SOFTWARE REQUIREMENTS SPECIFICATION

for

MELT Chess

Version 0.1

29. April 2021

Inhaltsverzeichnis

1 Einführung

1.1 Projektanforderungen

Entwickelt werden soll ein Schach Programm, welches es ermöglicht gegeneinander als auch gegen eine künstliche Intelligenz Schach zu spielen. Das Spiel soll dafür sowohl über eine graphische als auch über eine konsolenbasierte Benutzerschnittstelle verfügen. Das Spiel soll in englischer Sprache umgesetzt werden. Die Entwicklung soll sich dabei in drei Iterationen gliedern. In diesem Dokument sind die Anforderungen an die erste Iteration dargelegt.

1.2 Zielgruppe des Dokuments und weitere Resourcen

Dieses Anforderungsdokument richtet sich zum einen an die beteiligten Entwickler und dient zur Orientierung ob die Funktionalität des Projekt gemäß den Anforderungen umgesetzt wird, und zum anderen an die Kontaktperson(en) des Moduls um die Planung des Projekts zu überprüfen.

Einen weiteren Überblick bieten die Storycards, welche im Gitlab des Projekts zu finden sind und in Form von Issues umgesetzt werden.

2 Umfang des Projekts

2.1 Funktionale Anforderungen

Zum Umfang gehört in der 1. Iteration des Projekts die Bereitstellung einer Textbasierten Konsolen-Schnittstelle in der es möglich ist Mensch-gegen-Mensch Spiele zu spielen. Dazu ist die Umsetzung des Pakets model nötig, in dem der Zustand des Bretts sowie die Schachregeln[1] implementiert werden.

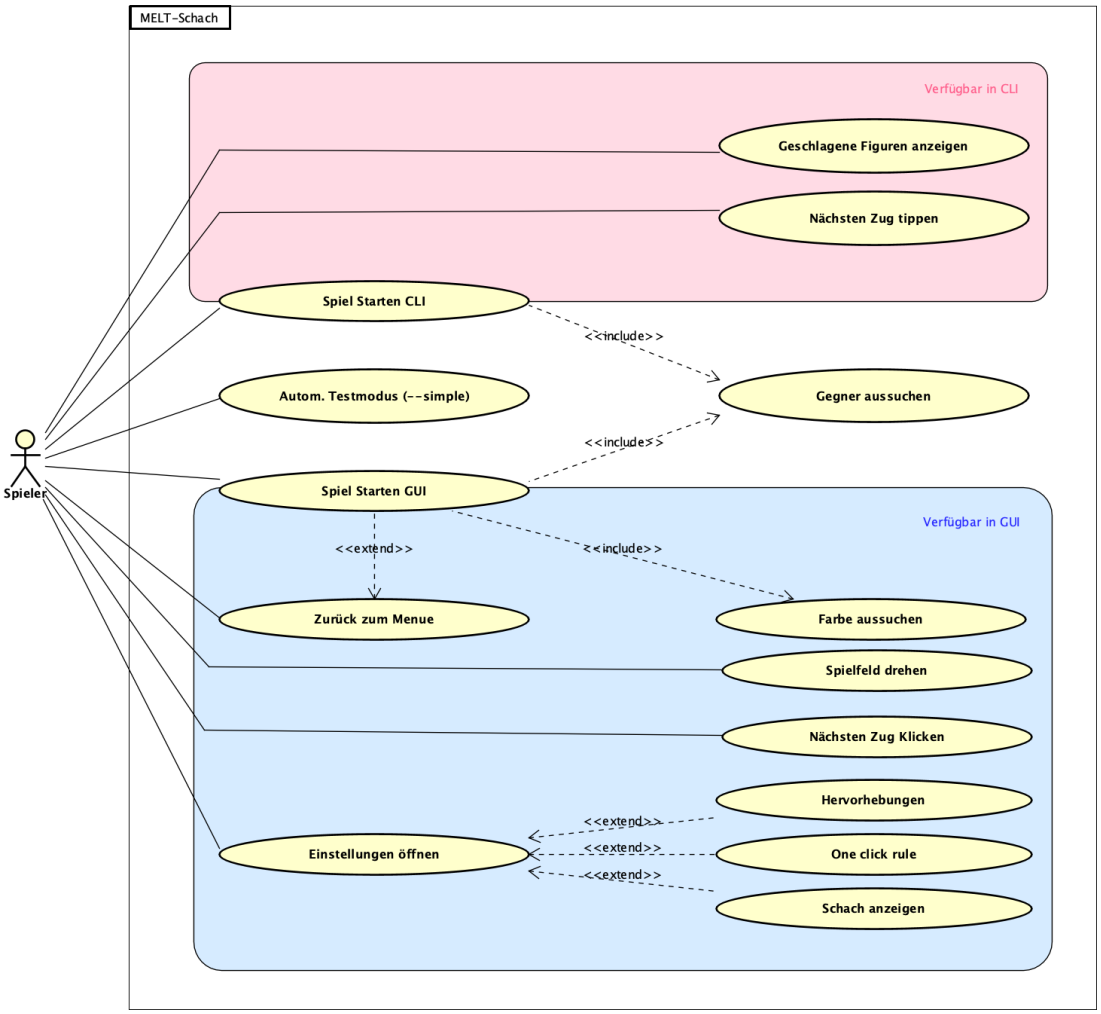
In der 2. Iteration wird die geschaffene Basis durch eine 2D-GUI unter Verwendung des JavaFX Moduls erweitert und eine rudimentäre Schach KI im Modul engine erstellt.

