Karlsruher Institut für Technologie

Programmieren – Wintersemester 2016/2017

Software-Design und Qualität (SDQ)
`https://sdqweb.ipd.kit.edu/wiki/Programmieren`
Jun.-Prof. Dr.-Ing. Anne Koziolek

Übungsblatt 3

Ausgabe: 29.11.2016 – 17:00
Abgabe: ~~14.12.2016 – 13:00~~
15.12.2016 – 07:00

- Achten Sie darauf nicht zu lange Zeilen, Methoden und Dateien zu erstellen.
- 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` und `java.io`, es sei denn die Aufgabenstellung erlaubt ausdrücklich weitere Pakete. Verwenden Sie also insbesondere keine Klassen aus `java.util` und Unterpaketen `java.util.[...]`.[1]
- Achten Sie auf fehlerfrei kompilierenden Programmcode.[1]
- Halten Sie alle Whitespace-Regeln ein.
- Halten Sie die Regeln zu Variablen-, Methoden und Paketbenennung ein und wählen Sie aussagekräftige Namen. Legen Sie die von Ihnen definierten Klassen in Unterpaketen `edu.[...]`, `com.[...]`, `org.[...]`, `net.[...]` oder `de.[...]` ab, beispielsweise in `edu.kit.informatik`.

Abgabemodalitäten

Die Praktomat-Abgabe wird am **Mittwoch, den 07. Dezember 2016, um 13:00 Uhr**, freigeschaltet.

- Geben Sie Ihre Antworten zu Aufgabe A als `.java`-Dateien ab.

Bitte beachten Sie, dass das erfolgreiche Bestehen der öffentlichen Tests für eine erfolgreiche Abgabe nötig ist. Planen Sie entsprechend Zeit für Ihren ersten Abgabeversuch ein.

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

Allgemeine Hinweise

- Machen Sie sich bei der Bearbeitung der Aufgaben klar, an welcher Stelle Sie Referenzen auf veränderbare (*mutable*) Datenstrukturen als Eingabe bekommen oder als Rückgabe ausgeben. Gegebenenfalls müssen Objekte kopiert werden, um eine ungewollte Änderung des Zustands Ihrer Instanzen von außen zu vermeiden.
- Übernehmen Sie die Methodensignaturen und Klassennamen genau wie in der Aufgabenstellung angegeben. Fügen Sie den Klassen keine weiteren **öffentlichen** (`public`) Felder, Methoden oder Konstruktoren hinzu. Es empfiehlt sich jedoch, private Hilfsmethoden und Konstanten hinzuzufügen um gegebenenfalls Code-Duplikate zu vermeiden und Ihren Code besser zu strukturieren. Setzen Sie ggf. **sparsam** `package private (default)`-Sichtbarkeit bei zusätzlichen Methoden in den spezifizierten Klassen ein, falls Sie zusätzliche Implementierungsdetails zwischen den von Ihnen implementierten Klassen teilen müssen. Sie können weitere eigene Klassen definieren.
- Sie können beim Lösen *dieses Übungsblattes* davon ausgehen, das die Argumente für die beschriebenen Methoden immer valide sind. Das bedeutet beispielsweise, dass nie `null` übergeben wird, es sei denn, dies ist explizit in der Methodenbeschreibung erwähnt. Weiterhin haben übergebene Felder bspw. immer korrekte Länge und Aufbau. Wenn Sie die Methoden aus Ihrem eigenen Code aufrufen müssen Sie natürlich trotzdem dafür sorgen, dass Sie nur valide Argumente übergeben.

A Kachelung (20 Punkte)

In dieser Aufgabe entwickeln Sie ein Programm, das prüft, ob eine gegebene Kachelung eines Spielfeldes zu einem Regelsatz konform ist.

Grundlagen

Bei einer Kachelung, auch Parkettierung[2] genannt, werden gleichartige geometrische Flächen dazu verwendet, um eine Fläche lückenlos abzudecken. Mit gleichseitigen Sechsecken (Hexagonen) ist dies für zweidimensionale Ebenen im euklidischen Raum möglich. Dies ist beispielhaft in Abbildung 1 dargestellt.

Basierend auf dieser mathematischen Idee existieren zahlreiche Spiele, in denen sechseckige Spielsteine nach festgelegten Regeln aneinandergelegt werden. Ein Beispiel für ein derartiges Spiel ist *Tantrix*[3]. Spielsteine, die zusätzlich mit Linien versehen sind, wie beispielhaft in Abbildung 2 dargestellt, werden von den verschiedenen Spielern abwechselnd unter Einhaltung von vorgegebenen Regeln an einen initialen Spielstein angelegt.

Bitte beachten Sie folgenden Hinweis: Die Aufgabe, die Sie auf diesem Blatt bearbeiten sollen, entspricht **nicht** genau dem im vorherigen Abschnitt beschriebenen Beispiel. Dies soll lediglich den Kontext für die Aufgabe illustrieren. Sie müssen nur die im Folgenden explizit geforderte Funktionalität implementieren, und beispielsweise **keine** zusätzlichen Regeln oder eine Punktevergabe für den Spielausgang entwickeln.

