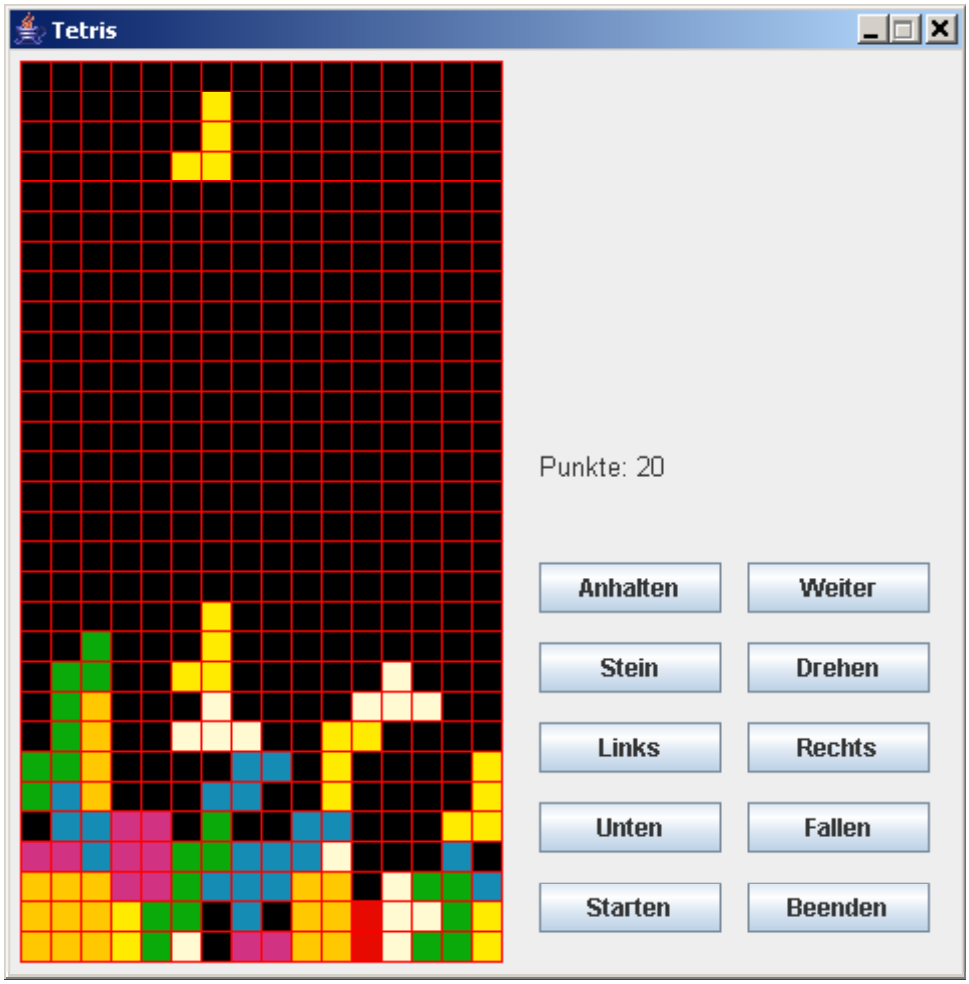
Tetris

Tetris ist ein Computerspiel, bei dem man herunterfallende, immer aus vier Teilen bestehende Bausteine so drehen muss, dass sie möglichst ohne Lücken eine Mauer bilden. Daher rührt auch der Name Tetris, welcher ein Derivat des griechischen Wortes für vier, "tetra", ist. Sobald eine Reihe von Steinen komplett ist, wird sie entfernt. Das Spiel endet, sobald sich die nicht abgebauten Bausteine bis zum oberen Spielfeldrand aufgetürmt haben. Je länger man diesen Zustand hinauszögern kann, desto mehr Punkte erhält man. Wenn eine bestimmte Zahl an Reihen entfernt wurde wird das Level und somit die Geschwindigkeit erhöht. (nach Wikipedia)



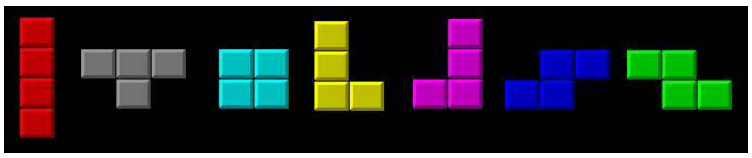
Tetris soll als Computerspiel mittels Java realisiert werden, wobei ein objektorientierter Ansatz gewählt wird.

