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. Welche Werte diese Attribute sinnvollerweise haben können, ist einem `Spot` völlig egal. 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° gegen den Uhrzeigersinn gedreht und sodann um `dx` waagrecht und um `dy` senkrecht 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.

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 hoch)` erzeugt ein Feld gegebener Breite und Höhe. 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 getBreit()` und `int getHoch()` liefern Breite und Höhe 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()` erzeugt einen neuen Stein mit solchen Spot-Koordinaten, dass er sich oben waagrecht mittig im Feld befindet. (Für den Anfang reicht eine Steinform, später wird zufällig aus mehreren Möglichkeiten ausgewählt.) Diese Methode ruft `setStein` auf und gibt deren Erfolg zurück. Später wird sie derart erweitert, dass sie über Misserfolg Bericht erstattet – später.
- `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 ge-

nau, 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 hoch);

  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:

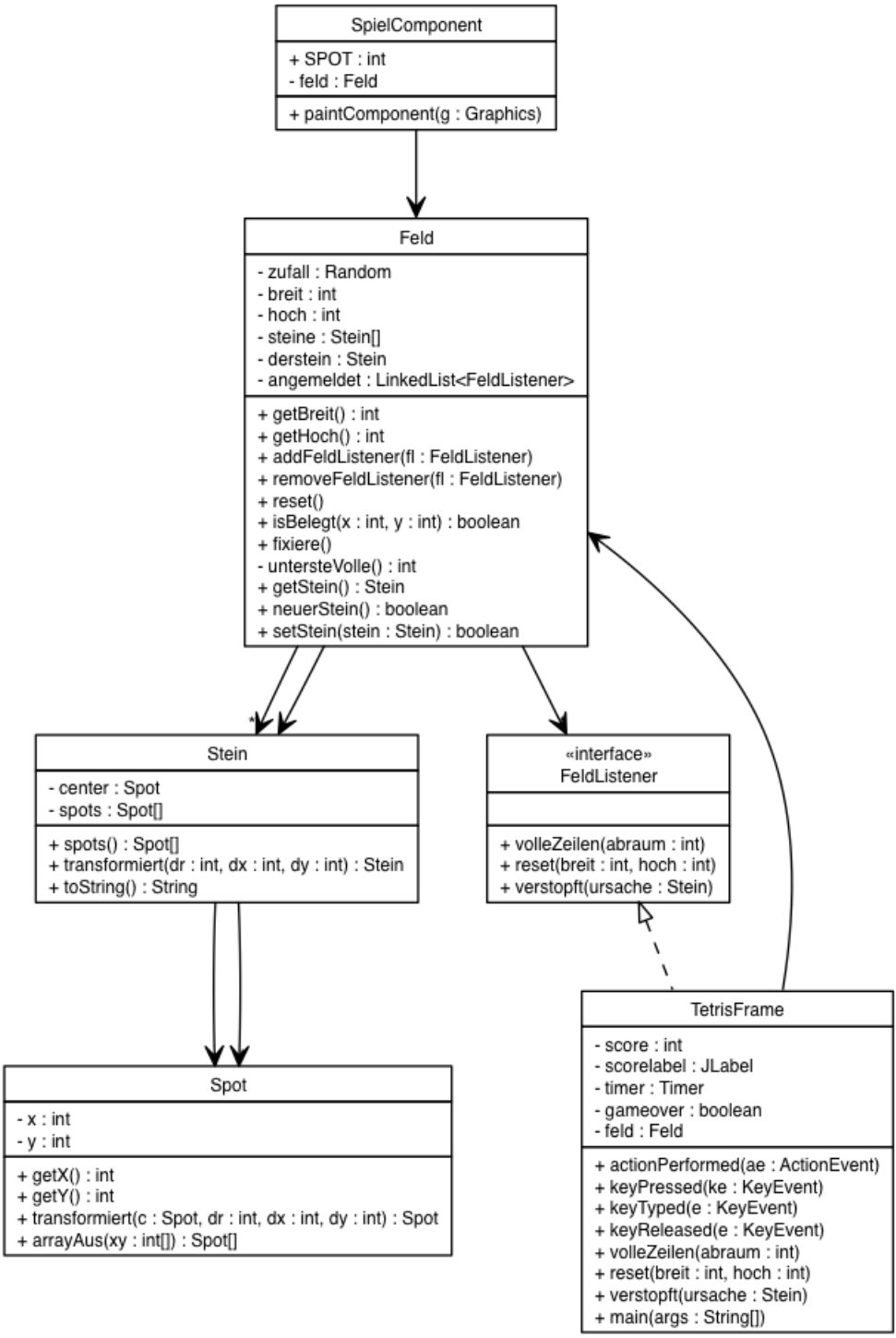
```
public class Dummy implements FeldListener{
  void volleZeilen(int abraum){
    System.out.println("Zeile mit "+abraum+" Feldern gelöscht.");
  }

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

  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`.


SpielComponent
+ SPOT : int
- feld : Feld
+ paintComponent(g : Graphics)
Feld
- zufall : Random
- breit : int
- hoch : int
- steine : Stein[]
- derstein : Stein
- angemeldet : LinkedList<FeldListener>
+ getBreit() : int
+ getHoch() : int
+ addFeldListener(fl : FeldListener)
+ removeFeldListener(fl : FeldListener)
+ reset()
+ isBelegt(x : int, y : int) : boolean
+ fixiere()
- untersteVolle() : int
+ getStein() : Stein
+ neuerStein() : boolean
+ 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
- feld : Feld
+ actionPerformed(ae : ActionEvent)
+ keyPressed(ke : KeyEvent)
+ keyTyped(e : KeyEvent)
+ keyReleased(e : KeyEvent)
+ volleZeilen(abraum : int)
+ reset(breit : int, hoch : int)
+ verstopft(ursache : Stein)
+ main(args : String[])