
Abschlussaufgabe 2 Match Three

Ausgabe: 24.02.2017 – 13:00
Abgabe: 24.03.2017 – 13:00

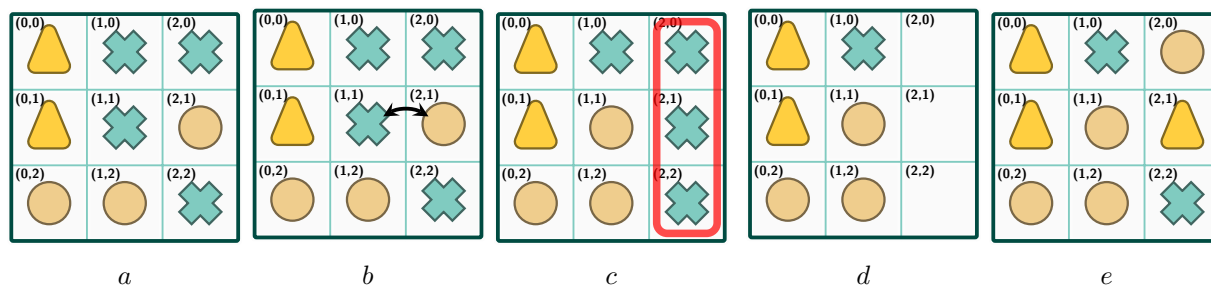
- Achten Sie darauf nicht zu lange Zeilen, Methoden und Dateien zu erstellen¹
- Programmcode muss in englischer Sprache verfasst sein
- Kommentieren Sie Ihren Code angemessen: So viel wie nötig, so wenig wie möglich
- Wählen Sie geeignete Sichtbarkeiten für Ihre Klassen, Methoden und Attribute
- Verwenden Sie keine Klassen der Java-Bibliotheken ausgenommen Klassen der Pakete `java.lang`, `java.io` und `java.util` und deren Unterpakete, es sei denn die Aufgabenstellung erlaubt ausdrücklich weitere Pakete¹
- Achten Sie auf fehlerfrei kompilierenden Programmcode¹
- Halten Sie alle Whitespace-Regeln ein¹
- Halten Sie die Regeln zu Variablen-, Methoden und Paketbenennung ein und wählen Sie aussagekräftige Namen¹
- Halten Sie die Regeln zu Javadoc-Dokumentation ein¹
- Nutzen Sie nicht das default-Package¹
- Halten Sie auch alle anderen Checkstyle-Regeln ein
- `System.exit` und `Runtime.exit` dürfen nicht verwendet werden¹

Abgabemodalitäten

Die Praktomat-Abgabe wird am **Freitag, den 10. März 2017, um 13:00 Uhr**, freigeschaltet. Geben Sie die Java-Klassen als `.java`-Dateien ab. Geben Sie **nicht** die bereitgestellten Klassen in `edu.kit.informatik.matchthree.framework`, `edu.kit.informatik.matchthree.framework.exceptions` und `edu.kit.informatik.matchthree.framework.interfaces` ab.

Bitte beachten Sie, dass das erfolgreiche Bestehen der öffentlichen Tests für eine erfolgreiche Abgabe nötig ist. Planen Sie ausreichend Zeit für Abgabeveruche ein, sollte der Praktomat Ihre Abgabe wegen einer Regelverletzung ablehnen.

¹Der Praktomat wird die Abgabe zurückweisen, falls diese Regel verletzt ist.


 Abbildung 1: Einfaches Beispiel für ein *match three*-Spiel




Einführung

In dieser Aufgabe implementieren Sie verschiedene Teile eines *match three*-Spiels. Diese Art von Spiel hat das Ziel, ein Spielbrett, das mit Spielsteinen gefüllt ist, durch Spielzüge so zu verändern, dass sich vorgegebene Muster ergeben, sogenannte Treffer (engl. *matches*). Die betroffenen Spielsteine werden dann vom Spielbrett entfernt und durch andere Spielsteine ersetzt. Der Name *match three* kommt daher, dass klassischerweise die Anzahl der Spielsteine in einem Treffer drei oder mehr beträgt. Populäre Vertreter dieser Art von Spiel sind beispielsweise Bejeweled² oder Candy Crush Saga³.

Die auf diesem Aufgabenblatt betrachtete Variante dieser Art von Spiel ist die folgende: Elemente auf einem schachbrettförmigen Spielbrett werden vom Spieler nach definierten Zugregeln umgeordnet. Nach jedem Zug werden alle Muster, die von einem festgelegten Regelsatz abgedeckt sind, mit Punkten belohnt und komplett vom Spielbrett entfernt. Dadurch entstehende Lücken werden durch von oben nachrückende Steine aufgefüllt. Entweder handelt es sich bei den nachrückenden Steinen um bereits auf dem Spielbrett vorhandene oder, falls eine Lücke am oberen Rand entsteht, durch neue, zufällig ausgewählte Steine. Bei diesem Nachrücken kann es dazu kommen, dass direkt weitere Muster entstehen und wiederum vom Spielbrett entfernt und mit Punkten belohnt werden. Dies wird als *Kettenreaktionen* bezeichnet.

Variationen in möglichen Spielinstanzen ergeben sich durch die Größe des Spielbretts, die Menge an Regeln für erlaubte Züge, die Menge von Regeln für Treffer und die Menge an vorkommenden Spielsteinen.

In Abbildung 1 ist ein Beispiel für eine einfache Instanz eines solchen Spiels gezeigt. Die Regeln **für diese Instanz** sind die folgenden:

- Das Spiel findet auf einem 3x3-Spielbrett statt.
- Ein Treffer sind drei direkt aufeinanderfolgende Spielsteine des gleichen Typs in einer Reihe oder in einer Spalte.
- Der einzige erlaubte Zug ist das Vertauschen von zwei horizontal benachbarten Steinen.
- Die vorkommenden Spielsteine sind ,  und .

In Abbildung 1a liegt zunächst kein Treffer vor. Durch das Vertauschen der Spielsteine (1,1) und (2,1) (Abbildung 1b) ergibt sich der in c markierte Treffer in Spalte 3 ((2,0), (2,1), (2,2)). Diese Steine werden entfernt (d) und durch zufällige Steine ersetzt (e). Ein anschließend möglicher Zug, der erneut zu einem Treffer führen würde, wäre das Vertauschen der Steine (1,0) und (2,0).


²<http://www.bejeweled.com/>









³<http://candycrushsaga.com>

Begrifflichkeit und Notation

Spielbrett

Ein *Spielbrett* hat die in Abbildung 2 dargestellte Form. Es besteht aus einer festgelegten Anzahl an Reihen und Spalten von *Feldern*. Jedes Feld hat eine eindeutige *Position*. Diese ist in der linken oberen Ecke des Felds dargestellt, beispielsweise (0,1). Hierbei steht an erster Position die Spalte (*x-Koordinate*) und an zweiter Position die Zeile (*y-Koordinate*). Beide werden von 0 ausgehend gezählt. Ein Feld an Position (*x,y*) wird auch kurz als „Feld (*x,y*)“ bezeichnet.

