Programmentwurf Tim-Robin Kalkhof

Inhaltsverzeichnis

Idee

Visuelles Schachprogramm und Schachbot

GUI-Programm, dass es ermöglicht Schach gegen einen 2.Menschen vor dem PC oder einen Schachbot zu spielen. Außerdem verschiedene Schachuhr Presets, für verschiedene Zeiteinstellungen. Hierbei wird die komplette Logik des Spieles Schach implementiert. Außerdem werden Negamax und Alpha-Beta Pruning Algorithmen verwendet, um einen Schachbot zu implementieren.

Hierbei verwendete Technologien: Java, Swing/FX, Junit, AssertJ, Mockito/Powermock, Gradle

Repository: https://github.com/CarbonTornado/ChessSWE

Unit Tests

Beim Testen der Square Klasse war eine große Priorisierung, dass die Tests Repeatable sein müssen, damit vermieden wird, dass die verschiedenen Tests für Interferenzen sorgen und unabhängig ausgeführt werden können.

Um die Abhängigkeiten bei der Image Loader Klasse zu reduzieren habe ich Mocks verwendet, damit man hier den Aspekt des Integration Tests vermeiden kann und trotzdem überprüfen kann, ob die Funktionalität korrekt ausgeführt wird. Weiterhin wird dadurch darauf geachtet, dass die Tests Independent ablaufen und keine äußeren Abhängigkeiten besitzen und im konkreten Fall kein Bild uaf der Maschine vorhanden sein muss.

Um die Tests auch Thorough umzusetzen wurde stets darauf geachtet möglichst alle Kritischen Aspekte zu testen. Aufgrund der Zeitbeschränkungen war es jedoch nicht möglich für jeden kritischen Bug einen Test nachzuliefern.

Ein weiter Aspekt der ATRIP Regeln, der hier beachtet wurde war der Aspekt, dass Tests automatisch ablaufen sollten. Dies ist hier der Fall, da ich auf externe Eingaben und Voraussetzungen verzichtet habe und so die Test autonom ablaufen können.

Testcode sollte leicht verständlich und professionell sein. Deswegen habe ich den Code in den einzelnen Methoden auf wenige Zeilen begrenzt und auch immer verschiedene Methoden für verschiedene Testszenarien verwendet und nicht eine Methode für mehrere. Damit ist es dann auch im fertigen Testreport besser einsehbar, welche Tests fehlschlagen und welche Softwarekomponente darauf basierend dann auch nicht funktioniert.

Bei der Code Coverage habe ich besonders den Wert auf die Branch Coverage gelegt, anstatt nur die Coverage Prozentzahl zu priorisieren. Der Grund hierfür war, dass es bei diesem Projekt nicht so viel Sinn ergeben hat auf die Code Coverage zu schauen, da die Zeit limitiert war und die Branch Coverage hier aussagekräftiger und einfacher zu erreichen ist.

Report

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	88.2% (15/17)	67.2% (45/67)	55.7% (205/368)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
<empty package name>	100% (1/1)	100% (3/3)	100% (5/5)
board	100% (2/2)	69.6% (16/23)	71.7% (99/138)
logic	0% (0/1)	0% (0/1)	0% (0/1)
pieces	90% (9/10)	61.3% (19/31)	40% (66/165)
utils	100% (2/2)	83.3% (5/6)	82.6% (19/23)
view	100% (1/1)	66.7% (2/3)	44.4% (16/36)

Für die Evaluation Klasse im Logic Package wurden keine Unit Tests implementiert, da diese Klasse selbst noch nicht implementiert worden ist. Deshalb ist die Coverage hier auch nicht vorhanden.

Clean Architecture

Für die Clean Architecture ist die Plugin Schicht Swift für das UI und die Test Frameworks AssertJ und Junit. Außerdem Mockito/PowerMock für Mocks, da sie externe Abhängigkeiten sind, die nur leicht gekoppelt sind.