Objektorientierte Analyse

In der objektorientierten Analyse werden die Wünsche und Anforderungen des Auftraggebers ermittelt und ein Fachkonzept entwickelt. Die Anforderungen können im Wesentlichen dem obigen Text und Bild entnommen werden. Das Spiel soll sich sowohl über Schalter als auch über die Tastatur bedienen lassen.

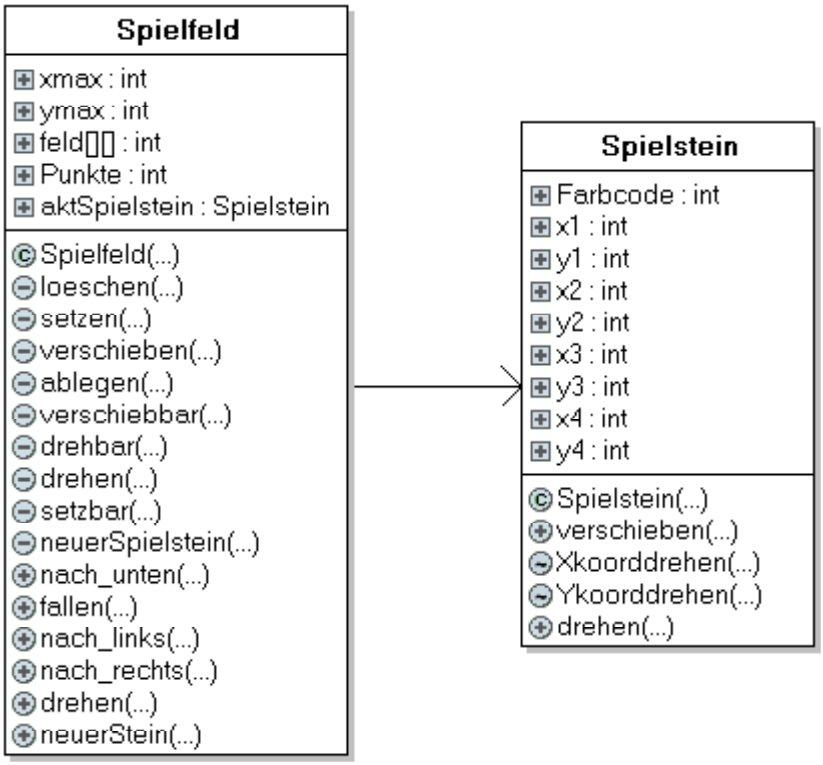
Fachkonzept

In das Fachkonzept gehen die Spielsteine und das Spielfeld ein. Jeder Spielstein besteht aus vier verbundenen Quadraten und hat eine Farbe. Eine genauere Analyse zeigt, dass es sieben verschiedene Formen von Spielsteinen gibt.



Das Spiel wird immer mit einem Spielstein gespielt. Dieser kann nach links und rechts verschoben, sowie um 90 Grad gedreht werden. Ein Taktgeber sorgt dafür, dass der Spielstein jeweils um eine Einheit nach unten verschoben wird. Kann der Spielstein nicht mehr weiter nach unten verschoben werden, so wird er im Spielfeld fixiert und ein neuer Spielstein erzeugt. Um die Lage eines Spielsteins während des Spiels beschreiben zu können, benötigt man die x- und y-Koordinaten der 4 Quadrate. Ein Spielstein kann daher wie nebenstehend beschrieben werden.