Ein Feld eines Spielbretts kann mit genau einem *Spielstein* (engl. *token*) belegt sein. Jeder Spielsteintyp ist fest mit einem Textsymbol assoziiert, welches auf diesem Aufgabenblatt zusätzlich graphisch dargestellt wird. Beispielsweise hat das Feld (0,0) in Abbildung 2 den Spielstein . Der Spielstein auf Feld (0,0) wird im Folgenden auch kurz als „Spielstein (0,0)“ bezeichnet. Einen Sonderfall bildet ein nicht belegtes Feld, welches durch den Text `null` dargestellt wird.

(0,0) 	(1,0) 	(2,0) 
(0,1) 	(1,1) 	(2,1) null
(0,2) 	(1,2) 	(2,2) 

`A*s;OX␣;+Y=`

Abbildung 2: Spielbrett

Unter dem Spielbrett ist im Folgenden immer eine textuelle Repräsentation des Spielbretts gegeben. In Abbildung 2 ist dies `A*s;OX␣;+Y=`. Hierbei ist jede Zeile durch die darin vorkommenden Spielsteine repräsentiert. Die einzelnen Zeilen sind durch Semikola (;) getrennt. Ein nicht belegtes Feld wird durch ein Leerzeichen (␣) repräsentiert.

Treffer

In der Beschreibung wird für ein auf dem Spielbrett gefundenes Muster der Begriff „Treffer“ verwendet.

Positionen und Deltas

Aus Platzgründen werden auf dem Aufgabenblatt Kurzschreibweisen verwendet. Falls nicht anders notiert, steht (*x,y*) für die Position oder das Delta mit den angegebenen *x*- und *y*-Koordinaten, auch innerhalb von Java-Code. Ein Delta ist eine Tupel aus *x*- und *y*-Distanz. Ein Delta kann auf eine Position addiert werden und ergibt erneut eine Position. In Codeausschnitten ist entsprechend (*x,y*) mit `new Position(x, y)`, `Position.at(x, y)`, `new Delta(x, y)` oder `Delta.dxy(x, y)` gleichzusetzen.

Mengen (Sets)

Mengen werden in mathematischer Mengennotation dargestellt. Auch Java-Mengen (`java.util.Set<T>`) werden durch die Schreibweise `{ a, b, c, ... }` ausgedrückt.

Aufgabenstellung

Laden Sie sich als Grundlage für die Implementierung das auf der Webseite⁴ zur Verfügung gestellte Framework⁵ herunter. Dieses enthält neben den von Ihnen zu implementierenden Schnittstellen auch ein Grundgerüst (ohne Kommentare oder implementierte Funktionalität), das Sie als Ausgangspunkt für Ihre Implementierung verwenden können. Weiterhin enthalten ist ein beispielhafter JUnit⁶-Test (`edu.kit.informatik.matchthree.tests`), den Sie als Ausgangspunkt für eigene Tests verwenden **können**. Tests werden **nicht** mit abgegeben oder bewertet, sondern können Ihnen helfen, die Qualität Ihrer Abgabe zu sichern.

Laden Sie die Klassen aus dem Rahmenwerk (im Paket `edu.kit.informatik.matchthree.framework`) **nicht** mit Ihrer Abgabe hoch. Ändern Sie die vorgegeben Klassen nicht, sondern arbeiten Sie zur Erweiterung mit Komposition oder Vererbung. Erstellen Sie keine `main`-Methode im von Ihnen abgegebenen Code. Legen Sie alle von Ihnen implementierten Klassen in das Paket `edu.kit.informatik.matchthree`. Verwenden Sie die im Rahmenwerk enthaltenen Exceptions (in `edu.kit.informatik.matchthree.framework.exceptions`). Die Klassen `DeterministicStrategy` und `RandomStrategy` müssen Sie nicht zwingend verstehen oder verwenden, können dies aber zu Testzwecken tun.

Achten Sie bei der Implementierung darauf, nicht gegen Ihre Implementierung, sondern gegen die definierten Schnittstellen zu programmieren. Das bedeutet, dass Sie beispielsweise bei der Implementierung von `Matcher`, `Move` oder `Game` nicht davon ausgehen können, dass deren Methoden nur mit Ihrer Implementierung (von `MatchThreeBoard`) aufgerufen werden, sondern während dem automatisierten Testen im Praktomat auch mit Implementierungen der Musterlösung aufgerufen werden, um Punktabzüge durch Folgefehler zu vermeiden. Daher sollen Sie insbesondere **nicht** mit `instanceof` und Typ-Konvertierungen (engl. *casts*) eine Bindung an Ihre Implementierung erzeugen. Erstellen Sie in den vorgegebenen Klassen `MatchThreeBoard`, `MoveFactoryImplementation`, `MaximumDeltaMatcher` und `MatchThreeGame` daher neben den auf dem Aufgabenblatt vorgegebenen öffentlichen Konstruktoren und den durch die Schnittstelle vorgegebenen Methoden nur eigene **private** Methoden. Sie können neben den vorgegebenen Klassen weitere öffentliche Klassen, Schnittstellen und Enumerables mit beliebigen (auch statischen) Methoden mit von Ihnen gewählter Sichtbarkeit anlegen.

Die Aufgabe ist im Folgenden in die vier Teile A bis D gegliedert. Sie können diese Aufgabenteile prinzipiell unabhängig voneinander bearbeiten. Insbesondere können Sie eine Untermenge der Aufgabenteile implementieren und abgeben, ohne die anderen Teile komplett oder korrekt bearbeitet zu haben. Mindestvoraussetzung für die Abgabe ist der öffentliche (*required*) Test im Praktomat. Um Ihre Implementierung über den im Praktomat sichtbaren öffentlichen Test hinaus auch selbst zu testen und um Ihr Verständnis zu erleichtern empfiehlt sich aber die (vollständige) Bearbeitung in der beschriebenen Reihenfolge.

⁴https://sdqweb.ipd.kit.edu/wiki/Vorlesung_Programmieren_WS16/17

⁵<https://sdqweb.ipd.kit.edu/lehre/WS1617-Programmieren/final02-framework.zip>

⁶Ein Framework zum Testen von Java-Programmen (<http://junit.org>). Eine kurze Einführung in das Framework wurde in der Vorlesung gegeben und ist auf der Webseite der Vorlesung verfügbar (https://sdqweb.ipd.kit.edu/lehre/WS1617-Programmieren/19_Introduction_JUnit.pdf)

A Spielbrett

Die vorgegebene Schnittstelle `Board` spezifiziert die Funktionalität eines Spielbretts. Implementieren Sie eine Klasse `MatchThreeBoard` im Paket `edu.kit.informatik.matchthree` die diese Schnittstelle implementiert. Im Folgenden sind die in der Schnittstelle enthaltenen Methoden kurz erläutert. Weitere Beschreibungen finden Sie im JavaDoc der Schnittstelle.

