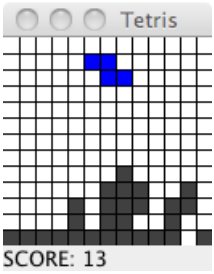
Tetris

Wir programmieren das allseits bekannte Spiel Tetris, um daran etliche OOP- und Java-Konzepte zu erlernen. Bei den Konzepten wollen wir von Anfang an einen weiten Horizont im Auge behalten, während das Spiel selber vorerst so wenig Zuneigung erfährt, wie gerade noch tragbar ist. Am Ende kann es dann jeder nach eigenem Geschmack zurechtmachen.

Spielregeln

Ein Stein (im Bild blau dargestellt) fällt so lange runter, bis er nicht mehr weiter kann. Nur während dieser Zeit wirken folgende Tasten:



- Cursor links: um eine Position nach links verschieben
- Cursor rechts: um eine Position nach rechts verschieben
- Cursor auf: um 90° drehen
- Cursor ab: schnell fallen lassen
- Leertaste: pausieren/weiterspielen
- Enter: am Ende ein neues Spiel starten

Wenn der Stein so weit gefallen ist, dass er feststeckt, wird er mit der Umgebung verschmolzen (grau dargestellt); oben entsteht dann ein neuer (blauer) Stein. Punkte gibt es, wenn eine vollständige Zeile entsteht. Die wird dann entfernt, und der Spieler hat Platz zum Weiterspielen. Reagiert der Spieler nicht schnell genug, wächst die Umgebung so hoch, dass er nicht mehr spielen kann. Dann ist GAME OVER.

1 Spots und Steine

Machen Sie jetzt nicht den Lieblingsfehler aller Anfänger, Ihre Gedanken an das sichtbare Spielfeld zu heften! Stellen Sie sich lieber auf den abwegig scheinenden Standpunkt, dass das Spiel auch dann spielbar sein kann, wenn man es überhaupt nicht sieht. Wenn Sie das durchhalten, werden Sie am Ende stabil und unabhängig funktionierende Bausteine geschaffen haben. Die Sichtbarmachung ist dann nur noch eine Kleinigkeit.

Die folgenden Ideen sind die kurze Wiedergabe eines längeren Denkprozesses, auch wenn es vielleicht nicht so aussieht:

Ein Stein besteht aus mehreren Stücken, nennen wir sie Spots. Üblicherweise sind es vier (deshalb *Tetris*), aber da können wir problemlos mehr Freiheit einplanen.

Wenn ein Stein bewegt oder gedreht werden soll, kann es sein, dass dafür gar kein Platz ist, was man aber erst prüfen kann, wenn die Veränderung durchgeführt

ist. Deshalb werden wir für Veränderungen aus dem aktuellen Stein einen neuen erzeugen mit neuen Spots. Wenn der Platz reicht, vergessen wir den alten Stein und arbeiten mit dem neuen weiter. Wenn der Platz nicht reicht, haben wir noch den unveränderten alten Stein und können seine Spots mit der Umgebung verschmelzen.

Damit verzichten wir auf die Verlockung, einen Stein als veränderliches Objekt zu betrachten und modellieren ihn *immutable*, wie man sagt. Damit ist nebenbei gesichert, dass ein Stein, den außer uns noch jemand anders in Händen hat, nicht unerwartet verändert wird. (Jemand anders können natürlich auch wir selber an einer anderen Stelle sein.)

1.1 Spot

- Ein `Spot` ist ein Ort auf dem Spielfeld, der evtl. nur für Rechnungen dient (z.B. wenn um ihn eine Drehung durchgeführt wird). Es kann aber auch sein, dass er irgendwie angezeigt wird, z.B. als Teil eines Steins.
- Ein `Spot` hat je einen ganzzahligen `x`- und `y`-Wert, der beim Erzeugen festgelegt wird und danach nicht mehr verändert werden kann. (In Java ist es üblich, dass (0|0) links oben liegt, die x-Achse nach rechts wächst und die y-Achse nach unten.) Erfragt werden können die Werte mit `getX()` und `getY()`.
- `Spot transformiert(Spot c, int dr, int dx, int dy)` erzeugt einen neuen Spot, der im Vergleich zu *this* Spot `dr` mal um den Drehpunkt `c` um 90° im Uhrzeigersinn gedreht und sodann um `dx` nach rechts und um `dy` nach unten verschoben wurde. Z.B. habe *this* die Werte $(x|y) = (6|2)$ und habe `c` die Werte $(x|y) = (1|-1)$. Die Argumente für `transformiere` seien $dx = dy = 0$ und $dr = 1$. Dann muss `transformiere` einen `Spot` erzeugen mit $(x|y) = (-2|4)$.