Das rechteckige Spielfeld hat die Größe 16 x 30. Mit den fixierten Spielsteine am unteren Rand kann man nicht mehr spielen, es sind demnach keine Spielsteine mehr sondern nur farbige Quadrate. Im Spielfeld müssen daher 16 x 30 farbige Quadrate verwaltet werden. Wenn eine Reihe voll wird, erhält der Spieler Punkte und die betreffende Reihe wird gelöscht. Der Punktestand wird als Attribut des Spielfelds modelliert. Zum Spielfeld gehört ein Spielstein, mit dem der Spieler aktuell spielen kann. Also besteht zwischen der Spielfeld- und der Spielstein-Klasse eine gerichtete Assoziation. Um den Spielstein auf dem Spielfeld bewegen zu können, braucht man einige Methoden: setzen, löschen, drehen, verschieben, ablegen, verschiebbar, drehbar, fallen, nach_links, nach_rechts. Daraus ergibt sich das folgende Klassendiagramm als Fachkonzept.

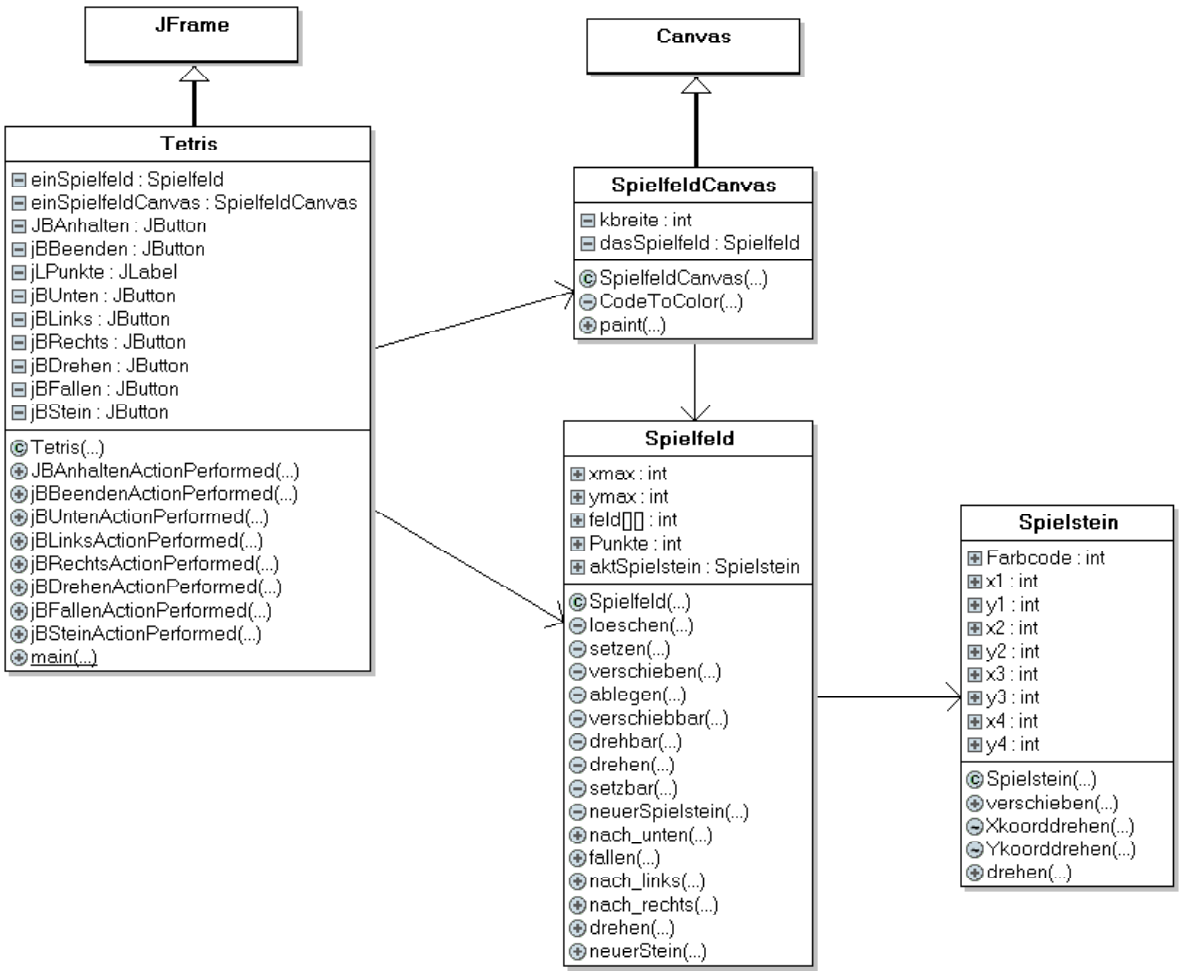


Objektorientierter Entwurf

Im objektorientierten Entwurf ergänzen wir gemäß der drei Schichten Architektur das Fachkonzept um eine Benutzungsoberfläche. Eine Datenhaltung wäre auch möglich, um beispielsweise Highscores zu verwalten. Die Benutzungsoberfläche soll wie auf Seite 1 dargestellt aussehen. Links wird das Spielfeld samt aktuellem Spielstein dargestellt, rechts wird der Punktestand angezeigt und lässt sich das Spiel schrittweise über die Schaltflächen spielen. Das Spielfeld muss aber auch über Tastatur gespielt werden können, weswegen wir folgende Tastenzuordnungen festlegen:

Taste	**Aktion**
S	neuer Spielstein
Pfeil links	Nach links verschieben
Pfeil rechts	Nach rechts verschieben
Pfeil unten	Nach unten verschieben
Pfeil oben	Um 90 Grad drehen
Leertaste	Spielstein fallen lassen
Pause	Timer anhalten
Eingabetaste	Weiter

Zur Anzeige des Spielfeldes benutzen wir eine Canvas-Komponente von der wir unsere Spielfeld-Canvas-Klasse ableiten müssen. Die Klasse Canvas enthält eine Methode paint, die wir in der SpielfeldCanvas-Klasse überschreiben. In dieser Methode erzeugen wir die Grafik für das Spielfeld. Dazu muss Zugriff auf die Spielfeld-Klasse bestehen, also haben wir eine gerichtete Assoziation zwischen SpielfeldCanvas und Spielfeld.



