

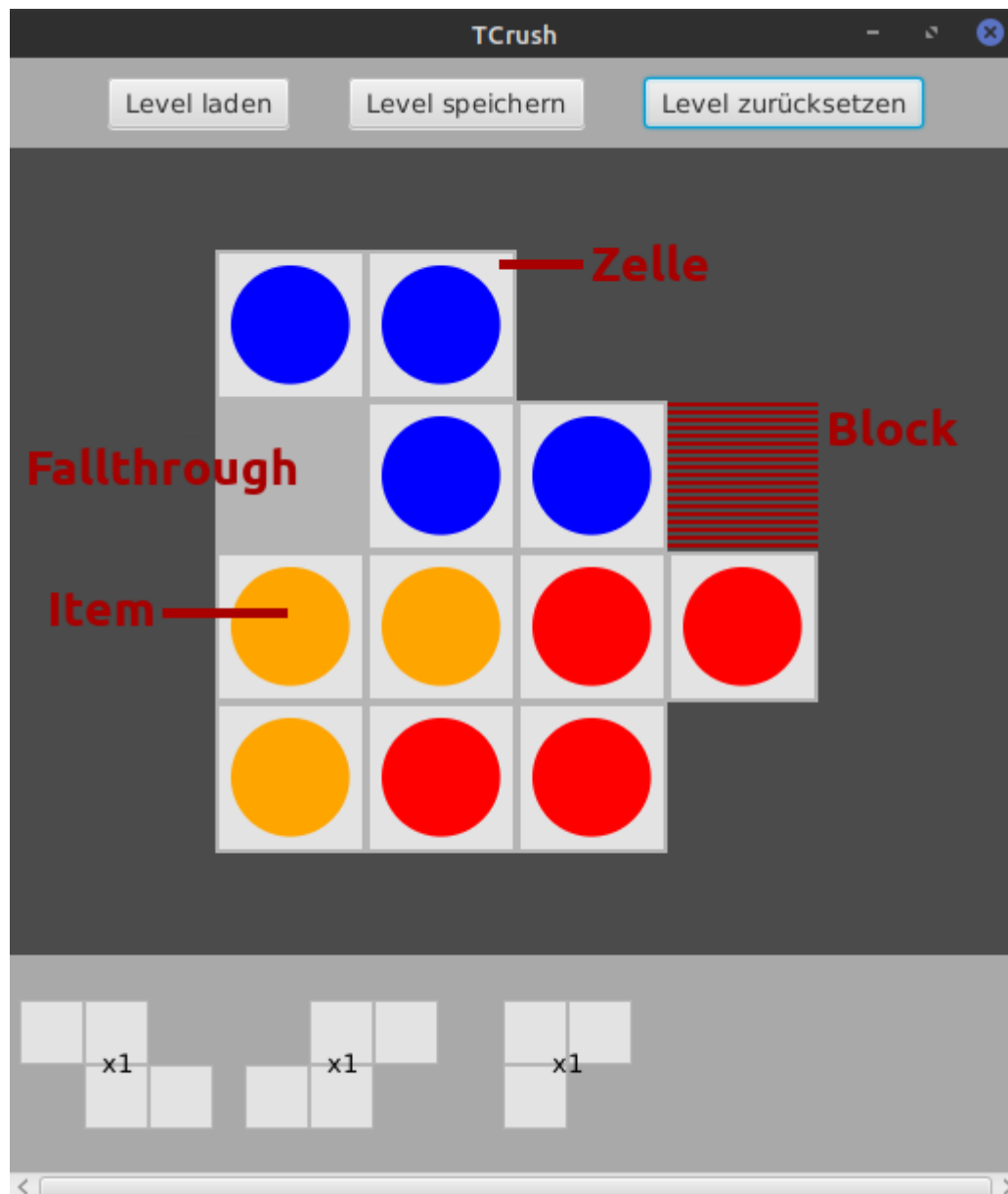
TCrush

JavaFX mit Gradle

Achtung: Da JavaFX seit Version 8 nicht mehr im JDK von Oracle enthalten ist, müssen die entsprechenden Bibliotheken extra eingebunden werden. Aus diesem Grund ist in den heruntergeladenen Dateien ein Gradle-Script enthalten, in dem die entsprechenden Abhängigkeiten festgelegt sind. Übliche Entwicklungsumgebungen sind in der Lage dieses Script zu erkennen und die nötigen Inhalte automatisch zu importieren. Zum Beispiel fragt IntelliJ nach dem Auschecken des PABS-SVN-Repositories, ob das entsprechende Gradle-Projekt importiert werden soll oder importiert es automatisch. Achten Sie darauf dies auszuführen und sollten Sie anfangs die Meldung verpasst haben, können über das Gradle-Menü (üblicherweise am rechten Bildschirmrand) Projekte nachträglich importiert werden.

TCrush ist ein levelbasiertes 2D-Spiel, das an schon bekannte Spiele dieser Art angelehnt ist. Die Spieloberfläche besteht dabei aus einem Spielfeld, sowie einer Auswahl an Formen, die in das Spielfeld eingesetzt werden können. Eine Form kann dabei einen Bereich gleichfarbiger Felditems entfernen. Ziel des Spieles ist dabei alle Items aufzulösen, wobei Items, die über Entfernten liegen, "herunterfallen". Für jedes Level existieren festgelegte Formen, mit deren Hilfe sich das Level lösen lässt.

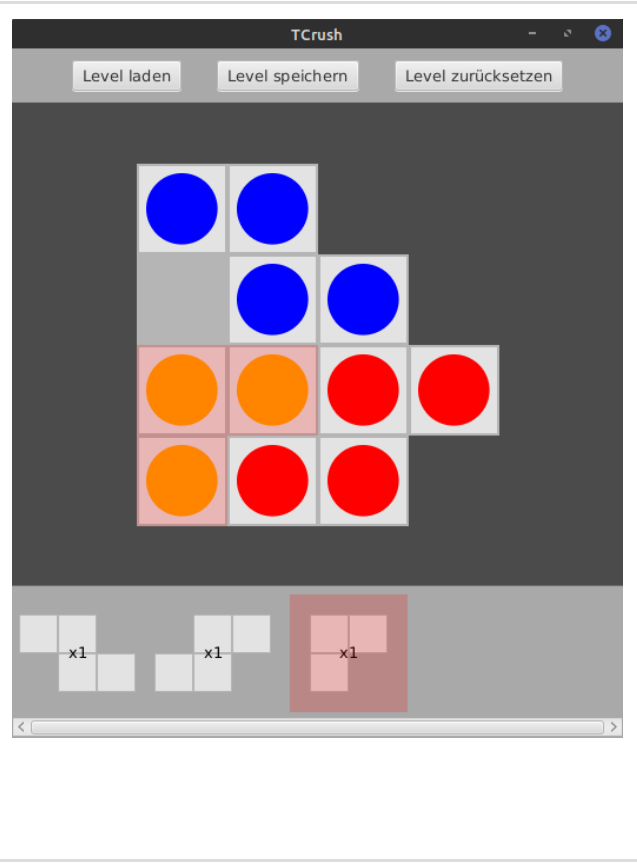
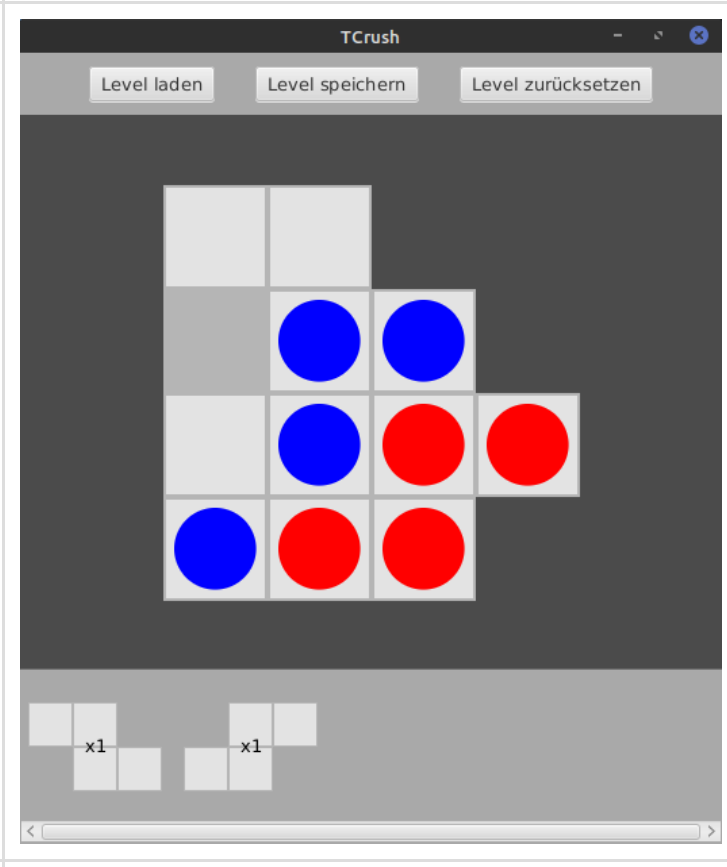
So könnte ein Level aussehen:



Dabei gibt es drei verschiedene Typen von Spielfeldern:

- *Zellen*: In diesen können sich Items befinden.
- *Blöcke*: Blockieren sich bewegende Items und können auch keine Items beinhalten.
- *Fallthroughs*: Bewegende Items können durch sie hindurchfallen, sie können aber keine beinhalten.

Dazu ein beispielhafter Spielzug:

vorher	nachher
 <p>Das Bild zeigt den TCrush-Spielstand vor einer Aktion. Die Spielplatte ist mit verschiedenen farbigen Kreisen (blau, orange, rot) besetzt. Die untere Leiste zeigt die verfügbaren Steine, die jeweils mit 'x1' markiert sind.</p>	 <p>Das Bild zeigt den TCrush-Spielstand nach einer Aktion. Die Spielplatte ist mit verschiedenen farbigen Kreisen (blau, orange, rot) besetzt. Die untere Leiste zeigt die verfügbaren Steine, die jeweils mit 'x1' markiert sind.</p>
<p>Positionen, die durch die Form betroffen sind, werden vorher markiert.</p>	<p>Das linke blaue Item fällt durch das Fallthrough-Element und die Form ist nicht mehr verfügbar.</p>

Ziel dieser Aufgabe ist es dieses Spiel umzusetzen. Dabei soll zusätzlich ein Parser implementiert werden, der es ermöglicht Level aus Dateien zu laden und in diese zu speichern. Abschließend soll mit Hilfe von JavaFX eine GUI geschrieben werden mit der sich TCrush spielen lässt.

Wichtige Hinweise:

- Alle Klassen müssen die geforderten Methoden zur Verfügung stellen, können und sollen aber auch beliebig um Hilfsmethoden erweitert werden.
- Auch das zu definierende Interface können Sie **in dieser Aufgabe** um Methoden erweitern. In anderen Aufgaben in denen Interfaces vorgegeben sind ist dies unter Umständen aber **nicht** möglich!
- Die verschiedenen Teile dieser Aufgabe sind stark miteinander verknüpft. Ebenso sind es die Tests. Das heißt, dass Sie unbedingt die Reihenfolge der zu implementierenden Klassen einhalten sollten und auch bei einem nichtnachvollziehbarem Testergebnis zunächst überprüfen sollten, ob sie alle Tests für die vorherigen Teile bestanden haben.

1. Die Spiellogik

1.1 Hilfsklassen

Alle Klassen aus diesem Teil sind im Paket `jpp.tcrush.gamelogic.utils` anzulegen.

Die Klasse `Coordinate2D`

Instanzen dieser Klasse repräsentieren eine Position in einem zweidimensionalen Koordinatensystem. Stellen Sie folgenden Konstruktor zu Verfügung:

```
public Coordinate2D(int x, int y)
```

Dieser erstellt eine neue Koordinate mit den Koordinaten `x` und `y`. Implementieren Sie auch folgende Methoden:

```
public int getX() : Gibt die x-Koordinate zurück.
```

```
public int getY() : Gibt die y-Koordinate zurück.
```

Überschreiben Sie außerdem die Methoden `equals` und `hashCode`, sodass Instanzen von `Coordinate2D` als gleich angesehen werden, wenn diese die gleiche Position repräsentieren.

Außerdem soll die `toString` Methode so angepasst werden, dass die Objekte diese Stringrepräsentation besitzen:

```
(<x>,<y>)
```

Die Klasse `Move`

Um später Informationen über die Bewegungen nach einem Spielzug zu speichern wird die Klasse `Move` verwendet. Objekte dieser Klasse speichern dabei ihren Start- und Zielpunkt. Der Konstruktor lautet:

```
public Move(Coordinate2D from, Coordinate2D to)
```

Die übergebenen Koordinaten sind dabei der Start- und Endpunkt dieser Bewegung. Sollte eines der Argumente `null` sein werfen Sie eine `IllegalArgumentException`. Die Klasse enthält folgende Methoden:

```
public Coordinate2D getFrom() : Gibt die Startposition zurück.
```

```
public Coordinate2D getTo() : Gibt die Zielposition zurück.
```

1.2 Spielfeld

Implementieren Sie die Klassen dieses Teils im Paket `jpp.tcrush.gamelogic.field`.

Die Enumeration `GameFieldType`

Diese ist eine Aufzählung der verschiedenen Typen von Spielitems. Folgende Felder sind zu implementieren:

```
BLUE  
GREEN  
RED  
YELLOW  
PURPLE  
BLACK  
ORANGE
```

Die Klasse `GameFieldItem`

Ein Objekt vom Typ `GameFieldItem` repräsentiert ein Item im Spielfeld. Um später die verschiedenen Bewegungen nach einem Zug zu analysieren implementieren diese Objekte ein paar Methoden um an diese Informationen zu kommen. Dabei speichern die Objekte auch die Information, ob sie sich aktuell in einem Bewegungsprozess befinden. Dieser Konstruktor sollte zur Verfügung stehen:

```
public GameFieldItem(GameFieldItemType type)
```

