Ergänzungen zum Buch

Dietmar Ratz, Jens Scheffler, Detlef Seese, Jan Wiesenberger

Grundkurs Programmieren in Java

Band 2: Einführung in die Programmierung kommerzieller Systeme

Inhaltsverzeichnis

Ergänzung 1

Zum Aufwärmen

„Phantasie ist wichtiger als Wissen, denn Wissen ist begrenzt.“ Dieses Zitat stammt von niemand Geringerem als Albert Einstein. Wer wären die Autoren dieses Bandes, dem zu widersprechen?
Tasächlich mag es etwas seltsam anmuten, das Kapitel eines Lehrbuchs mit einem solchen Satz zu beginnen. Schließlich sind die Autoren ja angetreten, Ihnen Wissen zu vermitteln. In diesem Buch werden Sie mit einer Vielzahl neuer Techniken und Bereiche sowohl aus der allgemeinen Welt des objektorientierten Programmierens als auch konkret aus der Programmiersprache Java konfrontiert. Sie erwerben also neues Wissen.
Bevor wir jedoch diesen Schritt wagen, sollten wir uns erst bewusst machen, dass wir bereits eine Menge gelernt haben. Wenn Sie den ersten Band und seine Übungsaufgaben intensiv durchgearbeitet haben, dann kennen Sie die Sprache Java selbst wie Ihre Westentasche. Sie kennen sich mit objektorientierten Strukturen aus und haben gelernt, auch an komplex erscheinende Problemstellungen analytisch heranzugehen und eine Lösung zu finden. Sie haben ein stabiles Fundament gelegt, auf dem wir nun mit diesem Band weiter aufbauen werden.
Betrachten wir einmal das in Abbildung 1.1 beschriebene Springer-Problem:[1]

[1] Dieses Schachproblem taucht übrigens in ähnlicher Form im Computerspiele-Klassiker *The 11th hour* auf, aus dem wir es auch übernommen haben.

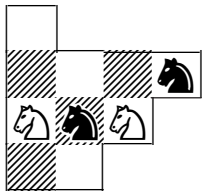

Abbildung 1.1: Das Springer-Problem

Auf einem beschnittenen Schachbrett stehen vier Springer, jeweils zwei der Farben Schwarz und Weiß. Ziel der Aufgabe ist es, dass die Pferde ihre Positionen tauschen, das heißt, an Stelle der schwarzen Springer sollen weiße stehen und umgekehrt. Die Springer dürfen die beim Schach üblichen Züge machen. Sie dürfen hierbei nicht das Feld verlassen. Ferner wird bekannt gegeben, dass das Problem in unter 40 Zügen lösbar ist.[2]
Wie Sie sicher sehen, handelt es sich hierbei nicht gerade um ein triviales Problem. Setzen Sie sich einmal an ein Schachbrett und versuchen Sie, die Lösung zu finden. Am Ende dieses Kapitels werden wir unseren Computer so programmiert haben, dass er uns die Lösung dieses Problemes berechnet. Wir werden keine anderen Mittel verwenden als das Wissen, das wir uns im ersten Band angeeignet haben. Abgesehen natürlich von unserer Phantasie ...

1.1 Schach dem Computer!

1.1.1 Aufgabenstellung

Wir wollen in diesem Abschnitt ein virtuelles Schachbrett programmieren. Auf dem Schachbrett sollen sich die bei diesem Spiel üblichen Figuren befinden: Bauer, Turm, Springer, Läufer, Dame und König. Der Benutzer soll in der Lage sein, diese Figuren auf dem Brett zu bewegen. Er soll dies aber nur im Rahmen der üblichen Zugregeln tun dürfen. Der Einfachheit halber nehmen wir Besonderheiten wie etwa die Rochade explizit aus.
Zu Anfang soll das Brett wie in Abbildung 1.2 beschrieben aufgebaut sein. Der Benutzer soll jederzeit in der Lage sein, das Brett wieder in diesen Ursprungszustand zurückzuversetzen. Weitere Funktionalitäten (Anzeige von „Schach“ bzw. „Matt“ oder Überwachung der Zugreihenfolge) sollen nicht realisiert werden.

1.1.2 Designphase

In der Design- oder Modellierungsphase werden wir nun die zu realisierende Aufgabe analysieren und versuchen, die gegebene Situation in einem Mo-

[2] Wir werden später sehen, dass die optimale Lösung tatsächlich 34 Züge benötigt

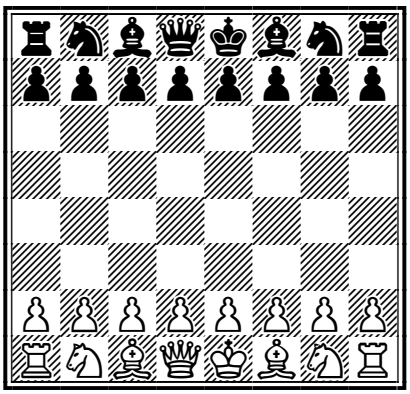

Abbildung 1.2: Ausgangssituation beim Schachspiel

dell von Klassen widerzuspiegeln. Wir benutzen zu diesem Zweck die UML-Klassendiagramme, in denen wir unsere wichtigsten Klassen und deren Zusammenhänge skizzieren. Von diesem Modell ausgehend, werden wir uns an eine konkrete Implementierung wagen.
Beginnen wir mit dem Entwurf des Schachbretts, dargestellt in Abbildung 1.3. Ein Schachbrett setzt sich zusammen aus dem Brett an sich und den Figuren, die sich auf ihm befinden. Um diesen Aspekt auf dem Computer nachzubilden, sehen wir zwei Klassen vor:

- Ein Klasse `Chessman`[3] repräsentiert auf dem Computer eine konkrete Art von Schachfigur (z. B. schwarze Dame, weisser Turm). Wir legen die Schachfigur hier als ein Interface an, das heißt wir geben keine konkrete Implementierung vor. Für jeden Typ von Schachfigur kann es eine eigene Klasse geben.

  Wir schreiben in unserem Interface hierbei vier verschiedene Methoden vor:

  1. Die Methode `toChar` liefert eine Darstellung der Figur in Form eines einzelnen Zeichens. Dies kann für eine Dame beispielsweise der Buchstabe Q (für die englische Bezeichnung Queen) sein.
  2. Die Methode `canTake` überprüft, ob diese Figur eine andere Figur zu schlagen in der Lage ist. Beim Schach ist dies immer der Fall, wenn die andere Figur der gegnerischen Farbe angehört.[4]
  3. Die Methode `equals` vergleicht zwei Chessman-Objekte auf Gleichheit. Diese Methode ist eigentlich schon in der allgemeinen Oberklasse aller

[3] Hinweis: Der Name Chessman ist unter Schachkundigen ein wenig umstritten. Normalerweise werden Sie stattdessen die Bezeichnung *Chesspiece* lesen, welche gebräuchlicher ist. Offiziell stellt ein Chesspiece jedoch lediglich eine der „Haupt-“Figuren, also keine Bauern, dar. Wir wählen aus diesem Grunde hier die präzise, wenn auch unübliche Übersetzung.

[4] Noch einmal zur Verdeutlichung: unser `Chessman` stellt keine Figur dar, die sich konkret auf dem Brett befindet. Sie speichert keine Koordinaten, sondern modelliert lediglich die allgemeinen Charakteristika der Figur. Daher ist obige Bedingung im Allgemeinen vollkommen ausreichend.

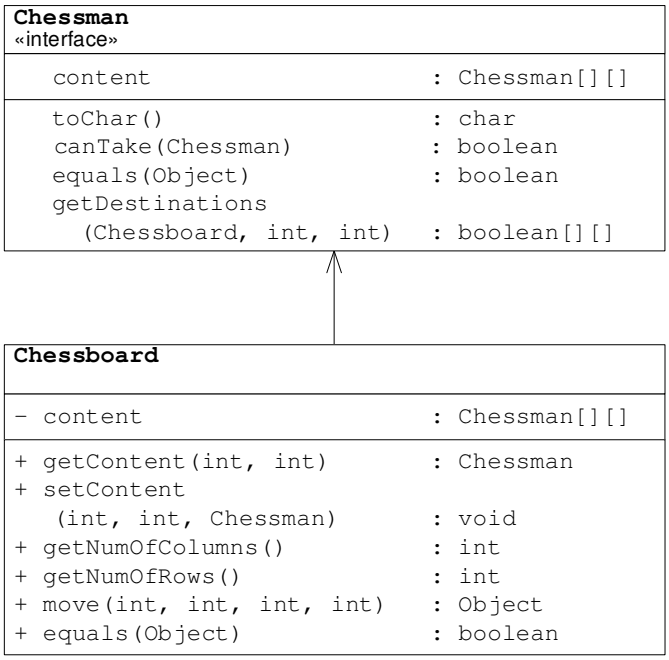

Abbildung 1.3: Der Aufbau von Schachbrett und -figur

Klassen, `java.lang.Object`, vordefiniert. Wir führen sie hier aber noch einmal explizit auf um zu erinnern, dass wir die Methode überschreiben müssen.

4. Die Methode `getDestinations` berechnet anhand einer konkreten Konstellation auf dem Schachbrett (das `Chessboard`) alle Positionen, auf die sich die Figur von einer übergebenen Position (die beiden `int`-Zahlen) aus bewegen könnte. Das zurückgegebene Feld von Booleschen Werten besagt, ob ein Feld mit gewissen Koordinaten betreten werden kann.

- Unser eigentliches Schachbrett repräsentieren wir durch eine Klasse namens `Chessboard`. Die Klasse beinhaltet ein zweidimensionales Feld (`content`), in dem wir die aktuelle Stellung speichern. Das Feld vereinbaren wir mit Sichtbarkeit `protected`, sodass Subklassen Zugriff auf die Instanzvariable besitzen. Verweist ein Feldeintrag auf `null`, so befindet sich auf diesem Feld keine Figur. Ist das Feld besetzt, speichern wir an der entsprechenden Stelle die Art der Figur ab, die auf das Brett gesetzt wurde (also den `Chessman`). Wir sehen ferner folgende Methoden vor:

  - Mit Hilfe von get- und set-Methoden können wir den Inhalt des Feldes manipulieren. Wir führen diese Methoden aus Gründen der Datenkapselung ein. Weitere Informationen zum Thema Datenkapselung finden Sie

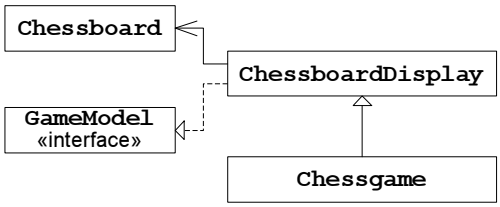

Abbildung 1.4: Die Schnittstelle zum Benutzer

im ersten Band dieses Kurses.

- Wir gehen davon aus, dass unser Schachbrett immer rechteckig ist. Wir gehen aber nicht unbedingt davon aus, dass es (wie beim Schach) tatsächlich acht mal acht Felder besitzt. Die Breite und Höhe des Schachbretts (ausgedrückt in der Anzahl der Zeilen und Spalten) lässt sich daher in den Methoden `getNumOfColumns` und `getNumOfRows` erfragen.
- Um Figuren auf dem Brett bewegen zu können, definieren wir eine Methode namens `move`. Dieser Methode übergeben wir die Start- und Endkoordinaten unseres Zuges. Die Methode soll nun überprüfen, ob sich auf dem Startfeld eine Figur befindet, und ob das Zielfeld entweder leer ist, oder aber eine schlagbare Figur enthält. Anschließend soll der Zug durchgeführt werden. Wird eine Figur vom Feld genommen, so drückt sich dies durch den Rückgabewert der Methode aus.

Nachdem wir nun das eigentliche Schachbrett entworfen haben, müssen wir uns nur noch Gedanken über die Schnittstelle zum Benutzer machen. Die Grundidee hierzu ist in Abbildung 1.4 skizziert. Aus dem ersten Band kennen wir das Interface `GameModel`,[5] mit dessen Hilfe wir auf einfachem Wege grafische Oberflächen für brettartige Spiele definieren können. Wir werden diesen Mechanismus erneut verwenden und definieren eine Klasse namens `ChessboardDisplay`, die eben dieses Interface implementiert. Das `ChessboardDisplay` hat eine Assoziation zu einem `Chessboard`, das heißt es wird eine Instanzvariable von diesem Typ besitzen. Eben dieses Brett wird das `GameModel` dann auf dem Bildschirm darstellen.
Da abzusehen ist, dass wir im Laufe dieses Kapitels mehr als eine Darstellung von Schachbrettern brauchen, beinhaltet unser `ChessboardDisplay` selbst keine Logik bezüglich der Ablaufsteuerung unseres Schachbretts. Diese konkrete Logik verpacken wir in einer Klasse `Chessgame`, die wir vom `ChessboardDisplay` ableiten. Diese Subklasse wird „wissen“, wie sie mit dem Spieler zu interagieren hat.

[5] Vorsicht: Hierbei handelt es sich um keine mit Java mitgelieferte Standardklasse. Die Klassen `GameModel` und die dazugehörige `GameEngine` sind vielmehr Bestandteil einer Sammlung von Hilfsklassen aus dem ersten Band – den so genannten **`Prog1Tools`**. Sie können diese Klassen von der Homepage zum Buch herunterladen.