Das Anwendungsprogramm entwerfen wir als Swing-Programm, das daher von der Klasse JFrame abgeleitet wird. Es kennt sowohl das Spielfeld als auch die SpielfeldCanvas-Komponente.

Objektorientierte Programmierung

Zum Abschluss des objektorientierten Entwicklungsprozesses kommt die bei weitem schwierigste Aufgabe, die Implementierung des Entwurfs.

Implementierung des Spielsteins

Mit dem Konstruktor Spielstein(int i) erzeugen wir den gewünschten Spielstein:

```
public Spielstein(int i) {
  switch (i) {
    case 1:
      x1 = 7; y1 = 2;
      x2 = 7; y2 = 1;
      x3 = 7; y3 = 3;
      x4 = 8; y4 = 3;
      Farbcode = 1;
      break;
    case 2:
      x1 = 7; y1 = 2;
      x2 = 7; y2 = 1;
      x3 = 7; y3 = 3;
      x4 = 6; y4 = 3;
      Farbcode = 2;
      break;
   ...
}
```

Die Methode **public void** verschieben(int x, int y) ist ganz einfach zu realisieren. x und y sind die relativen Werte, um die verschoben wird also -1, 0 oder +1.

Am schwierigsten stellt sich die Implementierung der Methode *drehen* heraus. Ein Kästchen muss als Drehpunkt festgelegt werden, ansonsten würde ein Spielstein nicht auf ganzzahlige Koordinaten gedreht. Eine Ausnahme kann man beim 4er-Quadrat machen. Als Drehpunkt verwenden wir (x1/y1). Zum Drehen benutzen wir zwei Hilfsfunktionen, die man sich geometrisch klar machen kann.

```
int Xkoorddrehen(int y) {
  return x1 + y - y1;
}

int Ykoorddrehen(int x) {
  return y1 - x + x1;
}

public void drehen() {
  int x, y;

  x = Xkoorddrehen(y2);
  y = Ykoorddrehen(x2);
  x2 = x;
  y2 = y;
  ...
}
```

Implementierung des Spielfeldes

Das Spielfeld deklarieren wir so:

```
public int xmax = 16;
public int ymax = 30;
public int feld[][] = new int[xmax+2][ymax+2];
```

legen also außen herum einen Rahmen, um beim Prüfen, ob ein Stein verschieb- oder drehbar ist, keine Probleme mit ungültigen Feldindexen zu bekommen.