Durch diesen wird ein neues `GameFieldItem` vom Typ `type` erstellt. Der Typ ist dabei für jedes Objekt unveränderlich, da Items nicht die Farbe wechseln können. Lediglich an welcher Stelle sich ein Item befindet kann sich im Laufe eines Spieles verändern. Die Objekte stellen folgende Methoden zur Verfügung:

```
public GameFieldItemType getType() : Gibt den Typ des Items zurück.
```

```
public void startMove(Coordinate2D startPosition) : Startet auf diesem Objekt einen neuen  
Bewegungsprozess von der Position startPosition aus. Sollte startPosition null sein, werfen Sie  
eine IllegalArgumentException und sollte sich das Item schon in einem Bewegungsprozess befinden  
werfen Sie eine IllegalStateException.
```

```
public Move endMove(Coordinate2D endPosition) : Beendet einen Bewegungsprozess zu der Position  
endPosition und gibt das diese Bewegung beschreibende Move-Objekt zurück. Sollte endPosition  
null sein, werfen Sie eine IllegalArgumentException und sollte sich das Item in keinem  
Bewegungsprozess befinden werfen Sie eine IllegalStateException.
```

```
public boolean isOnMove() : Gibt an, ob sich das Item gerade bewegt oder nicht.
```

Die Enumeration `GameFieldElementType`

In der Beschreibung der Spiellogik wurden schon die verschiedenen Arten von Spielfeldern angesprochen. Diese Klasse ist eine Aufzählung dieser Typen und hat diese Felder:

```
CELL  
BLOCK  
FALLTHROUGH
```

Das Interface `GameFieldElement`

Um nun alle Elemente des Spielfelds später gleich behandeln zu können, soll nun das Interface `GameFieldElement` geschrieben werden. Es statet die Objekte, die das Spielfeld bilden mit den nötigen Funktionen aus, um Veränderungen der Spielfläche auszuführen. Gleichzeitig enthält es drei statische Methoden, über die die verschiedenen Elemente erzeugt werden können. Geben Sie durch das Interface folgende Instanz-Methoden vor:

```
public Optional<GameFieldItem> getItem() : Gibt das Item dieses Feldelements zurück. Sollte es kein  
Item besitzen, soll hier ein leeres Optional zurückgegeben werden. Diese Methode wird nur von Zellen  
unterstützt.
```

```
public GameFieldElementType getType() : Gibt den Typ des Feldelements an.
```

```
public Optional<GameFieldElement> getPredecessor() : Gibt den Vorgänger dieser Feldposition an. Der  
Vorgänger ist dabei das Feldelement, von dem dieses Element im Falle einer Veränderung (z.B. durch  
Auflösung des Items dieses Elements) ein neues Item erhält. Logischerweise kann es auch sein, dass es  
keinen Vorgänger gibt, da sich dieses Element im Level ganz oben oder unter einem Block befindet. In diesem  
Fall soll ein leeres Optional zurückgegeben werden. Diese Methode wird nur von Zellen unterstützt.
```

```
public Optional<GameFieldElement> getSuccessor() : Gibt den Nachfolger dieser Feldposition an.  
Analog zum Vorgänger ist der Nachfolger das Element, an das dieses Element sein Item weitergibt, wenn der  
Nachfolger kein Item mehr besitzt. Auch hier kann es sein, dass es Elemente ohne Nachfolger gibt. Das sind  
dann Solche, die sich im Level ganz unten oder über Blöcken befinden. Diese Methode wird nur von Zellen  
unterstützt.
```

```
public Coordinate2D getPos() : Gibt die Position dieser Zelle im Spielfeld an.
```

```
public void setItem(Optional<GameFieldItem> item) : Setzt das Item dieses Elements. Möchte man  
das Item löschen wird hier ein leeres Optional übergeben. Diese Methode wird nur von Zellen unterstützt.
```

`public void setPredecessor(GameFieldElement field) :` Setzt den Vorgänger dieses Elements. Sollte das Argument `null` sein werfen Sie eine `IllegalArgumentException`. Diese Methode wird nur von *Zellen* unterstützt.

`public void setSuccessor(GameFieldElement field) :` Setzt den Nachfolger dieses Elements. Sollte das Argument `null` sein werfen Sie eine `IllegalArgumentException`. Diese Methode wird nur von *Zellen* unterstützt.

`public void update(Collection<Move> moves) :` Aktualisiert die Position des Items dieses Elements und aller anderen Items, die durch diese Veränderung betroffen sind. Die daraus entstehenden Bewegungen sollen in der übergebenen `Collection` gespeichert werden. Diese Methode wird nur von *Zellen* unterstützt.

Wie Sie sicher festgestellt haben, wird die "Schwerkraft" in TCrush durch doppelt verkettete Listen simuliert. Jede Zelle kennt dabei ihren Vorgänger und Nachfolger. So können sehr leicht Spielfeldelemente wie *Fallthroughs* umgesetzt werden, da sich hier lediglich die Zellen über und unter dem *Fallthrough*-Element kennen. Durch die `update`-Methode soll man nun dafür sorgen können, dass sich eine Verkettung von Zellen aktualisiert und ihre Items weitergeben. Implementieren Sie diese Methode am besten rekursiv, da sich in jeder Ebene die gleichen Dinge abspielen. Dazu ein Beispiel:

```
//Verkettete Zellen sind hier als Liste dargestellt; "oben" ist links
Kette: [grün, blau, kein item, kein item]

//Nach einem Aufruf von update auf der zweiten Zelle von links
//sollte die Situation so aussehen:
Kette: [kein item, kein item, grün, blau]

//Gleichzeitig sollen in der übergebenen Move-Collection zwei Bewegungen hinzugekommen
sein:
Moves: [0 -> 2, 1-> 3] (indexnotierte Zahlen)
```

Der grobe Ablauf bei einem Aufruf von `update` ist dabei so:

1. Die Abbruchbedingung ist, dass diese Zelle kein Item besitzt
2. Besitzt diese Zelle einen Nachfolger und hat dieser aktuell kein Item, so muss das Item dieser Zelle weitergegeben werden.
 1. Nach einer Weitergabe wird die Referenz auf das Item zurückgesetzt.
 2. Um die aus der Weitergabe resultierenden Bewegungen zu vollenden, muss der Nachfolger, sowie der Vorgänger (insofern vorhanden) aktualisiert werden.
3. Besitzt diese Zelle keinen Nachfolger oder besitzt dieser aktuell ein Item, so muss überprüft, werden ob diese Zelle ihr Item durch eine vorherige Aktualisierung erhalten hat. Falls ja, muss die daraus entstandene Bewegung der `Collection` hinzugefügt werden, da sich das Item nun offensichtlich an einem Endpunkt befindet.

Im Interface `GameFieldElement` sollen nun auch diese drei Factory-Methoden zur Verfügung stehen:

`public static GameFieldElement createCell(GameFieldItem item, Coordinate2D pos) :` Erzeugt eine neue Zelle mit entsprechendem Item und Position. Sollte `pos` `null` sein werfen Sie eine `IllegalArgumentException`. Sollte `item` `null` sein, besitzt diese Zelle aktuell kein Item.

`public static GameFieldElement createBlock(Coordinate2D pos) :` Erzeugt einen neuen Block mit entsprechender Position. Sollte `pos` `null` sein werfen Sie eine `IllegalArgumentException`.

`public static GameFieldElement createFallthrough(Coordinate2D pos) :` Erzeugt einen neuen *Fallthrough* mit entsprechender Position. Sollte `pos` `null` sein werfen Sie eine `IllegalArgumentException`.