A.1 `public class MatchThreeBoard extends Board`

Ein Spielbrett hat immer mindestens 2 Zeilen und 2 Spalten, ansonsten wird beim Erstellen eine `BoardDimensionException` geworfen. Für ein Spielbrett müssen immer mindestens 2 unterschiedliche Tokens (neben dem nicht vorhandenen Token) valide sein, sonst wird eine `IllegalArgumentException` geworfen. Wird an eine Methode eine Position übergeben, die nicht auf dem Spielbrett liegt, wird eine `BoardDimensionException` geworfen. Dies gilt nicht für die Methode `#containsPosition(Position)`. Werden an eine Methode sonstige nicht-valide Parameter übergeben wird eine `IllegalArgumentException` geworfen.

Ihre Klasse muss neben den in der Schnittstelle festgelegten Methoden die folgenden Konstruktoren anbieten:

A.1.1 Konstruktor: `MatchThreeBoard(Set<Token> tokens, int columnCount, int rowCount)`

Erzeugt ein neues leeres Spielbrett mit der vorgegebenen Spalten- und Zeilenanzahl, auf dem die in `tokens` enthaltenen Spielsteine vorkommen können.

A.1.2 Konstruktor: `MatchThreeBoard(Set<Token> tokens, String tokenString)`

Erzeugt ein neues Spielbrett, auf dem die in `tokens` enthaltenen Spielsteine vorkommen können. Das Spielbrett wird mit dem gegebenen `tokenString` initialisiert dieser besteht aus durch Semikola (;) getrennten Beschreibungen der Zeilen des Spielbretts. Jedes Zeichen in der Beschreibung einer Zeile steht für genau einen Spielstein. Ein Leerzeichen () steht für einen nicht vorhandenen Spielstein. Für den `tokenString` `A*s;OX;+Y=` ergibt sich das in Abbildung 2 dargestellte Spielbrett. Der `tokenString` `;;` ergibt ein leeres Spielbrett der Höhe 2 und Breite 2. Jede Zeile des `tokenStrings` muss genau die gleiche Länge haben.

Der Konstruktor wirft eine `TokenStringParseException`, falls nicht genau das spezifizierte Format eingehalten wird, oder falls ein Spielstein in `tokenString` vorkommt, der nicht in `tokens` enthalten ist.

A.1.3 `String toTokenString()`

Gibt eine textuelle Repräsentation des Spielbretts zurück, die genau die für Konstruktor `#MatchThreeBoard(Set<Token>, String)` beschriebene Form hat.

Es muss also insbesondere immer möglich sein, eine Kopie eines Spielbretts `board` anzulegen, indem `new MatchThreeBoard(board.getAllValidTokens(), board.toTokenString())` aufgerufen wird.

A.1.4 `int getColumnCount()` / `int getRowCount()`

Gibt die Spaltenanzahl (Breite) bzw. Zeilenanzahl (Höhe) des Spielbretts zurück.

A.1.5 `Set<Token> getAllValidTokens()`

Gibt alle Tokens zurück, die aufgrund der Initialisierung im Konstruktor auf dem Spielbrett platziert werden können.

A.1.6 `Token getTokenAt(Position position)`

Gibt den Token an der gegebenen Position zurück. Falls an der gegebenen Position kein Token vorhanden ist, gibt die Methode `null` zurück.

A.1.7 void setTokenAt(Position position, Token token)

Setzt den Token an der gegebenen Position. Hat `token` den Wert `null`, wird der Token an der gegebenen Position entfernt.

A.1.8 boolean containsPosition(Position position)

Gibt genau dann `true` zurück, falls die übergebene Position auf dem Spielbrett liegt.

A.1.9 void swapTokens(Position positionA, Position positionB)

Vertauscht die Spielsteine an den gegebenen Positionen. Ist für eine übergebenen Positionen das entsprechende Feld leer, wird der Spielstein vom vollen auf das leere Feld bewegt. Sind beide Felder leer, passiert nichts.

A.1.10 void removeTokensAt(Set<Position> positions)

Entfernt für alle Positionen in `positions` den Spielstein. Felder, die nicht mit einem Spielstein belegt sind, werden ignoriert. Ist eines der Felder nicht auf dem Spielbrett wird eine passende Exception geworfen, und keines der Felder des Spielbretts verändert sich.

A.1.11 void setFillingStrategy(FillingStrategy strategy)

Setzt eine Füllstrategie für das Spielbrett. Diese wird nur in der Methode `#fillWithTokens()` verwendet. Wird für `strategy` der Wert `null` übergeben, wird eine `NullPointerException` geworfen.

Hinweis: Zwei beispielhafte Implementierungen von Strategien, die das Spielbrett mit zufälligen und vorgegebenen Tokens füllen sind in `RandomStrategy` bzw. `DeterministicStrategy` im Framework enthalten.

A.1.12 void fillWithTokens()

Wendet die in `#setFillingStrategy(FillingStrategy)` gesetzte Strategie auf das Spielbrett an. Dies geschieht mit `FillingStrategy#fill(Board)`. Wurde keine derartige Strategie zuvor durch die entsprechende Methode gesetzt, wird eine `NoFillingStrategyException` geworfen.

A.1.13 Set<Position> moveTokensToBottom()

Bewegt alle auf dem Spielbrett vorhandenen Spielsteine soweit wie möglich nach unten. Dies bedeutet eine Bewegung in die Position mit der gleichen x - und der höchsten möglichen unbelegten y -Koordinate, die ohne ein Überspringen anderer Spielsteine erreichbar ist. Somit sind nach Ausführen der Methode nur noch freie Felder am oberen Rand des Spielbretts vorhanden oder es sind keine Felder mehr frei:

$$\forall x, y : ((x, y) \text{ ist null} \Rightarrow (\forall y' : 0 \leq y' < y \Rightarrow (x, y') \text{ ist null}))$$

Der Vorgang ist beispielhaft in Abbildung 3 dargestellt. Die Menge aller Positionen auf dem entstandenen Spielbrett, deren Belegung sich geändert hat wird zurückgegeben. Dies kann auch Felder beinhalten, deren Belegung sich nicht verändert hat, aber die trotzdem bewegt wurden, wie (2,2) in Abbildung 3. (3,2) würde in in diesem Beispiel nicht zurückgegeben.