A.1 LineType

In diesem Übungsblatt gibt es nur die Farben Rot (`RED`), Grün (`GREEN`) und Gelb (`YELLOW`) (anders als im vorher genannten Beispiel). Verwenden Sie im Folgenden den in Codeauflistung 1 definierten Aufzählungs-Datentypen `LineType` für die möglichen Linienarten. Dieser enthält zusätzlich einen Wert `NONE`, der eine nicht vorhandene Linie repräsentiert. Kopieren Sie den Code in eine Datei `LineType.java`, fügen Sie eine Paketdeklaration zur Datei hinzu und geben Sie die Datei mit ab. Der Code ist auf der Lehrstuhl-Webseite verfügbar.[4]

Der untenstehende Code enthält zusätzliche Logik, die für jeden Wert eine Abkürzung (Englisch: *abbreviation*) festlegt. Diese kann über die Methode `getAbbreviation()`

[2] `https://de.wikipedia.org/wiki/Parkettierung`
[3] `https://en.wikipedia.org/wiki/Tantrix`, `http://www.tantrix.com/`
[4] `https://sdqweb.ipd.kit.edu/lehre/WS1617-Programmieren/LineType.java`

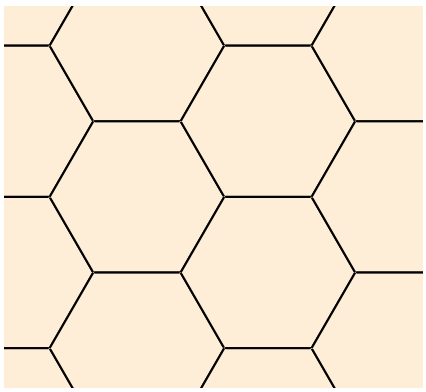


Abbildung 1: Lückenlose Kachelung mit Sechsecken

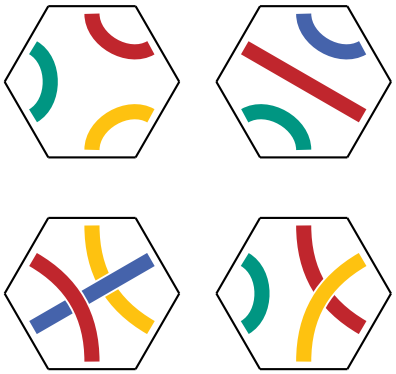


Abbildung 2: Beispielhafte Spielsteine

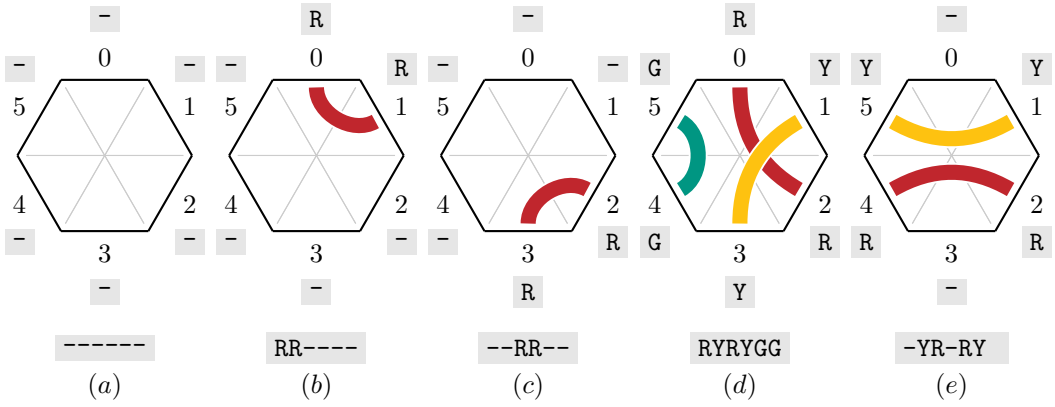


Abbildung 4: Kacheln mit Codierungen

als `char` abgerufen werden, bspw. durch `LineType.RED.getAbbreviation()` oder `LineType lineType = LineType.GREEN; char lineTypeAbbreviation = lineType.getAbbreviation()`.

Weiterhin gibt für einen Wert die Methode `isColor()` zurück, ob es sich bei dem Wert um eine Farbe handelt, d.h. ob der Wert `RED`, `GREEN` oder `YELLOW` ist.

Sie müssen die Definition für `LineType` nicht im Detail verstehen, wichtig ist allerdings, dass Sie die Methoden `getAbbreviation()` und `isColor()` kennen und, falls notwendig, nutzen können.