Um diese Methoden umzusetzen sollten Sie drei verschiedene Klassen schreiben, die alle das `GameFieldElement` -Interface implementieren. **Achtung:** Verwenden Sie dabei keine anonymen Klassen, da es durch die spezielle Testumgebung dann nicht möglich ist Methoden aus diesen Klassen aufzurufen. Die Methoden, die von *Blöcken* und *Fallthroughs* nicht unterstützt werden, sollen bei unerlaubtem Aufruf eine `UnsupportedOperationException` werfen.

Außerdem sollten alle Ihre `GameFieldElement` -Implementierungen die `toString` Methode nach folgendem Schema überschreiben:

```
'b' für Zellen mit Items des Typs BLUE
'g' für Zellen mit Items des Typs GREEN
'r' für Zellen mit Items des Typs RED
'y' für Zellen mit Items des Typs YELLOW
'p' für Zellen mit Items des Typs PURPLE
'B' für Zellen mit Items des Typs BLACK
'o' für Zellen mit Items des Typs ORANGE
'n' für Zellen ohne Item
'#' für Blöcke
'+' für Fallthroughs
```

1.3 Die Formen

Im Folgenden wird nun

Die Klasse Shape

im Paket `jpp.tcrush.gamelogic` implementiert. Objekte dieser Klasse repräsentieren bestimmte, vollkommen frei wählbare Formen, die in das Spielfeld eingesetzt werden können. Eine Form ist dabei über eine Menge von Punkten sowie eine Anzahl definiert. Letztere speichert, wie viele Formen von dieser Art verfügbar sind, falls eine Form in einem Level mehrmals verwendet werden darf. Übliche Formen können außerdem mit einem Namen konstruiert werden, um sie später über eine einfache Syntax in einer Datei zu definieren. Daher besitzt die Klasse diese zwei Konstruktoren:

```
public Shape(Collection<Coordinate2D> points, String name, int amount)
```

und

```
public Shape(Collection<Coordinate2D> points, int amount)
```

Sollte `points` oder `name` `null`, `amount` kleiner oder gleich `0` oder `points` leer sein, so werfen Sie eine `IllegalArgumentException`.

Die Klasse enthält folgende Methoden:

```
public Collection<Coordinate2D> getPoints() : Gibt die Menge der Punkte, die diese Form definieren zurück. Achtung: Die zurückgegebene Menge darf nicht modifizierbar, also veränderbar sein.
```

```
public int getAmount() : Gibt die Anzahl der verfügbaren Formen dieses Typs an.
```

```
public boolean reduceAmount() : Verringert die Anzahl der verfügbaren Formen dieses Typs um eins. Sollten dadurch noch mehr als null Formen vorhanden sein, soll true zurückgegeben werden.
```

Überschreiben Sie außerdem mindestens die Methoden `equals` und `toString`. Zwei Formen sind demnach gleich, wenn ihre Punktmengen die gleichen Punkte beinhalten, unabhängig von der Anzahl. Das heißt, dass Sie nicht überprüfen müssen, ob zwei Formen die gleiche logische Form ergeben. Die Stringrepräsentation eines `Shape` -Objekts sieht dabei so aus:

```
//Wenn die Form einen Namen hat:
<amount>:<name>

//Wenn die Form keinen Namen hat:
<amount>:<Menge der Punkte, nach jedem Punkt kommt ein ';'>
//also:
<amount>:(<x1>,<y1>);(<x2>,<y2>); ... (xN,yN);
```

Nachdem diese Instanzmethoden und Konstruktoren implementiert wurden, sollen Sie jetzt einige statische Methoden schreiben, mit deren Hilfe man gänge Formen (nämlich solche mit Namen) erzeugen kann. Dazu zunächst ein Beispiel, wie eine Punktmenge einer Form aussehen kann:

Die Form



besitzt die Punktmenge:

```
[(0,0);(1,0);(0,1)]
```

Während die Form




die Punktmenge

```
[(0,0);(1,0);(0,-1)]
```

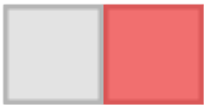
besitzt. Beachten Sie, dass jede Form so theoretisch unendlich viele Repräsentationen als Punktmenge besitzt, da man ja immer alle Punkte gleich weit in eine Richtung verschieben könnte. Für die Formen, die Sie im Folgenden konstruieren sollen ist deshalb zu jeder Form ein Referenzpunkt markiert. Dieser soll im Ergebnis $(0,0)$ entsprechend. Ausgehend von dieser Referenz können Sie dann die restlichen Punkte bestimmen. Diese Formen mit Namen sollen über die angegebenen Methoden erstellt werden:

```
public static Shape getPointShape(int amount)
```

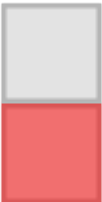
Name	Form
------	------

Name	Form
point	


public static Shape getRowShape(int amount)

Name	Form
sRow	


public static Shape getColumnShape(int amount)

Name	Form
sColumn	


public static Shape getRowShape(int amount)

Name	Form
row	


public static Shape getColumnShape(int amount)

Name	Form
column	

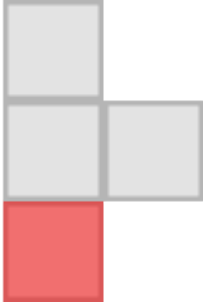
```
public static Shape getUpTShape(int amount)
```

Name	Form
upT	

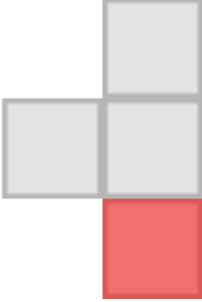
```
public static Shape getDownTShape(int amount)
```

Name	Form
downT	

```
public static Shape getRightTShape(int amount)
```

Name	Form
rightT	

```
public static Shape getLeftTShape(int amount)
```

Name	Form
leftT	

1.4 Das Level

Zum Abschluss der Spiellogik wird nun eine Klasse geschrieben, die alle bisherigen Elemente zusammenführt:

Die Klasse `Level`

Auch diese soll sich im Paket `jpp.tcrush.gamelogic` befinden. Instanzen dieser Klasse speichern den Status des Spielfelds, sowie welche Formen verfügbar sind. Der Konstruktor lautet:

```
public Level(Map<Coordinate2D,GameFieldElement> fieldMap, Collection<Shape>
allowedShapes)
```

In der `Map` sind dabei Positionen im Spielfeld zu den an diesen Stellen zugehörigen Feldelementen gespeichert. Beachten Sie, dass im Spielfeld $(0,0)$ sich oben links befindet und die y -Koordinate mit sinkender Reihe steigt. Das steht im Gegensatz zu der Notation der Formen, die quasi in einem herkömmlichen Koordinatensystem angegeben sind. Grundsätzlich kann außerdem davon ausgegangen werden, dass alle Positionen immer ein ausgefülltes Rechteck ergeben. Sollten die Dimensionen dieses Rechtecks hinsichtlich Höhe oder Breite aber kleiner zwei sein, werfen Sie hier eine `IllegalArgumentException`. Diese muss auch im Falle einer leeren Form-Menge geworfen werden, oder wenn einer der übergebenen Parameter `null` ist.