Ist das Spielbrett bereits lückenlos besetzt oder Lücken bereits soweit möglich von oben aufgefüllt, passiert nichts. Somit hat ein wiederholter Aufruf der Methode ohne dazwischenliegende andere Änderungen nach dem ersten Aufruf keinen weiteren Effekt. Dann gibt die Methode eine leere Menge (nicht `null`) zurück.

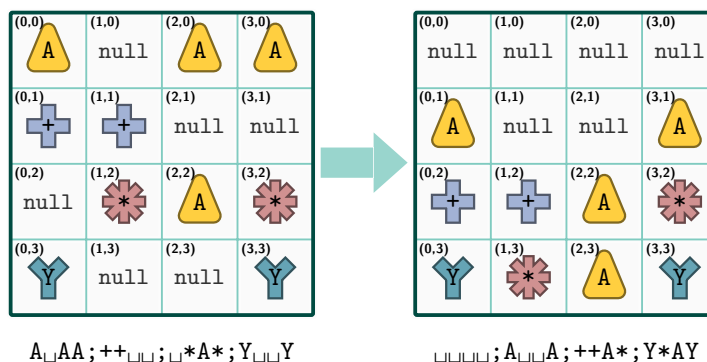


Abbildung 3: Bewegung der Spielsteine nach unten.

B Züge

Implementieren Sie eine Klasse `MoveFactoryImplementation`, die das Interface `MoveFactory` implementiert. Entwerfen Sie hierfür eine oder mehrere sinnvolle Klassen, die das Interface `Move` implementieren und geben Sie beim Aufruf der jeweiligen Methode von `MoveFactoryImplementation` einen `Move` zurück, der den im Folgenden beschriebenen Effekt hat. Im Folgenden wird zunächst die Schnittstelle `Move` und daraufhin die Schnittstelle von `MoveFactory` beschrieben, die auch die von Ihnen zu implementierenden konkreten Züge beschreibt.

B.1 `public interface Move`

Bitte beachten Sie, dass ein `Move` nicht bei der Erstellung „automatisch“ auf einem Spielbrett ausgeführt wird, sondern erst beim Aufruf der Methode `apply`. Ein Spielzug kann auch auf leere Felder angewandt werden.

B.1.1 `boolean canBeApplied(Board board)`

Gibt zurück, ob der Spielzug auf dem gegebenen Spielbrett ausgeführt werden kann. Ein Spielzug kann genau dann ausgeführt werden, wenn nicht auf Felder außerhalb des Spielbretts zugegriffen wird. Beispielsweise kann der Zug `moveFactory.flipRight((2,0))` auf dem in Abbildung 4 dargestellten Spielbrett (grün) nicht ausgeführt werden, auf dem in Abbildung 5 dargestellten (blau) aber schon.

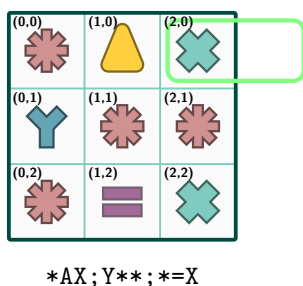


Abbildung 4: Beispiel für nicht anwendbaren Zug

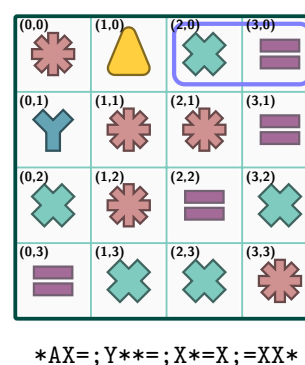


Abbildung 5: Beispiel für anwendbaren Zug

B.1.2 `void apply(Board board)`

Führt den angegebenen Zug auf dem Spielbrett `board` aus. Falls der Zug nicht ausgeführt werden kann, weil die zu manipulierenden Felder nicht zur Größe des Spielbretts passen, wird eine `BoardDimensionException` geworfen.

B.1.3 `Move reverse()`

Gibt einen Zug zurück, der die „Umkehrung“ dieses Zugs bedeutet.

Umkehrung bedeutet in diesem Kontext, dass für einen `Move m` und ein festes `Board b` immer gilt, dass das Ausführen von `m.apply(b); m.reverse().apply(b);` das Spielbrett wieder in den Ausgangszustand versetzt, ebenso ein Aufruf von `m.reverse().apply(b); m.apply(b);`. Insbesondere erhält man durch `Move m2 = m.reverse().reverse()` einen Zug `m2`, dessen Anwendung die gleiche Auswirkung auf jedes Spielbrett hat, wie die Anwendung von `m`.

B.1.4 `Set<Position> getAffectedPositions(Board board)`

Gibt alle Positionen auf dem Spielbrett `board` zurück, die von dem Zug betroffen sein werden, also verändert werden. Es werden auch Positionen zurückgegeben, deren Spielsteintyp sich effektiv nicht ändert, da er durch den gleichen Spielsteintyp ersetzt wird.

B.2 `public class MoveFactoryImplementation implements MoveFactory`

Die zu implementierenden Züge sind beispielhaft in den Abbildungen 6 bis 13 dargestellt. Hierbei sind die Ausführungen mit `#apply(Board)` und `#reverse()` mit einem auf das Ergebnis angewandten `#apply(Board)` entsprechend markiert.

B.2.1 Konstruktor: `public MoveFactoryImplementation()`

Ihre Klasse muss einen Konstruktor anbieten, der keine Parameter entgegennimmt.

B.2.2 `Move flipRight(Position position)`

Gibt einen Zug zurück, der bei Durchführung die gegebene Position `position` und das Feld rechts davon vertauscht (Abbildung 6). Die umgekehrte Ausführung dieses Zugs (`reverseApply`) ist gleich der normalen Ausführung.

B.2.3 `Move flipDown(Position position)`

Gibt einen Zug zurück, der bei Durchführung die gegebene Position `position` und das Feld darunter vertauscht (Abbildung 7). Die umgekehrte Ausführung dieses Zugs (`reverseApply`) ist gleich der normalen Ausführung.

B.2.4 `Move rotateSquareClockwise(Position position)`

Gibt einen Zug zurück, der das Rechteck aus den folgenden Feldern im Uhrzeigersinn „dreht“:

(i) `position`, *(ii)* `position + (1,0)`, *(iii)* `position + (1,1)`, *(iv)* `position + (0,1)`.

(i) nimmt also den Platz von *(ii)* ein, *(ii)* den von *(iii)*, *(iii)* den von *(iv)* und *(iv)* den von *(i)*. Dies ist in Abbildung 8 und Abbildung 9 dargestellt.

B.2.5 `Move rotateColumnDown(int columnIndex)`

Rotiert die angegebene Spalte nach oben und fügt das oberste Feld in der Spalte am unteren Rand des Spielbretts wieder ein. Dies ist in Abbildung 10 und Abbildung 11 exemplarisch dargestellt.