  Hätte man $dx = 10$ und $dy = 20$, so bekäme man natürlich $(x|y) = (8|24)$, aber mit der Verschiebung werden Sie eh keine Probleme haben.

  Um die nicht ganz so leichte Mathematik der Drehung einzusehen, sollten Sie sich ein paar solche Fälle auf kariertem Papier aufzeichnen und per Hand durchrechnen, bis Sie die Formel gefunden haben.
- Eine `static`-Methode gibt es noch, die gut in die Klassendatei von `Spot` passt: `Spot[] arrayAus(int... xy)` bekommt eine unbestimmte Anzahl von ganzen Zahlen und interpretiert diese als abwechselnd `x`- und `y`-Werte von Spots. Die Methode erzeugt daraus ein Array von Spots mit diesen Koordinaten. Die Methode soll eine `IllegalArgumentException` werfen, wenn die Anzahl der übergebenen Zahlen nicht gerade ist. (Informieren Sie sich über *Varargs*, wenn Sie sowas noch nie gesehen haben.)

Implementieren Sie das in einem Grobgerüst, das nicht funktioniert, aber kompilierbar ist. Bauen Sie also alle Attribute und Methoden ein, aber machen Sie sie leer. Schreiben Sie z B in `getX` nur `return 0` und in `transformiert` nur `return null`. Dann sind Sie schnell fertig und können schon an die nächste Klasse denken.

1.2 Stein

- Ein `Stein` hat einen Spot als Drehzentrum und ein Array von Spots, die definieren, welchen Platz er einnimmt. Zwei Konstruktoren scheinen angemessen.
- Ein `Stein` kann erzeugt werden aus dem Dreh-`Spot` und einem `Spot[]`.
- Ein `Stein` kann erzeugt werden aus zwei Ganzzahlen, die den Dreh-`Spot` definieren und einer unbestimmten Menge von Ganzzahlen, die die restlichen Spots definieren (Varargs).
- `spots()` gibt die Menge der belegten Spots als Array zurück. Da ein Empfänger aber stets in der Lage ist, ein Array zu verändern, ist es sicherer, nicht die Original-Spots zu liefern, sondern eine mit `clone` erzeugte Kopie.
- `transformiert(int dr, int dx, int dy)` tut dasselbe, wie die entsprechende Methode in `Spot`, nur dass natürlich ein `Stein` zurück gegeben wird und kein Drehspot mehr als Argument nötig ist, weil ein `Stein` ja schon selber einen solchen besitzt. Diese Methode kann diejenige in `Spot` gut verwenden.
- Die `toString()`-Methode könnte sich für Testzwecke als hilfreich erweisen.

1.3 Partnerarbeit

Erstellen Sie die Klasse `Stein` ebenso schnell und schlampig – aber immerhin kompilierbar – wie `Spot`. Suchen Sie sich dann einen Partner, sodass nur einer von Ihnen `Spot` und der andere `Stein` präzisiert. Beide sind vergleichbar einfach.

Solange Sie an Ihrer Klasse arbeiten, sollten Sie eine `main`-Methode einbauen, in der Sie viele Tests durchführen. Kümmern Sie sich möglichst immer nur um eine Methode. Machen Sie immer nur kleine Änderungen und kompilieren Sie oft!

Der Implementer von `Stein` kann seine eigene `Spot` immer wieder ein bisschen aufbessern, um Tests durchzuführen. Wenn der andere dann `Spot` fertig hat, funktioniert plötzlich alles problemlos. Die `main`-Methode wird *ganz* am Ende entfernt, nur die Hauptklasse wird eine `main` haben.

1.4 Steinmetz

Im Lauf des Spiels werden immer wieder neue zufällige Steine gebraucht. Dies übernimmt die Methode `erzeugeStein()` der Klasse `Steinmetz`. Implementieren Sie vorerst nur so viel, dass sie immer die gleiche Sorte erzeugt. Später können Sie die Methode per Zufall verbessern.

Die erzeugten Steine sollen ganz links oben liegen; achten Sie also darauf, dass das Minimum der x- und y-Koordinaten der Spots den Wert 0 haben.