A.2 Kachel

In dieser Aufgabe entwickeln Sie eine Klasse `Tile`, die eine Kachel repräsentiert. Eine Kachel kann zwischen 0 und 3 Verbindungslinien besitzen (anders als im unter Grundlagen genannten Beispiel). Im Fall von 0 Linien nennen wir die Kachel *leer*. In Abbildung 4 (ignorieren Sie zunächst die grau hinterlegten Bezeichner und Zahlen) sind Beispiele für valide Kacheln dargestellt. Eine Kachel hat immer eine feste Orientierung, so sind beispielsweise Kacheln (*b*) und (*c*) in Abbildung 4 nicht identisch.

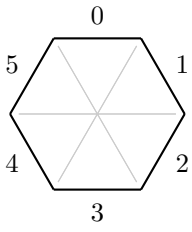


Abbildung 3: Aufbau einer Kachel

Weiterhin kann es mehrere Instanzen Ihrer Klasse geben, die zwar die gleichen Verbindungslinien haben und gleich orientiert sind, aber nicht die selbe Instanz darstellen. Dies hängt damit zusammen, dass die Klasse *mutable* ist. Beispielsweise ändert die Methode `rotateClockwise()` die Instanz, auf der die Methode aufgerufen wird.

Implementieren Sie eine Klasse `Tile`, mit die im Folgenden beschriebenen Methoden.

```java
/**
 * Represents all possible types of lines that can be present on
 * tiles.
 *
 * <p>The type contains values for three colors ({@link #RED},
 * {@link #GREEN} and {@link #YELLOW}) and one value
 * that represents the absence of a line ({@link #NONE}).
 *
 * <p>Each value has an <em>abbreviation</em> that can be
 * obtained using the {@link LineType#getAbbreviation()} method.
 */
public enum LineType {
    /** Non-existent line. */
    NONE('-'),
    /** Line with color red. */
    RED('R'),
    /** Line with color green. */
    GREEN('G'),
    /** Line with color yellow. */
    YELLOW('Y');

    private final char abbreviation;

    LineType(char abbreviation) {
        this.abbreviation = abbreviation;
    }

    /**
     * Returns the abbreviation of the line type.
     *
     * <p>Each color is represented by the first character of its {@linkplain #name() name}
     * ({@linkplain #RED R}, {@linkplain #GREEN G} or {@linkplain #YELLOW Y}), the line type
     * {@linkplain LineType#NONE NONE} is represented by an ASCII minus sign '-' ('\u002D').
     *
     * @return the abbreviation
     */
    public char getAbbreviation() {
        return abbreviation;
    }

    /**
     * Returns whether this is a color.
     *
     * @return {@code true} if this is a color, {@code false} if this is {@link #NONE}
     */
    public boolean isColor() {
        return this != NONE;
    }
}
```

Codeauflistung 1: LineType.java

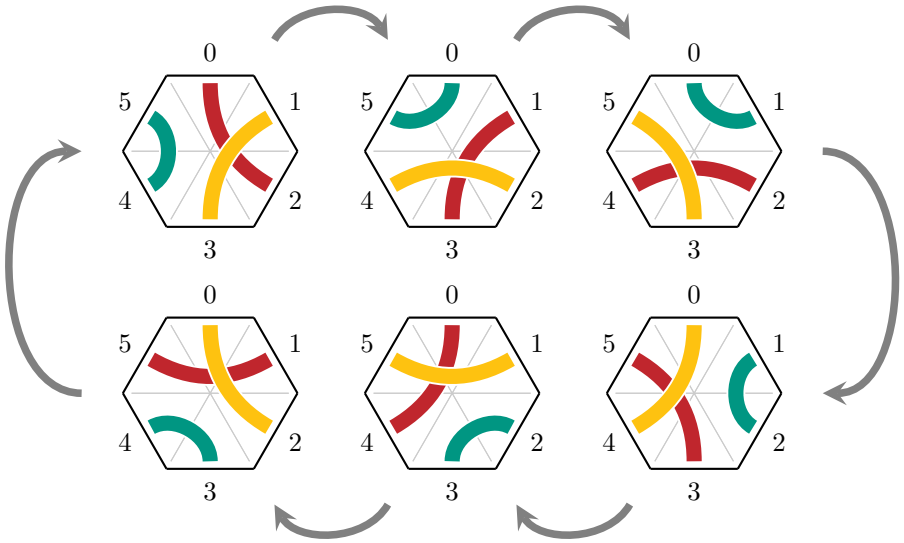


Abbildung 5: Drehen eines Felds

A.2.1 Konstruktor: `public Tile(LineType[] lineTypes)`

Erstellt eine neue Kachel und initialisiert Sie mit Linien, deren Position und Typ im Parameter `lineTypes` kodiert ist. Die übergebenen Typen sind hierbei im Uhrzeigersinn von der oberen Seite ausgehend sortiert. Jeder Eintrag bezeichnet die Farbe der anliegenden Verbindungslinie (`LineType.RED`, `LineType.GREEN`, `LineType.YELLOW`), oder das Nichtvorhandensein einer Linie (`LineType.NONE`). Die Reihenfolge ist in Abbildung 3 dargestellt. Beispielhafte Belegungen sind in Abbildung 4 dargestellt. Hierbei steht in der Abbildung jeweils die Abkürzung eines `LineType`-Werts `c` (d.h. der Wert, den `c.getAbbreviation()` zurückgibt) für den entsprechenden Linientyp.