1.1.3 Implementierung

1.1.3.1 Das Schachbrett an sich

Wir wollen uns nun daran machen, unseren Entwurf in konkreten Java-Code umzusetzen. Wir beginnen mit der Schachbrett/Schachfiguren-Konstellation und realisieren das Interface `Chessman`:

```
1  package de.uni.karlsruhe.aifb.prog2.chess;
2
3  /** Dieses Interface repraesentiert eine beliebige Figur,
4    * die auf ein Schachbrett gestellt werden kann. Das Interface
5    * beinhaltet hierbei keine Positionsangaben auf dem Brett, d.h.
6    * Instanzen dieses Interfaces koennen auch mehrmals auf demselben
7    * Brett bzw. auf mehreren Brettern zugleich verwendet werden. Sie
8    * repraesentieren somit lediglich den Typ einer Schachfigur
9    * (also Laeufer, Springer usw).
10   */
11 public interface Chessman {
12
13   /** Liefert ein einzelnes Zeichen, das den Figurentyp
14     * repraesentiert. Diese Methode kann fuer einfache textuelle
15     * Darstellungen verwendet werden.
16     **/
17   public char toChar();
18
19   /** Testet, ob diese Figur eine andere Figur theoretisch zu schlagen
20     * in der Lage ist.
21     * @param die Schachfigur, die es zu ueberpruefen gilt
22     * @return true genau dann, wenn die Figur geschlagen werden kann
23     **/
24   public boolean canTake(Chessman chessman);
25
26   /** Ueberprueft, auf welche Felder sich die Figur im naechsten Zug
27     * theoretisch bewegen kann.
28     * @return ein Boolesches Feld in der Dimension des Schachbretts.
29     * Ein Eintrag ist genau dann true, wenn die Figur auf dieses Feld
30     * ziehen darf.
31     * @param board das Schachbrett, auf dem agiert wird
32     * @param row die Nummer der Zeile, in der sich die Figur befindet.
33     *   Die Methode muss nicht ueberpruefen, ob unter der angegebenen
34     *   Position wirklich eine entsprechende Figur steht. Die Nummer
35     *   wird von 0 an gezaehlt
36     * @param col die Nummer der Spalte, in der sich die Figur
37     *   befindet.
38     *   Die Methode muss nicht ueberpruefen, ob unter der angegebenen
39     *   Position wirklich eine entsprechende Figur steht. Die Nummer
40     *   wird von 0 an gezaehlt
41     **/
42   public boolean[][] getDestinations
43     (Chessboard board,int row,int col);
44
45   /** Ueberprueft, ob es sich bei zwei Schachfiguren um dieselbe Figur
46    *  handelt.
47    *  @return true genau dann, wenn es sich um dieselbe Figur handelt
```

```
48      */
49     public boolean equals(Object o);
50
51 }
```

Ihnen werden an diesem Quelltext wahrscheinlich zwei Dinge aufgefallen sein:

- Die Klasse befindet sich nicht (wie im ersten Teil üblich) im Standard-Package. Wir haben stattdessen mit Hilfe der **`package`**-Anweisung eine Paketzugehörigkeit definiert. Diese kleine aber feine Neuerung werden wir nun, da wir uns im Kurs für Fortgeschrittene befinden, einführen. Auf diese Art und Weise verursachen unsere Klassennamen keine Konflikte zu anderen Programmen, die von sonstigen Herstellern und Softwareentwicklern verbreitet werden.
- Der Quelltext besteht aus mehr Kommentar als aus sonstigen Java-Konstrukten. Dies ist ein weiterer Punkt, den wir im kommerziellen Programmieren noch stärker betonen werden. Wenn wir eine neue Klasse definieren, insbesondere wenn es sich um eine nach aussen sichtbare Schnittstelle handelt, müssen wir immer sehr genau angeben, was wir uns bei der Erstellung dieser Klasse gedacht haben. Wir müssen dokumentieren, wozu unser Programm gut ist und wie man es anwendet. Dies hat zum einen den Grund, dass vielleicht einmal ein Anderer unsere Programme weiterverwenden oder sogar erweitern muss. Andererseits gibt es auch für einen selbst kaum etwas Mühsameres, als Code nachzuvollziehen, den man vor einem oder zwei Jahren geschrieben hat. Detailliertes Dokumentieren spart somit Arbeit, Nerven und Geld.

Zurück aber zu unserem Schachbrett. Wie im Design vorgesehen, definieren wir unsere Klasse `Chessboard` und versehen sie mit einem Feld von Schachfiguren:

```
package de.uni.karlsruhe.aifb.prog2.chess;

/** Diese Klasse repraesentiert ein Schachbrett, auf dem sich Figuren
  * befinden. Diese Figuren duerfen durch entsprechende Befehle
  * bewegt werden.
  **/
public class Chessboard {

  /** In diesem Array wird die aktuelle Belegung des Schachbretts
    * gespeichert.
    **/
  protected Chessman[][] content;
```

Um unser Schachbrett erzeugen zu können, definieren wir ferner einen Konstruktor:

```
  /** Konstruktor
    * @param rows die Anzahl der Zeilen
    * @param cols die Anzahl der Spalten
    **/
  public Chessboard(int rows,int cols) {
    content = new Chessman[rows][cols];
  }
```

Beachten Sie, dass wir den Konstruktor im UML-Diagramm nicht vorgesehen hatten. Tatsächlich werden wir im Laufe der Implementierung immer wieder Methoden einfügen, die sich aus den Diagrammen nicht ergeben. Hierbei handelt es sich lediglich um Hilfsmethoden, die wir intern zur Realisierung anderer wichtiger (und auch geplanter) Bestandteile benötigen. Diese Dinge haben wir in der Design-Phase bewusst ausser Acht gelassen, da wir uns dort ja vor allem auf das prinzipielle Konzept und das Zusammenspiel der einzelnen Bestandteile konzentriert haben. Dasselbe gilt beispielsweise auch für folgende Hilfsmethode, die überprüft, ob sich zwei gegebene Koordinaten noch innerhalb des gültigen Bereiches (im Falles des Standard-Schachbretts also zwischen 0 und 7) befinden:

```
public boolean isValid(int row,int col) {
  return row >=0 && col >= 0 &&
         row < content.length && col < content[0].length;
}
```

Kommen wir nun aber wieder zu den Methoden, die Sie auch aus dem UML-Diagramm kennen. Da wären zum einen die get- und set-Methoden zum Zugriff auf die private Instanzvariable. Ebenso wie die Methoden `getNumOfRows` und `getNumOfColumns` sind sie mit nur wenigen Zeilen programmiert:

```
/** Gibt die Anzahl der Spalten des Schachbretts zurueck */
public int getNumOfColumns() {
  return content[0].length;
}

/** Gibt die Anzahl der Zeilen des Schachbretts zurueck */
public int getNumOfRows() {
  return content.length;
}

/** Gibt den Inhalt des Bretts an einer gegebenen Position an.
  * @param row der Zeilenindex, gezaehlt ab Index 0
  * @param col der Spaltenindex, gezaehlt ab Index 0
  * @return den Inhalt des Bretts an der gegebenen Position,
  *   null, wenn dort keine Figur steht
  **/
public Chessman getContent(int row, int col) {
  return content[row][col];
}

/** Besetzt ein Feld mit einer Spielfigur. Diese Methode ueberprueft
  * nicht, ob der Zug gemaess den Schachregeln erlaubt ist.
  * @param chessman die zu setzende Spielfigur oder null
  * @param row der Zeilenindex, gezaehlt ab Index 0
  * @param col der Spaltenindex, gezaehlt ab Index 0
  * @return den Inhalt, der zuvor auf diesem Feld war.
  **/
public Chessman setContent(int row,int col,Chessman chessman) {
  Chessman result = content[row][col];
  content[row][col] = chessman;
  return result;
}
```

Es verbleibt also nur noch die `equals`-Methode, um die Anforderungen des Interface zu erfüllen. Zwei Schachbretter gelten als gleich, wenn sie dieselben Dimensionen haben und ferner mit den gleichen Figuren belegt sind:

```
    public boolean equals(Object o) {
      if (o instanceof Chessboard) {
        Chessboard board = (Chessboard) o;
        // Vergleiche die Dimensionen der Felder
        if (board.content.length != content.length ||
            board.content[0].length != content[0].length)
        {
          return false;
        }
        // Ueberpruefe die Feldinhalte
        for (int i = 0; i < content.length; i++) {
          for (int j = 0; j < board.content[i].length; j++) {
            Chessman man1 = content[i][j];
            Chessman man2 = board.content[i][j];
            if (man1 == null && man2 == null)
              continue;
            if (man1 != null && man2 != null && man1.equals(man2))
              continue;
            return false;
          }
        }
        return true;
      }
      return false;
    }
```

Da man aber, wenn man die Methode `equals` überschreibt, auch die Methode `hashCode` neu definieren muss, dürfen wir in dieser Klasse eine letzte weitere Methode definieren. Zwei Schachbretter, deren `equals`-Methode den Wert `true` zurückgibt, müssen ebenfalls denselben Rückgabewert in der `hashCode`-Methode haben. Wir verwenden den Inhalt des Schachbretts, um dies zu gewährleisten:

```
     *  equals-Methode ueberschreibt, muss auch immer die
     *  hashCode-Methode anpassen!
     */
    public int hashCode() {
      int res = 0;
      for (int i = 0; i < content.length; i++)
        for (int j = 0; j < content[i].length; j++)
          if (content[i][j] != null)
            res += content[i][j].hashCode();
      return res;
    }
  }
```

Damit sind die Klassen `Chessman` und `Chessboard` vollständig definiert. Machen Sie sich bei dieser Gelegenheit bewusst, dass wir erst beide Klassen programmieren mussten, bevor wir den Compiler das erste Mal starten können. Die Klasse `Chessboard` kann ohne das Interface `Chessman` nicht existieren, da es

in der Variable `content` auf ein Feld dieser Objekte verweist. Umgekehrt verlangt `Chessman` in seiner Methode `getDestinations` jedoch ein Schachbrett zur Zugberechnung.
Nun, da wir aber beide Klassen definiert haben, können wir uns endlich an den Entwurf der konkreten Schachfiguren machen.

1.1.3.2 Die Klasse `AbstractChessman`

Im nächsten Abschnitt werden wir die ersten konkreten Schachfiguren implementieren - die Figur des Bauern (englisch Pawn) und die des Springers oder Pferdes (englisch Knight). Bevor wir uns jedoch diesen Figuren widmen, können wir uns für die Zukunft noch eine Menge Arbeit ersparen.
Was haben alle Schachfiguren, die wir im Folgenden implementieren, eigentlich gemeinsam? Jede Schachfigur besitzt eine Farbe (schwarz oder weiß). Sie kann eine andere Figur genau dann schlagen, wenn diese eine andere Farbe hat. Sie liefert bei der `toChar`-Method ein einzelnes Zeichen als Ergebnis zurück. Dieses Zeichen ist im Schachspiel eindeutig, d. h. zwei Figuren sind genau dann vom gleichen Typ, wenn sie dasselbe Zeichen haben.
All diese Dinge sind für die Figuren gleich. Wir müssen sie also nicht immer wieder aufs Neue implementieren. Wir nutzen das Mittel der **Generalisierung** und erzeugen eine Oberklasse, die besagte Bedingungen erfüllt. Um zu verdeutlichen, dass diese Klasse keine konkrete Schachfigur realisiert, machen wir die Klasse abstrakt:

```
package de.uni.karlsruhe.aifb.prog2.chess;

/** Die abstrakte Implementierung der Schachfigur fuegt dem
  * urspruenglichen Interface eine Farbe (Schwarz oder Weiss)
  * hinzu und implementiert die canTake und toChar-Methoden.
  */
public abstract class AbstractChessman implements Chessman {
```

Innerhalb der Klasse definieren wir nun zwei Instanzvariablen. Die Variable `description` speichert einen Großbuchstaben, der die Figur beschreibt (beispielsweise „Q“ für Queen, also die Dame). Die Variable `isBlack` wiederum besagt, ob es sich bei unserer Figur um eine schwarze oder eine weisse Figur handelt. Beide Werte wollen wir im Konstruktor unserer Figur setzen:

```
  /** Diese Zeichen wird zur Darstellung der Figur verwendet.
    **/
  private char description;

  /** Dieser Boolesche Wert besagt, ob es sich um eine schwarze Figur
    * handelt.
    **/
  private boolean isBlack;

  /** Konstruktor.
    * @param isBlack ist true genau dann, wenn die Figur schwarz ist.
    * @param description eine Darstellung der Figur als Zeichen.
```

```
  *   Dieser Wert muss ein Grossbuchstabe (zwischen A und Z) sein.
  **/
public AbstractChessman(boolean isBlack,char description) {
  this.description = description;
  this.isBlack = isBlack;
}
```

Für die Abfrage der Farbe werden wir der Variablen `isBlack` eine get-Methode spendieren[6]:

```
/** Gibt true genau dann zurueck,
  * wenn die Schachfigur schwarz ist
  **/
public boolean isBlack() {
  return isBlack;
}
```

Die Variable `description` verwenden wir, um die Methode `toChar` allgemein realisieren zu können. Im Falle einer weissen Spielfigur liefern wir den Inhalt der Variablen zurück. Bei schwarzen Spielfiguren wandeln wir den übergebenen Buchstaben von Groß- nach Kleinschreibung.[7] Dies Aktion führen wir in der Methode `toChar` durch:

```
/** Liefert ein einzelnes Zeichen, das die Figur repraesentiert.
  **/
public char toChar() {
  return (char) (description + ((isBlack) ? ('a' - 'A') : 0));
}
```

So weit so gut. Neben der `toChar`-Methode können wir auch die Methode `canTake` allgemeingültig formulieren, indem wir aufgrund der Farbe entscheiden:

```
/** Testet, ob diese Figur eine andere Figur theoretisch zu schlagen
  * in der Lage ist. Dies ist immer der Fall, wenn die andere Figur
  * nicht dieselbe Farbe hat.
  **/
public boolean canTake(Chessman chessman) {
  if (!(chessman instanceof AbstractChessman))
    return false;
  return isBlack != ((AbstractChessman)chessman).isBlack;
}
```

Zu guter Letzt werden wir noch die Gelegenheit nutzen, die Methoden `equals` und `hashCode` anzupassen. Zwei Schachfiguren sollen hierbei als vom gleichen Typ gelten, wenn ihre `toChar`-Methode denselben Wert zurückliefert:

```
/** Vergleicht zwei Schachfiguren auf Typ-Gleichheit. Zwei
  * Schachfiguren gelten hierbei als gleich vom gleichen Typ,
```

[6] Bei booleschen Werten ist es üblich, im Methodennamen `is` anstelle von `get` zu verwenden.