2 Ein Feld und seine Listener

2.1 Feld

Ein `Feld` ist etwas, das sich merkt, welche Orte schon belegt sind. Es kennt den aktuellen Stein und kann feststellen, ob er in einer veränderten Fassung noch Platz hat. Genau wie `Spot` und `Stein` arbeitet ein `Feld` ohne dass man es sehen muss.

Implementieren Sie, wie schon bei den anderen Klassen, ein möglichst leeres, wohl aber kompilierbares Grundgerüst mit folgenden Bestandteilen:

- `Feld(int breit, int tief)` erzeugt ein Feld gegebener Breite und Tiefe. Fügen Sie ein geeignetes Attribut hinzu, in dem sich gemerkt wird, welche Orte des Feldes schon belegt sind. Sie können ein ein- oder zweidimensionales Array verwenden. Im Grunde reicht ein `boolean[]`. Der Konstruktor sorgt jedenfalls dafür, dass dieses Attribut so initialisiert wird, dass das Feld als leer betrachtet werden kann.
- `int getBreite()` und `int getTiefe()` liefern Breite und Tiefe des Feldes.
- `boolean setStein(Stein)` macht den übergebenen Stein zum Spielstein, falls alle seine Positionen noch frei sind; es wird dann `true` zurück gegeben. Andernfalls bleibt der bisherige Stein erhalten und `false` wird zurück gegeben.
- `boolean neuerStein(Stein s)` rutscht den übergebenen Stein waagrecht in die Mitte. Diese Methode ruft `setStein` auf und gibt den mittig gerutschten Stein zurück. (Kommt es dabei jedoch zu Kollisionen, so wird `null` zurück gegeben.)
- `Stein getStein()` gibt den aktuellen Stein zurück.
- `boolean belegt(int x, int y)` gibt zurück, ob eine Feldposition schon belegt ist. Der Stein hat hierauf keinen Einfluss, der kann ja ggf. selber abgefragt werden. Wenn ein `Feld` nach Koordinaten gefragt wird, die gar nicht existieren, meldet es diese als belegt. Damit ist ein `Feld` im Grunde immer unendlich groß, hat aber nur eine kleine Menge unbelegter Plätze.
- `reset()` leert das Feld.
- `fixiere()` markiert die Positionen, die der aktuelle Stein einnimmt, im Feld als belegt. Damit gilt der Stein als mit dem Feld verschmolzen und das Feld hat keinen Stein mehr (wird aber bald darauf per `neuerStein` einen neuen zugewiesen bekommen).

Es mag Ihnen sinnlos vorkommen, Methoden nur soweit auszuformulieren, dass sie kompilierbar sind. Aber gerade bei dieser Tätigkeit begreift man in Ruhe, was später mal sein wird. Insbesondere werden Sie die nun folgende Technik leichter verstehen.

2.2 FeldListener

Ein `FeldListener` ist jemand, der eine Arbeit verrichtet, sobald sich was Interessantes in dem `Feld` tut, für das er zuständig ist. (In unserem Fall gibt es nur ein Feld,

aber das ist egal.) Damit er diese Arbeit tun kann, muss er natürlich benachrichtigt werden, was passiert und wann.

Im Leben eines Feldes gibt es drei spannende Momente:

- Eine Zeile kann voll werden und verschwinden.
- Wenn das Feld bis oben voll gelaufen ist, hat der neue Stein keinen Platz mehr.
- Das Feld kann für eine neue Runde gelöscht werden.

Wenn jemand über ein Feld Bescheid wissen will, muss er über diese Vorgänge informiert werden.

Natürlich ist Tetris ein sehr kleines Projekt und wir wissen schon ziemlich genau, dass im Falle voller Zeilen Punkte zu addieren sind. Und natürlich könnte diese Aufgabe das Feld selber leicht erledigen. Aber hier soll es ja darum gehen, wie man auch mit größeren Problemen zurecht kommt. Versuchen wir also gar nicht erst, uns jetzt schon vorzustellen, *was dann getan werden muss*, sondern bleiben wir bei der Frage, welche Ereignisse es gibt. In Java haben wir die Möglichkeit, komplexe Ideen in mehrere Teile zerlegt auszuformulieren. Erzeugen Sie also erst einmal die Datei `FeldListener.java` mit dem Inhalt

```
public interface FeldListener {
  void volleZeilen(int abraum);

  void reset(int breit, int tief);

  void verstopft(Stein ursache);
}
```

und beachten Sie folgende Details:

- Es werden drei Methoden deklariert, aber nicht implementiert. Die Zeilen enden einfach mit einem Semikolon.
- Wo sonst `class` steht, finden Sie hier das Wort `interface`. Für den Compiler bedeutet das, er möge sich nicht aufregen, wenn er irgendwo mal den undefinierten Begriff `FeldListener` sieht. Er möge bitte keine Fehlermeldung ausgeben und abbrechen. Stattdessen soll er dann einfach mal locker bleiben und so tun, als hätte er es mit einer ganz normalen Klasse zu tun, in der die drei genannten Methoden ordentlich implementiert sind.
- Ein Interface ist ganz schnell hingeschrieben, weil man es eben nicht gleich implementieren muss. Das *nicht gleich implementieren* ist also so wichtig, dass es sogar ein eigenes Sprachkonstrukt dafür gibt.

2.3 Zusammenarbeit

Jetzt, wo der Compiler vorgewarnt ist, können wir die Klasse `Feld` fertig formulieren.

- Wir brauchen eine Liste und wollen sie nicht selber implementieren. Fügen Sie also in der ersten Zeile hinzu: `import java.util.LinkedList`.
- Fügen Sie das Attribut `LinkedList<FeldListener> angemeldet` hinzu. Hierin wird sich das Feld merken, wer über die interessanten Vorgänge informiert

werden muss. Die Klammern < und > bewirken, dass man der Liste nicht alle möglichen Objekte hinzufügen darf, sondern nur `FeldListener`-Instanzen.

- Implementieren Sie die Methode `void addFeldListener(FeldListener fl)`. Diese Methode bekommt einen `FeldListener` und fügt ihn der Liste hinzu. Mehr muss sie nicht tun.
- Implementieren Sie auch `void removeFeldListener(FeldListener fl)`, die das Gegenteil tut. Wir werden sie zwar nicht brauchen, aber das gehört sich so.

Damit ist der Plan abgesteckt und es folgt die Arbeit. Implementieren Sie nun alle angerissenen Methoden der Klasse `Feld` aus. Sie werden einige Attribute und vielleicht Hilfsmethoden brauchen.

Ein Aha-Erlebnis werden Sie hoffentlich haben, wenn Sie `reset()` programmieren. Sie werden das Feld geeignet leeren und müssen dann in einer Schleife alle in `angemeldet` gespeicherten `FeldListener` darüber benachrichtigen. Das geht so: Der Compiler geht davon aus, dass ein `FeldListener` die Methode `reset(int, int)` hat. Rufen Sie also die auf und übergeben Sie als Wert die Maße des Feldes. Ganz schön frech, wenn man bedenkt, dass es gar keine Klasse mit dieser Methode gibt!

Auch in `fixiere()` müssen Sie aufpassen. Da kann es passieren, dass eine Zeile voll wird. Löschen Sie sie dann und vergessen Sie nicht, alle in `angemeldet` darüber zu benachrichtigen durch Aufruf der `volleZeilen(int)`. Geben Sie als Argument die Anzahl der gelöschten Spots mit, denn wer sich angemeldet hat, wird wissen wollen, wie viele Spots wegfallen. Bedenken Sie auch, dass die wegfallende Zeile nicht die unterste sein muss und nicht die einzige!

Schließlich kann es noch in `neuerStein` passieren, dass der neue Stein keinen Platz mehr hat. Dann müssen Sie von allen in `angemeldet` die Methode `verstopft` aufrufen und den problematischen Stein übergeben.

Während der nun folgenden längeren Arbeitsphase wollen Sie `Feld` oft kompilieren und in Tests laufen lassen. Da wäre es doch hilfreich, schon mal jemand zu haben, der auch wirklich benachrichtigt werden kann, also ein richtiger `FeldListener`, nicht nur ein im Interface behaupteter. So einen Dummy baut man leicht so:

```
public class Dummy implements FeldListener{
  void volleZeilen(int abraum){
    System.out.println("Zeile mit "+abraum+" Feldern gelöscht.");
  }

  void reset(int breit, int tief){
    System.out.println("Feld geleert: Breite="+breit+" Tiefe="+tief);
  }

  void verstopft(Stein u){
    System.out.println(u+" hat alles zum Überlaufen gebracht.");
  }
}
```

In der `main`-Methode, die zum Testen von `Feld` dient, können Sie dann schreiben `feld.addFeldListener(new Dummy());` und schon werden auf dem Bildschirm alle

Ereignisse vermerkt. Wenn Sie zwei Dummys anmelden, wird alles zweimal vermerkt. Testen Sie auch das.