Sie können davon ausgehen, dass `lineTypes` immer valide ist, also beispielsweise nur valide Kacheln definiert werden (bspw. nie `R-R-R-`), d.h. jeder Linentyp außer `LineType.NONE` an genau 0 oder 2 Stellen im Array steht.

Dadurch ergibt sich eine eindeutige Darstellung der Verbindungslinien auf einer Kachel.

A.2.2 Konstruktor: `public Tile()`

Gibt eine neue leere Kachel zurück.

A.2.3 `public LineType getLineTypeAtIndex(int index)`

Gibt einen Eintrag der Kodierung der Verbindungslinien auf der Kachel zurück. Die Reihenfolge der Farben kodiert die Verbindungslinien genau gleich, wie in der Beschreibung des Konstruktors `Tile(LineType[])` erläutert. Sie können davon ausgehen, dass der übergebene `index` stets eine Zahl zwischen einschließlich 0 und 5 ist.

A.2.4 `public int getNumberOfColors()`

Gibt die Anzahl der verschiedenen **Farben** auf der Kachel zurück. Dies ist immer 0, 1, 2 oder 3. Für die Kacheln in Abbildung 4 wäre die Rückgabe dieser Funktion beispielsweise: 0, 1, 1, 3 und 2 für (a), (b), (c), (d) und (e).

A.2.5 `public boolean isExactlyEqualTo(Tile otherTile)`

Gibt genau dann `true` zurück, wenn die Kachel und die übergebene Kachel `otherTile` *gleich* sind, d.h. die gleichen Verbindungslinien in den gleichen Farben haben und gleich orientiert ist. Beispielsweise ist eine Kachel immer zu sich selbst *gleich*.

A.2.6 `public Tile copy()`

Gibt eine **neue** Kachel zurück, die die gleichen Linien in den gleichen Farben in der gleichen Orientierung enthält. Für eine Kachel `t` ist `t.isExactlyEqualTo(t.copy())` immer wahr.

A.2.7 `public void rotateClockwise()`

Dreht die Kachel um 60° im Uhrzeigersinn. Dieser Vorgang ist in Abbildung 5 beispielhaft dargestellt. Jeder der Pfeile steht für einen Aufruf von `rotateClockwise()`.

A.2.8 `public void rotateCounterClockwise()`

Dreht die Kachel gegen den Uhrzeigersinn. Ein Aufruf von `rotateClockwise()` und ein direkt darauffolgender Aufruf von `rotateCounterClockwise()` stellt immer den vorherigen Zustand wieder her.

A.2.9 `public boolean isEmpty()`

Gibt genau dann `true` zurück, wenn die Kachel keine Verbindungslinien enthält. Beispielsweise ist `(new Tile()).isEmpty()` immer wahr.

A.2.10 `public boolean isRotationEqualTo(Tile otherTile)`

Gibt genau dann `true` zurück, wenn die Kachel und `otherTile` *rotationsgleich* sind. Hierbei nennen wir zwei Kacheln *rotationsgleich* genau dann, wenn man eine der Kacheln durch drehen in die andere überführen kann. Zwei Kacheln, die *gleich* sind, sind immer *rotationsgleich*, umgekehrt gilt dies jedoch nicht immer. Beispielsweise sind (*b*) und (*c*) in Abbildung 4 *rotationsgleich*, allerdings nicht *gleich*.

A.2.11 `public boolean canBeRecoloredTo(Tile otherTile)`

Gibt genau dann `true` zurück, wenn die Kachel durch Umfärben in `otherTile` überführt werden kann, *oder wenn beide Kacheln bereits gleich sind*. Umfärben bedeutet das Ändern der Farbe bereits vorhandener Verbindungslinien, ohne dass neue Linien hinzugefügt werden. Die leere Kachel ist bspw. nur zu sich selbst umfärbbar. Wenn für `Tile t1` und `Tile t2` gilt `t1.isExactlyEqualTo(t2)`, dann folgt daraus immer `t1.canBeRecoloredTo(t2)`.

A.2.12 `public boolean dominates(Tile otherTile)`

Gibt genau dann `true` zurück, wenn die Kachel die Kachel `otherTile` *dominiert*. Eine Kachel K_1 wird dann von einer anderen Kachel K_2 *dominiert*, wenn K_1 durch das Hinzufügen von *mindestens einer zusätzlichen* Linie in K_2 überführt werden *könnte*. Eine Kachel dominiert sich nie selbst. Jede Kachel außer der leeren Kachel dominiert die leere Kachel.