B.2.6 `Move rotateRowRight(int rowIndex)`

Rotiert die angegebene Zeile nach rechts und fügt das rechteste Feld in der Zeile am linken Rand des Spielbretts wieder ein. Dies ist in Abbildung 12 und Abbildung 13 exemplarisch dargestellt.

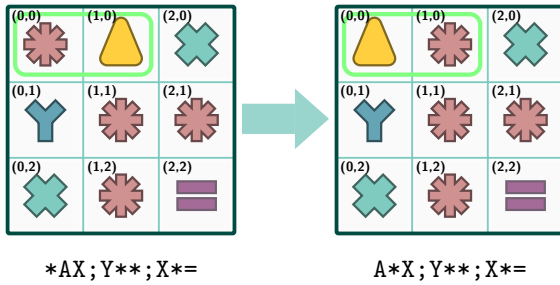


Abbildung 6:
`flipRight((0,0)).apply/.reverse().apply`

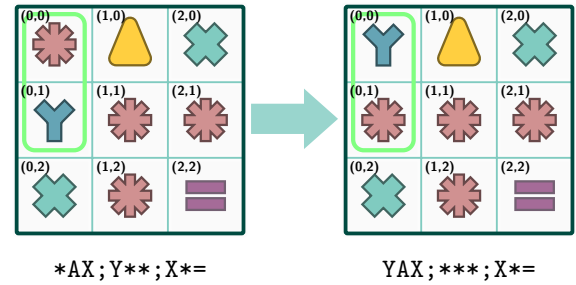


Abbildung 7:
`flipDown((0,0)).apply/.reverse().apply`

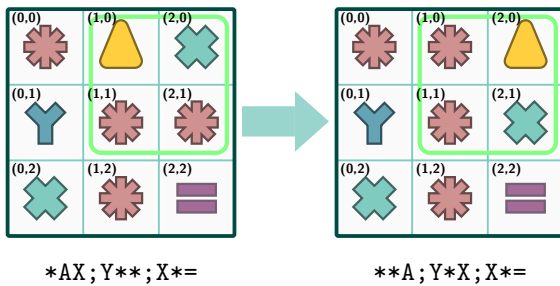


Abbildung 8:
`rotateSquareClockwise((1,0)).apply`

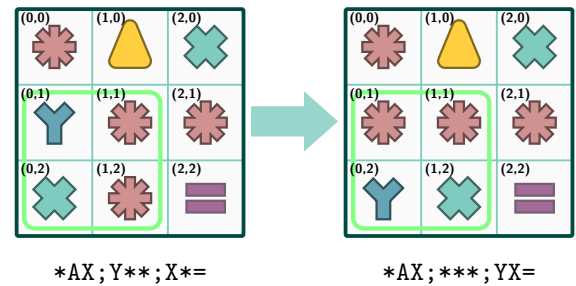


Abbildung 9:
`rotateSquareClockwise((0,1)).reverse().apply`

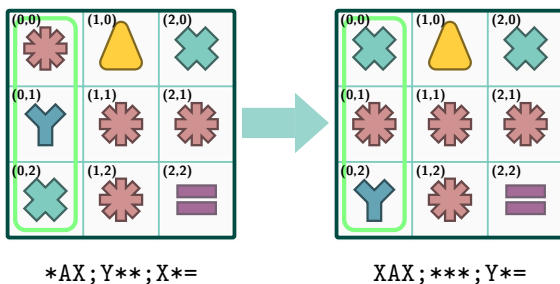


Abbildung 10: `rotateColumnDown(0).apply`

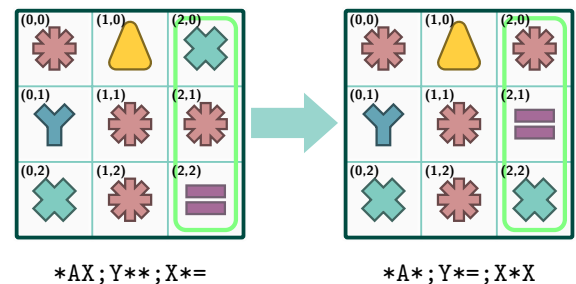


Abbildung 11:
`rotateColumnDown(2).reverse().apply`

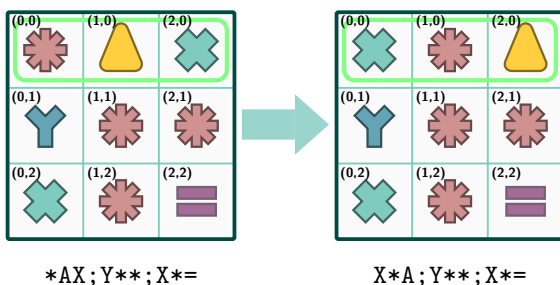


Abbildung 12: `rotateRowRight(0).apply`

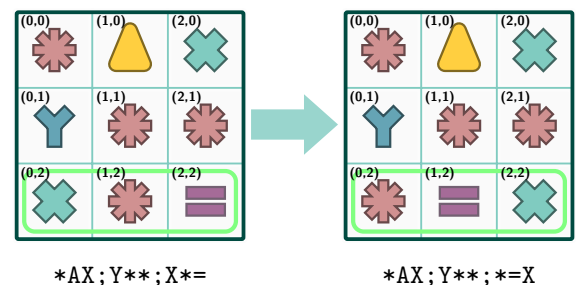


Abbildung 13:
`rotateRowRight(2).reverse().apply`

C Matching

C.1 `public interface Matcher`

Welche Felder einen Treffer ergeben, hängt von den Regeln des Spiels ab und wird in konkreten Implementierungen der Schnittstelle ausgeprägt. Die allgemeine Schnittstelle für eine Klasse, die Treffer erkennt, ist in `Matcher` definiert. In der Java-Schnittstelle ist ein Treffer durch eine Menge von Positionen, also `Set<Position>` repräsentiert. Damit ist eine Menge von Treffer durch ein `Set` von `Set<Position>`s repräsentiert, was zum Typ `Set<Set<Position>>` führt. Leere Felder sind nie in einem Treffer enthalten. Im Folgenden werden zunächst kurz die Methoden der Schnittstelle `Matcher` und daraufhin der konkrete von Ihnen zu implementierende `Matcher` beschrieben:

C.1.1 `Set<Set<Position>> match(Board board, Position initial)`

Gibt alle von einer Position `initial` ausgehenden Treffer zurück. Dies beinhaltet auch Treffen, die weniger als drei Positionen enthalten. Ein Treffer enthält jedoch mindestens eine Position.

C.1.2 `Set<Set<Position>> matchAll(Board board, Set<Position> initial)`

Gibt eine Menge von Treffer zurück, wobei für jede Position in `initial` alle Treffer bestimmt wird. Die Vereinigungsmenge aller so gefundenen Treffer wird zurückgegeben.