[7] Wir erreichen dies, indem wir die Differenz zwischen den Zeichen „A“ und „a“ auf den Wert addieren. Wie Sie aus dem ersten Teil dieses Kurses bereits wissen, lassen sich in Java einzelne Zeichen, also `char`-Werte, wie Zahlen auffassen und behandeln. Die Differenz zwischen demselben Buchstaben in Groß- und Kleinschreibweise ist hierbei immer konstant.

```
    * wenn sie dieselbe Zeichendarstellung (toChar) haben.
    */
  public boolean equals(Object o) {
    if (o instanceof Chessman)
      return ((Chessman)o).toChar() == toChar();
    return false;
  }

  /** Da wir die equals-Methode ueberschrieben haben, muessen
    * wir auch die hashCode-Methode ueberschreiben.
    **/
  public int hashCode() {
    return toChar();
  }
```

Nun ist es dann aber auch (fast) genug mit abstrakten Klassen – wir wollen uns nun einige konkrete Figurentypen ansehen.

1.1.3.3 König, Dame, Turm und Läufer

In diesem Abschnitt wollen wir vier der existierenden Schachfiguren (König, Dame, Turm und Läufer) realisieren. Bevor wir das aber tun, wollen wir ein letztes Mal generalisieren und die Eigenschaften dieser Figuren zusammenfassen:
Was haben König, Dame, Turm und Läufer gemeinsam? Alle vier Figuren bewegen sich in gerader Richtung – entweder horizontal und vertikal (Turm) oder in diagonale Richtungen (Läufer). König und Dame können sich sowohl horizontal als auch vertikal bewegen. Hierbei haben die Figuren eine unterschiedliche Reichweite: die einen können sich nur ein Feld weit bewegen (König), die anderen können sich mehrere Felder weit bewegen (Dame).
Auch wenn die vier Schachfiguren sehr unterschiedlich erscheinen, bewegen sie sich also doch nach einem Algorithmus des gleichen Schemas:

- Für jede Zugrichtung[8] führe folgende Schritte durch:
  1. Beginne bei der Ausgangsposition.
  2. Gehe einen Schritt in die zu überprüfende Richtung. Markiere das Feld als einen möglichen Kandidaten für einen Zug.
  3. Falls wir außerhalb des Brettes gelandet sind, brich die Iteration ab.
  4. Falls wir auf einem belegten Feld gelandet sind, brich die Iteration ab.
  5. Falls wir am Ende der Reichweite der Figur angekommen sind, brich die Iteration ab.
  6. Wenn die Abbruchkriterien nicht zutreffen, setze die Iteration mit dem zweiten Schritt fort.
- Eliminiere von den möglichen Kandidaten solche, auf denen sich eine nicht schlagbare Figur befindet.

[8] Gemeint sind, im Sinne eines Kompass gesprochen, Nord, Süd, West, Ost, Nordwest, Nordost, Südwest und Südost.

Wir wollen diese Idee nun in die Tat umsetzen. Zu diesem Zweck definieren wir eine Klasse namens `LinearChessman` (also eine Schachfigur, die sich geradlinig bewegt:[9])

```
package de.uni.karlsruhe.aifb.prog2.chess;

/** Basis-Klasse der meisten Schachfiguren. Der LinearChessman ist
  * in der Lage, sich in gerader Linie (linear) fortzubewegen. Dies
  * trifft, ausser beim Pferd, auf alle Figuren zu.
  **/
public class LinearChessman extends AbstractChessman {

  /** Besagt, dass sich die Figur geradlinig bewegen laesst. */
  private boolean horizontalAndVertical;

  /** Besagt, dass sich die Figur diagonal bewegen laesst. */
  private boolean diagonal;

  /** Reichweite einer Figur in Feldern. */
  private int range;
```

Wir haben unserer Klasse hierbei drei Instanzvariablen spendiert:

- Eine Variable `horizontalAndVertical` besagt, ob sich unsere Figur senk- und waagerecht (Kompassrichtungen Nord, Süd, West und Ost) bewegen kann.
- Eine Variable `diagonal` besagt, ob sich die Figur diagonal (Nordwest, Nordost, Südwest, Südost) bewegen kann.
- Eine Variable `range` beinhaltet die Reichweite der Figur in Feldern pro Zug. Bauer oder König können sich beispielsweise nur ein Feld pro Zug fortbewegen, während dies bei der Dame oder dem König beliebig viele sein können.

Beachten Sie, dass sich die Klasse von der Superklasse `AbstractChessman` ableitet – sie erbt also somit auch die Eigenschaft, eine Farbe und ein beschreibendes Zeichen zu besitzen (`description` und `isBlack`). In unserem Konstruktor werden wir auch diese Werte jeweils setzen:

```
  /** Konstruktor.
    * @param isBlack ist true genau dann, wenn die Figur schwarz
    *  ist.
    * @param description eine Darstellung der Figur als Zeichen.
    *  Dieser Wert muss ein Grossbuchstabe (zwischen A und Z) sein.
    * @param horizontalAndVertical ist true genau dann, wenn sich
    *   die Figur horizontal und vertikal bewegen kann
    * @param diagonal ist true genau dann, wenn sich die Figur
    *   diagonal bewegen kann.
    * @param range die Anzahl der Felder, die die Figur maximal
    *   pro Zug ziehen kann
    **/
  public LinearChessman(boolean isBlack,char description,
```

[9] Genaugenommen gilt dies auch für den Bauern. Wegen seinen komplexeren Zugregeln wollen wir uns mit diesem Figurentyp allerdings erst später beschäftigen

```
      boolean horizontalAndVertical, boolean diagonal,
      int range) {
  super(isBlack,description);
  this.horizontalAndVertical = horizontalAndVertical;
  this.diagonal = diagonal;
  this.range = range;
}
```

Nun aber zur Realisierung des Algorithmus. Wir rollen das Feld von hinten auf und implementieren zuerst den letzten Schritt. Gegeben seien ein Schachbrett, eine Schachfigur und ein Feld von möglichen Zugpositionen (realisiert durch ein Feld von Booleschen Werten). Wir eliminieren nun alle Kandidaten, auf denen sich eine nicht schlagbare Figur befindet:

```
/** Hilfsmethode, die zur Programmierung der
  * getDestinations-methode verwendet werden kann.
  * @param destinations alle Felder, auf die die Figur ziehen
  *   kann, ohne dass der Weg dorthin von einer anderen Figur
  *   blockiert wuerde (einschliesslich der blockierenden Figur
  *   selbst). Genau diese Eintraege im Booleschen Feld muessen dann
  *   auf true gesetzt sein. Die Methode setzt in diesem Feld exakt
  *   jene Werte auf false zurueck, die mit einer nicht schlagbaren
  *   Figur besetzt sind.
  * @param chessboard das zugehoerige Schachbrett.
  * @param chessman die Schachfigur, die bewegt werden soll
  **/
public static void cleanBlocked(boolean[][] destinations,
                 Chessboard chessboard,Chessman chessman) {
  for (int i = 0; i < destinations.length; i++)
    for (int j = 0; j < destinations[i].length; j++)
      if (destinations[i][j]) { // muss geprueft werden?
        Chessman opponent = chessboard.getContent(i,j);
        if (opponent != null && !chessman.canTake(opponent))
          destinations[i][j] = false;
      }
}
```

Beachten Sie hierbei, dass die Methode `cleanBlocked` sehr allgemein gehalten ist. Sie könnte uns bei der Programmierung anderer Schachfiguren ebenfalls noch nützlich sein. Wir realisieren sie deshalb nicht in unserer gerade entwickelten Klasse, sondern fügen sie in die Superklasse `AbstractChessman` ein. Ein weiterer interessanter Punkt ist, dass die Methode das Ergebnis ihrer Berechnungen nicht mittels eines `return`-Statements liefert, sondern das (per Referenz übergebene) Feld direkt manipuliert. Wir nutzen hierbei also so genannte Seiteneffekte aus.

Wir tasten uns nun einen Schritt weiter vor. Wir wollen eine Zugrichtung überprüfen. Zu diesem Zweck schreiben wir eine Methode namens `check`, die diesen Schritt für uns erledigt. Als Parameter übergeben wir ihr das Schachbrett, die Position der Figur, und das Feld, in das sie mögliche Kandidaten eintragen soll. Die zu überprüfende Zugrichtung übergeben wir als Feld namens `direction`, wobei

der erste Eintrag für die Veränderung der Zeile[10] und der zweite Eintrag für die Veränderung der Spalte steht:

```
/** Ueberprueft eine Zugrichtung */
private void check(int[] direction,Chessboard board,
                   int row,int col,boolean[][] destinations) {
  try
  {
    for (int i = 1; i <= range; i++) {
      // Gehe einen Schritt
      row = row + direction[0];
      col = col + direction[1];
      // Markiere das Feld als Kandidaten
      destinations[row][col] = true;
      // Ist das Feld schon besetzt?
      if (board.getContent(row,col) != null)
        break;
    }
  }
  catch(ArrayIndexOutOfBoundsException e)
  {
    // Wir sind am Rande des Feldes angekommen!
  }
}
```

Die konkrete Implementierung entspricht dem zuvor entwickelten Algorithmus und ist in den Kommentarzeilen des Programms dokumentiert. Beachten Sie, dass wir die Gültigkeit der Position (also, ob wir uns noch im Feld befinden) nicht explizit überprüfen. Sollten wir uns ausserhalb des Feldes aufhalten, dann wird beim Eintrag in das Feld `destinations` eine `ArrayIndexOutOfBoundsException` entstehen. Diese Exception fangen wir und beenden dann unseren Methodenaufruf. Vergegenwärtigen Sie sich auch noch einmal den Unterschied zur Methode `cleanBlocked`. In jener Methode mussten wir die konkrete Figur als Parameter übergeben. Dies war notwendig, da wir die Methode als **Klassenmethode** formuliert haben. Unsere Methode `check` ist aber eine **Instanzmethode** - das heißt, sie gehört zu einer bestimmten Schachfigur und hat somit auch Zugriff auf deren Instanzvariablen (wie etwa auf die Reichweite `range`).
Nun, da wir den Kern unseres Algorithmus implementiert haben, ist es bis zur Realisierung der Methode `getDestinations` nur noch ein kleiner Schritt. Zuerst hinterlegen wir die möglichen Zugrichtungen in zwei konstant definierten Feldern:

```
/** Gerade Richtungen */
private final static int[][] DIRECTIONS_HV =
  {{1,0},{-1,0},{0,1},{0,-1}};

/** Diagonale Richtungen */
private final static int[][] DIRECTIONS_DIAG =
  {{1,1},{1,-1},{-1,1},{-1,-1}};
```

[10] $+1$ steht für einen Schritt nach unten, -1 für einen Schritt nach oben. 0 steht für den Stillstand

Anschliessend formulieren wir unsere Methode `getDestinations`, die lediglich noch die Methoden `check` und `cleanBlocked` in der richtigen Reihenfolge aufzurufen hat:

```
/** Ueberprueft, auf welche Felder sich die Figur im naechsten Zug
  * theoretisch bewegen kann.
  **/
public boolean[][] getDestinations
  (Chessboard board,int row,int col) {
  // Bis das Gegenteil bewiesen ist,
  // darf unsere Figur nirgends hin
  boolean[][] res =
    new boolean[board.getNumOfRows()]
               [board.getNumOfColumns()];
  // In einer Schleife berechnen wir alle Positionen,
  // die theoretisch moeglich sind.
  if (horizontalAndVertical)
    for (int i = 0; i < DIRECTIONS_HV.length; i++)
      check(DIRECTIONS_HV[i],board,row,col,res);
  if (diagonal)
    for (int i = 0; i < DIRECTIONS_DIAG.length; i++)
      check(DIRECTIONS_DIAG[i],board,row,col,res);
  // Eliminiere nun die Felder,
  // die nicht betreten werden koennen
  cleanBlocked(res,board,this);
  // Gib das Ergebnis zurueck
  return res;
}
```

Weitere Methoden sind nicht mehr zu definieren; was wir ansonsten benötigen, erbt unsere Klasse von ihrem Elternteil `AbstractChessman`.
Unsere konkreten Klassen für König (`King`), Dame (`Queen`), Turm (`Rook`) und Läufer (`Bishop`) sehen nun jeweils sehr einfach aus. Wir müssen lediglich einen Konstruktor definieren, der die notwendigen Werte (Zugrichtungen, Reichweite, Beschreibung) setzt. Für die Reichweite verwenden wir beim König `1`, bei allen anderen Figuren `Integer.MAX_INT` (beliebig weit). Lediglich die Farbe der Spielfigur wird in unserem Konstruktor nicht explizit gesetzt:

```
1  package de.uni.karlsruhe.aifb.prog2.chess;
2
3  /** Diese Klasse repraesentiert den Koenig */
4  public class King extends LinearChessman {
5
6    /** Konstruktor.
7      * @param isBlack ist true genau dann, wenn die Figur schwarz ist.
8      **/
9    public King(boolean isBlack) {
10     super(isBlack,'K',true,true,1);
11   }
12 }
```

```
1  package de.uni.karlsruhe.aifb.prog2.chess;
2
3  /** Diese Klasse repraesentiert die Dame */
```

```
4  public class Queen extends LinearChessman {
5
6    /** Konstruktor.
7      * @param isBlack ist true genau dann, wenn die Figur schwarz ist.
8      **/
9    public Queen(boolean isBlack) {
10     super(isBlack,'Q',true,true,Integer.MAX_VALUE);
11   }
12 }
```

```
1  package de.uni.karlsruhe.aifb.prog2.chess;
2
3  /** Diese Klasse repraesentiert den Turm */
4  public class Rook extends LinearChessman {
5
6    /** Konstruktor.
7      * @param isBlack ist true genau dann, wenn die Figur schwarz ist.
8      **/
9    public Rook(boolean isBlack) {
10     super(isBlack,'R',true,false,Integer.MAX_VALUE);
11   }
12 }
```

```
1  package de.uni.karlsruhe.aifb.prog2.chess;
2
3  /** Diese Klasse repraesentiert den Laeufer */
4  public class Bishop extends LinearChessman {
5
6    /** Konstruktor.
7      * @param isBlack ist true genau dann, wenn die Figur schwarz ist.
8      **/
9    public Bishop(boolean isBlack) {
10     super(isBlack,'B',false,true,Integer.MAX_VALUE);
11   }
12 }
```

Beachten Sie, dass wir vier komplette Schachfiguren mit nur wenigen Zeilen Programmcode definieren konnten. Den komplizierten Algorithmus der Zugberechnung konnten wir mit Hilfe des objektorientierten Ansatzes vereinheitlichen und somit erheblichen Programmieraufwand einsparen.