A.2.13 `public boolean hasSameColorsAs(Tile otherTile)`

Gibt genau dann `true` zurück, wenn beide Kacheln die gleichen **Farben** enthalten. Diese müssen nicht unbedingt in den gleichen Verbindungslinien ausgedrückt sein. Ob auf der Kachel der Typ `LineType.NONE` vorkommt, ist für die Rückgabe der Methode unerheblich.

A.2.14 `public String toString()`

Gibt eine textuelle Repräsentation der Kachel zurück. Hierbei wird ein `String` zurückgegeben, der immer genau 6 Zeichen lang ist. Jedes der Zeichen kodiert die Farbe einer an eine Seite der Kachel anliegenden Verbindungslinie, oder das Nichtvorhandensein einer anliegenden Linie. Die Kodierung kann über das Feld `abbreviation` des `LineType`-Typs erhalten werden. Die Reihenfolge ist gleich wie die in der Übergabe als Feld an den Konstruktor `Tile(LineType[])`.

Beispiele für die gewünsche Rückgabe sind in Abbildung 4 dargestellt, also beispielsweise `"RR----"` für Kachel (*b*).

```
Tile t1 = new Tile();
t1.isEmpty(); // true
t1.getNumberOfColors(); // 0
t1.toString(); // "------"
Tile t2 = new Tile(new LineType[] {
  LineType.NONE, LineType.YELLOW, LineType.NONE, LineType.NONE, LineType.NONE, LineType.YELLOW });
t2.toString(); // "-Y---Y"
t1.canBeRecoloredTo(t2); // false
Tile t3 = new Tile(new LineType[] { // Hinweis: (e) aus Abbildung 4
  LineType.NONE, LineType.YELLOW, LineType.RED, LineType.NONE, LineType.RED, LineType.YELLOW });
t3.dominates(t1); // true
t3.dominates(t2); // true
t2.canBeRecoloredTo(t3); // false
Tile t4 = t3.copy();
t4.isExactlyEqualTo(t3); // true
t4.rotateClockwise();
t4.rotateClockwise();
t4.rotateClockwise();
t4.toString(); // "-RY-YR"
t4.isExactlyEqualTo(t3); // false
t4.isRotationEqualTo(t3); // true
t4.canBeRecoloredTo(t3); // true
```

Codeauflistung 2: Beispielinteraktionen mit der Klasse `Tile`

Beispielinteraktionen

In Codeauflistung 2 ist ein Code-Ausschnitt und der Wert für bestimmte Methodenaufrufe als Kommentar dargestellt.

A.3 Kacheln zusammenfügen

Erweitern Sie zunächst die Klasse `Tile` um die folgende Methode:

A.3.1 `public boolean fitsTo(Tile otherTile, int position)`

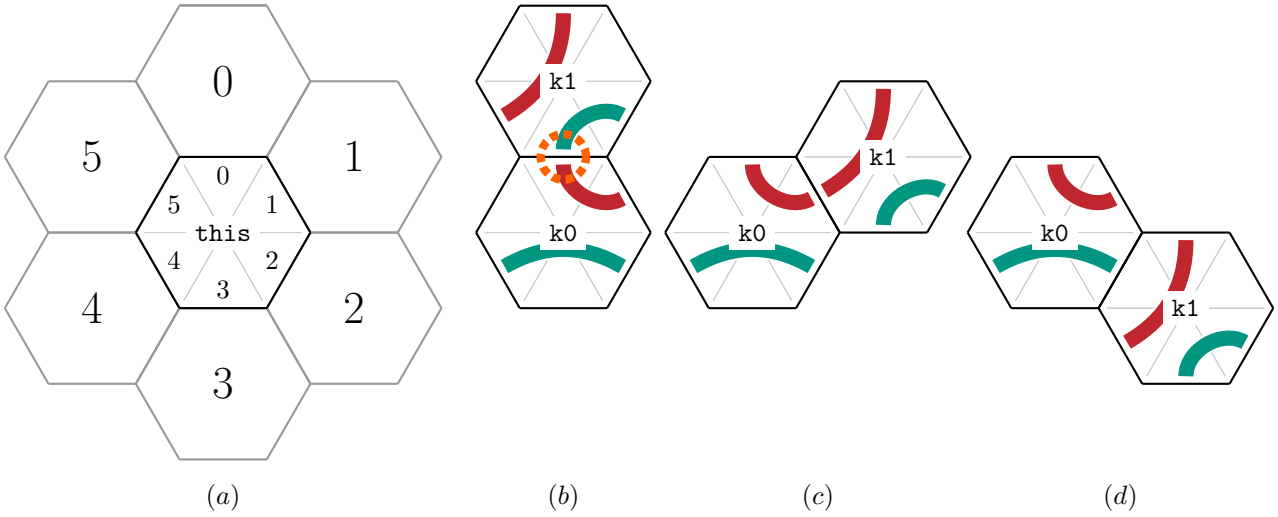
Gibt genau dann `true` zurück, wenn die Kachel und `otherTile` *zusammenpassen*. `position` gibt hierbei an, an welcher Seite `otherTile` an die Kachel angelegt würde. Die möglichen Anlegepositionen sind genau wie die Farbangaben für Verbindungslinien kodiert, d.h. ausgehend von der oberen Seite im Uhrzeigersinn von 0 bis 5. Dies ist in Abbildung 6 (*a*) dargestellt. `this` bezeichnet hierbei die Kachel auf der die Methode aufgerufen wird.