Diese Methoden müssen implementiert werden:

```
public int getHeight() : Gibt die Höhe des Levels zurück. Diese ergibt sich aus der übergebenen Map
aus dem Konstruktor.
```

```
public int getWidth() : Gibt die Breite des Levels zurück. Auch diese ergibt sich aus der übergebenen
Map .
```

```
public Map<Coordinate2D, GameFieldElement> getField() : Gibt den aktuellen Spielfeldzustand zurück.
Achten Sie darauf, dass die zurückgegebene Map nicht modifizierbar ist.
```

```
public Collection<Shape> getAllowedShapes() : Gibt eine Menge der aktuell zur Verfügung stehenden
Formen zurück. Auch diese Menge darf nicht modifizierbar sein.
```

```
public Optional<Collection<Coordinate2D>> fit(Shape shape, Coordinate2D position) :
Überprüft ob die übergebene Form an die übergebene Position im Spielfeld passt. Eine Form passt, wenn
durch sie nur Zellen überdeckt werden, die Items des gleichen Types beinhalten. Falls die Form passt soll der
Rückgabewert eine Menge der Positionen enthalten, die von der Form überdeckt werden. Wenn sie nicht
passt soll ein leeres Optional zurückgegeben werden. Achtung: Diese Methode ändert nicht den Status des
Spielfelds, sondern informiert lediglich ob ein entsprechender Zug möglich ist. Ist einer der Parameter null
werfen Sie eine IllegalArgumentException .
```

```
public Collection<Move> setShapeAndUpdate(Shape shape, Coordinate2D position) : Diese
Methode setzt nun die übergebene Form an der übergebenen Position im Spielfeld. Dabei werden die
betroffenen Items gelöscht und die Feldelemente aktualisiert, sodass betroffene Items an ihre neue Position
```

gelangen. Der Rückgabewert enthält dabei alle durch die Löschung der Items entstandenen Bewegungen. Außerdem soll die Menge der erlaubten Formen um die jeweilige reduziert werden (beachten Sie dabei den Unterschied zwischen einer Form von der es nun eine Version weniger gibt und einer, die durch dieses Aktualisierung nicht mehr verfügbar ist). Es ist weiterhin möglich, dass das übergebene `shape`-Objekt nicht Teil der Menge der erlaubten Formen ist und trotzdem zur Verfügung steht, da es laut `equals` mit einem Objekt aus eben dieser Menge gleich ist. Achten Sie also darauf, dass Sie auch in diesem Fall die Anzahl des richtigen Objekts verändern. Werfen Sie in diesen Fällen eine `IllegalArgumentException` :

- Die Form passt nicht an der angegebenen Stelle
- Die Form ist aktuell nicht verfügbar
- Einer der Parameter ist `null`

`public boolean isWon()` : Gibt an, ob das aktuelle Level gelöst ist. Ein Level ist gelöst, wenn alle Items zum Verschwinden gebracht wurden.

`public boolean canMakeAnyMove()` : Gibt an, ob im aktuellen Levelzustand ein Zug möglich ist.

Zum Abschluss dieser Klasse sollen Sie nun noch die `toString`-Methode überschrieben werden. Dabei folgt die Stringrepräsentation eines Levels diesem Schema:

```
TCrush-LevelDefinition:
<Spielfeld: eine Zeile = eine Zeile im Feld>
//erlaubte Symbole sind hier : 'b','g','r','y','p','B','o','n','#','+'

Shapes:
<Stringrepräsentation der ersten Form>
<Stringrepräsentation der zweiten Form>
....
```

Ein Beispiel:

```
TCrush-LevelDefinition:
bb
bb

Shapes:
2:point
1:(0,0);(1,0);(0,-1);
```

2. Parser

Nachdem die Spiellogik abgeschlossen ist, soll nun ein Parser implementiert werden, der es erlaubt Level in beliebigen Zuständen zu speichern, sowie wieder einzulesen. Dazu werden zunächst ein paar Hilfsmethoden implementiert, die Ihnen die spätere Arbeit erleichtern. Die Klassen dieses Teils müssen im Paket `jpp.tcrush.parse` angelegt werden.

2.1 Die Klasse `ParserUtils`

Sie enthält die folgenden Hilfsmethoden:

`public static Coordinate2D parseStringToCoordinate(String s)` : Konstruiert eine neue Koordinate aus dem übergebenen String. Dabei ist dieser String in der schon für Koordinaten bekannten Form formatiert. Sollte diese Formatierung fehlerhaft sein werfen Sie eine `InputMismatchException` .

`public static Shape parseStringToShape(String s)` : Hier soll eine Form erstellt werden. Die Formatierung dieses Strings entspricht ebenfalls der, die schon für Formen bekannt ist. Das bedeutet, dass diese Methode auch in der Lage sein soll Strings wie "3:point" richtig zu verarbeiten. Auch hier soll eine `InputMismatchException` geworfen werden, wenn die Formatierung nicht passt.

`public static Map<Coordinate2D, GameFieldElement> parseStringToFieldMap(String s)` : Diese Methode wandelt einen String in ein Spielfeld um. In den Strings steht eine Zeile für eine Zeile im Spielfeld. Die Symbole 'b', 'g', 'r', 'y', 'p', 'B', 'o', 'n', '#', '+' sind dabei erlaubt. Achten Sie darauf, dass die in der Map enthaltenen Feldelemente mit den richtigen Items und Positionen erzeugt werden. Außerdem sollte die Verkettung der Zellen also die Vorgänger und Nachfolger richtig gesetzt werden. In folgenden Fällen soll dabei eine `InputMismatchException` geworfen werden:

- Wenn der String ein unerlaubtes Symbol enthält.
- Wenn der String kein Rechteck ergibt. Beispiel:

```
bb#
+r
bbb
```

- Wenn *Fallthroughs* so positioniert sind, dass ihnen keine Zellen nachfolgen oder vorgehen. Beispiel:

```
+bb
yy#
rBB
```

oder

```
#bb
yy+
rB+
```

Die Herausforderung dieser Methode liegt darin die Vorgänger und Nachfolger der Zellen zu setzen. Gerade bei den *Fallthrough*-Elementen muss darauf geachtet werden, dass diese auch verkettet werden können und so Zellen Vorgänger und Nachfolger besitzen können, die mehrere Zeilen von einander entfernt sind.

Wie dieses Problem gelöst wird bleibt Ihnen überlassen. Hier ein Hinweis für eine elegante Lösung:

Sie können im Interface `GameFieldElement` eine weitere Methode definieren und in allen dazugehörigen Klassen implementieren. Diese kann, wenn sie die richtigen Parameter erhält die Verkettung rekursiv vornehmen, wenn schon alle Elemente mit Position existieren.

2.2 Die Klasse `LevelParser`

Diese Klasse stellt zwei Methoden zur Verfügung mit denen nun ein Level eingelesen und gespeichert werden kann:

`public static Level parseStreamToLevel(InputStream inputStream)` : Liest die Informationen aus dem übergebenen `InputStream` und gibt ein entsprechendes Level zurück. Diese Daten sind dabei gemäß der `toString`-Methode der `Level`-Klasse formatiert. Auch hier soll eine `InputMismatchException` geworfen werden, wenn diese Informationen in irgendeiner Weise fehlerhaft sind. Um diese Methode auszutesten sind Ihnen in ihrem Verzeichnis einige Beispieldateien mitgeliefert.

