Mittelpunkt steht.

Des weiteren sollte es immer nur einen Player geben, was bedeutet, dass sich hier das Singleton Pattern anbietet.

Die legalen Aktionen die der User eingeben kann, bzw. bei dessen Eingabe auch wirklich etwas ausgeführt wird, werden innerhalb der Sequenz durch einen Array bestimmt.
Es wäre auch denkbar, dass der User selber auch einen Action-Array an allen Aktionen, die er ausführen kann besitzt und die Schnittmenge aus diesem Array und dem Array der Aktionen die dann in der spezifischen Sequenz erlaubt sind, werden dann in dem InputHandler Instanziiert.
So könnte man zum Beispiel dem User Zugang zu neuen Aktionen gewähren, wenn dieser spezielle Items findet. Beispiel: Im Inventar ist eine Karte -> Eintrag in seinen Action: string[] Array mit dem identifier der Aktion, der dem Eintrag in der Enumeration, die alle möglichen Aktionen speichert, entspricht.

Dieser Gedanke bleibt im Hinterkopf, da das eine Erweiterung ist und ich mich fortan auf die Umsetzung der Grundanforderungen fokussieren werde.

Konkret:
Planung der Character Sub-Klassen

Player wird im Singleton Pattern gemacht.

Bei der Planung des Players sind mir viele Neuerungen aufgefallen, um das Spiel noch strukturierter zu machen. (Auch in Anbetracht der load/save Optionen)
Morgen setze ich die Änderungen um.

30.07.2020

Neue Actions hinzugefügt, Sequenzen so aufgebaut, dass sie immer nur die für sie Relevanten Objekte beinhalten

-Kleine Anpassungen in der Art und Weise, wo Actions registriert werden (InputHandler existiert nicht mehr), Action-Objekte werden direkt in der jeweiligen Sequenz beim erzeugen der Instanz erzeugt.

-Neue Aktions klassen erstellt (unter anderem für die commands „inventory“, „examine“, „look at“) und lasse diese in den jeweiligen Sequenzen, wo sie verfügbar sein sollen in der neuen Methode instantiateActions() jeweils mit new erstellen.

-Die RoomSequenz hat nicht mehr einen Verweis auf currentRoom.
Der Player hat stattdessen den Verweis – und jede Sequenz einen Verweis auf ihn. (was auch Sinn macht, da der Player immer im Mittelpunkt sein sollte).

-Regex entfernt, Grund: da die Commands in sehr unterschiedlichen Mustern vorkommen können und zum teil aus einem „Identifier“ (commands) und zum teil aus zwei (look at) bestehen und dann wiederum einige davon ein target beschreiben ({itemname} oder {direction}) ist ein regex nicht sehr sinnvoll, bzw. wäre es für mich zu schwer so eines zu beschreiben
Darum -> Überprüfung ob der userInput mit einem actionIdentifier der ActionObjekte im actions Array der Sequenz anfängt, wenn ja, dann ab dem letzten index der noch dem actionIdentifier entspricht bis zum allerletzten buchstaben substring bilden und das als targetName beachten und innerhalb der Action Objekte dann gucken, ob ein Objekt in dem für die Action relevanten Arrays mit diesem Namen existiert.

-Da der Player als Singleton existiert, muss den Sequenzen nicht mehr die Referenz übergeben werden sondern sie können die Instanz per Player.getInstance() im Konstruktur holen.

Die MainMenuSequence ist neu die AusgangsSequenz, die erzeugt wird um dem User die erste Wahl zu geben, ob er ein neues Spiel anfangen möchte, oder einen alten Spielstand laden möchte.
Die MainMenuSequence besitzt dann die Action-Objekte vom typ StartNewGame und LoadGame

Diese Action Objekte haben über die sequence Referenz zugriff zu dem LoaderAndSaver Objekt, dass die JSON-Lade funktionen und instanziierung des GameState Objekts (samt allen Rooms und Player) übernimmt.

Die currentRoom property der GameState Klasse ist obsolet geworden, dadurch, dass player nun immer weiss, in welchem room er sich befindet (und der currentRoom sollte auch immer im Zusammenhang mit dem Player stehen, da es um den Player zentral geht).

Ein Problem, dass nun mit der Spielerbewegung hinzukommt, ist dass bisher einige Commands wie z.b. „inventory“ neue „Sub-Sequenzen“ gestartet haben, was auch gewollt ist, da die main der „Super-Sequenz“ (RoomSequence) „angehalten“ werden soll (oder viel mehr warten soll, Stichwort: await), so lange bis PlayerInventorySequence.isSequenceEnd wahr ist (der Player also die Inventar Anzeige schließt) und somit die main-Methode der „Sub-Sequence“ ihr Ende erreicht.

Mit dieser Idee bin ich sehr zufrieden.

Ein Problem hierbei ist allerdings, dass bisher wenn der User einen Kampf beginnen möchte mit (fight with {enemy}), die Methode doAction(_targetName) des FightWith Objekts aufgerufen wird und (gleich nachdem das target bestimmt wurde, falls eins existiert) ein neues FightSequence Objekt erstellt wird und noch innerhalb der doAction Methode dessen main() aufgerufen wird.
Bei den Subsequenzen ist dies nicht weiter tragisch.
Bei den RoomSequence Objekten allerdings (bzw. bei DEM RoomSequence Objekt, da immer nur eins existieren sollte) ist das ein Problem.
Durch die „rekursive“ Art der Sequenzobjekt Erzeugung, würde, da bei der Action StartNewGame von der MainMenuSequence gleich dessen main Methode aufgerufen wird, das Ende nie erreicht werden, wenn die Implementation der „Go“ Aktion, also der Spielerbewegung, die den currentRoom vom Spieler Anpassen soll und eine neues RoomSequence -Objekt erstellt und dessen main() aufruft (wie es bei den Subsequenzen der Fall ist), nie das Ende der ursprünglichen RoomSequence erreicht werden, da immer wieder neu ineinandergeschachtelte main methoden aufgerufen werden würden, ohne dass je das isSequenceEnd- Attribut eines der RoomSequence-Objekten auf true gesetzt wird.