C.2 `public class MaximumDeltaMatcher implements Matcher`

Implementiert die Schnittstelle `Matcher` so, dass ein maximaler Treffer entsprechend einer Menge von möglichen *Deltas* zurückgegeben werden. Ein solcher Treffer wird als *Maximum-Delta-Match* bezeichnet. Die Deltas geben an, wie bereits gefundene Matches um weitere Spielsteine *erweitert* werden können. Die möglichen Deltas werden im Konstruktor übergeben. Der Konstruktor hat die Signatur `public MaximumDeltaMatcher(Set<Delta> deltas)`. Wird dem Konstruktor nicht mindestens ein valides Delta übergeben wird eine `MatcherInitializationException` geworfen. Das gleiche gilt, falls eines der Deltas (0,0) oder `null` ist.

Dann besteht ein einfacher Algorithmus zum Finden eines *Maximum-Delta-Match* zu einer Initialposition p aus den folgenden Schritten:

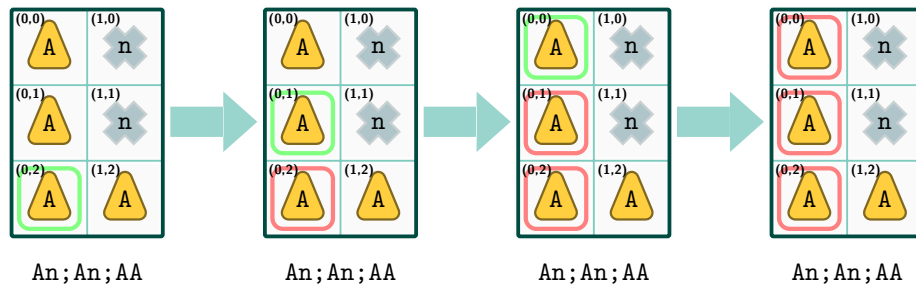
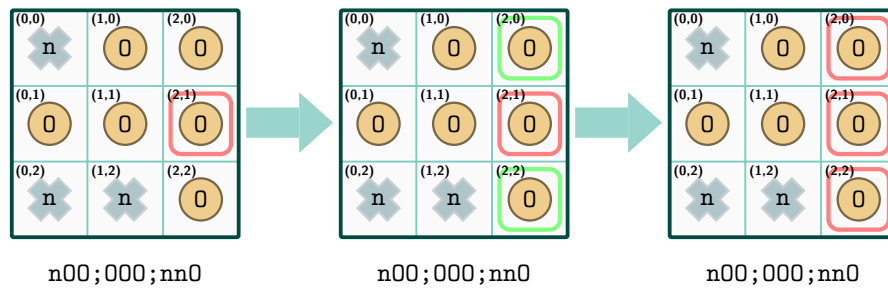
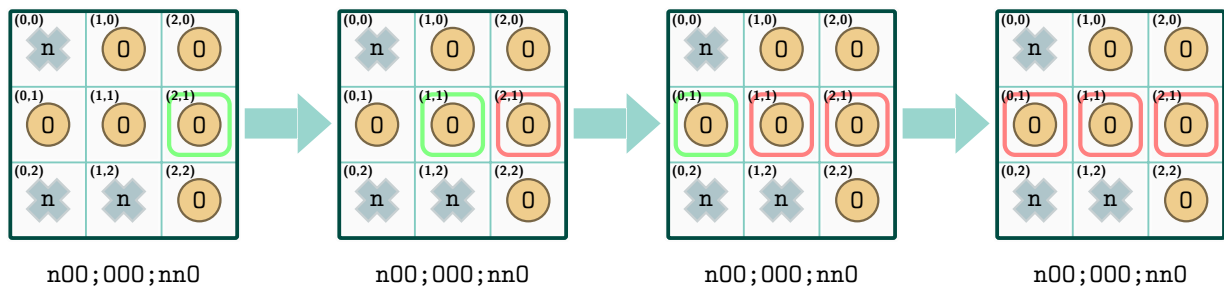
1. Setze die Menge an gematchten Positionen P auf $\{p\}$.
2. Solange es eine Position die nicht in P enthalten ist gibt, die von einem Feld in P durch eine **Vorwärts- oder Rückwärts**-Bewegung um ein gegebenes Delta erreichbar ist und mit dem gleichen Spielstein wie die Felder in P belegt ist, füge diese Position zu P hinzu.
3. P ist dann ein *Maximum-Delta-Match*.
- (4. Gebe P zurück.)

Beispiele für den Ablauf und Ergebnisse des beschriebenen Maximum-Delta-Matchings sind in den Abbildungen 14 bis 19 zu sehen. Tabelle 1 gibt jeweils die im Konstruktor übergebene Delta-Menge, die initiale Markierung und das Ergebnis an.

Ihre Implementierung `MaximumDeltaMatcher` gibt beim Aufruf von `#match(Board, Position)` immer nur einen Treffer, also ein `Set<Set<Position>>`, das genau ein `Set<Position>` enthält, zurück.

Abbildung	deltas	initial	match(b, initial)
14	$\{(0, 1)\}$	$\{(0, 2)\}$	$\{(0, 0), (0, 1), (0, 2)\}$
15	$\{(0, -1)\}$	$\{(2, 1)\}$	$\{(2, 0), (2, 1), (2, 2)\}$
16	$\{(1, 0)\}$	$\{(2, 1)\}$	$\{(0, 1), (1, 1), (2, 1)\}$
17	$\{(1, 1), (1, -1)\}$	$\{(2, 1)\}$	$\{(0, 1), (1, 0), (2, 1)\}$
18	$\{(1, 0), (0, 1)\}$	$\{(2, 1)\}$	$\{(0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (2, 2)\}$
19	$\{(0, -2)\}$	$\{(0, 0)\}$	$\{(0, 0), (0, 2), (0, 4)\}$

Tabelle 1: Eingabe und Ausgabe des Maximum-Delta-Matching


 Abbildung 14: Schrittweise Bestimmung eines Maximum-Delta-Match für $\text{deltas} = \{(0, 1)\}$ und $\text{initial} = \{(0, 2)\}$.

 Abbildung 15: Schrittweise Bestimmung eines Maximum-Delta-Match für $\text{deltas} = \{(0, -1)\}$ und $\text{initial} = \{(2, 1)\}$.

 Abbildung 16: Schrittweise Bestimmung eines Maximum-Delta-Match für $\text{deltas} = \{(1, 0)\}$ und $\text{initial} = \{(2, 1)\}$.