Nochmal langsam: `Dummy implements FeldListener` sagt dem Compiler, dass er überall, wo ein `FeldListener` auftaucht, bitte auch ein `Dummy` akzeptieren soll, denn `Dummy` hat ja offensichtlich die gewünschten Methoden eines `FeldListeners`.

3 Kontakt zum Menschen

Bis jetzt haben Sie die Zahnräder und Hebel der Maschine gebaut. Nun müssen wir uns der Tatsache stellen, dass Tetris etwas ist, das ein Mensch auch sehen und beeinflussen will; wir brauchen ein Gehäuse, genannt GUI. Wie gut, dass Sie die Idee mit den Listenern schon verstanden haben, die sind nämlich in GUI-Programmen ein unumgänglicher Standard.

Eine graphische Oberfläche ist ein komplexes Ding. Also werden Sie eine Menge Details erfahren, eine erschreckende Menge! Manche Leute verwenden eine IDE wie NetBeans, *die ihnen alles abnimmt*. Es ist aber eher ein Wahrnehmungsfehler, wenn die Einfachheit einer Sache danach bemessen wird, dass man die Maus länger und die Tastatur später benutzt. In diesem Projekt hilft Ihnen nicht eine IDE, sondern Ihr Lehrer. Und ich helfe Ihnen nicht mit Klickanweisungen, sondern mit einer Vorlage, an Hand derer ich die Zusammenhänge aufzeige. Das Schöne: Nachher bleibt nicht mehr viel zu lernen. Nur das erste Projekt ist schlimm. Beim zweiten weiß man, wo man nachschauen kann und ab dem dritten wird man langsam selbständig – ehrlich!

3.1 SpielComponent (quick and dirty)

Wir brauchen eine Komponente, die in der Lage ist, den Zustand des Spiels fortlaufend darzustellen und auf Eingaben zu reagieren. Da es eine solche nicht gibt, müssen wir sie selber erschaffen. Wir leiten ab von `JComponent`, einer rechteckigen graphischen Komponente, die als Vorfahre für vieles dient. Schreiben Sie in die Datei `SpielComponent.java` die Zeilen

```
import javax.swing.JComponent;
import java.awt.Color;
import java.awt.Graphics;

public class SpielComponent extends JComponent{
  public SpielComponent(Feld feld){ }

  public void paintComponent(Graphics g){
    g.setColor(Color.BLACK);    //auf die Schnelle: ein schräger
    g.drawLine(0, 0, 200, 100); //Strich. Das wird später entfernt.
  }
}
```

Wesentlich ist wieder einmal, dass wir schnell etwas Kompilierbares haben. Der Konstruktor wird ein `Feld` bekommen. In der `paintComponent`-Methode wird im Moment nur eine schräge, schwarze Linie gezeichnet, damit man die Komponente erkennt, wenn man sie sieht.

Die Philosophie der Selbstdarstellung soll hier kurz angeschnitten werden:

- Wenn eine Komponente aufgedeckt wird, weil z B ein darüberliegendes Fenster weggenommen wird, sagt das Betriebssystem der JVM (Java Virtual Machine), dass seine Oberfläche jetzt sichtbar ist und bitte gezeichnet werden soll.
- Die JVM ermittelt, welcher Teil der eigenen Oberfläche jetzt neu dargestellt werden muss und ruft von allen betroffenen Komponenten die Methode `paint` auf. Dabei wird ein `Graphics`-Objekt mitgegeben, das man sich am besten vorstellt als ein Stück des Bildschirms. Die Komponente kann darauf zeichnen, was sie für angemessen hält.
- In der `paint` von `JComponent` werden mehrere Teilzeichnungs-Methoden aufgerufen, eine davon ist `paintComponent`. Die muss später so programmiert werden, dass man das Spielfeld mit dem Stein sehen kann. Im Moment reicht aber der schräge Strich.
- In unserem Programm entsteht regelmäßig die Notwendigkeit der Neudarstellung allein dadurch, dass sich der Stein bewegt hat. Dann müssen wir und nicht das Betriebssystem die JVM darum bitten, die `paint`-Methode einer Komponente aufzurufen. Das geht, indem man die `repaint()`-Methode dieser Komponente aufruft.