Eine Kachel *passt* genau dann **nicht** zu einer anderen Kachel, wenn die Seiten der Kacheln, die zusammenliegen, in den Kacheln von Linien unterschiedlicher Farbe berührt werden. In allen anderen Fällen passen die Kacheln zusammen. In Abbildung 6 sind weiterhin drei Beispiele (*b*), (*c*) und (*d*) für die Verwendung der Funktion dargestellt. Allen drei Beispielen liegt zunächst folgender Code zugrunde:

```
Tile k0 = new Tile(new LineType[] {
    LineType.RED, LineType.RED, LineType.GREEN,
    LineType.NONE, LineType.GREEN, LineType.NONE });
Tile k1 = new Tile(new LineType[] {
    LineType.RED, LineType.NONE, LineType.GREEN,
    LineType.GREEN, LineType.RED, LineType.NONE });
```

In Abbildung 6 ist dargestellt:

- (*b*): warum `k0.fitsTo(k1, 0)` den Wert `false` zurück gibt.



(a) (b) (c) (d)

Abbildung 6: Mögliche Anlegepositionen für Kacheln (a) und Beispiele (b)-(d)

- (c): warum `k0.fitsTo(k1, 1)` den Wert `true` zurück gibt.
- (d): warum `k0.fitsTo(k1, 2)` den Wert `true` zurück gibt.

A.4 Das Spielfeld `Board`

In dieser Aufgabe entwickeln Sie eine Klasse `Board`, die ein Spielbrett darstellt und auf Korrektheit prüft. Dies wird im Folgenden genauer erklärt.

Die Klasse soll folgende Methoden implementieren:

A.4.1 Konstruktor: `public Board()`

Erstellt ein neues leeres Spielbrett. Ein Spielbrett besteht immer aus genau 12 Kacheln, die wie in Abbildung 7 dargestellt angeordnet sind. Nach der Initialisierung des Spielbretts sind zunächst alle 12 Kacheln leere `Tile`-Instanzen.

A.4.2 `public Tile getTile(int position)`

Gibt die Kachel an der angegebenen Position auf dem Spielbrett zurück.

A.4.3 `public void setTile(int position, Tile newTile)`

Setzt die Kachel an der angegebenen Position auf dem Spielbrett.

A.4.4 `public void removeTile(int position)`

Ersetzt die Kachel an der angegebenen Position auf dem Spielbrett durch eine leere Kachel.

A.4.5 `public boolean isEmpty()`

Gibt genau dann `true` zurück, wenn alle Kacheln auf dem Spielbrett leer sind.

A.4.6 `public void rotateTileClockwise(int position)`

Dreht die Kachel an der angegebenen Position im Uhrzeigersinn.

A.4.7 `public void rotateTileCounterClockwise(int position)`

Dreht die Kachel an der angegebenen Position gegen den Uhrzeigersinn.

A.4.8 `public int getNumberOfColors()`

Gibt die Anzahl an verschiedenen **Farben** auf dem Spielbrett zurück. Die Rückgabe ist immer genau 0, 1, 2 oder 3.

A.4.9 `public boolean isValid()`

Prüft, ob das Spielbrett *regelkonform* ist. Ein Spielbrett ist genau dann regelkonform, wenn alle Kacheln wie in der Methode `fitsTo(Tile, int)` der Klasse `Tile` beschrieben zusammenpassen.

In Abbildung 9 ist ein Beispiel für ein regelkonformes Spielbrett dargestellt. In Abbildung 10 ist ein Beispiel für ein nicht regelkonformes Spielbrett dargestellt.

A.4.10 `public LineType getConnectedPathColor(int[] positions)`

Prüft, ob es auf dem Spielfeld einen Pfad in einer Farbe gibt, der alle angegebenen Spielfelder in der Reihenfolge durchläuft, die in `positions` steht, und gibt diese Farbe zurück. Dabei ist es nicht ausschlaggebend, ob der Pfad noch weitere Felder vor oder nach den Positionen in `positions` durchläuft. Das bedeutet, in Abbildung 10 gibt es einen Pfad der Farbe Grün von Kachel 4 nach Kachel 5, obwohl dieser in den Kacheln 6 und 7 weitergeführt wird.Falls es keinen solchen Pfad gibt, gibt die Methode `LineType.NONE` zurück. Der Pfad muss hierbei in `positions[0]` beginnen und in `positions[positions.length - 1]` enden.