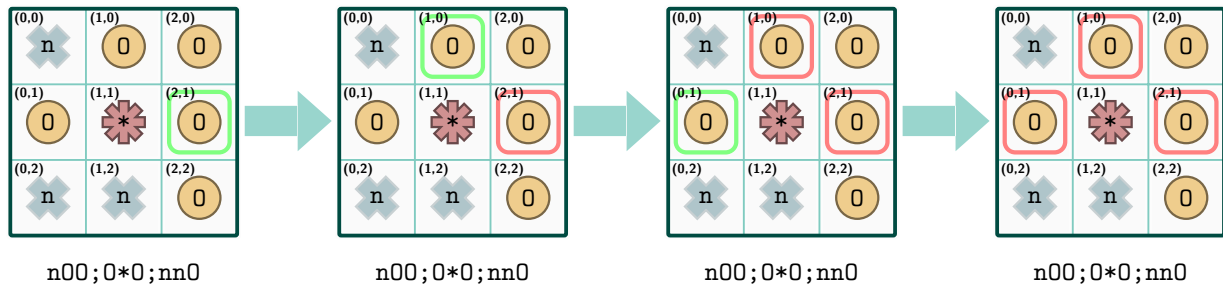


Abbildung 17: Schrittweise Bestimmung eines Maximum-Delta-Match für $\text{deltas} = \{(1,1), (-1,1)\}$ und $\text{initial} = \{(2,1)\}$.

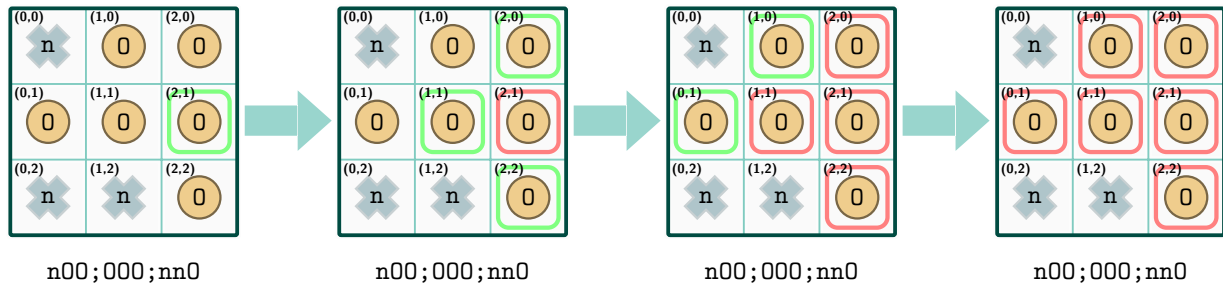


Abbildung 18: Schrittweise Bestimmung eines Maximum-Delta-Match für $\text{deltas} = \{(1,0), (0,1)\}$ und $\text{initial} = \{(2,1)\}$.

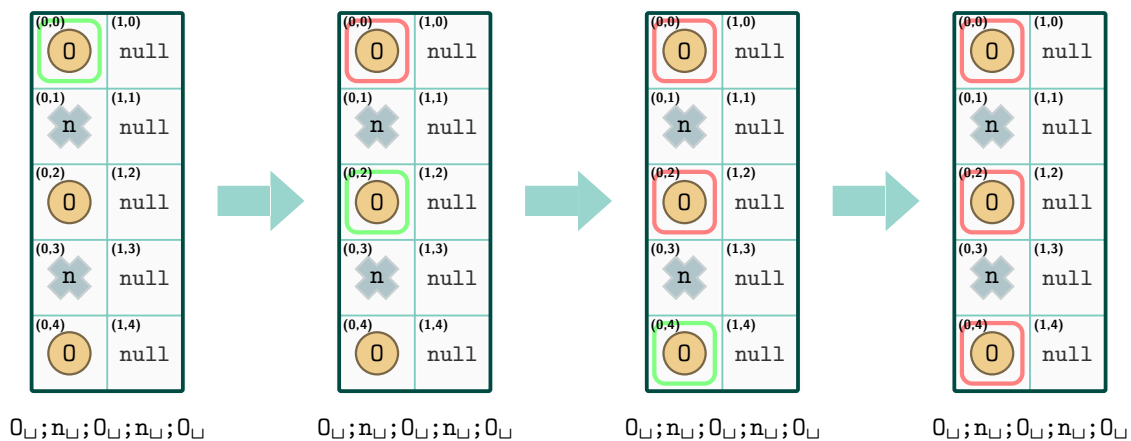


Abbildung 19: Schrittweise Bestimmung eines Maximum-Delta-Match für $\text{deltas} = \{(0,-2)\}$ und $\text{initial} = \{(0,0)\}$.

D Spielablauf und Punktevergabe

In diesem Aufgabenteil implementieren Sie die Schnittstelle **Game** in einer von Ihnen entwickelten Klasse **MatchThreeGame**, die einen Spielablauf inklusive Punktevergabe darstellt. Strukturieren Sie die hierzu nötige Logik ggf. in sinnvolle Klassen, Schnittstellen und Methoden.

D.1 `public class MatchThreeGame implements Game`

Ihre Klasse muss einen Konstruktor anbieten, der die folgende Signatur hat: `public MatchThreeGame(Board board, Matcher matcher)`. Das Spielbrett eines Spiels kann nicht durch ein anderes Spielbrett ausgetauscht werden. Der **Matcher** kann durch den in der Schnittstelle definierten *setter* `#setMatcher(Matcher)` ersetzt werden. Dabei ist es nicht erlaubt, den **Matcher** auf `null` zu setzen. Der Aufruf des Konstruktors ändert das Spielbrett nicht direkt. Zu Beginn eines Spiels ist der Punktestand 0.

Im Folgenden werden die Methoden der Schnittstelle **Game** beschrieben.

D.2 `interface Game`

D.2.1 `void initializeBoardAndStart()`

Lässt alle Steine des aktuellen Spielbretts falls nötig nach unten rücken (`Board#moveTokensToBottom()`) und füllt das Spielbrett mit zufälligen Steinen auf (`Board#fillWithTokens()`).

Nach dem initialen Füllen des Spielbretts werden Treffer auf allen Feldern des Spielbretts erkannt und genau so behandelt, als wäre die aktuelle Situation durch eine Bewegung des gesamten Spielbretts entstanden. Wie Treffer hier definiert sind, und wie sie behandelt werden, wird in der folgenden Methode beschrieben. Das bedeutet insbesondere, dass gegebenenfalls Kettenreaktionen durchgeführt werden.

D.2.2 `void acceptMove(Move move)`

Führt den gegebenen Spielzug auf dem Spielbrett aus. Wirft gegebenenfalls eine sinnvolle Exception, falls der Spielzug nicht auf dem Spielbrett ausführbar ist.

Für das Finden von Treffern wird der gesetzte **Matcher** verwendet. Übergeben Sie diesem nicht *alle* Positionen des Spielbretts, sondern nur die, die in der letzten Aktion (Zug oder Nachrücken von Steinen) beeinflusst wurden.