`public static void writeLevelToStream(Level level, OutputStream outputStream)` : Schreibt das übergebene Level auf den übergebenen `OutputStream`.

3. Testen

Um Ihre Spiellogik und den Parser als Gesamtes zu testen finden Sie im Ordner `lib` das Archiv `tcrush_console.jar`. Dieses enthält ein Konsolenprogramm mit dem Sie Ihre Spiellogik und Parser testen können, wenn Sie alle bis zu diesem Zeitpunkt verlangten Klassen und Methoden implementiert haben. Um das Programm zu starten müssen Sie das Archiv als Bibliothek in Ihr Projekt einbinden (In IntelliJ geht das am einfachsten durch *Rechtsklick auf das Archiv -> Add as Library*). Beachten Sie hierbei, dass Sie die Bibliothek als Modul Bibliothek zum Modul `tcrush.main` hinzufügen, wenn Sie das Projekt über das Gradle-Script aufgesetzt haben. Ansonsten sind die Inhalte der Bibliothek nicht für den Compiler verfügbar.

Danach können Sie folgenden Befehl in einer `main`-Methode ausführen:

```
TCrushConsole.startConsoleForLevel(<Pfad zu einer Leveldatei>);
```

Im Ordner `resources` sind Ihnen Beispiellevel mitgeliefert, sodass Sie hier einfach Pfade auf diese Dateien übergeben können:

```
TCrushConsole.startConsoleForLevel("resources/testlevel_1");
```

Nach dem Start wird Ihnen in der Konsole der aktuelle Levelzustand angezeigt. Sie können über diese Notation eine Form einsetzen:

```
<Zeile der Form, die eingesetzt werden soll>:<x-Koordinate>,<y-Koordinate>
```

Beispielhafter Ablauf:

TCrush-LevelDefinition:

gb

+r

+r

br

Shapes:

1:point

1:column

1:sRow

Eingabe: 2:1,3

TCrush-LevelDefinition:

gn

+n

+n

bb

Shapes:

1:point

1:sRow

Eingabe: 2:1,3

TCrush-LevelDefinition:

nn

+n

+n

gn

Shapes:

1:point

Eingabe: 1:0,3

Level gelöst!

4. Die graphische Oberfläche

Klassen aus diesem Teil sollen im Paket `jpp.tcrush.gui` angelegt werden.

Da graphische Oberflächen nicht automatisiert getestet werden können, gibt es für diesen Teil der Aufgabe lediglich einen Test, der überprüft ob sich in der Klasse `TCrushGui` eine `main` Methode befindet. Diese soll von Ihnen zur Verfügung gestellt werden und die GUI starten.

Problem: GUI startet nicht

Sollte es Ihnen nicht möglich sein Ihre `main` Methode in `TCrushGui` direkt zu starten, so können Sie auch den Gradle-Task `run` ausführen. Dies funktioniert nur, wenn Sie JavaFX zuvor über das mitgelieferte Gradle-Script importiert haben. Um den Task in IntelliJ zu starten, öffnen Sie das Gradle-Menü am rechten Bildschirmrand. Öffnen Sie dann *Tasks -> application -> Doppelklick auf run*. Dies startet Ihre GUI aus der Klasse `TCrushGui`. Danach befindet sich diese Option auch unter Ihren verschiedenen Run-Configurations im oberen rechten Bildschirmbereich und kann direkt von dort gestartet werden, sodass Sie nicht immer auf das Gradle-Menü wechseln müssen.

Anforderungen

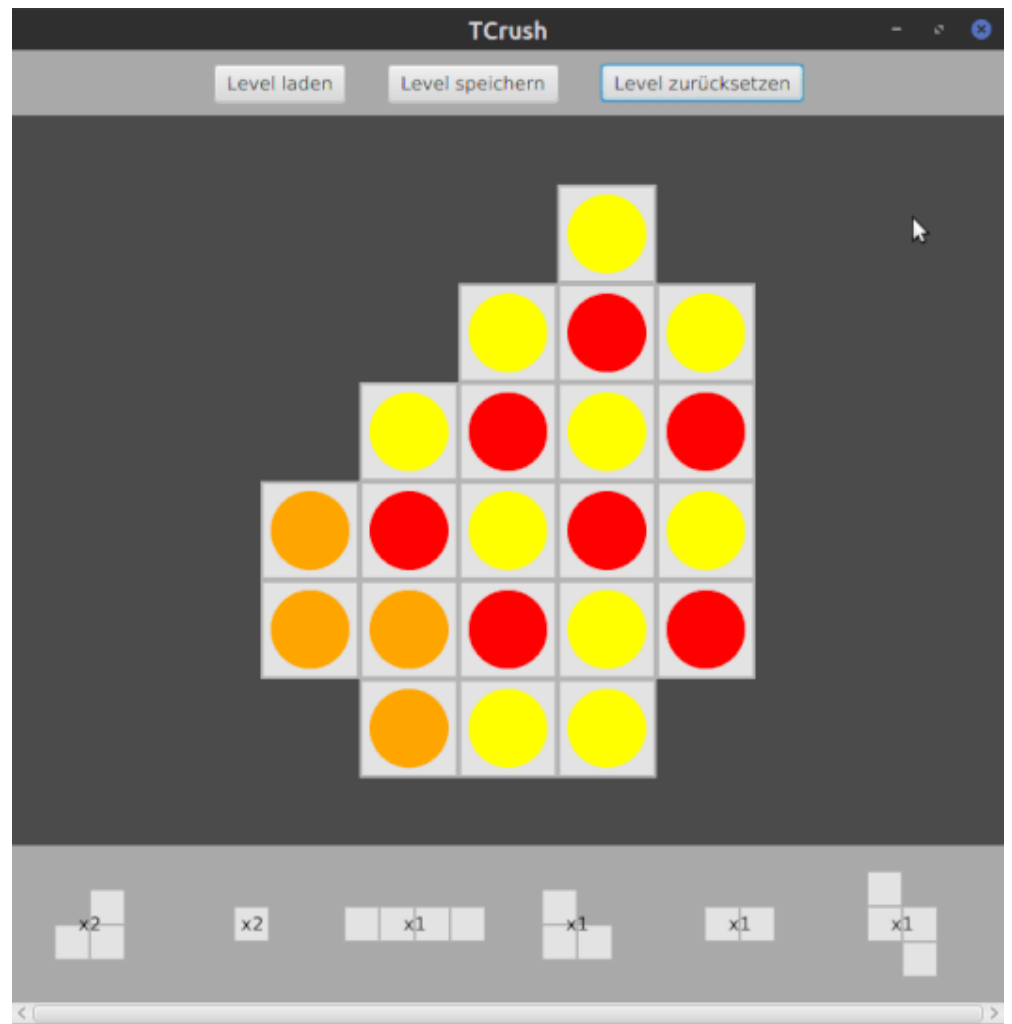
- Eine Schaltfläche über die ein Level geladen werden kann. Diese soll einen Dialog zur Dateiauswahl öffnen und nach der Auswahl einer Datei diese parsen. Wenn eine Datei ausgewählt wird, die zu keinem TCrush-Level verarbeitet werden kann, soll das aus der GUI durch eine Fehlermeldung erkennbar sein.
- Eine Schaltfläche über die das aktuelle Level gespeichert werden kann. Auch diese soll einen Dateidialog öffnen, über den die Zieldatei ausgewählt werden kann. Wenn gerade kein Level geladen ist, soll es auch nicht möglich sein ein Level zu speichern.
- Eine Schaltfläche über die das aktuelle Level zurückgesetzt werden kann. Diese soll die GUI auf den ursprünglichen Levelzustand zurücksetzen. Gemeint ist also der Zustand, nach dem das letzte Level geladen wurde. Auch diese Funktion soll nur möglich sein, wenn ein Level geladen wurde.
- Teilen Sie Ihre GUI so wie in den Beispielen auf. Also in Spielfeld und Formauswahl.
- Formauswahl:
 - Diese soll beliebig viele Formen beinhalten können. Implementieren Sie daher eine Scroll-Funktion für die Auswahl.
 - Sorgen Sie für aussagekräftige Vorschaubildern für die einzelnen Formen, die komplett dynamisch erzeugt werden. So sollen beliebig große Formen aller Art durch ihre GUI ein Vorschaubild bekommen. Hierzu der Hinweis auf Canvas . Mit diesen Objekten können Sie die Vorschau umsetzen, indem Sie auf einem Canvas Rechtecke zeichnen.
 - Zu jeder Form soll angezeigt werden, wie oft diese verwendet werden darf.
 - Wenn eine Form aufgebraucht wurde, soll sie aus der Auswahl entfernt werden.
 - Eine Form soll durch einen Mausklick ausgewählt werden können, um in das Spielfeld eingesetzt zu werden. Markieren Sie die ausgewählte Form, sodass der Nutzer nachvollziehen kann, welche Form er gerade einsetzt. Man kann immer nur eine Form auswählen, wenn also eine weitere Form angeklickt wird, muss die Markierung der ersten entfernt werden.
- Spielfeld:
 - Setzen Sie die verschiedenen Spielfeldelemente (*Zelle*, *Block* oder *Fallthrough*) optisch unterscheidbar um.
 - Zeichnen Sie die verschiedenen Items in ihren jeweiligen Farben in die *Zellen*.
 - Sorgen Sie für eine Vorschau des Bereichs auf den die aktuell ausgewählte Form Auswirkungen hätte. Diese Vorschau soll erfolgen, wenn eine Form ausgewählt wird und sich die Maus über das Spielfeld bewegt. Markieren Sie dann die Feldelemente, die an der aktuellen Mausposition von der aktuellen Form überdeckt würden, unabhängig davon, ob die Form passt oder nicht.
 - Durch einen Mausklick kann die Form eingesetzt werden und das Spielfeld wird aktualisiert. (Sie müssen keine Gravitationsanimation umsetzen)
 - Außerdem soll sich die Spielfeldgröße (also die Größe der Elemente) dynamisch in den zur Verfügung stehenden Platz einpassen, sodass beliebig große Level nicht abgeschnitten werden.
- Benachrichtigen Sie im Falle eines gelösten Levels oder eines fehlgeschlagenen Lösungsversuchs den Spieler über die GUI.