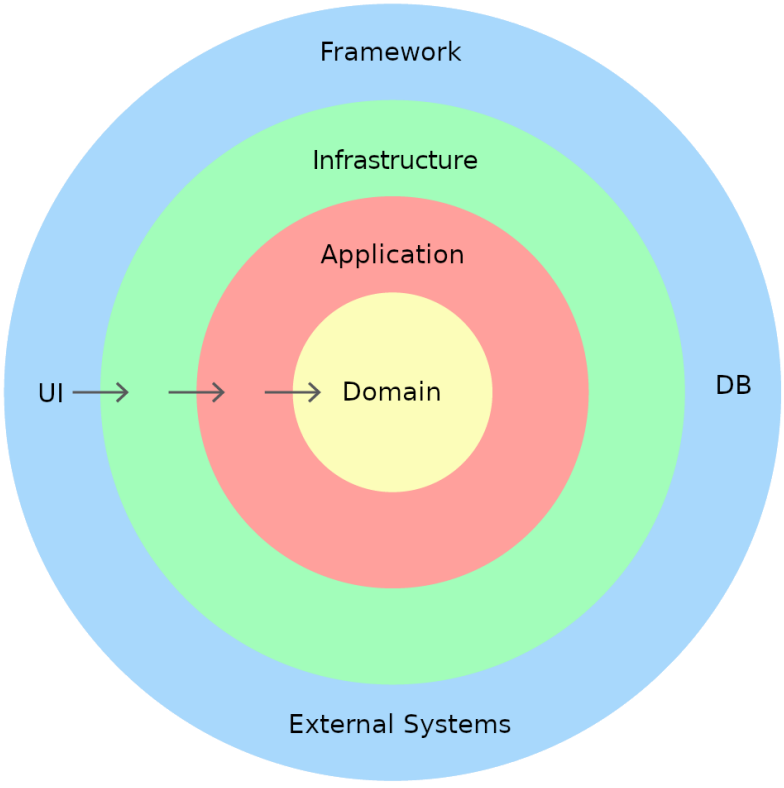
Auf der Adapter Ebene ist dann die GUI Klasse vertreten. Diese sorgt für die Verbindung zwischen den Plugins und der Applikation.

Auf der Applikationsebene ist die Board Klasse zu nennen, die Entities wie Beispielsweise Squares und Pieces auf der Domainebene verwaltet.

Weiterhin sind auf der Domain Ebene die Pieces als Entities vertreten.

Als Abstraction Ebene habe ich den Negamax Algorithmus bestimmt, da dieser auf mathematisch fundierten Grundlagen beruht und immer von Grund auf korrekt ist. Um den Negamax Algorithmus auf die Abstraction Ebene zu bringen musste das Dependecy Inversion Principle verwendet werden, da die Evaluationsklasse in der der Algorithmus enthalten war eine Abhängigkeit zur Board Klasse hatte.

Als Adapters Klasse kann man die Board Klasse nennen, da diese Die Logik zwischen den Figuren und dem GUI darstellt.



Plugins: Swing, Junit, AssertJ, (Game/Main)

↓

Adapters: GUI

↓

Application: Board

↓

Domain: Piece, Square

↓

Abstraction: Evaluate

Refactoring

Die Initierungsmethode der Figuren in der Board Klasse ist sehr Lang und Komplex und enthält viele Duplikate und bietet sich daher sehr für ein Refactoring an. Hierbei könnten viele Aufrufe durch Schleifen verkürzt werden und außerdem wird durch das bei den Pieces angewandte Factory Pattern die Erstellung der Figuren vereinfacht. Der Code Smell hierbei war die Long Method, da die Initialisierung nicht optimiert war. Als Refactoring wurde hier Extract Method verwendet. Das Commit zum Refactoring ist hier zu finden:

https://github.com/CarbonTornado/ChessSWE/commit/ea3686a6a20dda528da6759b9836da2c78043800