Für ein Randkästchen tragen wir -1 ein, ein unbelegtes Kästchen erhält den Wert 0. Belegte Kästchen erhalten den Farbwert des Spielsteines. Bei abgelegten Spielsteinen wird der Farbcode um 8 erhöht.

Ein Spielstein wird wie folgt verschoben:

```
private void verschieben(Spielstein einSpielstein, int x, int y) {
  loeschen(einSpielstein);
  einSpielstein.verschieben(x, y);
  setzen(einSpielstein);
}
```

Die Hilfsmethoden *loeschen* und *setzen* verändern die Werte im Feld. Vor dem Verschieben sollte man prüfen, ob der Stein überhaupt verschiebbar ist. Daher:

```
private boolean verschiebbar(Spielstein einSpielstein, int x, int y) {
  Spielstein verschobenerSpielstein = einSpielstein.Kopie();
  verschobenerSpielstein.verschieben(x, y);
  return setzbar(verschobenerSpielstein);
}
```

Wir brauchen also in der Klasse Spielstein noch eine Funktion, um eine Spielstein-Kopie herzustellen. Diese wird dann verschoben und dann wird geprüft, ob der verschobene Spielstein gesetzt werden kann.

```
private boolean setzbar(Spielstein s) {
  int f = aktSpielstein.Farbcode;
  return (feld[s.x1][s.y1] == f || feld[s.x1][s.y1] == 0) &&
         (feld[s.x2][s.y2] == f || feld[s.x2][s.y2] == 0) &&
         (feld[s.x3][s.y3] == f || feld[s.x3][s.y3] == 0) &&
         (feld[s.x4][s.y4] == f || feld[s.x4][s.y4] == 0);
}
```

Analog dazu geht man beim Drehen vor. Beim Ablegen eines Steins erhöht man seinen Farbcode. Dadurch wird obige Implementierung von setzbar gegen gleichfarbige, schon fixierte Kästchen abgesichert.

```
private void ablegen(Spielstein einSpielstein) {
  einSpielstein.Farbcode = einSpielstein.Farbcode + 8;
  setzen(einSpielstein);
}
```

Als Schnittstelle zum Hauptprogramm benötigt man noch Methoden für die möglichen Benutzeraktionen, zum Beispiel ein Schritt *nach_unten*

```
public void nach_unten() {
  if (verschiebbar(aktSpielstein, 0, 1)) {
    verschieben(aktSpielstein, 0, 1);
  }
}
```

Implementierung des SpielfeldCanvas

Zur Darstellung des Spielfeldes platzieren wir eine Canvas-Komponente auf dem GUI-Formular. Diese erhält die Breite xmax * kbreite + 1 = 16 * 15 + 1 = 241 und die Höhe ymax * kbreite + 1 = 30 * 15 + 1 = 451, wobei kbreite die Abkürzung für Kästchenbreite ist. Nachdem die Canvas-Komponente konfiguriert ist, ersetzt man im Quelltext der GUI-Klasse den Datentyp Canvas durch SpielfeldCanvas.

Der Konstruktor der SpielfeldCanvas-Klasse erhält als Parameter das Spielfeld, das dargestellt werden soll:

```
public SpielfeldCanvas(Spielfeld einSpielfeld) {
  dasSpielfeld = einSpielfeld;
}
```