1.1.3.4 Von Bauern und Pferden

Kommen wir nun zu den verbleibenden Schachfiguren Bauer und Pferd. Beide Figuren besitzen zu viele besondere Eigenschaften, als dass wir bei ihnen ohne eine Sonderbehandlung auskämen. So ist das Pferd etwa die einzige Schachfigur, die sich nicht gerade, sondern in Winkeln bewegt. Der Bauer wiederum darf nur gerade ziehen und diagonal schlagen, so dass man hier besondere Rücksicht auf den Inhalt des Schachbretts nehmen muss.
Beginnen wir mit der Figur des Springers. Das Vorgehen bei der Zugberechnung entspricht prinzipiell dem des `LinearChessman`. Wir speichern alle möglichen Zugbewegungen in einem konstanten Feld namens `DIRECTIONS`:

```
  /** Richtungen, in die sich das Pferd bewegen kann. Der erste
    * Eintrag ist stets die Richtung in der Zeile, die zweite die
    * Richtung in der Spalte.
    **/
  private final static int[][] DIRECTIONS =
    {{-2,-1},{-2,1},{-1,-2},{-1,2},{1,-2},{1,2},{2,-1},{2,1}};
```

In einer Schleife iterieren wir nun durch die verschiedenen möglichen Zugrichtungen. Ist eine errechnete Position innerhalb des Schachfeldes, markieren wir sie als möglichen Kandidaten. Anschließend eliminieren wir mit Hilfe der Methode `cleanBlocked` alle nicht erlaubten Züge. Dieses Vorgehen sollte aus dem letzten Abschnitt geläufig sein, so dass wir die Klasse `Knight` in ihrer Gesamtheit betrachten können;

```
1  package de.uni.karlsruhe.aifb.prog2.chess;
2
3  /** Die Springer-Figur des Schachspiels. */
4  public class Knight extends AbstractChessman {
5
6    /** Konstruktor.
7      * @param isBlack ist true genau dann, wenn die Figur schwarz ist.
8      **/
9    public Knight(boolean isBlack) {
10     super(isBlack,'N');
11   }
12
13   /** Richtungen, in die sich das Pferd bewegen kann. Der erste
14     * Eintrag ist stets die Richtung in der Zeile, die zweite die
15     * Richtung in der Spalte.
16     **/
17   private final static int[][] DIRECTIONS =
18     {{-2,-1},{-2,1},{-1,-2},{-1,2},{1,-2},{1,2},{2,-1},{2,1}};
19
20   /** Ueberprueft, auf welche Felder sich die Figur im naechsten Zug
21     * theoretisch bewegen kann.
22     **/
23   public boolean[][] getDestinations
24     (Chessboard board,int row,int col) {
25     // Bis das Gegenteil bewiesen ist, darf unsere Figur nirgends hin
26     boolean[][] res =
27       new boolean[board.getNumOfRows()]
28                  [board.getNumOfColumns()];
29     // In einer Schleife berechnen wir alle Positionen,
30     //  die theoretisch moeglich sind.
31     for (int i = 0; i < DIRECTIONS.length; i++) {
32       int x = row + DIRECTIONS[i][0];
33       int y = col + DIRECTIONS[i][1];
34       // Sind die Koordinaten im gueltigen Bereich?
35       if (board.isValid(x,y))
36         res[x][y] = true;
37     }
38     // Eliminiere nun die Felder, die nicht betreten werden koennen
39     cleanBlocked(res,board,this);
40     // Gib das Ergebnis zurueck
41     return res;
```

```
42    }
43  }
```

Kommen wir nun zur kompliziertesten aller Schachfiguren: dem Bauern. Wir definieren eine Klasse `Pawn`, die sich von der Klasse `AbstractChessman` ableitet. Für diese Klasse definieren wir einen Konstruktor, in dem wir die Beschreibung der Figur durch das Zeichen „P“ festlegen:

```
package de.uni.karlsruhe.aifb.prog2.chess;

/** Die Bauern-Figur
  */
public class Pawn extends AbstractChessman {

  /** Konstruktor.
    * @param isBlack ist true genau dann, wenn die Figur schwarz ist.
    **/
  public Pawn(boolean isBlack) {
    super(isBlack,'P');
  }
```

Wenn wir das zu implementierende Interface betrachten, so stellen wir erneut fest, dass wir bis auf die Methode `getDestinations` bereits alle gestellten Anforderungen erfüllt haben. Wir wollen uns jetzt um diesen letzten Punkt kümmern.
Der Bauer bewegt sich beim Schach immer um ein Feld vorwärts, nie zurück. Er bewegt sich gerade, wenn das Feld vor ihm frei ist, und schlägt diagonal. Wurde der Bauer im Spiel noch nicht bewegt, darf sein erster Zug auch zwei Felder geradeaus sein[11].
Genau diese Regeln müssen wir nun in unserer Methode beachten. Wir gehen von einem Feld von `false`-Werten aus, d. h. die Figur darf sich in keine einzige Richtung bewegen:

```
  /** Ueberprueft, auf welche Felder sich die Figur im naechsten Zug
    * theoretisch bewegen kann.
    **/
  public boolean[][] getDestinations
    (Chessboard board,int row,int col) {
    // Bis das Gegenteil bewiesen ist, darf unsere Figur nirgends hin
    boolean[][] res =
      new boolean[board.getNumOfRows()]
                 [board.getNumOfColumns()];
```

Nun werden wir die verschiedenen Zugmöglichkeiten durchtesten. Wir gehen davon aus, dass sich wie in Abbildung 1.2 die schwarzen Figuren zu Anfang oben befinden (je weiter oben, desto kleiner der Zeilen-Index im Feld). Schwarze Figuren ziehen also, indem sie nach unten wandern (den Zeilen-Index erhöhen). Weisse Figuren ziehen nach oben (und erniedrigen den Zeilen-Index):[12]

```
    // Im Falle "schwarz" gehen wir nach unten, andernfalls nach oben
    row = isBlack() ? row + 1 : row - 1;
```

[11] Komplexere Schachregeln, wie etwa die Umwandlung des Bauern oder die Rochade, schließen wir zur Vereinfachung aus.

[12] Schach-Freunde aufgepasst: Bitte beachten Sie die Übungsaufgabe auf Seite 36.

Ist das resultierende Feld im Schachbrett noch nicht belegt, haben wir einen möglichen Zug gefunden. Zuvor verwenden wir allerdings die `isValid`-Methode, um die Gültigkeit der errechneten Koordinaten zu überprüfen:

```
// Pruefe, ob wir noch im Feld sind
if (!board.isValid(row,col))
  return res;
// Pruefe nun, ob der Bauer geradeaus gehen kann
if (board.getContent(row,col) == null)
  res[row][col] = true;
```

Unser nächster Test gilt der Möglichkeit, dass der Bauer zu schlagen in der Lage ist. Zu diesem Zweck erhöhen bzw. erniedrigen wir den Spaltenindex (wir verwenden hierzu eine Schleife) und überprüfen,

- ob die errechneten Koordinaten überhaupt gültig sind (`isValid`) und
- ob sich an dieser Stelle eine schlagbare Figur befindet.

Sind diese Bedingungen erfüllt, haben wir einen gültigen Zug gefunden:

```
// Pruefe nun, ob der Bauer schlagen kann
for (int i = -1; i < 2; i += 2) {
  int column = col + i;
  if (!board.isValid(row,column))
    continue;
  Chessman opponent
    = board.getContent(row,column);
  if (opponent != null && canTake(opponent))
    res[row][column] = true;
}
```

Es verbleibt nur noch ein Test. Wir müssen überprüfen, ob unser Bauer zwei Felder auf einmal ziehen darf. Dies ist der Fall, wenn

- der Bauer noch nicht bewegt wurde. Das ist genau dann der Fall, wenn er schwarz ist und das mögliche Ziel die vierte Reihe von oben ist (oder er weiss und das Ziel die vierte Reihe von unten ist) und
- der Bauer nicht schon beim Test gescheitert ist, ob er *ein* Feld geradeaus gehen kann.
- Ferner muss natürlich wieder gelten, dass das Zielfeld im gültigen Bereich und nicht bereits besetzt ist.

Mit diesem letzten Test haben wir sämtliche zu beachtenden Regeln bedacht; unsere Methode ist somit komplett:

```
// Pruefe, ob der Bauer zwei Felder gehen darf
// Voraussetzung: er darf zumindest EIN Feld
// geradeaus gehen.
if (res[row][col]) {
  row = isBlack() ? row + 1 : row - 1;
  if (board.isValid(row,col) &&
      ((row == res.length - 4 && !isBlack())
      || (row == 3 && isBlack())) &&
```

```
            board.getContent(row,col) == null)
        res[row][col] = true;
    }
```

Wir haben somit alle Figuren definiert, die wir zur Realisierung eines Schachbrettes brauchen. Wir wollen uns nun damit befassen, wie wir den Spieler mit diesen Objekten interagieren lassen.

1.1.3.5 Zur Wiederholung: die Klassen `GameModel` und `GameEngine`

Der folgende Abschnitt gibt eine kleine Einführung in die Klassen `GameModel` und `GameEngine`. Er ist für jene Leser gedacht, für die der erste Teil dieses Kurses schon ein Weilchen zurückliegt – oder für fortgeschrittene Programmierer, die sofort mit dem zweiten Band begonnen haben. Sollten Ihnen die Klassen zu Genüge vertraut sein, können sie die folgenden Seiten auch überspringen und direkt auf Seite 29 weiterlesen.
Sie werden in diesem Band lernen, grafische Oberflächen zu programmieren. Im Moment sind wir allerdings noch nicht so weit; und wir wollten uns schließlich auch schließlich auf das beschränken, was wir aus dem ersten Band an Vorwissen mitbekommen haben.
Auch im ersten Teil des Kurses waren wir mit dem Problem konfrontiert, ohne Kenntnis der entsprechenden Klassen Grafik für einfache Spiele programmieren zu müssen. Um dieses Problem zu lösen, befinden sich in den `Prog1Tools`[13] die Klassen `GameEngine` und `GameModel`. Wir werden diese Klassen hier wiederverwerten – denn schließlich ist die Wiederverwertung von bereits definierten Klassen eine der Stärken der objektorientierten Programmierung.
Unsere grafische Oberfläche setzt sich aus zwei grundlegenden Komponenten zusammen:

1. Ein Interface namens `GameModel` erlaubt es uns, beinahe beliebige Brettspiele (oder Spiele, die in ihrem Aufbau einem Brettspiel ähneln) durch ein und dieselbe Oberfläche darstellbar zu machen. Die verschiedenen Methoden, die auf den Seiten 26 und 27 im JavaDoc-Format spezifiziert sind, werden im folgenden Abschnitt für unser Schachspiel realisiert.
2. Die Klasse `GameEngine` übernimmt die komplette Steuerung und den grafischen Aufbau unseres Spieles. Alles, was wir tun müssen, ist, ein Objekt der `GameEngine` zu erzeugen und dem Konstruktor eine Instanz des `GameModel` zu übergeben:

   ```
   new GameModel(konkretesModell);
   ```

   Wir haben im ersten Band mit Hilfe dieser einen Klasse mehrere Spiele grafisch dargestellt (man erinnere sich etwa an das Game Of Life). Die Klasse kann mit beliebigen `GameModel`-Implementierungen arbeiten, ohne nähere Angaben über deren konkrete Bedeutung haben zu müssen.

[13] Diese Klassen-Bibliothek ist auf der Webseite zum Kurs kostenlos erhältlich.

Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Prog1Tools

Interface GameModel

public abstract interface **GameModel**

Klassen, die dieses Interface implementieren, können von der GameEngine als Spiel dargestellt und gesteuert werden.

Method Summary

void	**buttonPressed**(int row, int col) Signalisiert, dass ein bestimmter Button gedrückt wurde.
int	**columns**() Gibt die Anzahl der Spalten des Spielbretts zurück.
void	**firePressed**() Signalisert, dass der Feuer-Button gedrückt wurde.
char	**getContent**(int row, int col) Gibt den aktuellen Inhalt eines bestimmten Feldes zurück.
java.lang.String	**getFireLabel**() Gibt den Text zurück, der aktuell auf dem Feuer-Button stehen soll.
java.lang.String	**getGameName**() Gibt den Namen des Spieles als String zurück.
java.lang.String	**getMessages**() Gibt den Text zurück, der in der aktuellen Runde im Meldefenster stehen soll.
int	**rows**() Gibt die Anzahl der Zeilen des Spielbretts zurück.