Außerdem war in der Queen Klassen noch Dead Code vorhanden, der dadurch entstanden ist, dass die Movement Logik sich in die Piece Klasse verschoben hatte. Das Refactoring in der dieser Code entfernt wurde ist hier zu finden: https://github.com/CarbonTornado/ChessSWE/commit/edcaedeed8c00a09fe5d51373ecc8a733c1b52ac

Weiterhin bat sich die Duplikation in der Pawn Klasse auch sehr für ein Refactoring an. Hierbei wurde dann der Code für beide Farben in ein If/Else Statementgebaut. Beim Editieren vom Code ist mir dort aufgefallen, dass es dort immer zu Shotgun Surgery, da es bei Änderungen der Bewegungsmuster des Bauern dazu kommt, dass für beide Farben die Logik geändert werden muss, obwohl sie sich nicht Grundlegend unterscheidet. Dies wurde mithilfe des Extract Variable Refactorings durchgesetzt und ist hier zu finden: https://github.com/CarbonTornado/ChessSWE/commit/c4d04aab74f5357e0d730c358cd17ecfad7abe14

Programming Principles

Die Piece Klasse wurde als abstrakte Klasse definiert, damit sich dann auch die verschiedenen Figuren daran orientieren können und keine eigene Methode für Bewegungen implementieren müssen.

Beim Single Responsibility Principle habe ich darauf geachtet, dass jede Klasse nur eine Zuständigkeit hat. Sodass sich die Square Klasse Beispielweise nur ändert wenn sich etwas an der Darstellung eines Feldes ändert. Zum Beispiel die Feldfarbe oder auch die besetzende Figur. Weiterhin habe ich für jede Klasse nur eine Aufgabe bestimmt.

Diese sind dann im Detail, dass die GUI Klasse nur für die grafische Darstellung verantwortlich ist (Anzeige des Boards und des Timers). Die ChessClock (Schachuhr) hat auch nur die genannte Funktion und zwar die einer Schachuhr.

Als Pure Fabrication Klasse hat die Image Loader Klassen keine Domainspezifische Aufgabe sondern ist lediglich verantwortlich für das Laden der Bilder, welche die verschiedenen Figuren darstellen.

Die Main(Game) Klasse hat die Verantwortung des Erstellens eines neuen Spiels und dazu das initiieren eines Threads. Die nicht implementierte Klasse Evaluate hätte nur die Funktionalität, dass sie den besten Evaluierten Zug zurückgeben würde für die gegebene Brettkonfiguration. Außerdem sind noch die ganzen Pieces zu nennen, die jeweils nur die Eigenschaften der Figuren abbilden und die Bewegung ermöglichen. Die kritische Klasse ist vor allem die Board Klasse, da sie für die Verwaltung der Figuren und der Felder verantwortlich ist und damit 2 Verantwortlichkeiten hat.

Beim Open / Closed Principle sind zwei Beispiele zu nennen. Einerseits die Enum Klasse Piece Color, welche es über Erweiterung ermöglicht mehr Farben hinzufügen. Andererseits gibt es hier auch Klassen die dagegen verstoßen. Dies lässt sich auch nicht vermeiden. Als Beispiel kann man hier die Erbenden Klassen der abstrakten Piece Klasse und der Piece Klasse generell nennen, bei denen man zur Änderung an den Bewegungsmustern Modifikationen vornehmen muss.

Die Piece Klasse folgt dem Liskov Substitution Principle, da sie Kovariant die Beziehung zwischen der Abstrakten Figur und der konkreten Figur darstellt.

Bei der Interface Segregation ist zu beachten, dass kein Interface im Programmentwurf implementiert wurde. Allerdings existiert die abstrakte Klasse Piece, die hier auch zutreffen könnte. Hierbei könnte man behaupten, dass diese nicht sehr spezifisch ist, da sie nicht nur Eigenschaften sondern auch noch die Bewegungsmöglichkeiten abbildet. Hier würde sich möglicherweise noch eine neue Klasse, Beziehungsweise ein weiteres Interface anbieten.

