Programmentwurf für Check-Mate

Moritz Knapp
SICK AG

DAvid Schmidt
SICK AG

March 23, 2022

Contents

1 Intro

Da wir, David und Moritz, schon immer gerne Schach gegeneinander spielten um auch in Lernpausen unseren Geist nicht zu unterfordern, entschieden wir uns ein eigenes Schach zu programmieren. Zudem sind wir davon überzeugt, dass Schach eine perfekte Grundlage für objektorientiertes Programmieren bietet, aufgrund der vielen Figuren die jeweils gerade und schräge Züge individuell ausführen.

2 Clean-Architecture

Bereits vor dem Studium hat David schonmal ein Schach programmiert. Dabei handelte es sich um ca 3000 Zeilen hochineffizienten Arduino Code. In diesem Projrekt wollten wir unseren früheren Ichs beweisen wie viel eleganter so etwas umsetzbar ist. Der erste Schritt dazu ist die Clean-Architecture.

Zusammengefasst ermöglicht eine Clean Architecture die Veränderung der Umgebung, bzw. des Systems, ohne den eigentlichen Kern des Codes anpassen zu müssen. Das bedeutet, dass wir theoretisch unser Spiel um ein echtes Spielbrett mit LEDs und Arduino- Raspberry-Steuerung erweitern könnten, ohne unseren Schach-Code zu verädnern.

2.1 Entitys

Die Entitys sind der Kern unseres Spiels. Unabhängig von der Umgebung sind die einzelnen Figuren virtuell auf einem Feld positioniert und können umplatziert werden. Dadurch ergeben sich grundlegend Folgende Klassen:

- Board
- Square
- Piece

Ein Board besteht aus vielen Squares, auf denen jeweils ein Piece stehen kann. Um dies zu implementieren und später auch verschiedene Pieces zu benutzen, zeigt Figure 1 alle Klassen welche unser Kern der Clean-Architektur beinhalten soll.