3.2 Hauptfenster (quick and dirty)

Obwohl in der `SpielComponent`-Klasse noch sehr viel fehlt, kann man schon eine Instanz erzeugen und sie in ein Hauptfenster einbauen. Tun Sie das folgende:

- Erzeugen Sie die Klasse `TetrisFrame` als Nachfahre von `JFrame`. Sie werden die `score`-Punkte mitzählen wollen, geben Sie sich ein geeignetes Attribut.
- Im Konstruktor ist folgendes nötig:
  - Setzen Sie auf `SOUTH` ein `JLabel` mit dem Inhalt `SCORE: 0`. Merken Sie sich aber dieses JLabel als Attribut; Sie werden es später ändern wollen.
  - Erzeugen Sie eine Instanz von `Feld` mit einer vernünftigen Größe.
  - Erzeugen Sie eine Instanz von `SpielComponent`, die das soeben erzeugte Feld bekommt und setzen Sie sie auf `CENTER`.
  - Melden Sie sich beim Feld als `FeldListener` an. Dazu müssen Sie natürlich erst einmal ein kompetenter `FeldListener` sein. Implementieren Sie also die nötigen Methoden. Im Moment reicht es, wenn diese nichts tun.
- Erzeugen Sie in der `main` eine Instanz eines `TetrisFrames`.

Wenn alles gut gegangen ist, sieht man ein Fenster mit einer schrägen Linie und einer Score-Zeile unten. Nun kann die fehlende Funktionalität nachgeliefert werden.

3.3 SpielComponent (verbessert)

```
1  public class SpielComponent extends JComponent{
2    public static final int SPOT=10; //Seitenlänge eines Spots in Pixeln
3    private Feld feld;
4
5    public SpielComponent(Feld feld){
6      this.feld=feld;
7      setPreferredSize(new Dimension(feld.getBreite()*SPOT, feld.getTiefe()*SPOT));
8    }
9
10   public void paintComponent(Graphics g){
11     // Schleife: zeichne weiße oder graue Quadrate der
12     //           Seitenlänge SPOT je nach dem, wie feld belegt ist
13     // zeichne blauen Stein
14     // lege schwarzes Gitter drüber
15   }
16 }
```

2 Die Größe eines Spots wird hier als Konstante definiert.

3 Das Feld muss sich gemerkt werden, weil es für die Zeichnung der Anzeige gebraucht wird.

7 Irgendwann wird der Layoutmanager fragen, wie viel Platz man gern hätte. Hier wird die Antwort berechnet.

11-14 Hier muss nun jeder Spot in der richtigen Farbe gezeichnet werden, ebenso der blaue Stein und das Gitter. Wie das geht, liest man in der Dokumentation von `Graphics`.

3.4 Hauptfenster (verbessert)

```
1  public class TetrisFrame extends JFrame
2              implements FeldListener, ActionListener, KeyListener{
3    private ...// all die bisherigen Attribute
4    private Feld feld;
5    private javax.swing.Timer timer;
6    private Steinmetz steinmetz;
7
8    public TetrisFrame(){
9      super("Tetris");
10     timer=new Timer(500, this);//melde this als ActionListener an
11     timer.setDelay(300);
12     timer.start();
13     steinmetz=new Steinmetz();
14     // feld mit vernünftiger Größe erzeugen
15     // mit steinmetz einen ersten Stein erzeugen und dem feld geben
16     // SpielComponent sc für feld erzeugen und CENTER setzen
17     // Scoreanzeige auf SOUTH setzen
18     sc.addKeyListener(this);
19     feld.addFeldListener(this);
```

```
20    }
21
22    public void actionPerformed(ActionEvent ae){
23      removeKeyListener(this);
24      // was zu tun ist, wenn der Timer wieder mal abgelaufen ist
25      addKeyListener(this);
26      repaint();
27    }
28
29    public void keyPressed(KeyEvent ke){
30      Stein stein=feld.getStein();
31      switch(ke.getKeyCode()){
32        case KeyEvent.VK_LEFT: //eine Position nach links
33          break;
34        case KeyEvent.VK_RIGHT://eine Position nach rechts
35          break;
36        case KeyEvent.VK_UP:   //90° im Uhrzeigersinn drehen
37          break;
38        case KeyEvent.VK_DOWN://fallen lassen
39          break;
40        case KeyEvent.VK_SPACE://Pause ein- bzw. ausschalten
41          break;
42        case KeyEvent.VK_ENTER://neues Spiel
43          break;
44      }
45      feld.setStein(stein);
46      repaint();
47    }
48
49    public void keyTyped(KeyEvent e){}    //nicht verwendet
50    public void keyReleased(KeyEvent e){} //nicht verwendet
51
52    public void volleZeilen(int abraum){
53      // score hochzählen und anzeigen
54    }
55
56    public void reset(int breit, int tief){
57      // score auf 0 setzen und anzeigen
58    }
59
60    public void verstopft(Stein ursache){}
61
62    public static void main(String[] args){
63      TetrisFrame tetris=new TetrisFrame();
64    }
65  }
```

2 Der Compiler wird gebeten, sich davon zu überzeugen, dass `SpielComponent` nicht nur eine ordentliche `JComponent` ist (per Vererbung), sondern auch alle Anforderungen eines `KeyListeners` und eines `ActionListeners` erfüllt.