Etwas diffizil ist die Darstellung des Spielfeldes, weil es hierbei auf jedes Pixel ankommt. Die Kästchenbreite ist zwar 15, aber die eigentlichen Kästchen werden nur mit 14 Pixel Breite dargestellt, damit die Gitterlinien Platz haben.

```
public void paint(Graphics g) {
  g.setColor(Color.red);
  for (int i = 0; i <= dasSpielfeld.xmax; i++) {
    g.drawLine(i * kbreite, 0, i * kbreite, dasSpielfeld.ymax * kbreite);
  }
  for (int j = 0; j <= dasSpielfeld.ymax; j++) {
    g.drawLine(0, j * kbreite, dasSpielfeld.xmax * kbreite, j * kbreite);
  }

  for (int i = 0; i < dasSpielfeld.xmax; i++ ) {
    for (int j = 0; j < dasSpielfeld.ymax; j++ ) {
      Color Farbe = CodeToColor(dasSpielfeld.feld[i + 1][j + 1]);
      g.setColor(Farbe);
      zeichneKaestchen(g, i, j);
    }
  }
}

private void zeichneKaestchen(Graphics g, int x, int y) {
  g.fillRect(x * kbreite + 1, y * kbreite + 1, kbreite - 1, kbreite - 1);
}
```

Im Spielfeld ist nur ein Farbcode gespeichert. Dieser muss in eine Farbe umgesetzt werden. Dazu verwenden wir diese Methode:

```
private Color CodeToColor(int Farbcode) {
  Color Farbe;
  if (Farbcode > 7) {
    Farbcode = Farbcode - 8;
  }
  switch (Farbcode) {
    case 0: Farbe = Color.black;
```

```
        break;
      case 1: Farbe = new Color(230, 10, 0);
        break;
      case 2: Farbe = new Color(255, 235, 0);
        break;
      case 3: Farbe = new Color(10, 170, 10);
        break;
      case 4: Farbe = new Color(20, 140, 180);
        break;
      case 5: Farbe = new Color(255, 250, 210);
        break;
      case 6: Farbe = new Color(210, 50, 130);
        break;
      case 7: Farbe = Color.orange;
        break;
      default:
        Farbe = Color.red;
    }
   return Farbe;
  }
```

Implementierung des Hauptprogramms

Im Hauptprogramm werden unter anderem ein Spielfeld und ein SpielfeldCanvas erzeugt.

```
// Anfang Variablen
private Spielfeld einSpielfeld = new Spielfeld();
private SpielfeldCanvas einSpielfeldCanvas =
            new SpielfeldCanvas(einSpielfeld);
```

In den Ereignismethoden für die Schaltflächen werden die entsprechenden Methoden des Spielfeldes aufgerufen und anschließend mit *repaint* dafür gesorgt, dass das Spielfeld neu dargestellt wird.

```
public void jBUntenActionPerformed(ActionEvent evt) {
  einSpielfeld.nach_unten();
  einSpielfeldCanvas.repaint();
}
```

Optimierung der Animation

Damit ist eine Grundversion des Programms realisiert. Im Weiteren geht es um die Verbesserung des Programms. Beim Spielen fällt auf, dass die Anzeige flackert. Das hängt damit zusammen, dass bei jeder kleinen Bewegung des Spielsteins das Spielfeld komplett dargestellt wird. Das Flackern kann dadurch beseitigt werden, dass man nicht das ganze Spielfeld neu zeichnet, sondern nur die Kästchen, die sich ändern. Da bei einer Bewegung zuerst ein Stein gelöscht und dann an der neuen Position wird gezeichnet wird, ist es sinnvoll, wenn das Löschen und Neu zeichnen innerhalb der entsprechenden Spielfeldmethoden ausgelöst wird. Dazu benötigt die Spielfeldklasse Kenntnis vom SpielfeldCanvas. Aus der gerichteten Assoziation wird also eine bidirektionale:

```
public SpielfeldCanvas(Spielfeld einSpielfeld) {
  dasSpielfeld = einSpielfeld;
  dasSpielfeld.einSpielfeldCanvas = this;
}
```

Zum Zeichnen eines Kästchens erhält die SpielfeldCanvas-Klasse diese Methode:

```
public void zeichneSpielstein(Spielstein einSpielstein, int Farbcode) {
  Graphics g = getGraphics();
  g.setColor(CodeToColor(Farbcode));
```

```
    zeichneKaestchen(g, einSpielstein.x1 - 1, einSpielstein.y1 - 1);
    zeichneKaestchen(g, einSpielstein.x2 - 1, einSpielstein.y2 - 1);
    zeichneKaestchen(g, einSpielstein.x3 - 1, einSpielstein.y3 - 1);
    zeichneKaestchen(g, einSpielstein.x4 - 1, einSpielstein.y4 - 1);
  }
```