Die Dependency Inversion wäre noch implementiert worden, da der in der Evaluate Klasse enthaltene Negmax Algorithmus im Clean Architecture Model auf der Abstraction Ebene hätte sein sollen. Allerdings wurde dieser aus Zeitgründen nicht implementiert.

Bei den GRASP Principles ist für High Kohesion die Verbindung zwischen der Klasse Piece und Square mit der Klasse Board zu nennen.

Bei der Pure Fabrication Regel kommt die Image Loader und die PieceFactory Klassen zum Einsatz. Diese sollen sich natürlich auf wenige Klassen begrenzen. Sie sind vom Rest des Codes isoliert und dienen rein nur zum Laden der Bilder der Spielfiguren und zum instanziieren der Spielfiguren.

Weiterhin wurde hier der Polymorphismus verwendet um den verschiedenen Spielfiguren einen Aufbau vorzugeben und um zu erzwingen, dass sie die Move Methode implementieren. Weiterhin ist auch die Piece Klasse als abstract implementiert um sie vor kompletter Implementierung zu schützen und als Protected Variation darzustellen.

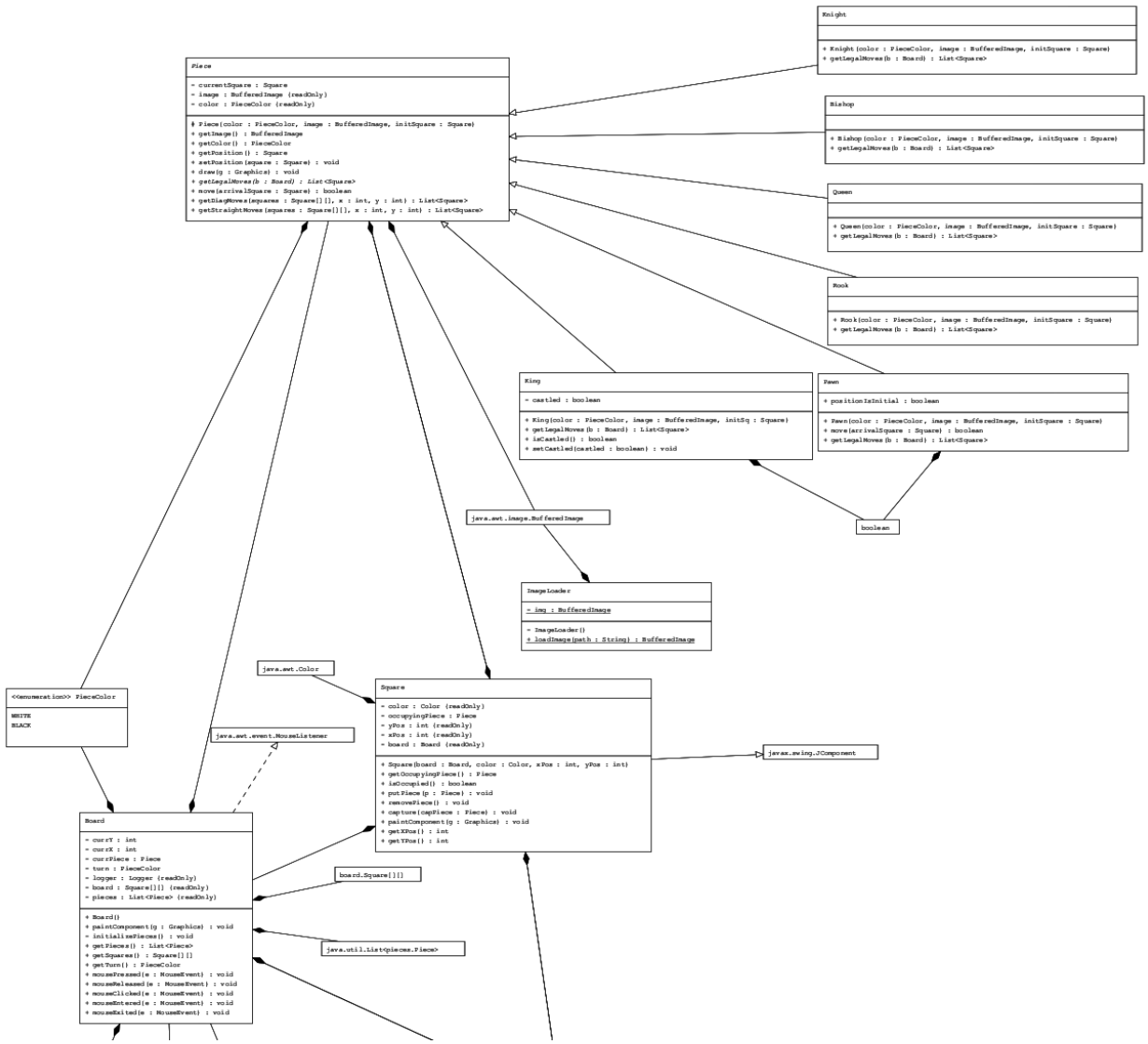
Low Coupling wurde in der Board Klasse implementiert, da dort auf die polymorphe Move Methode der Spielfiguren zugegriffen wird und diese so sehr leicht austauschbar sind.