Um so also mögliche Arbeitsspeicher-Probleme (die bei einem Textadventure wahrscheinlich nicht allzu groß werden), muss etwas getan werden.

Fazit:
RoomSequence benötigt also einen Mechanismus, um nicht selber neue RoomSequence Objekte zu erzeugen, da die main()-Methode ihr Ende erreicht haben soll, bevor die main Methode der nächsten RoomSequence aufgerufen werden soll.

Ein weiteres Problem bei der „Go“ Aktion, also der Spielerbewegung, ist, dass das State-beschreibende JSON Objekt, jedem „adjacentRooms“-Attribut von einem Room Objekt ein Key-Value Pair in Form von Richtung-RaumId zuweist (damit die Größe und Unübersichtlichkeit des JSON-Objekts bei diesen ganzen Querverweisen nicht ins unermässliche steigt).
Das Problem besteht nun darin, dass die Sequenzen keine Referenz auf das GameState-Objekt (welches die Room Objekte beinhält) besitzen.
Das Go-Action Objekt hat höchstens eine Referenz auf den currentRoom (da jede Aktion eine Referenz auf Player hat), und innerhalb des adjacentRooms Attributs sind nur die IDs gespeichert, somit könnte garkeine neue RoomSequence gestartet werden, da für die Instanziierung einer solchen ein Room Objekt nötig ist (für die NPCs und das inventory, etc.)

Es gibt also 2 gute Gründe warum etwas an der Struktur geändert werden muss.

Idee:

Die MainMenuSequence dahingehend anpassen, dass ein Klassenattribut nextRoomSequence hinzugefügt wird, dann wird bei new game oder load game ein erstes mal ein in der funktion doAction lokales Objekt von RoomSequence erzeugt (nachdem player und alle anderen objekte geladen un instanziiert wurden), dann nach dem await getPlayerInput() innerhalb der main eine neue while Schleife gesetzt, die überprüft, ob nextRoomSequence nicht null ist, wenn ja dann führe die main() aus, ansonsten befindest du dich noch in deiner eigenen while(!this.isSequenceEnd) schleife und der user kommt zu zuürck an die hauptmenu funktionen.

Bei GoDirection muss dann, sofern ein Room in der Richtung in die der Player gehen möchte existiert, ein neues RoomSequence Objekt erstellt und per MainMenuSequence.getInstance() . (Singleton) dessen nextRoomSequence Attribut auf das neue gesetzt werden.

Dann wird noch die isSequenceEnd der aktuellen RoomSequence auf true gesetzt, damit nach dem PlayerInput auch die main abgeschlossen (und das Promise resolved) wird und so der initale newgame/loadgame .doAction() aufruf verlassen werden kann und direkt nach dem getplayerinput folgt dann eben die while(this.nextRoomSequence) schleife.

Sobald der Spieler zum Hauptmenu zurückkehren möchte, muss eine Quit-Actions Klasse mit einer Implementation der doAction Menthode existieren, die einfach das isSequenceEnd Attribut auf true setzt und **kein** nextRoomSequence der MainMenuSequence setzt.

31.07.2020

Allgemein einige neue Aktivitäten und Klassendiagramm änderungen samt Implementation.

Morgen noch die letzten Änderungen am Aktivitätsdiagramm vornehmen und fehlende Methoden implementieren, dann eine kleine Story erzählen und für die Abgabe relevante Dokumente erstellen.

02.08.2020

Die neuesten Aktivitätsdiagramme sind implementiert und ich habe noch ein letztes mal kleinigkeiten verändert, unter anderem hab ich mich dafür entschieden die shortcuts komplett zu kicken, da ich mittlerweile (mit allen Sequenzen) bei 15/16 commands bin und das langsam keinen sinn mehr macht für jedes ein „kürzel“ zu bestimmen zumal viele sowiesso schon sehr kurz sind (take, buy, go)

Die aktivitätsdiagramme und das klassendiagram wurden noch einmal angepasst, da einige Ideen nicht genau so umgesetzt werden konnten.

Im großen und ganzen bin ich mit der Planung aber sehr zufrieden.

Für die Zukunft kann ich mir sehr viele neue Features vorstellen, beispielsweise eine ConversationSequence mit wiederum spezifischen Aktionen wie „ask“ oder „threaten“ und so weiter.
Allgemein ist der Aufbau des Spiels sehr „modular“ ich denke diese ConversationSequence samt aktionen wäre sehr schnell implementiert.

Weitere Erweiterungsideen:

-Darstellung in der Konsole verbessern:

-„Fehlermeldungen“ wie z.b. „XY is not a valid command“ rot einfärben

-Body-background image dynamisch anpassen, je nachdem in was für einer Art Room sich der player befindet

-Kampfsystem erweitern, mit „attack with {weapon}“ und dann typen von gegnern mit immunitäten und schwächen erstellen

-Items in weitere Subklassen unterteilen, Potion könnte von einer Superklasse „Consumable“ erben, und dann HealPotion, DefensePotion etc. Klassen erstellen

-Allgemein verschiedene Stats hinzufügen

-Erfahrungs/ Level system

-Karte: beim öffnen der Karte im inventar, lässt sich ein kleiner überblick über besuchte Räume geben. (wäre mit dem momentanen Aufbau sehr schnell umgesetzt)