Method Detail

rows

```
public int rows()
```

Gibt die Anzahl der Zeilen des Spielbretts zurück. Die Anzahl darf sich im Laufe des Spieles nicht mehr verändern.

columns

```
public int columns()
```

Gibt die Anzahl der Spalten des Spielbretts zurück. Die Anzahl darf sich im Laufe des Spieles nicht mehr verändern.

Abbildung 1.5: Dokumentation der Klasse `GameModel` (Seite 1)

---

getFireLabel

```
public java.lang.String getFireLabel()
```

Gibt den Text zurück, der aktuell auf dem Feuer-Button stehen soll.

---

getMessages

```
public java.lang.String getMessages()
```

Gibt den Text zurück, der in der aktuellen Runde im Meldefenster stehen soll.

---

getGameName

```
public java.lang.String getGameName()
```

Gibt den Namen des Spieles als String zurück.

---

getContent

```
public char getContent(int row,
                       int col)
```

Gibt den aktuellen Inhalt eines bestimmten Feldes zurück.
Parameters:
`row` - die Zeile, von 0 an gezählt
`col` - die Spalte, von 0 an gezählt

---

buttonPressed

```
public void buttonPressed(int row,
                          int col)
```

Signalisiert, dass ein bestimmter Button gedrückt wurde.
Parameters:
`row` - die Zeile, von 0 an gezählt
`col` - die Spalte, von 0 an gezählt

---

firePressed

```
public void firePressed()
```

Signalisert, dass der Feuer-Button gedrückt wurde.

Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Abbildung 1.6: Dokumentation der Klasse `GameModel` (Seite 2)

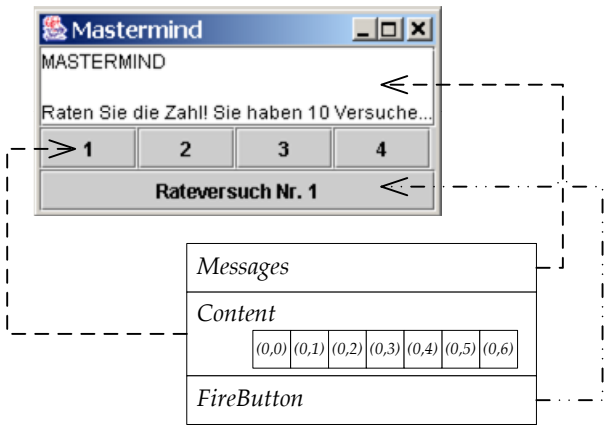

Abbildung 1.7: Die grafische Oberfläche und das zugehörige Modell

Wie können wir uns den Aufbau unserer Spieleoberfläche am besten vorstellen? Abbildung 1.7 skizziert das Datenmodell, das unserer Oberfläche zugrunde liegt, am Beispiel des Mastermind-Spieles aus Band 1:

- Für die Ausgabe von Nachrichten an die Spielerin bzw. den Spieler gibt es ein spezielles Textfenster. Die `GameEngine` holt sich aus dem Datenmodell die darzustellenden Texte, indem sie die Methode `getMessages` des `GameModel`-Objektes aufruft. Wie dieser Text anschließend auf dem Bildschirm dargestellt wird, braucht uns als Entwickler nicht zu kümmern. Wir geben den Text lediglich als einen `String` zurück.

- Das eigentliche Spielbrett wird im Datenmodell als „Content" bezeichnet. Wie bei einem Schachbrett oder einer Tabelle setzt es sich aus einer festen Anzahl von Zeilen und Spalten zusammen. Die Anzahl der Zeilen bzw. Spalten ist fest und kann aus dem Modell durch die Methoden `rows` bzw. `columns` erfragt werden.

  Jedes Spielfeld, das durch das Modell angesprochen werden kann, wird anhand seiner Zeilen- und Spaltennummer angesprochen (vgl. Abbildung 1.7). Es gibt im `GameModel` zwei Aktionen für ein Feld: der Inhalt des Spielfeldes kann abgefragt werden (`getContent`) und der Benutzer bzw. die Benutzerin kann auf ein einzelnes Spielfeld mit der Maus klicken (`buttonPressed`). Die durch das Klicken ausgelösten Aktionen (etwa, dass sich der Inhalt des Feldes verändert), werden innerhalb der Modellklasse vollzogen. Anschließend werden diese Änderungen von der `GameEngine` automatisch auf dem Bildschirm dargestellt. Der Inhalt eines Content-Feldes ist hierbei ein einzelnes Character-Zeichen (**`char`**) – in unserem Schach-Spiel könnte dies die Bezeichnung einer einzelnen Schachfigur sein.

- Außerdem existiert noch der so genannte Feuer-Knopf (fire-button), der mit einem beliebigen Text belegt werden kann. Der Feuer-Knopf ist insbesondere für rundenbasierte Spiele von Interesse. Die Beschriftung des Knopfes wird durch die Methode `getFireLabel` erfragt. Das Drücken des Knopfes wird durch die Methode `firePressed` symbolisiert.

1.1.3.6 Die Schnittstelle zum Benutzer

Wir werden nun die Schnittstelle zur Außenwelt definieren, wie sie in Abbildung 1.4 modelliert wurde. Dazu beginnen wir mit einer einfachen Implementierung des `GameModel`-Interface. Die Klasse `ChessboardDisplay` besitzt eine Instanzvariable `chessboard`, in der das zu visualisierende Schachbrett gehalten wird. Die Methoden `rows` und `columns` werden einfach an das `Chessboard` weitergereicht, das auf Anfrage die Zahl seiner Zeilen und Spalten nennt. Die Methode `getContent` liest das Schachbrett an der entsprechenden Stelle aus und liefert das Ergebnis der `toChar`-Methode der vorgefundenen Schachfigur. Alle weiteren Methoden werden mit einem einfachen Standardverhalten gefüllt (tue nichts bzw. tue so wenig wie möglich), das bei der Definition von entsprechenden Subklassen überschrieben werden kann:

```
1  package de.uni.karlsruhe.aifb.prog2.chess;
2
3  import Prog1Tools.GameModel;
4
5  /** Basisimplementierung fuer die Darstellung von Schachbrettern.
6    * Laesst sich durch Bilden von Unterklassen anpassen.
7    **/
8  public abstract class ChessboardDisplay implements GameModel {
9
10   /** Das dargestellte Schachbrett */
11   protected Chessboard chessboard;
12
13   /** Konstruktor.
14     * @param chessboard das darzustellende Schachbrett
15     **/
16   public ChessboardDisplay(Chessboard chessboard) {
17     this.chessboard = chessboard;
18   }
19
20   /** Gibt die Anzahl der Zeilen des Spielbretts
21     * zurueck. Die Anzahl darf sich im Laufe des
22     * Spieles nicht mehr veraendern.
23     **/
24   public int rows() {
25     return chessboard.getNumOfRows();
26   }
27
28   /** Gibt die Anzahl der Spalten des Spielbretts
29     * zurueck. Die Anzahl darf sich im Laufe des
30     * Spieles nicht mehr veraendern.
31     **/
32   public int columns() {
```

```
33      return chessboard.getNumOfColumns();
34    }
35
36    /** Gibt den Text zurueck, der aktuell auf dem
37      * Feuer-Button stehen soll. Defaultwert ist NEXT MOVE
38      **/
39    public String getFireLabel() {
40      return "NEXT MOVE";
41    }
42
43    /** Gibt den Text zurueck, der in der aktuellen
44      * Runde im Meldefenster stehen soll. Defaultwert ist
45      * ein leerer String.
46      **/
47    public String getMessages() {
48      return "";
49    }
50
51    /** Gibt den Namen des Spieles als String zurueck.
52      * Standardwert ist CHESS
53      **/
54    public String getGameName() {
55      return "CHESS";
56    }
57
58    /** Gibt den aktuellen Inhalt eines bestimmten Feldes zurueck
59      * @param row die Zeile, von 0 an gezaehlt
60      * @param col die Zeile, von 0 an gezaehlt
61      **/
62    public char getContent(int row, int col) {
63      Chessman chessman = chessboard.getContent(row,col);
64      if (chessman == null)
65        return ' ';
66      return chessman.toChar();
67    }
68
69    /** Signalisiert, dass ein bestimmter Button gedrueckt wurde.
70      * Standardimplementierung ist es, nichts zu tun.
71      * @param row die Zeile, von 0 an gezaehlt
72      * @param col die Zeile, von 0 an gezaehlt
73      **/
74    public void buttonPressed(int row,int col) {
75    }
76
77    /** Signalisert, dass der Feuer-Button gedrueckt wurde.
78      * Standardimplementierung ist es, nichts zu tun.
79      **/
80    public void firePressed() {
81    }
82
83  }
```

Kommen wir nun zu unserem eigentlichen Schachspiel, das wir von der Klasse `ChessboardDisplay` ableiten:

```
package de.uni.karlsruhe.aifb.prog2.chess;
```

```
import Prog1Tools.GameEngine;

/** Einfache Schachapplikation. Stellt ein Schachbrett auf dem
  * Bildschirm dar, auf dem der Spieler Zuege machen kann.
  **/
public class Chessgame extends ChessboardDisplay {
```

Zuerst einmal wollen wir uns darüber Gedanken machen, wie wir ein Schachbrett mit der üblichen Belegung von Figuren erzeugen könnten. Als Erstes erzeugen wir von jeder unserer Schachfiguren exakt eine Instanz die wir in einem konstanten Feld von Schachfiguren verwalten:

```
  /** Die verwendeten Schachfiguren. */
  public final static Chessman[] CHESSMEN =
  {
    new Pawn(true),    //  0: schwarzer Bauer
    new Pawn(false),   //  1: weisser Bauer
    new Rook(true),    //  2: schwarzer Turm
    new Rook(false),   //  3: weisser Turm
    new Knight(true),  //  4: schwarzer Springer
    new Knight(false), //  5: weisser Springer
    new Bishop(true),  //  6: schwarzer Laeufer
    new Bishop(false), //  7: weisser Laeufer
    new Queen(true),   //  8: schwarze Dame
    new Queen(false),  //  9: weisse Dame
    new King(true),    // 10: schwarzer Koenig
    new King(false)    // 11: weisser Koenig
  };
```

Anschließend definieren wir in einer weiteren Konstante, wie ein Schachbrett prinzipiell zu Anfang belegt sein muss. Wir verwenden hierbei die Indizes der einzelnen Schachfiguren in dem Feld `CHESSMEN`, wobei der Wert `-1` für ein leeres Feld steht:

```
  /** Der Standardaufbau eines Schachbretts */
  public final static int[][] BOARD_AT_START =
  {
    { 2, 4, 6, 8,10, 6, 4, 2},
    { 0, 0, 0, 0, 0, 0, 0, 0},
    {-1,-1,-1,-1,-1,-1,-1,-1},
    {-1,-1,-1,-1,-1,-1,-1,-1},
    {-1,-1,-1,-1,-1,-1,-1,-1},
    {-1,-1,-1,-1,-1,-1,-1,-1},
    { 1, 1, 1, 1, 1, 1, 1, 1},
    { 3, 5, 7, 9,11, 7, 5, 3}
  };
```

Nun können wir eine Methode `createInitial` definieren, die uns anhand der obigen Felder ein neues Schachbrett aufbaut:

```
  /** Erzeugt ein neues, "frisches" Schachbrett */
  protected Chessboard createInitial()
  {
    Chessboard board = new Chessboard(8,8);
    for (int i = 0; i < 8; i++)
```

```
    for (int j = 0; j < 8; j++)
      if (BOARD_AT_START[i][j] > -1)
        board.setContent(i,j,CHESSMEN[BOARD_AT_START[i][j]]);
  return board;
}
```

Diese Methode können wir beim Aufruf des Konstruktors unserer Klasse somit einfach verwenden, um unser Schachbrett in Startkonstellation zu bringen:

```
/** Konstruktor.
  **/
public Chessgame() {
  super(null);
  chessboard = createInitial();
}
```

Kommen wir nun zum Hin- und Herschieben von Figuren. Der Benutzer soll zuerst eine Figur klicken dürfen und anschließend das Ziel markieren, auf das die Figur geschoben werden soll.
Damit eine derartige Aktion durchführbar ist, müssen wir zwischen zwei möglichen Zuständen unterscheiden[14]:

1. Der Benutzer hat eine Figur angewählt. Der nächste Mausklick wird das Ziel zeigen, wohin er die Figur schieben möchte. Handelt es sich um einen legalen Zug, dann soll die Aktion durchgeführt werden - aber nur dann! Auf jeden Fall hat aber der nächste Mausklick zur Folge, dass die Selektion aufgehoben wird.
2. Der Benutzer hat noch keine Figur ausgewählt. Klickt er nun auf eine Figur, dann wird diese Figur als ausgewählt markiert.