5 Damit der Stein schrittweise absinken kann, brauchen wir einen Timer, der regelmäßig abläuft. Bei einem Timer können wiederum `ActionListener` ange-

meldet sein, die immer dann benachrichtigt werden, wenn der Timer abgelaufen ist. Unser `TetrisFrame` möchte benachrichtigt werden, wenn das passiert, also muss er die Eigenschaften eines `ActionListeners` haben, ðlaut Dokumentation muss er die Methode `actionPerformed` haben.

10 Hier wird eine Timer-Instanz erzeugt mit einem Erstzeitraum von einer halben Sekunde. Da an Timern oft nur ein einziger `ActionListener` angemeldet wird, kann man dies hier gleich im Konstruktor tun. Man dürfte aber beliebig viele anmelden. Da `this` nun angemeldet ist, wird eine halbe Sekunde nach dem Timerstart die `actionPerformed` aufgerufen.

11,12 Die Zeit zwischen zwei aufeinander folgenden Timerdurchläufen soll 300 ms lang sein. Timer starten.

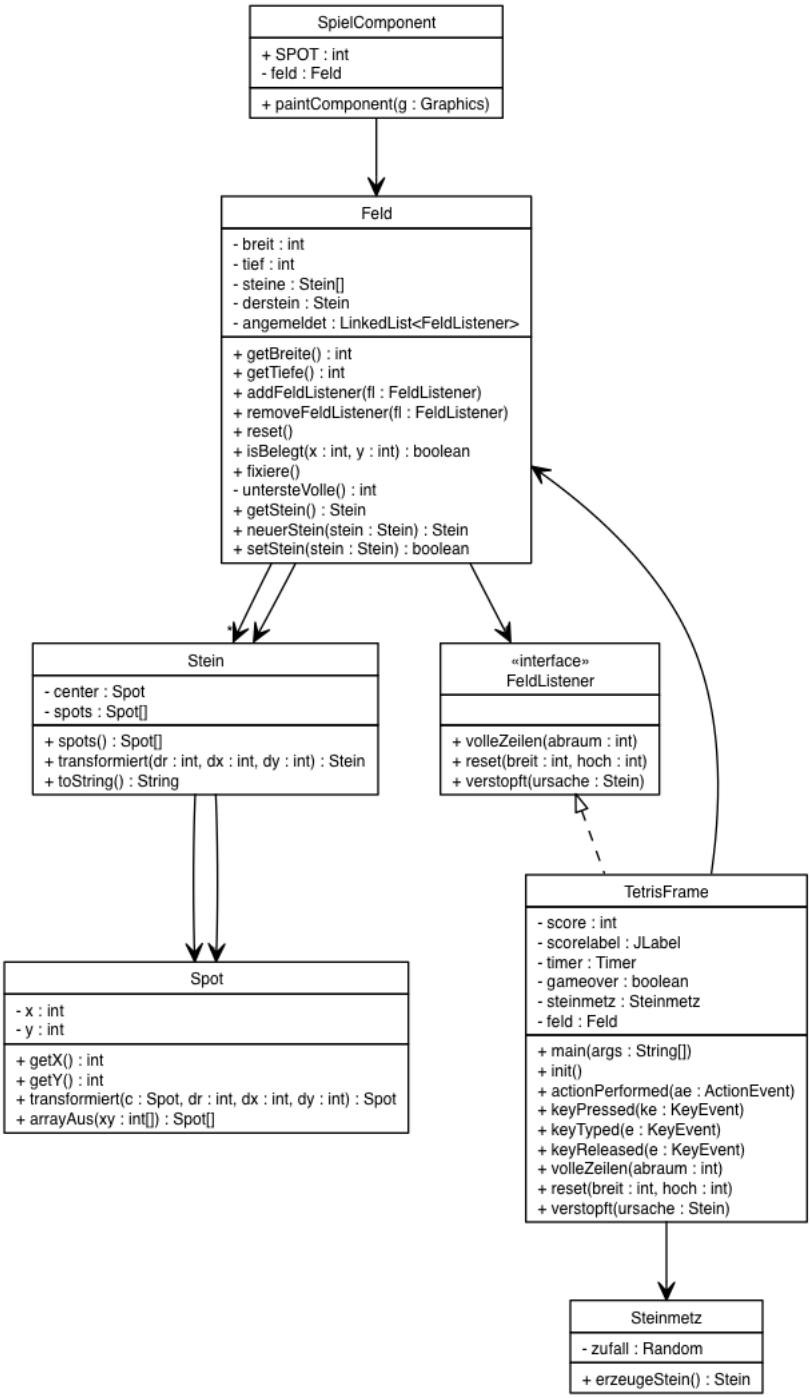
16 `SpielComponent` ist ein Nachfahre von `JComponent` und kann deshalb mit Tastaturereignissen umgehen. Wenn eine Taste gedrückt wird, werden alle angemeldeten `KeyListener` benachrichtigt. Wen könnten wir nun da anmelden? Laut Dokumentation muss es jemand sein, der die drei Methoden `keyPressed`, `keyReleased` und `keyTyped` hat. Wenn wir diese in `TetrisFrame` einbauen, kann sich ein solcher bei sich selber als `KeyListener` anmelden. Das ist ein klein wenig ungewöhnlich, etwa so wie ein Arzt, der sich selber als Patient hat.