In der 3. Iteration soll die Anwendung um Netzwerkfähigkeit mit der Anwendung der Gruppe 2 erweitert werden. Zusätzlich soll eine Auswahl von möglichen Erweiterungen umgesetzt werden, welche in Summe einen Aufwand von mindestens 10 Einheiten aufweisen müssen. Die Auswahl mit möglicher Punktzahl lauten:

- Verbesserte KI mithilfe Min-/Max-Suche mit α/β-Pruning (5)
- 3D-GUI (5)
- Eindimensionales Schach (5)
- Rückgängig-Machen von Zügen (3)
- Speichern/Laden von Spielen (3)
- Schachuhren (2)
- Zweisprachigkeit (2)
- Resizeable GUI (1)

[1] Nach den Regeln des Weltschachverbands (FIDE) in der deutschen Übersetzung von 2018

2.2 Usecase-Diagramm



2.3 Nicht-funktionale Anforderungen

2.3.1 Codingstyle, Metriken, Testabdeckung

Das Template verwendet die Plugins PMD und JaCoCo zur Generierung von Reports über Metriken, Codestyle und Testabdeckung. Es wird eine Testabdeckung des gesamten Codes von 90% Instruction Coverage erwartet, ausgenommen der GUI-Klassen. Im Template wird ein PMD-Regelsatz eingebunden. Diese Regeln sind für den Code und die Testfälle einzuhalten.

2.3.2 Überprüfungen

Es soll mindestens alle zwei Wochen ein Treffen mit dem zugewiesenen Tutor stattfinden. Neben den Deadlines der drei Iterationen gibt es außerdem noch folgende Termine:

- 23.04.2021: Abgabe Anforderungsanalyse, Vorgehensplan, Prüfung erfolgreiche Einrichtung der Infrastruktur
- 19.05.2021: Prüfung, ob auslieferbare Version vorliegt, die Anforderungen der ersten Iteration genügt mit automatisierten Tests und Theorie-Abfrage
- 16.06.2021: Prüfung, ob auslieferbare Version vorliegt, die Anforderungen der zweiten Iteration genügt mit Zwischenpräsentation
- 14.07.2021: Endabgabe mit Abschlusspräsentation

2.3.3 Abgabeformat

An den Schlusstagen der Iterationen, inklusive Endabgabe, muss der zu überprüfende/bewertende Stand mittels Tags (it1,it2,...) im git-Repository gekennzeichnet werden.

3 Vorgehensplan

Der Vorgehensplan wurde durch Storycards im Issuetracker von gitlab realisiert. Für die Dauer des Praktikums können diese hier[1] eingesehen werden. Im Folgenden befindet sich eine maschinell generierte Version aus dem csv Export von gitlab.

3.1 Storycard Issues

[1] https://projects.isp.uni-luebeck.de/melt-chess/melt-chess/-/issues

Schnittstelle Konsole

id: #3 Milestone: 1. Iteration

Priorisierung

Storypoints 5

Risiko 0

Der Benutzer hat die Möglichkeit das Schachspiel gegen einen anderen Benutzer zu spielen. Dazu werden die Züge in der Konsole eingegeben.