Um diese möglichen Zustände des Spiels modellieren zu können, definieren wir drei neue Instanzvariablen:

```
/** Die Reihe des letzten Buttons, der gedrueckt wurde. */
private int lastRow;

/** Die Spalte des letzten Buttons, der gedrueckt wurde. */
private int lastCol;

/** Ist momentan ein Button aktiviert? */
private boolean activeButton;
```

Die Variable `activeButton` enthält genau dann den Wert `true`, wenn der Spieler eine Figur ausgewählt hat. Ist dies der Fall, wird in `lastRow` und `lastCol` die Position des Feldes gehalten, auf das der Benutzer geklickt hat.
Wir können nun unsere Methode `buttonPressed` gemäß obiger Vorgabe implementieren:

[14] Selbstverständlich müssen wir bei der Umsetzung ferner beachten, dass Fehleingaben, wie das Klicken auf ein leeres Feld anstatt auf eine Figur, abgefangen werden.

```
/** Signalisiert, dass ein bestimmter Button gedrueckt wurde.
  * Standardimplementierung ist es, nichts zu tun.
  * @param row die Zeile, von 0 an gezaehlt
  * @param col die Zeile, von 0 an gezaehlt
  **/
public void buttonPressed(int row,int col) {
  // Falls erstmals auf eine Figur gedrueckt wird, merken
  // wir uns die Koordinaten
  if (!activeButton && chessboard.getContent(row,col) != null) {
    lastRow = row;
    lastCol = col;
    activeButton = true;
    return;
  }
  // Andernfalls teste, ob es sich um einen erlaubten Zug handelt
  if (activeButton)
  {
    Chessman chessman = chessboard.getContent(lastRow,lastCol);
    boolean[][] destinations =
      chessman.getDestinations(chessboard,lastRow,lastCol);
    if (destinations[row][col]) { // bewege Figur
      chessboard.move(lastRow,lastCol,row,col);
    }
  }
  // Zu guter Letzt, loesche die Selektion
  activeButton = false;
}
```

Die Grundfunktionalität unseres Schachspiels haben wir somit realisiert - was bleibt also noch zu tun?
Da wäre zum einen die Beschriftung unseres Spiels. Wir überschreiben die Methoden `getMessages` und `getGameName` so, dass sie eine halbwegs stimmige Bezeichnung zurückgeben.

```
/** Gibt den Text zurueck, der in der aktuellen
  * Runde im Meldefenster stehen soll. Defaultwert ist
  * ein leerer String.
  **/
public String getMessages() {
  return "Game Of Chess";
}

/** Gibt den Namen des Spiels als String zurueck.
  **/
public String getGameName() {
  return "Chessboard";
}
```

Nun wäre da noch unser Feuer-Knopf. Eigentlich müssten wir ihm nicht unbedingt ein besondere Funktion zuweisen. Aber da wir ihn nun schon einmal haben – wie wäre es, wenn wir mit seiner Hilfe das Brett wieder auf seine Ausgangsposition zurücksetzen könnten?
Die hierzu notwendigen Methoden sind schnell geschrieben. Zuerst überschreiben wir die Methode, die die Beschriftung unseres Feuer-Knopfes bestimmt:

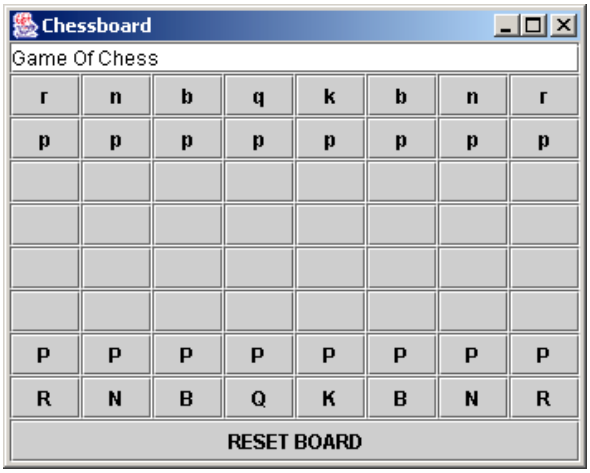

Abbildung 1.8: Das Schachspiel auf dem Bildschirm

```
/** Gibt den Text zurueck, der aktuell auf dem
  * Feuer-Button stehen soll.
  **/
public String getFireLabel() {
  return "RESET BOARD";
}
```

Anschließend kümmern wir uns um die Methode `firePressed`. Wir verwenden die Methode `createInitial`, um ein neues Schachbrett zu erzeugen:

```
/** Signalisert, dass; der Feuer-Button gedrueckt wurde.
  * Versetzt das Brett wieder in den Anfangszustand.
  **/
public void firePressed() {
  chessboard = createInitial();
  activeButton = false;
}
```

Wir haben nun alle Klassen definiert, um unser Schachspiel aufbauen und steuern zu können. Es fehlt nur noch eine Main-Methode, um unser Programm zu starten. Wir fügen diese Methode in unsere Klasse `Chessgame` ein:

```
/** Main-Methode zum Starten des Programms */
public static void main(String[] args) {
  new GameEngine(new Chessgame());
}
```

Wenn wir nun all unsere Klassen übersetzen und das Programm mit

Konsole

```
java de.uni.karlsruhe.aifb.prog2.chess.Chessgame
```

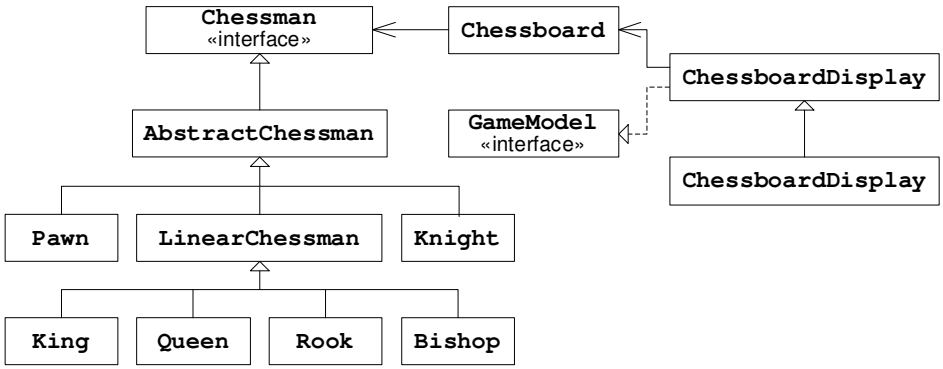

Abbildung 1.9: Das gesamte Klassenmodell im Überblick

starten, so sehen wir den in Abbildung 1.8 dargestellten Bildschirm. Versuchen wir nun, die Schachfiguren zu bewegen, so stellen wir fest, dass uns dies nur im Rahmen der gültigen Zugmöglichkeiten erlaubt ist. Die erste von uns zu lösende Aufgabenstellung wurde somit erfüllt.

1.1.4 Das Programm im Überblick

Wenn Sie den ersten Teil dieses Kurses kennen, so werden Sie am Ende jedes Praxis-Beispiels den Abdruck des gesamten Quelltextes erwarten. Wir werden in diesem Band ein wenig mit dieser Tradition brechen.
Der Grund für diese Neuerung ist schlicht und ergreifend der Umstand, dass wir mit mächtigen Schritten auf das Lager der fortgeschrittenen Software-Entwickler zuschreiten. Der zunehmende Umfang unserer Programme würde den Lesefluss eher behindern als fördern, was wir nach Möglichkeit vermeiden wollen.
Selbstverständlich werden sämtliche Quellen zu diesem Buch auch wieder auf den Internet-Seiten zu diesem Kurs erhältlich sein. Sie können sich also jede hier programmierte Zeile am heimischen Computer in Ihrer bevorzugten Entwicklungsumgebung betrachten.
Es gibt jedoch Situationen, in denen ein abschließender Überblick auch weiterhin sinnvoll sein wird. In unserem Fall beispielsweise haben wir eine Menge neuer Klassen definiert, die in der Designphase noch nicht bekannt waren. So hatten wir lediglich das Interface `Chessman` vorgegeben, aber noch keine einzige konkrete Schachfigur modelliert.
Um Ihnen einen Gesamtüberblick über alle verwendeten Klassen und ihre Beziehungen zu vermitteln, haben wir aus diesem Grund das komplette Objektmodell in Abbildung 1.9 zusammengestellt. Aus Gründen der Übersichtlichkeit haben wir Instanzvariablen und konkrete Methoden nicht aufgeführt. Das Klassendiagramm dürfte Ihnen aber dennoch helfen, einen Überblick über den Entwicklungsstand nach Beendigung dieses Abschnitts zu erhalten.

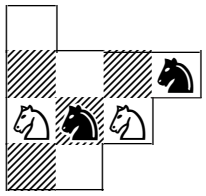

Abbildung 1.10: Das Springer-Problem

1.1.5 Übungsaufgaben

Aufgabe 1.1

Schach-Freunde aufgepasst: Im vorangehenden Abschnitt ist den Autoren ein „Fehler“ unterlaufen! Auf Seite 23 wird angenommen, dass weiße Figuren nach oben und schwarze Figuren nach unten wandern (den Zeilen-Index also erhöhen). Leider ist das so nicht korrekt – es sollte genau andersherum sein. Bitte korrigieren Sie diesen „Fehler“ der Autoren.

1.2 Nichts als Probleme

Erinnern wir uns an das eigentliche Ziel des Kapitels. Wir wollen eine Knobelaufgabe lösen – nämlich das in Abbildung 1.10 beschriebene Springer-Problem.
Auf den letzten Seiten dieses Kapitels haben wir einen ersten Schritt in Richtung Lösung gemacht. Wir haben ein Modell entworfen und implementiert, mit dem man Schachspiele (und auf dem Schachspiel basierende Situationen) realisieren kann. Was fehlt uns also noch zur endgültigen Lösung?
Die Antwort auf diese Frage liegt wie in der Frage selbst. Wenn wir ein *Schach-Problem* lösen wollen, benötigen wir zwei Dinge:

1. Ein Modell, mit dem wir das *Schach*-Spiel in den Computer bekommen und
2. einen Ansatz, der uns hilft, *Probleme* zu lösen.

Da wir die erste Bedingung ja bereits erfüllen, werden wir uns nun mit dem zweiten Teil unserer Aufgabenstellung näher befassen.

1.2.1 Aufgabenstellung

Zielsetzung dieses Abschnitts wird es sein, einen allgemeinen Lösungsweg für die Bewältigung von „Problemen“ zu entwickeln. Da wir natürlich nicht die allgemeine Lösung jede beliebige Aufgabe auf dieser Welt finden können, sollen die zu lösenden Probleme einige besondere Eigenschaften haben:

- Das Problem soll in einer beschränkten Zahl von Schritten lösbar sein.

- Das Problem soll eine definierte Ausgangs- und Endsituation haben. Das bedeutet, Startsituation[15] und Endsituation[16] sind als solche deutlich zu erkennen. Die Endsituation muss hierbei nicht unbedingt eindeutig sein, das heißt es kann auch mehr als eine Lösung geben.
- Nach jedem Zwischenschritt soll der aktuelle Zustand des Problems bewertet werden können.[17] Der Zustand (im Folgenden auch mit dem englischen Wort **Scenario**) bezeichnet soll hierbei unabhängig von den vorherigen Zügen sein – das heißt, es ist egal, ob man ihn mit drei oder dreißig Schritten erreicht hat

Ziel der Aufgabe ist die Entwicklung eines so genannten **Framework**, einem allgemeinen und wiederverwendbaren Rahmens zur Modellierung und Realisierung eines Problemlösers. Ähnlich wie bei der `GameModel`-`GameEngine`-Beziehung leisten wir hier also grundlegende Basisarbeit, deren Früchte wir erst im folgenden Abschnitt ernten werden ...

1.2.2 Designphase

Auch wenn sich die hier gestellte Aufgabe viel schwieriger anhört als das zuvor realisierte Schachspiel, ist dem doch nicht so. Sie werden vielmehr feststellen, dass (zumindest vom Entwurf her) die Realisierung mit weitaus weniger Tücken verbunden ist.
Wie würden wir eine Aufgabe wie etwa das Springer-Problem zu lösen versuchen, wenn wir nicht den Computer zur Verfügung hätten? Wir würden auf dem Brett verschiedene Züge ausprobieren und uns so Zug um Zug zur Lösung hangeln. Je nachdem, wie gut unser Gedächtnis ist, würden wir uns hierbei eventuelle Sackgassen merken, also Spielsituationen, die nicht zum Erfolg geführt haben. Auf diese Weise würden wir nicht zweimal denselben Fehler machen.
Dieses Verhalten wollen wir nun auf dem Computer nachbilden. Da wäre zuerst einmal eine einzelne Spielsituation (also etwa die akutelle Position der Figuren auf dem Brett), die wir durch eine Klasse namens `Scenario` modellieren werden. Jedes einzelne `Scenario` sollte hierbei folgende Eigenschaften besitzen:

- Man sollte bei einem `Scenario` immer genau wissen, ob es sich um eine Lösung des Problems handelt. Wir definieren aus diesem Grunde eine Methode namens `isSolution`.
- Man sollte zwei Spielsituationen immer miteinander vergleichen können – eben um herauszufinden, ob man denselben Zug schon einmal gemacht hat. Unser `Scenario` braucht aus diesem Grund eine `equals`-Methode.
- Man sollte von einer Spielsituation immer zu allen möglichen Folgesituationen kommen, die sich aus dem `Scenario` ergeben. Welche Zugalternativen besitzt

[15] beim Springerproblem etwa die Konstellation aus Abbildung 1.10.
[16] in unserem Fall dasselbe Brett – mit dem Unterschied, dass Schwarz und Weiß vertauscht sind
[17] etwa in der Form *gelöst* oder *nicht gelöst*

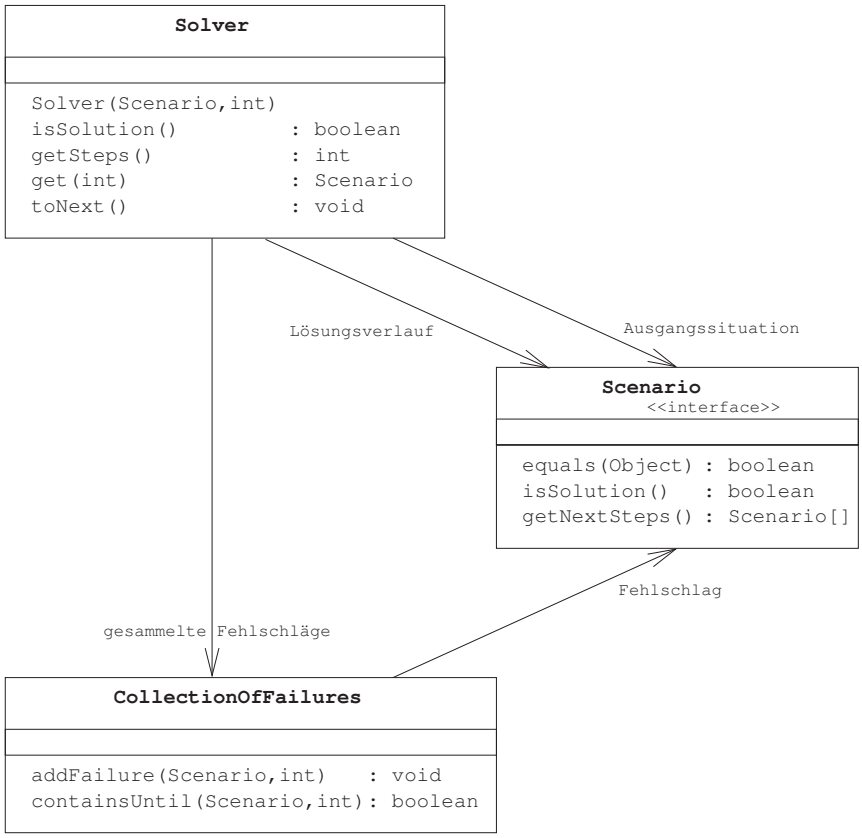

Abbildung 1.11: Design des Problemlösers

man? Zu diesem Zweck definieren wir eine Methode `getNextSteps`, die alle möglichen nächsten Züge zurückliefert.