Demo

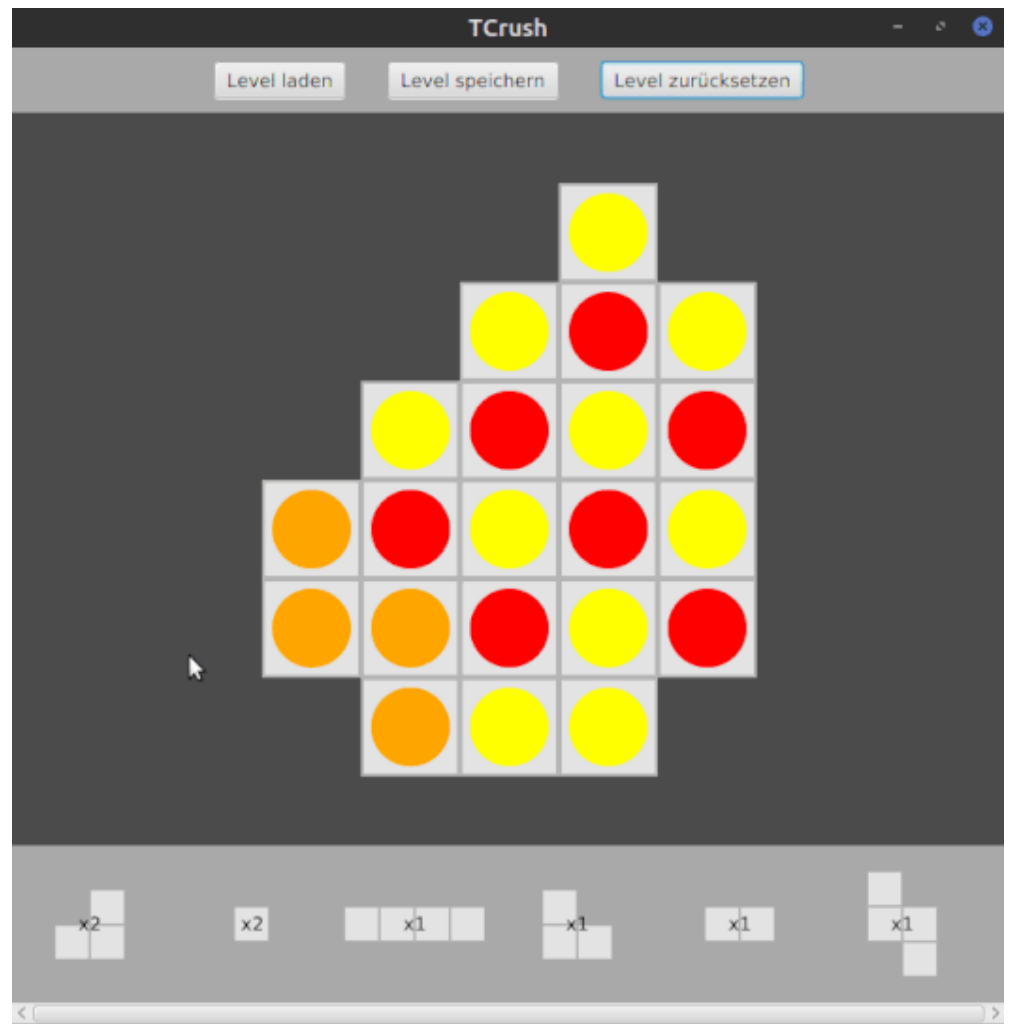
Lassen Sie sich von folgendem Animationen inspirieren. Sie demonstrieren die verschiedenen Anforderungen:



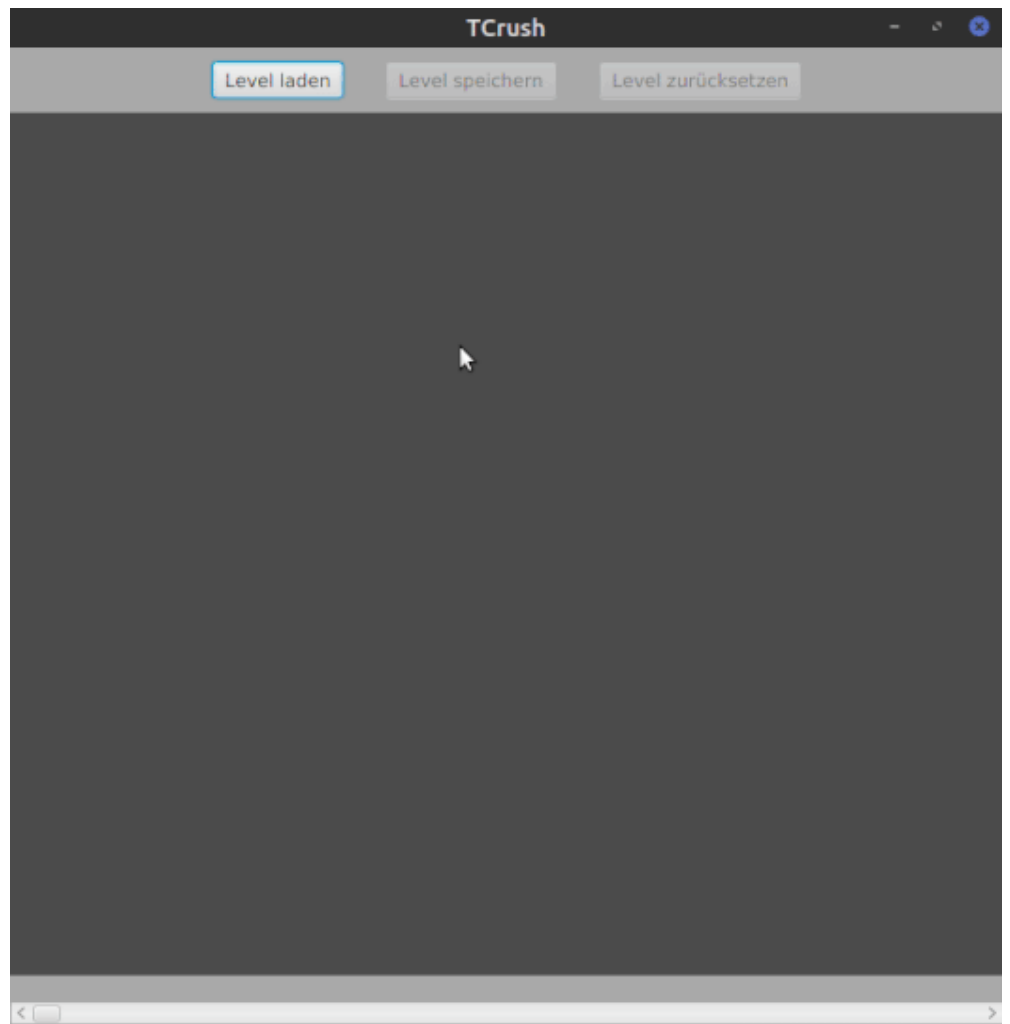
Skalierung

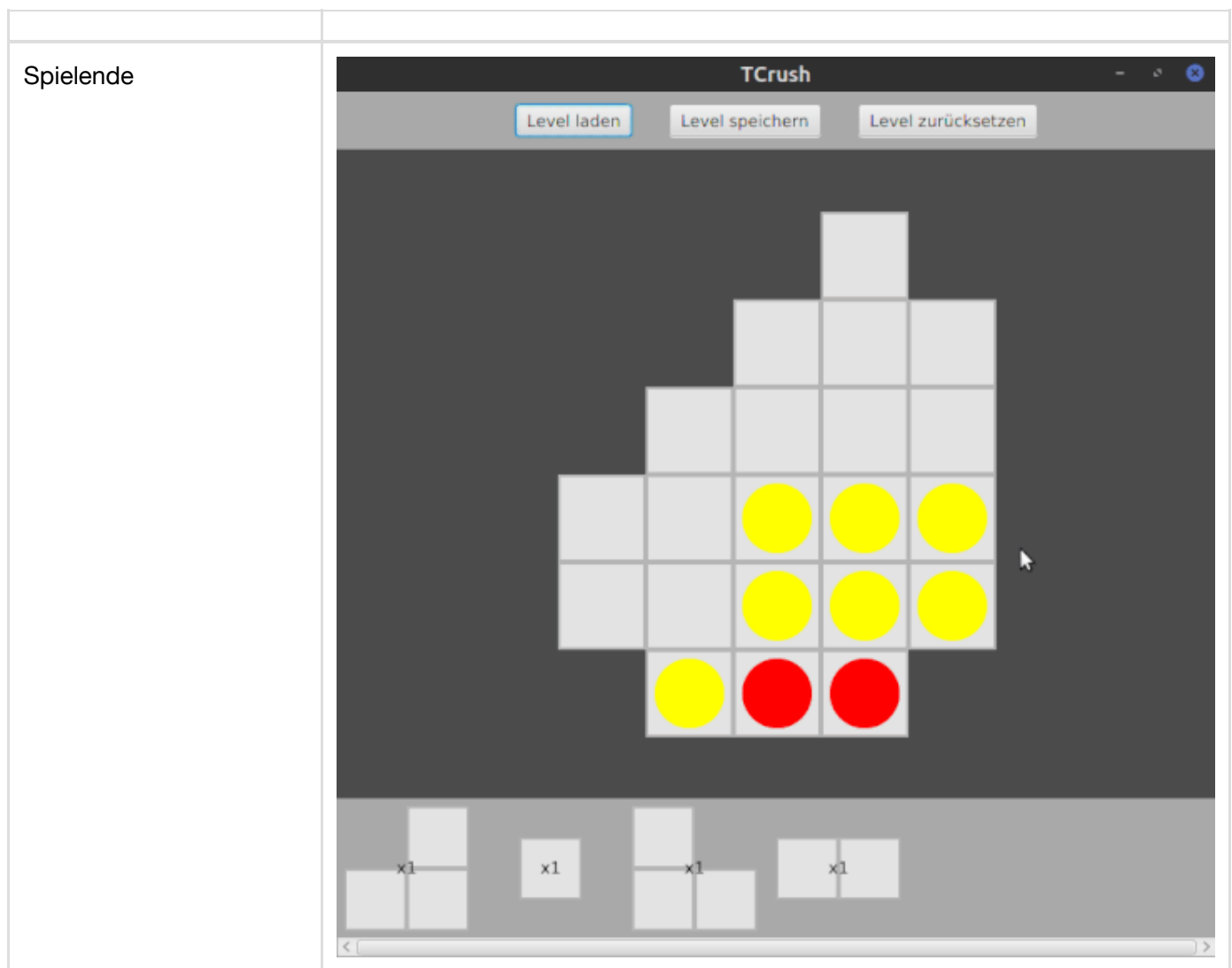


Vorschau auf dem
Spielfeld



Speichern und Laden





Punkteverteilung

Achtung: Diese Verteilung ist unverbindlich und dient nur zur Orientierung!

Abschnitt	Punkte
Coordinate2D	2
Move	2
GameFieldType	1
GameFieldItem	2
GameFieldTypeElement	1
GameFieldElement	3
Zellen-Implementierung	6
Block-Implementierung	2
Fallthrough-Implementierung	2
Shape	7

Abschnitt	Punkte
Level	12
ParserUtils	15
LevelParser	5
GUI	40

Viel Spaß!

PABS 3.10.3.8 - University of Würzburg - Impressum (<https://www.uni-wuerzburg.de/sonstiges/impressum/>) -
Datenschutz (<https://www.uni-wuerzburg.de/sonstiges/datenschutz/>)