Sie können für diese Aufgabe davon ausgehen, dass `positions` immer valide ist. Das bedeutet, dass `positions.length >= 2` gilt und beispielsweise

- für jedes i mit $0 \leq i <$ `positions.length` $- 1$ gilt: `positions[i] != positions[i + 1]`,
- für jedes j mit $0 \leq j <$ `positions.length` $- 2$ gilt: `positions[j] != positions[j + 2]`.
- zwei aufeinanderfolgende Einträge in `positions` immer benachbarte Felder sind.

Sie müssen nur die Kacheln in der angegebenen Reihenfolge prüfen und keine weiteren Pfade auf dem Spielbrett suchen, also bspw. ist für eine Belegung von `positions` mit `{1, 4}` nicht relevant, ob es einen Pfad von Kachel 1 über 2 nach 4 gibt, sondern ausschließlich die direkte Verbindung von Kachel 1 nach 4.

In der folgenden Tabelle finden Sie beispielhafte Eingaben in der ersten Spalte und entsprechende erwartete Ausgaben in der zweiten und dritten Spalte. Zur Steigerung der Übersichtlichkeit ist bei allen Einträgen der zweiten und dritten Spalte das führende `LineType.` ausgespart. Die beiden zugrundeliegenden Spielfelder sind das in Abbildung 9 dargestellte und in Abbildung 11 textuell ausgedrückte, welches als `boardValid` bezeichnet ist, und das in Abbildung 10 und entsprechend in Abbildung 12 ausgedrückte, welches als `boardInvalid` bezeichnet wird. Die zweite Spalte stellt die erwartete Ausgabe für `boardValid`, die dritte die erwartete Ausgabe für `boardInvalid` dar.

positions	boardValid ~~.hasConnectedPath(positions)~~ .getConnectedPathColor(positions)	boardInvalid
`{ 0, 3 }`	NONE	NONE
`{ 1, 4 }`	YELLOW	NONE
`{ 1, 2, 4 }`	NONE	NONE
`{ 4, 5, 7 }`	GREEN	GREEN
`{ 4, 5, 7, 6, 4, 5, 7 }`	GREEN	NONE
`{ 8, 10, 11 }`	YELLOW	YELLOW
`{ 8, 10, 11, 8 }`	YELLOW	NONE

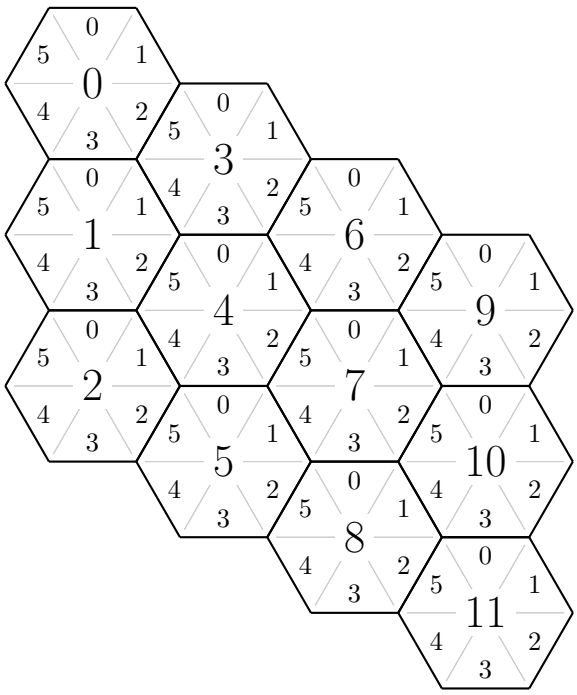


Abbildung 7: Aufbau des Spielfelds

```
------;------;------;
------;------;------;
------;------;------;
------;------;------;
```

Abbildung 8: Form der Rückgabe der Methode `toString()` der Klasse `Board`.

A.4.11 `public String toString()`

Gibt eine textuelle Repräsentation des Spielbretts zurück. Hierbei wird jede **Spalte** des Spielbretts durch eine **Zeile** im Rückgabewert repräsentiert. Die Kacheln 0, 1 und 2 bilden die erste **Spalte** des Spielbretts, die Kacheln 3, 4 und 5 bilden die zweite Spalte, und so weiter. Jede Zeile der Rückgabe besteht aus den textuellen Repräsentationen der Kacheln, wie sie von der Methode `toString()` der Klasse `Tile` zurückgegeben wird, jedoch jeweils mit einem abschließenden Semikolon (`";"`). Das bedeutet, jede Zeile hat die Form `------;------;------;`. Jeder Eintrag entspricht der Kachel auf dem jeweiligen Feld. Die Rückgabe besteht aus 4 derartigen Zeilen, hat also insgesamt die Form, die in Abbildung 8 dargestellt ist. Beispiele für Spielbretter und zugehörige Ergebnisse der `toString()`-Methode finden Sie in Abbildung 9 bzw. Abbildung 11 und Abbildung 10 bzw. Abbildung 12.