Abbildung 1.11 zeigt die Klasse `Scenario` mit ihren Methoden. Wir kozipieren das `Scenario` als Interface, so dass Anwender beliebige Realisierungen dieser Schnittstelle umsetzen können.
Neben dem einzelnen `Scenario` benötigen wir noch eine Klasse `Solver`, die uns die Lösung für unser Problem berechnet. Unser Problem-Löser muss natürlich mindestens eine Referenz zu einem `Scenario`-Objekt beinhalten – nämlich der Ausgangssituation. Ferner kann das Objekt noch eine beliebige Zahl weiterer Szenarien besitzen – nämlich den Lösungsweg. Im Klassendiagramm sind diese Referenzen durch die Verbindungspfeile zwischen den Klassen `Solver` und `Scenario` dargestellt.
Wie soll nun die Schnittstelle unserer Solver-Klasse zur Aussenwelt sein? Wir geben einen Konstruktor vor, in dem die Ausgangsituation und die maximale Zahl

von Lösungsschritten vorgegeben sind. Beim Aufruf des Konstuktors soll unser Objekt sofort nach der ersten Lösung suchen.
Im laufenden Betrieb können wir mit Hilfe der Methode `isSolution` erfragen, ob unser `Solver`-Objekt eine Lösung beinhaltet. Die Anzahl der Schritte bis zur Lösung erfahren wir durch die Methode `getSteps`. Die Methode `get` wird verwendet um auf die einzelnen Lösungsschritte zuzugreifen. Mit `get(0)` erhalten wir die Ausgangssituation, mit `get(solver.getSteps())` die Endsituation. Für den Fall, dass wir nach mehr als einer Lösung suchen wollen, stellen wir auch noch eine Methode `toNext` zur Verfügung, die den `Solver` die nächste Lösung suchen lässt.
Kommen wir zu einem letzten, aber alles andere als unwichtigen Punkt. Wir haben noch nicht entschieden, wie wir uns die verschiedenen Sackgassen merken wollen, in die unser Solver während der Lösungssuche hineinstolpern kann. Da die Verwaltung der Fehlschläge etwas komplizierter werden könnte, lagern wir diese in ein separates Objekt aus. Die `CollectionOfFailures` speichert alle Irrwege, auf denen wir uns bewegt haben. Zu diesem Zweck definieren wir eine Methode `addFailure`, der wir einen Fehlschlag und die Nummer des Zuges übergeben, in dem er aufgetreten ist. Um zu überprüfen, ob es sich bei einem eventuellen Zug um einen Fehlschlag handelt, legen wir ferner eine Methode `containsUntil` fest, der wir ein `Scenario` und eine maximale Suchtiefe `i` übergeben. Die Methode liefert genau dann `true` zurück, wenn zwischen dem ersten und dem `i`-ten Zug diese Situation schon einmal aufgetreten ist.
Auf den nun folgenden Seiten werden wir dieses Design in die Tat umsetzen. Wir beginnen mit der `CollectionOfFailures` und tasten uns dann zum eigentlichen `Solver` vor.

1.2.3 Implementierung

Auch wenn unser Löser im Design relativ einfach aussah, erfordert die Implementierung im Detail den einen oder anderen Kniff, der auf den ersten Blick vielleicht nicht völlig einsichtig sein wird. Seien Sie an dieser Stelle nicht enttäuscht, falls Sie nicht alles verstehen. Schlimmstenfalls können Sie diese Phase auch überspringen und direkt auf Seite 51 weiterlesen.

1.2.3.1 `Scenario` und `CollectionOfFailures`

Wir beginnen damit, das Interface `Scenario` auszuformulieren:

```
1  package de.uni.karlsruhe.aifb.prog2.solutions;
2
3  /** Stellt ein moegliches Szenario auf dem Weg zur
4   *  Problemloesung dar.
5   */
6  public interface Scenario {
7
8    /** Vergleicht zwei Szenarien auf Gleichheit */
9    public boolean equals(Object o);
```

```
10
11    /** Stellt dieses Szenario bereits die Problemloesung dar? */
12    public boolean isSolution();
13
14    /** Liefert alle Szenarien, zu denen man sich von diesem
15      * Szenario aus bewegen kann.
16      **/
17    public Scenario[] getNextSteps();
18
19  }
```

Dieser Quelltext stellt nur die die im Klassendiagramm bereits beschriebene Schnittstelle dar und soll an dieser Stelle deshalb nicht weiter erläutert werden. Beginnen wir nun mit der Klasse, die unsere gesammelten Fehlschläge verwalten soll: der `CollectionOfFailures`.

```
package de.uni.karlsruhe.aifb.prog2.solutions;

/** Diese Klasse kann verwendet werden, um fehlgeschlagene
  * Versuche zu speichern.
  **/
public class CollectionOfFailures {

  /** Speichert alle Fehlschlaege nach ihrer Zugnummer ab. */
  private Scenario[][] failures = new Scenario[0][0];
```

Unsere Klasse muss pro Zugtiefe ein bestimmte Anzahl von Felschlägen beinhalten. Wir realisieren dies, indem wir ein zweidimensionales Feld namens `failures` definieren. Der erste Indexeintrag beinhaltet hierbei jeweils die Zugnummer.
Wir wollen uns nun darum kümmern, wie ein solches Feld durchsucht werden kann. Nehmen wir uns zuerst den einfachen Fal vor, dass wir lediglich eine bestimmte Zugnummer überprüfen wollen. Falls in dieser Zugnummer ein gewisses `Scenario` enthalten ist, soll die Methode den Index im Feld zurückliefern. Andernfalls erhalten wir einen negativen Wert.

```
  /** Besitzt die Sammlung fuer eine gewisse Zugnummer
    * bereits dieses Scenario als registrierten Fehlschlag?
    * @return -1, wenn das Scenario noch nicht bekannt ist,
    * den Index innerhalb des internen Feldes ansonsten
    **/
  private int contains(Scenario scenario, int index)
  {
    if (index >= failures.length || index < 0)
    {
      return -1;
    }
    for (int i = 0; i < failures[index].length; i++)
    {
      if (failures[index][i].equals(scenario))
      {
        return i;
      }
    }
```

```
    return -1;
  }
```

Die besagte Methode `contains` ist relativ einfach programmiert. Wir durchlaufen das Feld in einer Schleife und verwenden die `equals`-Methode, um Objekte zu vergleichen. Wenn wir aber das Suchen für einen bestimmten Zug lösen, können wir die Suche auch sehr einfach auf einen kompletten Bereich von Zügen ausdehnen:

```
  /** Ist ein gewisses Scenario in der Sammlung ab einem
    * gewissen Index schon enthalten?
    * @param scenario das Szenario, das gesucht werden soll
    * @param maxIndex der Index, bis zu dem hoechstens
    * gesucht werden soll
    * @return den Zug, unter dem das Scenario abgespeichert
    *   wurde. -1, wenn es nicht gefunden wurde.
    */
  public int containsUntil(Scenario scenario,int maxIndex)
  {
    maxIndex = Math.min(maxIndex + 1,failures.length);
    for (int i = 0; i <= maxIndex; i++)
    {
      if (contains(scenario,i) > -1)
        return i;
    }
    return -1;
  }
```

Kommen wir nun zum Einfügen eines neuen Fehlschlages. Hierbei prüfen wir, ob das einzufügende Element schon in unserem Feld erfasst ist. Dies können wir mit Hilfe der `containsUntil`-Methode erfragen. Liegt die Zugnummer des bereits eingefügten Scenarios unterhalb des neuen Index, so ist der Fehlschlag bereits bekannt. Wir brauchen ihn also nicht noch einmal einzufügen. Ist der Index größer als die neue Zugnummer, so ist der Fehlschlag zwar bekannt, aber erst ab einer späteren Zugnummer. Wir löschen den alten Eintrag und fügen das `Scenario` unter der neuen Nummer ein. Auf diese Weise vermeiden wir, dass eine Spielsituation doppelt im Feld vorkommt (dies würde die Suche verlangsamen).

```
  /** Markiert ein Szenario als Fehlschlag.
    * @param scenario das Szenario, das nicht zur Loesung
    *  fuehrte
    * @param move die Nummer des Zuges, in der das Szenario
    *  aufgetreten ist. Es wird mit 0 zu zaehlen begonnen.
    **/
  public void addFailure(Scenario scenario, int move) {
    // Befindet sich das Scenario bereits in der Sammlung?
    int index = contains(scenario,failures.length);
    if (index > -1)
    {
      // Fall: das Scenario war schon bei einem frueheren
      // Zug ein Fehlschlag, ist also bekannt.
      if (index <= move)
        return;
      // Fall: der Zug war bislang erst spaeter als Fehlschlag
```

```
      // bekannt, muss also verschoben werden.
      remove(scenario,index);
    }
    // Prueft, ob das Array bis zu der gewuenschten
    // move-Nummer reicht. Andernfalls wird das Array erweitert
    if (move >= failures.length)
    {
      // Kopiere das alte Feld in ein neues.
      Scenario[][] old = failures;
      failures = new Scenario[move + 1][];
      System.arraycopy(old,0,failures,0,old.length);
      // Fuelle den Rest mit leeren Feldern.
      for (int i = old.length; i <= move; i++)
      {
        failures[i] = new Scenario[0];
      }
    }
    // Erweitere das Array an der gewuenschten Stelle.
    Scenario[] old = failures[move];
    failures[move] = new Scenario[old.length + 1];
    // Kopiere die alten Werte hinein und fuege den neuen an.
    System.arraycopy(old,0,failures[move],1,old.length);
    failures[move][0] = scenario;
  }
```

Das Löschen aus dem Feld haben wir hierbei in eine separate Hilfsmethode namens `remove` ausgegliedert:

```
  /** Loesche ein Scenario aus einem bestimmten Index */
  private void remove(Scenario scenario, int index)
  {
    int i = contains(scenario,index);
    if (i > -1)
    {
      Scenario[] old = failures[i];
      failures[i] = new Scenario[old.length - 1];
      if (i > 0)
        System.arraycopy(old,0,failures[i],0,i);
      if (i < old.length - 1)
        System.arraycopy(old,i+1,failures[i],i,
          old.length-i-1);
    }
  }
```

Beachten Sie, dass wir einen Großteil des Aufwandes in die Problematik gesteckt haben, unser Feld von Szenarien vernünftig zu verwalten. Wir werden in diesem Buch gewisse Hilfsklassen, die so genannten **Collections** kennen lernen, die uns das Leben hier deutlich erleichtern werden.