Indirection wurde verwendet in der Board Klasse um die Figuren zum Bewegen zu bringen indem die Bewegung an die Pieces delegiert wird. Dadurch muss sich dann die Board Klasse nicht mehr um die weitere Logik kümmern.

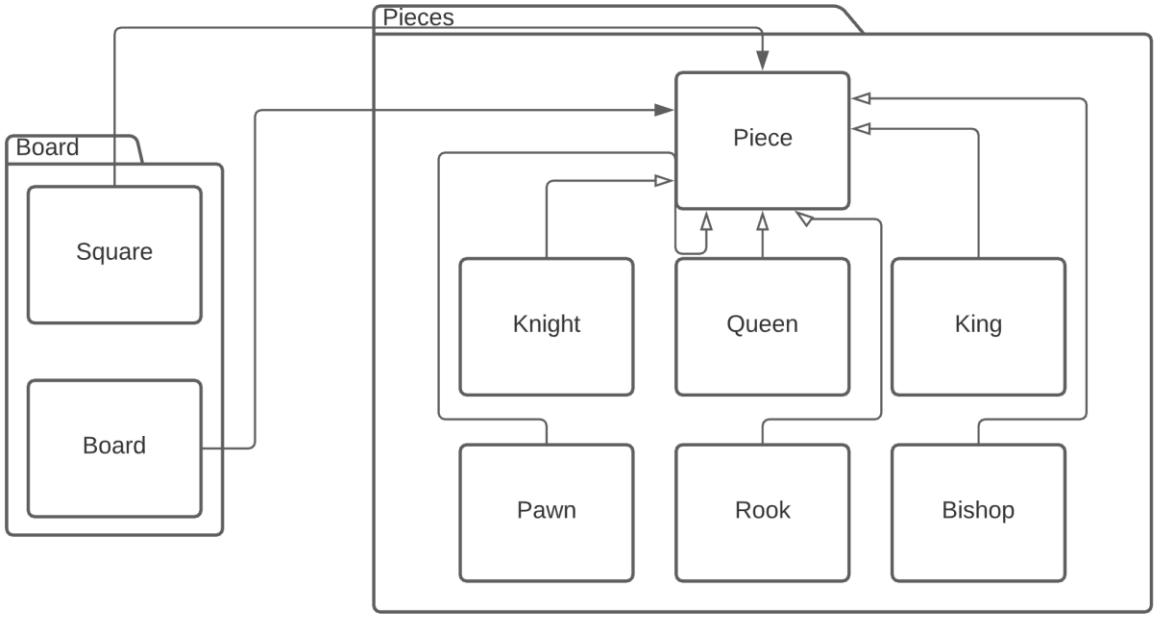
Entwurfsmuster

Für die Piece Klasse habe ich mich entschlossen eine Factory zu verwenden. Ich hatte mich für eine Factory Methode entschieden, da es das instanziieren der verschiedenen Figuren deutlich übersichtlicher macht, wenn ein Factory verwendet wird. Damit wird auch das Problem der Long Method in der Initialisierungsmethode des Boards verbessert und diese übersichtlicher gemacht (siehe Refactoring).