Abgeschlossen wenn

- ☐ Die Benutzereingabe wird korrekt geparsed (Tests für "parseUserMoveInput" bestanden)
- ☐ Gameloop ist etabliert (User Input einlesen, Schachbrettposition ausgeben, ...)
- ☐ Ausgabe der geschlagenen Figuren auf den Befehl "beaten"
- ☐ Ausgabe wenn ein Spieler im Schach steht
- ☐ Die Anforderungen an die Ausgabe bei gesetztem "
- ☐ Fehleingabe wird mit "!Invalid move" quittiert
- ☐ Gültige Eingabe wird mit "!¡Eingabe¿"quittiert
- ☐ Ein ungültiger Zug wird mit "!Move not allowed"quittiert
- ☐ Schachbrett wird nach jedem Zug geprintet

Erstellen der Models

id: #4 Milestone: 1. Iteration

Priorisierung

Storypoints

Risiko

Erstellen der Klassen im Paket `models` welche von den anderen Paketen `cli`, `engine` und `gui` benutzt werden

Abgeschlossen wenn

- ☐ Alle blockierenden Storycards sind abgeschlossen

Erstellen Board Klasse

id: #5 Milestone: 1. Iteration

Priorisierung

Storypoints 2

Risiko

Die `Board` Klasse im Paket `model` kapselt eine Position auf dem Brett, sowie die Information welcher Spieler am Zug ist. Die Position wird dabei als Array mit 64 Integerwerten zwischen 0 und 23 kodiert.

Abgeschlossen wenn

- ☑ Die Klasse repräsentiert eine Position auf dem Brett
- ☑ Ein neues Objekt kann durch übergeben eines FEN
- ☑ Ein neues Objekt kann durch die "makeMove" Methode aus einem bestehenden Objekt erzeugt werden
- ☑ Die Methoden "getPiecePositionsFor" und "getPieceAt" bestehen alle mithilfe von FEN

Erstellen Move Klasse

id: #6 Milestone: 1. Iteration

Priorisierung

Storypoints 1

Risiko 0

Die `Move` Klasse im Paket `model` soll einen Container für einen Spielzug verwirklichen. Sie merkt sich dazu von und zu welchem Feld der Zug gemacht wird, sowie eine Flag die besondere Umstände des Zugs (Rochade, Figurentausch, ...) beschreibt.

Abgeschlossen wenn

- ☑ Die Informationen zu einem Zug werden erfolgreich kodiert
- ☑ Die toString Methode gibt eine String repräsentation des Zuges gemäß den Anforderungen des Konsolen Clients zurück

Erstellen der MoveGenerator Klasse

id: #7 Milestone: 1. Iteration

Priorisierung

Storypoints 13

Risiko 1

Die `MoveGenerator` Klasse im Paket `model` soll mögliche Züge zu einer gegebenen Position unter berücksichtigung der Schachregeln generieren. Die werden sowohl von der GUI zum hervorheben der möglichen Felder beim bewegen einer Figur, als auch von der Engine bei der Suche nach dem nächsten Zug verwendet.

Abgeschlossen wenn

- ☑ Der Generator kann die verschiedenen Laufrichtungen zu den unterschiedlichen Schachfiguren berechnen
- ☑ Die "generate" Methoden bestehen sinnvolle Tests (Siehe die Flags in der Klasse "Move": werden auch diese Züge gefunden?)

Erstellen der Piece Klasse

id: #8 Milestone: 1. Iteration

Priorisierung

Storypoints 2

Risiko 0

Die Klasse `Piece`im Paket `model` soll die geplanten statischen Methoden zur Extraktion der kodierten Informationen zur Verfügung stellen. Typ und Farbe einer Schachfigur wird dabei als Integer zwischen 0 und 23 kodiert.

Abgeschlossen wenn

- ☑ Die statischen Methoden zur Rückgabe der in einem Integer Wert codierten Informationen bestehen alle Tests.
- ☑ Die toString Methode gibt die vorher festgelegten Symbole korrekt wieder

Erstellen MoveValidator Klasse

id: #11 Milestone: 1. Iteration

Priorisierung 1

Storypoints 8

Risiko 2

Der `MoveValidator` soll die vom `MoveGenerator` berechneten Züge auf gültigkeit überprüfen. Da die erlaubten Bewegungsrichtungen bereits in `MoveGenerator` korrekt implementiert sind, müssen hauptsächlich die Sonderregeln die zu schach des Königs führen beachtet werden.