Beispielinteraktionen

In Codeauflistung 3 ist ein Code-Ausschnitt und der Wert für bestimmte Methodenaufrufe als Kommentar dargestellt.

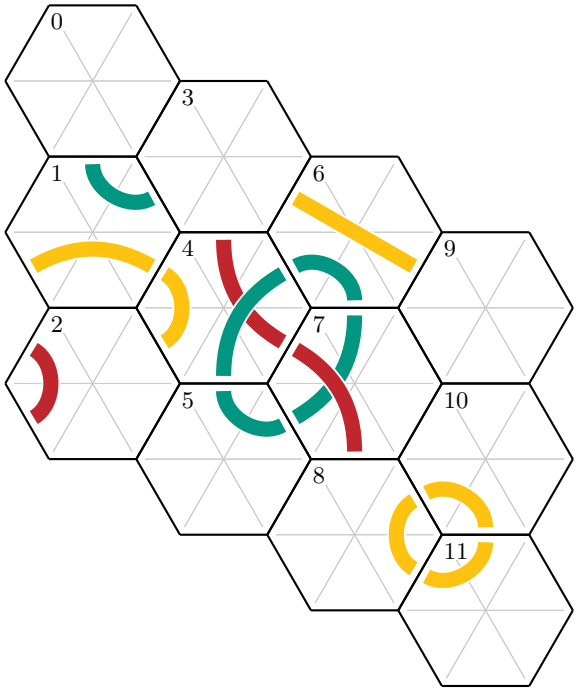


Abbildung 9: Regelkonformes Spielbrett

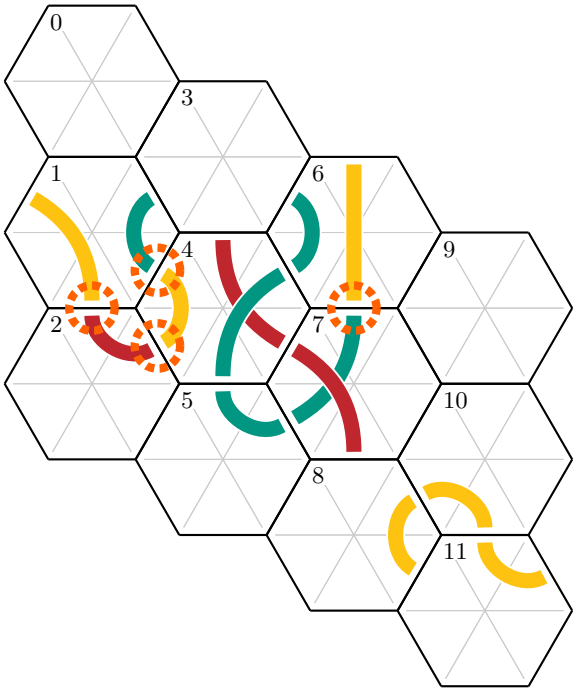


Abbildung 10: Nicht regelkonformes Spielbrett

```
------;GGY-Y-;----RR;
------;RGRGYY;GG----;
--YGGY;G--RGR;-YY---;
------;---YY-;Y----Y;
```

Abbildung 11: Ergebnis des Aufrufs von `toString()` auf dem Spielbrett aus Abbildung 9.

```
------;-GGY-Y;RR----;
------;RGRGYY;GG----;
Y--YGG;G--RGR;-YY---;
------;---YY-;YY----;
```

Abbildung 12: Ergebnis des Aufrufs von `toString()` auf dem Spielbrett aus Abbildung 10.

```
1  Board b = new Board();
2  b.isEmpty(); // true
3  b.setTile(1, new Tile(new LineType[] {
4    LineType.GREEN, LineType.GREEN, LineType.YELLOW, LineType.NONE, LineType.YELLOW, LineType.NONE }));
5  b.setTile(4, new Tile(new LineType[] {
6    LineType.RED, LineType.GREEN, LineType.RED, LineType.GREEN, LineType.YELLOW, LineType.YELLOW }));
7  b.isEmpty(); // false
8  b.isValid(); // true
9  b.toString(); // ------;GGY-Y-;------;
10               // ------;RGRGYY;------;
11               // ------;------;------;
12               // ------;------;------;
13 b.getConnectedPathColor(new int[] { 1, 4 }); // LineType.YELLOW
14 b.rotateTileClockwise(1);
15 b.isValid(); // false
16 b.toString(); // ------;-GGY-Y;------;
17               // ------;RGRGYY;------;
18               // ------;------;------;
19               // ------;------;------;
20 b.getConnectedPathColor(new int[] { 1, 4 }); // LineType.NONE
```

Codeauflistung 3: Beispielinteraktionen mit der Klasse `Board`