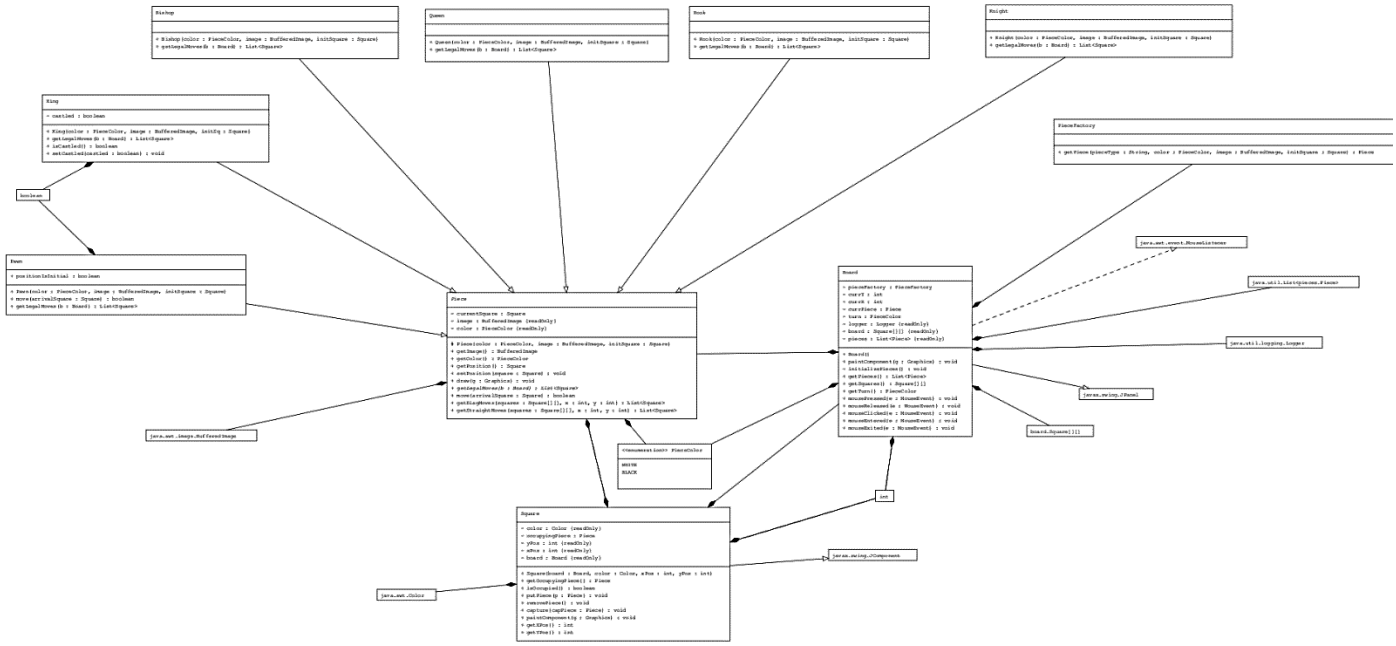
Vorher:



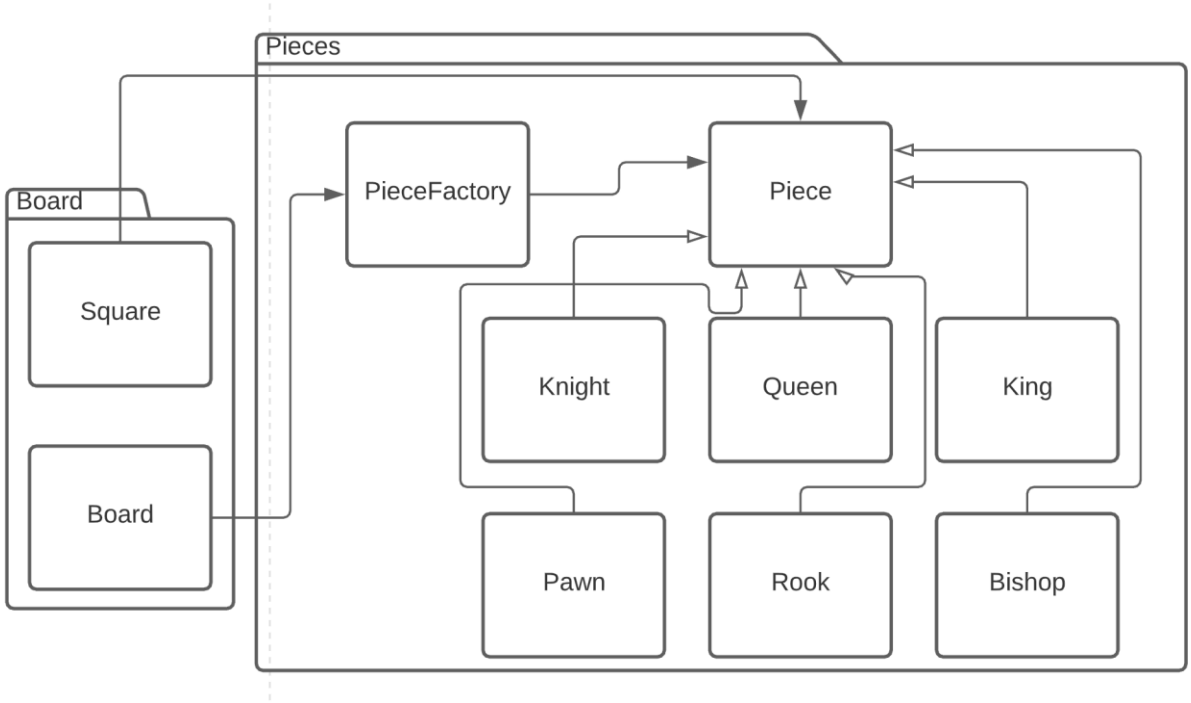
Besser Lesbar:



Nacher:



Besser Lesbar:


Pieces
Board
Square
Board
PieceFactory
Piece
Knight
Queen
King
Pawn
Rook
Bishop

Domain Driven Design

Bei Domain Driven Design habe ich viel Wert darauf gelegt, dass die Ubiquitous Language analysiert wurde und sinnvoll umgesetzt wird.

Die Benennung der verschiedenen Klassen wie auch entsprechende Spielfiguren, wurde Domainengerecht durchgesetzt.

Dies ist Beispielsweise an der Move Methode zu sehen, die jede Figur implementiert. Über die sich die Figur (Piece) von einem Feld auf dem Brett (Board) auf ein anderes bewegt.

Die Entities sind unter anderem die Klassen, die dem Piece Interface zugeordnet sind. Das sind Pawn (Bauer), King (König), Knight (Springer), Rook (Turm), Bishop (Läufer) und Queen (Dame). Weiterhin sind noch die Squares zu nennen, diese sind ebenfalls Entities, da sie ebenfalls veränderlich sind in Bezug auf die besetzende Figur sind.

Hierbei agiert dann die Klasse Board als Aggregate für alle Felder (Squares), da alles Squares nur über die Boardklasse ansprechbar sind, indem man sich die Liste aller verfügbaren Squares zurückgeben lässt.

Als Value Object ist dann Beispielsweise die Piece Color Klasse zu nennen, die die Farbe der Figur angibt. Die Farbe der Figur ist hierbei nicht veränderlich.