Die Löschen- und Setzen-Methode des Spielfeldes rufen diese Zeichenmethode auf, z. B.:

```
  private void loeschen(Spielstein einSpielstein) {
    feld[einSpielstein.x1][einSpielstein.y1] = 0;
    feld[einSpielstein.x2][einSpielstein.y2] = 0;
    feld[einSpielstein.x3][einSpielstein.y3] = 0;
    feld[einSpielstein.x4][einSpielstein.y4] = 0;
    einSpielfeldCanvas.zeichneSpielstein(einSpielstein, 0);
  }
```

Damit ist das Flackern behoben.

Implementierung der Tastatursteuerung

Die Tastatur reagiert normalerweise nur auf Komponenten, die für eine Betätigung durch Tastatur ausgelegt sind, also z. B. Textfeld oder Schaltfläche, aber nicht Label oder Canvas. Damit eine Canvas-Komponente auf Tastatureingaben reagiert, muss sie sich in die Liste derjenigen Objekte einreihen, die auf Tastaturereignisse reagieren und ein Objekt bereitstellen, an das die Tastaturereignisse gesendet werden.

```
    einSpielfeldCanvas.addKeyListener(new KeyAdapter() {
      public void keyPressed(KeyEvent event) {
        int key = event.getKeyCode();
        if (key == KeyEvent.VK_S)     einSpielfeld.neuerStein();
        if (key == KeyEvent.VK_LEFT)  einSpielfeld.nach_links();
        if (key == KeyEvent.VK_RIGHT) einSpielfeld.nach_rechts();
        if (key == KeyEvent.VK_DOWN)  einSpielfeld.nach_unten();
        if (key == KeyEvent.VK_UP)    einSpielfeld.drehen();
        if (key == KeyEvent.VK_SPACE) einSpielfeld.fallen();
      }
    });
```

Damit die SpielfeldCanvas-Komponente nach dem Programmstart den Eingabefocus erhält ist folgendes nötig:

```
    // Ende Komponenten
    setResizable(false);
    setVisible(true);
    einSpielfeldCanvas.requestFocusInWindow();  // richtige Position
```

Wird wechselweise mit Tastatur und Maus gespielt, so muss nach jedem Klick der Eingabefocus wieder auf die SpielfeldCanvas-Komponente gesetzt werden:

```
  public void jBUntenActionPerformed(ActionEvent evt) {
    einSpielfeld.nach_unten();
    einSpielfeldCanvas.requestFocusInWindow();
  }
```

Steuerung durch einen Timer

Die Java-API kennt drei verschiedene Timer-Klassen. Wir benutzen den **javax.swing.Timer.** Er wird mit

```
private int Tempo = 500;
private Timer einTimer;
einTimer = new Timer(Tempo, Fallen);
```

gestartet, wobei Tempo die Taktrate in Millisekunden ist und *Fallen* ein sogenannter ActionListener, also ein Objekt, das auf ein Ereignis reagiert. Der Timer löst dieses Ereignis immer nach der festgelegten Zeitspanne aus und ruft dann dessen Methode actionPerformed auf:

```
ActionListener Fallen = new ActionListener () {
  public void actionPerformed(ActionEvent evt){
    einSpielfeld.nach_unten();
    jLPunkte.setText("Punkte: " + einSpielfeld.Punkte);
  }
};
```

Das heißt, dass nach jeweils 500 Millisekunden der Spielstein um eine Einheit nach unten verschoben wird.

In die Tastatursteuerung wird der Timer wie folgt integriert:

```
if (key == KeyEvent.VK_PAUSE) einTimer.stop();
if (key == KeyEvent.VK_ENTER) einTimer.start();
```

Beim Starten wird der Time wie folgt aktiviert:

```
public void jBStartenActionPerformed(ActionEvent evt) {
  einSpielfeld.neuerStein();
  einSpielfeldCanvas.requestFocusInWindow();
  einTimer.start();
}
```