Abgeschlossen wenn

- ☑ Der König kann nicht in schach bewegt werden
- ☑ Gefesselte eigene Figuren können den König nicht in Schach setzen
- ☑ Ist ïm schachïmplementiert, wurde das generieren von Rochadezügen in "MoveGenerator" verhindert. Edit: Die Züge werden zwar generiert, aber vom "MoveValidator" abgelehnt.

Erstellen der Game Klasse

id: #12 Milestone: 1. Iteration

Priorisierung

Storypoints 3

Risiko 0

Die `Game` Klasse soll den Gesamtzustand einer Partie kapseln, aber hauptsächlich für die beiden GUI Module relevant sein.

Abgeschlossen wenn

- ☑ Funktion für neues Spiel starten"
- ☐ Verwaltet alle nötigen Objekte einer Partie und bietet ein einfaches Interface für die GUI Module
- ☐ Erkennt Schachmatt und Patt (#13)Bonus
- ☑ "Board" Objekte in einer Liste als Gamehistory merken (BONUS)
- ☐ Funktionen zum zurück oder vorwärs gehen in der Gamehistory
- ☐ Verwaltung der Schachuhren?
- ☐ Sichern und Laden des Spielzustands

Schachmatt und Patt

id: #13 Milestone: 1. Iteration

Priorisierung

Storypoints

Risiko

Es müssen die Fälle Patt (ein Spieler hat keine möglichen Züge) und Schachmatt (ein Spieler hat keine möglichen Züge und steht im Schach) implementiert werden

Abgeschlossen wenn

- ☐ Partie kann in Patt enden
- ☐ Partie kann durch schachmatt gewonnen/verloren werden

4 Anhang

4.1 Tabellarische Anforderungsanalyse

Klassifikation	Priorität	Kategorie	Kriterium	Iteration
funktional	must	Hauptfunktion	gegen eine KI spielen	2
funktional	must	Hauptfunktion	gegen einander spielen	1
funktional	must	Schnittstelle	graphisch	
funktional	must	Schnittstelle	konsolenbasiert	1
funktional	must	Sprache	Englisch	
funktional	must	Züge	nicht zulassen ungültiger Züge	1
funktional	must	GUI	2D	2
funktional	should	Features	Verbesserte KI mithilfe Min-/Max-Suche mit α/β\alpha/\betaαα/β-Pruning (5)	3
funktional	should	Features	3D-GUI (5)	3
funktional	should	Features	Eindimensionales Schach (5)	3
funktional	should	Features	Rückgängig-Machen von Zügen (3)	3
funktional	should	Features	Speichern/Laden von Spielen (3)	3
funktional	should	Features	Schachuhren (2)	3
funktional	should	Features	Zweisprachigkeit (2)	3
funktional	should	Features	Resizeable GUI (1)	3
funktional	must	Hauptfunktion	Netzwerkspiel	3
nicht-funktional	must	Codequalität	Einhaltung der Metriken	
nicht-funktional	must	Codequalität	Einhaltung Style-Convention	
nicht-funktional	should	Codequalität	Kommentierung des Codes	
Nebenbedingung	must	Testabdeckung	Abdeckung von 90 % des Codes (ohne GUI Klassen) durch JUnit-Tests	
nicht-funktional	must	Dokumentation	Stets aktualisierte Anforderungsdokumentation via Story-Cards	
nicht-funktional	must	Dokumentation	Stets aktualisierte Dokumentation der Architektur unter Zuhilfenahme von UML-Diagrammen (Klassen-, Objekt-, Sequenz-, Zustand-)	
nicht-funktional	must	Dokumentation	Stets aktualisierte Javadoc-Dokumentation des Programms	
nicht-funktional	must	Dokumentation	Stets aktualisierte Dokumentation der Programmverwendung (Bedienungsanleitung)	
Nebenbedingung	must	Sprache	Java, Version 11	
Nebenbedingung	must	Bibliothek	JavaFX	
Nebenbedingung	could	Bibliothek	externe Bibliotheken für die Umsetzung 3D-GUI	
Nebenbedingung	must	Version	Verslonsverwaltung in Git	
Nebenbedingung	must	Build-Managment	Maven	