Um bewertet und vom Spielbrett entfernt zu werden, müssen Treffer immer eine Größe von *mindestens* drei haben (d.h. drei Spielsteine oder mehr). Ist ein Treffer m_1 in einem anderen Treffer m_2 *komplett* enthalten – das bedeutet, die Positionen in m_1 sind eine Untermenge oder gleich der Positionen in m_2 – so wird m_1 **nicht bewertet**. Sind mehrere Treffer gleich, wird nur einer der Treffer bewertet. Sind die Positionen nur *teilweise* enthalten, werden beide Treffer vollständig bewertet.

Die Steine aller diese Bedingungen erfüllenden Treffer werden entfernt und die Punktzahl wird entsprechend der Beschreibung in Unterunterabschnitt D.2.3 erhöht.

Sind alle Steine entfernt, werden die Steine auf dem Spielbrett nach unten gerückt (`moveTokensToBottom`) und das Spielbrett wird mit zufälligen Steinen aufgefüllt (`fillRandomly`). Ergeben sich dadurch neue Treffer, wird so verfahren, als hätte sich die aktuelle Spielsituation durch einen Spielzug ergeben.

D.2.3 `int getScore()`

Gibt den aktuellen Punktestand des Spiels zurück. Der Punktestand wird beim Entfernen von Spielsteinen durch Treffer erhöht. Die Basis-Punktzahl eines Treffer ergibt sich wie folgt:

$$3 + (\text{Anzahl Steine} - 3) * 2$$

Ein Treffer mit 3, 4 oder 5 Steinen ergibt also die Basis-Punktzahl 3, 5, bzw. 7.

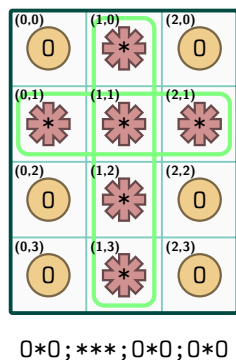


Abbildung 20: Situation mit zwei gleichzeitigen Treffern

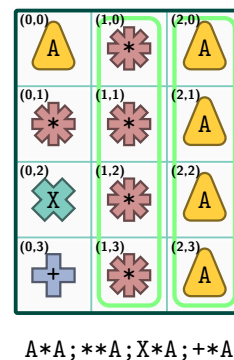


Abbildung 21: Situation mit zwei gleichzeitigen Treffern

Ergeben sich zu einem Zeitpunkt **mehrere Treffer gleichzeitig** (d.h. für ein Spielbrett), so wird die Summe der Basis-Punktzahlen aller Treffer mit der Anzahl an gleichzeitigen Treffern multipliziert. Gleichzeitig bedeutet hierbei vor dem Entfernen von Steinen und darauffolgendem Nachrücken. Ergibt sich also die Situation, wie in Abbildung 20 dargestellt, so würden 16 Punkte vergeben. Diese Zahl wird wie folgt bestimmt:

- Basis-Punktzahl $\{ (0, 1), (1, 1), (2, 1) \}$: 3.
- Basis-Punktzahl $\{ (1, 0), (1, 1), (1, 2), (1, 3) \}$: 5.
- Summe Basis-Punktzahlen: $3 + 5 = 8$.
- Multiplikator für zwei gleichzeitige Treffer: $8 * 2 = 16$.

Hierbei ist unerheblich, ob die Treffer die gleichen Symbole enthalten, oder nicht. Abbildung 21 zeigt eine Situation, die mit $(5 + 5) * 2 = 20$ Punkten bewertet werden würde.

Ergeben sich durch das oben beschriebene Nachrücken der Steine und das zufällige Auffüllen des Spielbretts neue Treffer, so werden Sie unmittelbar bewertet und wiederum entfernt, ohne dass ein weiterer Zug über die Methode `acceptMove` vorgenommen werden muss. Dies wird als **Kettenreaktion** bezeichnet.

Kettenreaktionen beeinflussen die Punktevergabe durch **Multiplikatoren**. Die Punkte der direkt durch einen Zug hergestellten Situation werden mit dem Faktor 1 multipliziert. Daraufhin rücken Steine nach und das Spielbrett wird aufgefüllt. Die Punkte für daraufhin entstehende Treffer werden mit dem Faktor 2 multipliziert. Nach einem erneuten Nachrücken und Auffüllen, wird der Faktor 3 angewandt, und so weiter. Der Faktor erhöht sich also pro Kettenreaktionsschritt um eins.

In Abbildung 22 ist ein solcher Vorgang beispielhaft dargestellt. Insgesamt würde sich die Punktzahl hierbei um $16 + 24 + 9 = 49$ Punkte erhöhen. Diese Punktzahl ergibt sich wie folgt:

1. Situation: $(3 + 5) * 2 = 16$ Punkte
2. Situation: $(3 + 3) * 2 = 12$ Punkte. Faktor Kettenreaktion: $2 \Rightarrow 12 * 2 = 24$ Punkte
3. Situation: 3 Punkte. Faktor Kettenreaktion: $3 \Rightarrow 3 * 3 = 9$ Punkte
4. Keine weiteren Treffer: Summe: $16 + 24 + 9 = 49$ Punkte

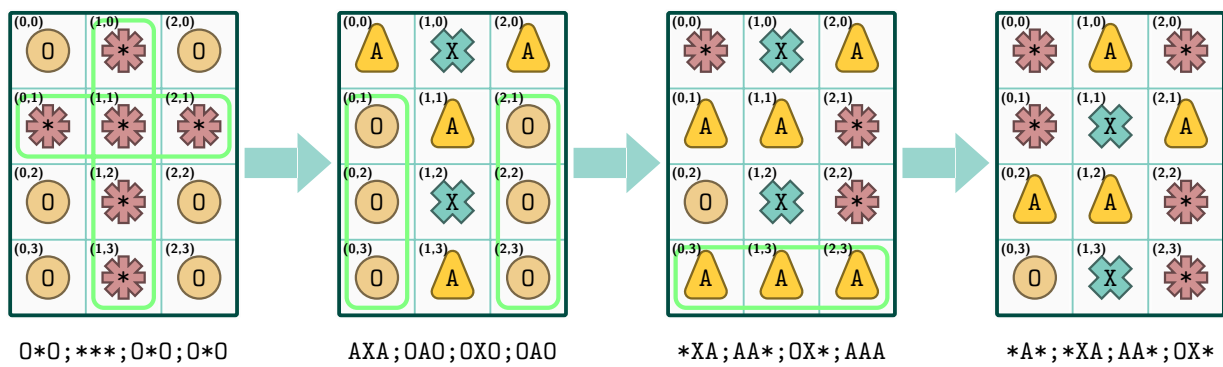


Abbildung 22: Kettenreaktion