23,25 Wenn der Timer abgelaufen ist, muss mit dem Stein etwas geschehen. Da wollen wir nicht, dass der Benutzer in dieser kurzen Zeit durch Tastendrücke dazwischen funkt. Wenn er es tut, bekommen wir es nicht mit, weil wir uns kurzzeitig abmelden. Genauer: `this` wird abgemeldet. `this` ist diese Instanz von `TetrisFrame`. Und von wem meldet er sich ab? Von sich selber. Man hätte genauer schreiben können `this.removeKeyListener(this)`.

29 Wenn die Methode `keyPressed` des `TetrisFrames` aufgerufen wird, liefert der Aufrufer (die `SpielComponent`) ein paar Informationen mit: welche Taste, genaue Zeit usw. Deshalb können wir hier entscheiden, was zu tun ist.

49-50 Da für unser Spiel nur das Drücken, nicht aber das Loslassen der Tasten einen Einfluss haben soll, können wir die letzten beiden Methoden einfach leer machen. Implementieren müssen wir sie aber, sonst regt sich der Compiler auf.

Das waren viele Details und dabei funktioniert noch nichts! Sie haben also noch eine Menge Arbeit selber zu tun. Immerhin sind Sie in der erfreulichen Situation, dass in diesem Moment alles fehlerfrei kompilierbar ist. Sorgen Sie dafür, dass es so bleibt!


SpielComponent
+ SPOT : int
- feld : Feld
+ paintComponent(g : Graphics)
Feld
- breit : int
- tief : int
- steine : Stein[]
- derstein : Stein
- angemeldet : LinkedList<FeldListener>
+ getBreite() : int
+ getTiefe() : int
+ addFeldListener(fl : FeldListener)
+ removeFeldListener(fl : FeldListener)
+ reset()
+ isBelegt(x : int, y : int) : boolean
+ fixiere()
- untersteVolle() : int
+ getStein() : Stein
+ neuerStein(stein : Stein) : Stein
+ setStein(stein : Stein) : boolean
Stein
- center : Spot
- spots : Spot[]
+ spots() : Spot[]
+ transformiert(dr : int, dx : int, dy : int) : Stein
+ toString() : String
«interface»
FeldListener
+ volleZeilen(abraum : int)
+ reset(breit : int, hoch : int)
+ verstopft(ursache : Stein)
Spot
- x : int
- y : int
+ getX() : int
+ getY() : int
+ transformiert(c : Spot, dr : int, dx : int, dy : int) : Spot
+ arrayAus(xy : int[]) : Spot[]
TetrisFrame
- score : int
- scorelabel : JLabel
- timer : Timer
- gameover : boolean
- steinmetz : Steinmetz
- feld : Feld
+ main(args : String[])
+ init()
+ actionPerformed(ae : ActionEvent)
+ keyPressed(ke : KeyEvent)
+ keyTyped(e : KeyEvent)
+ keyReleased(e : KeyEvent)
+ volleZeilen(abraum : int)
+ reset(breit : int, hoch : int)
+ verstopft(ursache : Stein)
Steinmetz
- zufall : Random
+ erzeugeStein() : Stein