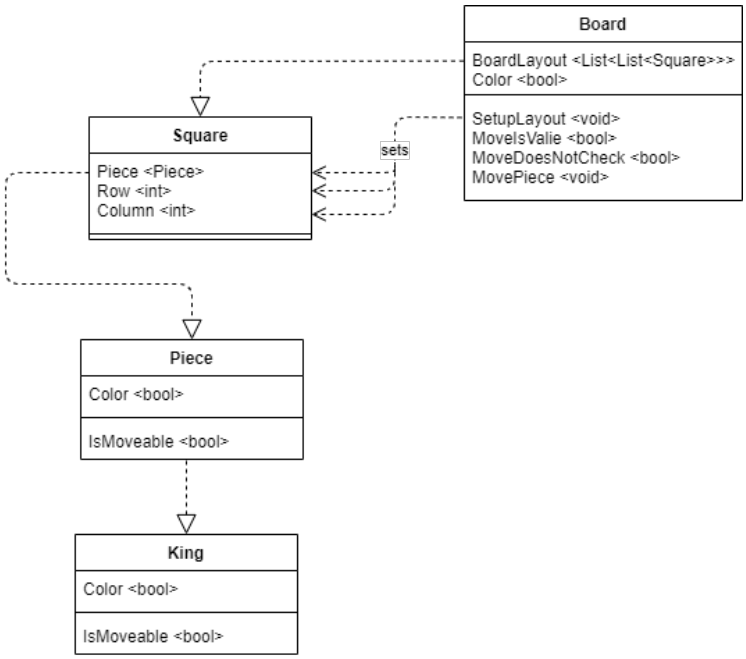

Figure 1: UML-Diagramm der Entitys

Das Diagramm zeigt die Schach-Logik im Kern unserer Clean Architecture, bzw. die Entitys. Das `Board` stellt das Schachbrett dar. Dieses besteht aus 64 `Squares`. Die Methode `SetupLayout` erstellt das Board und besetzt die richtigen Felder mit Pieces mit der entsprechenden Farbe. Von den eigentlichen `Pieces` ist hier nur der König gezeigt, da die anderen Figuren genau wie er die Methode `IsMoveable` implementieren. Das Board bietet drei Methoden, von denen nur eine tatsächlich von weiteren Schichten verwendet wird: `MovePiece`. Die Methode überprüft mit `MoveIsVali` ob es sich um einen Validen Zug handelt. `MoveIsValid` ruft dabei die von Figuren implementierte Methode `IsMoveable` auf um zu überprüfen ob die Figur theoretisch in der Lage wäre den Zug durchzuführen.
Der wesentliche Unterschied zwischen `MoveIsValid` und `IsMoveable` besteht darin, dass Figuren das Board nicht kennen und auch nicht wissen wo sie sich befinden. `IsMoveable` signalisiert lediglich, ob ein Zug rein von den allgemeinen Zugmöglichkeiten der Figur möglich ist (zB. Läufer kann nur schräg laufen). `MoveIsValid` ist eine Methode des Boards und kann somit auch situationsbedingte Auskunft über die Validität des Zuges geben. Wenn die Figur `IsMoveable` ist, wird von `MoveIsValid` noch überprüft ob weitere Figuren im Weg stehen und der Zug sich selbst in Schach setzen würde (`MoveDoesNotCheck`). Ist alles überprüft, wird der Zug durchgeführt.

2.2 Use-Case

Die Use-Cases bilden die zweite Schicht der Clean Architekture. Sie können informationen der ersten Schicht erhalten (Entitys), diese sind aber nicht abhängig von den Use-Cases.
In unserem Fall haben wir nur einen Use-Case: Das Spiel, bzw. `Game`. Diese Klasse erstellt das Board, bzw ruft die Funktion `SetupLayout` auf, hat aber noch weitere Informationen die über das einfache Ziehen von Figuren hinausgehen:

- Spielerposition (z.B. Spieler 1 spielt unten),
- Spielerfarbe (z.B. Spieler 1 ist weiß)
- und aktueller Spieler (z.B. Spieler 1 ist dran).

Der Use-Case initialisiert das Board, wodurch die Farbe des unten spielenden Spielers zufällig gewählt wird. Logisch betrachtet spielt für das `Game` Spieler 1 in jedem Fall *unten*. Da immer weiß beginnt, ist es zufällig ob Spieler 1 beginnen darf oder nicht. Welche Art von User-Interface nun Spieler 1 steuert, ist Aufgabe der dritten Schicht, welche in unserem Fall den Menschen an der Gui Spieler 1 und eine Schach-Engine Spieler 2 zuordnet. `Game` ist von dieser Information jedoch abgekapselt, da eine Zuganfrage in jedem Fall gleich aussieht. So eine Anfrage ist von `Game` definiert (z.B. 'e5,e6' = "Zug von e5 nach e6").
Wird nun eine solche Anfrage gestellt, verlangt `Game` von `Board` eine Auskunft über die Farbe der Figur, welche sich auf dem *von*-Feld befindet. Stimmt diese mit der Farbe des aktuell spielenden Spielers überein, kann der Zug nach Validitätsprüfung ausgeführt werden. Anhand dessen ob `MovePiece true` oder `false` zurückgibt, ist der andere Spieler daraufhin am Zug, oder nicht.

3 Programmierprizipien

3.1 pros/cons/wieso wir?

3.2 SOLID

3.2.1 Single Resposibility Principle

3.2.2 Open/Closed Principle

3.2.3 Liskov' Substitution Principle

3.2.4 Interface Segregation Principle

3.2.5 Dependency Inversion Principle

Auf Translator anwenden

3.3 GRASP

3.3.1 Low Coupling

3.3.2 High Cohesion

3.4 DRY (Don't Repeat Yourself)

Schon umgesetzt?

4 Unit Testing

4.1 pros/cons/wieso wir?

4.2 ATRIP

4.3 Code Coverage

4.4 Mocking

5 Refactoring

5.1 Beispiele und Begründungen

5.2 Code Smells

6 Entwurfsmuster

6.1 Diagramme und Begründungen

7 Fazit