1.2.3.2 Die Klasse `Solver`

Bevor wir mit der eigentlichen Realisierung unseres Problemlösers beginnen, benötigen wir noch eine winzig kleine Hilfsklasse. Für jeden Schritt, den wir in Richtung Lösung gehen, müssen wir nicht nur die aktuelle Spielsituation, son-

dern auch die möglichen nächsten Züge verwalten. Um dies zu meistern, definieren wir in unserer Klasse `Solver` eine innere Klasse `Iteration`. Diese Klasse beinhaltet ein `scenario` (so nennen wir die Variable) und ein Feld `next` mit allen Zügen, die von diesem `Scenario` aus möglich sind. Ein Zähler namens `current` speichert, welche der möglichen Schritte wir schon ausprobiert haben. Die Methoden `getNext()` und `getScenario()` ermöglichen Zugriff auf die Spielsituation und den nächsten noch nicht vollzogenen Schritt:

```
/** Innere Klasse, stellt einen Iterationszustand dar. */
private static class Iteration
{
  /** Das Scenario, um das es sich dreht. */
  private Scenario scenario;

  /** Die Schritte, zu denen man von dem Scenario aus
    * gehen kann.
    **/
  private Scenario[] next;

  /** Der aktuelle Schritt. */
  private int current;

  /** Konstruktor. */
  public Iteration(Scenario scenario)
  {
    this.scenario = scenario;
    next = scenario.getNextSteps();
    current = 0;
  }

  /** Gib das naechste, noch nicht bearbeitete Szenario
    * zurueck.
    **/
  public Scenario getNext()
  {
    if (current < next.length)
    {
      return next[current++];
    }
    return null;
  }

  /** Gib das Scenario zurueck, von dem ausgegangen wird. */
  public Scenario getScenario()
  {
    return scenario;
  }
}
```

Nun wollen wir uns um die Instanzvariablen unserer Klasse kümmern. Wir definieren ein Feld von `Iteration`-Objekten, das unsere Ausgangssituation und die durchgeführten Spielzüge abspeichert. Wir geben diesem Feld den Namen `solution`. Das Feld wird im Konstruktor die Länge `maxsteps+1` erhalten. Eine Variable namens `isSolution` besagt, ob der `Solver` gerade eine Lösung

beinhaltet. Die Variable `steps` zählt die gemachten Lösungsschritte und unsere Fehlschläge werden in einer `CollectionOfFailures` abgespeichert, der wir den Namen `failures` geben. Ferner definieren wir eine Variable `debug`, die wir intern benutzen, um während der Entwicklung detailliertere Ausgaben auf der Konsole zu erlauben:

```
/** Der Weg, der zur Loesung gegangen wird. */
private Iteration[] solution;

/** Beinhaltet das Objekt eine Loesung? */
private boolean isSolution;

/** Wie viele Schritte bis zur Loesung? */
private int steps;

/** Die Fehlschlaege auf dem Weg zur Loesung. */
private CollectionOfFailures failures;

/** Wollen wir Debug-Ausgaben haben? */
private boolean isDebug;
```

Der Zugriff auf unsere Variablen ist relativ einfach realisiert. Die Methode `isSolution()` entspricht einer Standard-get-Methode, wie wir sie auch aus dem ersten Band kennen.

```
/** Beinhaltet das Objekt momentan eine Loesung? */
public boolean isSolution()
{
  return isSolution;
}
```

Für die Anzahl der Schritte (`getSteps()`) benötigen wir allerdings schon eine Fallunterscheidung. Nur, wenn wir eine Lösung gefunden haben, steht nämlich in der Variablen `steps` ein sinnvoller Wert:

```
/** Gib die Anzahl der Schritte zur Loesung zurueck. */
public int getSteps()
{
  if (!isSolution)
  {
    return 0;
  }
  else
  {
    return steps;
  }
}
```

Ähnlich sieht es auch mit der Methode `get` aus, in der wir zusätzlich noch eine Überschreitung des maximalen Index überprüfen:

```
/** Gib das Scenario an der entsprechenden Stelle zurueck. */
public Scenario get(int index)
{
  // Im Fall des Start-Szenarios gib immer etwas zurueck.
```

```
    if (index == 0)
    {
      return solution[0].getScenario();
    }
    // Falls wir ansonsten keine Loesung haben, gib null zurueck.
    if (!isSolution)
    {
      return null;
    }
    // Falls wir oberhalb des maximalen Indexes sind, ebenfalls.
    if (index > steps)
    {
      return null;
    }
    // Andernfalls liefere das Ergebnis.
    return solution[index].getScenario();
  }
```

Betrachten wir nun die Methode `getNext`, die uns die nächste mögliche Lösung liefert. Wir gehen davon aus, dass wir bereits einmal erfolgreich eine Lösung gefunden haben. Die Variable `steps` beinhaltet also die aktuelle Zugnummer, die uns zur Lösung geführt hat. Für den Fall, dass das Problem keine Lösung mehr hat (`isSolution` hat den Wert `false`), führen wir die Methode überhaupt nicht aus. Andernfalls gehen wir einen Schritt zurück, also bevor wir die Lösung gefunden haben Hier beginnt dann unsere Suche:

```
  /** Setzt den Solver auf die naechste moegliche Loesung,
    * falls noch eine existiert.
    **/
  public void toNext()
  {
    // Muessen wir nach einer Loesung suchen?
    if (!isSolution)
    {
      return;
    }
    // Beginne mit der Iteration.
    steps--;
    bigLoop: while
      (steps >= 0 && !solution[steps].getScenario().isSolution())
```

Die folgende Suche nach einer Lösung wird in einer `while`-Schleife realisiert. Wir verwenden unser Feld von `Iterator`-Objekten, um uns für den aktuellen Schritt einen noch nicht gemachten Zug zu holen. Für diesen Zug überprüfen wir,

- ob wir diesen Zug im Verlaufe der Lösungsfindung schon einmal gemacht haben
- ob es sich bei diesem Zug um einen bekannten Fehlschlag handelt, oder
- ob wir am Ende der zur Verfügung stehenden Zahl von Zügen angelangt sind.[18]

[18] Die maximale Zahl erlaubter Züge ergibt sich aus der maximalen Größe des Feldes `solution`.

Trifft eine der Bedingungen auf das `Scenario` zu, müssen wir mit der nächsten Zugmöglichkeit weitermachen:

```
{
  // Ueberpruefe den naechsten moeglichen Schritt.
  Scenario next = solution[steps].getNext();
  if (next != null)
  {
    // Haben wir diesen Zug zuvor schon einmal gemacht?
    for (int i = 0; i <= steps; i++)
      if (next.equals(solution[i].getScenario())) {
        if (isDebug)
          System.out.println("Move already made in step "
             + i + ".");
        continue bigLoop;
      }
    // Ist das Ganze ein bereits bekannter Fehlschlag?
    if (failures.containsUntil(next,steps + 1) >= 0) {
      if (isDebug)
        System.out.println("Move already known as failure.");
      continue;
    }
    // Sind wir beim allerletzten Schritt und haben noch keine
    // Loesung?
    if (steps + 2 == solution.length && !next.isSolution())
    {
      if (isDebug)
        System.out.println("Out of steps.");
      continue;
    }
```

Andernfalls haben wir einen möglichen Zug gefunden. Wir speichern diesen Zug in einem `Iterator`-Objekt, erhöhen die Variable `steps` und machen mit dem nächsten Zug weiter:

```
    // Andernfalls gehe in den naechsten Schritt.
    steps++;
    if (isDebug)
      System.out.println("steps == " + steps);
    solution[steps] = new Iteration(next);
    continue;
  }
```

Kommen wir nun zu dem Fall, dass wir keine Zugmöglichkeit mehr haben, dass also `next==null` galt. In diesem Fall sind wir in einer Sackgasse gelandet. Wir speichern diese in der `CollectionOfFailures` und gehen einen Schritt zurück:

```
  // Andernfalls, gehe einen Schritt zurueck.
  failures.addFailure(solution[steps].getScenario(),steps);
  steps--;
  if (isDebug)
    System.out.println("steps == " + steps);
  if (steps < 0)
  {
    isSolution = false;
    return;
```

```
      }
    }
    // Das war's :-)
  }
```

Betrachten wir noch einmal unsere Schleifenbedingung:

```
bigLoop: while
  (steps >= 0 && !solution[steps].getScenario().isSolution())
```

Unsere Methode terminiert, wenn entweder eine Lösung gefunden wurde, oder aber der Inhalt der Variablen `steps` negativ wird. Im letztgenannten Fall konnten wir keine weitere Problemlösung finden.
Unsere Methode `getNext` funktioniert übrigens auch, wenn wir noch keine erste Problemlösung gefunden haben. Der Trick besteht hier darin, die Werte für die Instanzvariablen sinnvoll zu setzen. Und eben dies machen für uns die Konstruktoren:

```
/** Constructor. Dem Objekt werden das zu loesende
  * Anfangsszenario sowie eine maximale Zahl
  * erlaubter Loesungsversuche uebergeben.
  **/
public Solver(Scenario start, int maxSteps)
{
  this(start,maxSteps,false);
}

/** Constructor. Dem Objekt werden das zu loesende
  * Anfangsszenario sowie eine maximale Zahl
  * erlaubter Loesungsversuche uebergeben.
  **/
public Solver(Scenario start, int maxSteps, boolean isDebug)
{
  failures = new CollectionOfFailures();
  solution = new Iteration[maxSteps + 1];
  solution[0] = new Iteration(start);
  isSolution = true;
  steps = 1;
  this.isDebug = isDebug;
  toNext();
}
```

Mit diesem letzten Fragment ist unser Puzzle komplett - das Problemlösungs-Framework erfolgreich erstellt. Im nächsten Abschnitt können wir uns nun endlich daran machen, unser Springer-Problem zu lösen ...

1.2.4 Das Programm im Überblick

Bevor wir uns an die Realisierung des Springer-Problems machen, hier noch einmal die Klasse `Solver` in der Gesamtansicht:

```
1 package de.uni.karlsruhe.aifb.prog2.solutions;
2
3 /** Diese Klasse loest ein Problem, das durch ein
```

```
4     * Anfangsszenario gegeben ist.
5     **/
6   public class Solver {
7
8     /** Innere Klasse, stellt einen Iterationszustand dar. */
9     private static class Iteration
10    {
11      /** Das Scenario, um das es sich dreht. */
12      private Scenario scenario;
13
14      /** Die Schritte, zu denen man von dem Scenario aus
15        * gehen kann.
16        **/
17      private Scenario[] next;
18
19      /** Der aktuelle Schritt. */
20      private int current;
21
22      /** Konstruktor. */
23      public Iteration(Scenario scenario)
24      {
25        this.scenario = scenario;
26        next = scenario.getNextSteps();
27        current = 0;
28      }
29
30      /** Gib das naechste, noch nicht bearbeitete Szenario
31        * zurueck.
32        **/
33      public Scenario getNext()
34      {
35        if (current < next.length)
36        {
37          return next[current++];
38        }
39        return null;
40      }
41
42      /** Gib das Scenario zurueck, von dem ausgegangen wird. */
43      public Scenario getScenario()
44      {
45        return scenario;
46      }
47    }
48
49    /** Der Weg, der zur Loesung gegangen wird. */
50    private Iteration[] solution;
51
52    /** Beinhaltet das Objekt eine Loesung? */
53    private boolean isSolution;
54
55    /** Wie viele Schritte bis zur Loesung? */
56    private int steps;
57
58    /** Die Fehlschlaege auf dem Weg zur Loesung. */
```

```
59    private CollectionOfFailures failures;
60
61    /** Wollen wir Debug-Ausgaben haben? */
62    private boolean isDebug;
63
64    /** Constructor. Dem Objekt werden das zu loesende
65      * Anfangsszenario sowie eine maximale Zahl
66      * erlaubter Loesungsversuche uebergeben.
67      **/
68    public Solver(Scenario start, int maxSteps)
69    {
70      this(start,maxSteps,false);
71    }
72
73    /** Constructor. Dem Objekt werden das zu loesende
74      * Anfangsszenario sowie eine maximale Zahl
75      * erlaubter Loesungsversuche uebergeben.
76      **/
77    public Solver(Scenario start, int maxSteps, boolean isDebug)
78    {
79      failures = new CollectionOfFailures();
80      solution = new Iteration[maxSteps + 1];
81      solution[0] = new Iteration(start);
82      isSolution = true;
83      steps = 1;
84      this.isDebug = isDebug;
85      toNext();
86    }
87
88    /** Beinhaltet das Objekt momentan eine Loesung? */
89    public boolean isSolution()
90    {
91      return isSolution;
92    }
93
94    /** Setzt den Solver auf die naechste moegliche Loesung,
95      * falls noch eine existiert.
96      **/
97    public void toNext()
98    {
99      // Muessen wir nach einer Loesung suchen?
100     if (!isSolution)
101     {
102       return;
103     }
104     // Beginne mit der Iteration.
105     steps--;
106     bigLoop: while
107       (steps >= 0 && !solution[steps].getScenario().isSolution())
108     {
109       // Ueberpruefe den naechsten moeglichen Schritt.
110       Scenario next = solution[steps].getNext();
111       if (next != null)
112       {
113         // Haben wir diesen Zug zuvor schon einmal gemacht?
```

```
114         for (int i = 0; i <= steps; i++)
115           if (next.equals(solution[i].getScenario())) {
116             if (isDebug)
117               System.out.println("Move already made in step "
118                  + i + ".");
119             continue bigLoop;
120           }
121         // Ist das Ganze ein bereits bekannter Fehlschlag?
122         if (failures.containsUntil(next,steps + 1) >= 0) {
123           if (isDebug)
124             System.out.println("Move already known as failure.");
125           continue;
126         }
127         // Sind wir beim allerletzten Schritt und haben noch keine
128         // Loesung?
129         if (steps + 2 == solution.length && !next.isSolution())
130         {
131           if (isDebug)
132             System.out.println("Out of steps.");
133           continue;
134         }
135         // Andernfalls gehe in den naechsten Schritt.
136         steps++;
137         if (isDebug)
138           System.out.println("steps == " + steps);
139         solution[steps] = new Iteration(next);
140         continue;
141       }
142       // Andernfalls, gehe einen Schritt zurueck.
143       failures.addFailure(solution[steps].getScenario(),steps);
144       steps--;
145       if (isDebug)
146         System.out.println("steps == " + steps);
147       if (steps < 0)
148       {
149         isSolution = false;
150         return;
151       }
152     }
153     // Das war's :-)
154   }
155
156   /** Gib das Scenario an der entsprechenden Stelle zurueck. */
157   public Scenario get(int index)
158   {
159     // Im Fall des Start-Szenarios gib immer etwas zurueck.
160     if (index == 0)
161     {
162       return solution[0].getScenario();
163     }
164     // Falls wir ansonsten keine Loesung haben, gib null zurueck.
165     if (!isSolution)
166     {
167       return null;
168     }
```

```
169       // Falls wir oberhalb des maximalen Indexes sind, ebenfalls.
170       if (index > steps)
171       {
172         return null;
173       }
174       // Andernfalls liefere das Ergebnis.
175       return solution[index].getScenario();
176     }
177
178     /** Gib die Anzahl der Schritte zur Loesung zurueck. */
179     public int getSteps()
180     {
181       if (!isSolution)
182       {
183         return 0;
184       }
185       else
186       {
187         return steps;
188       }
189     }
190
191   }
```

1.3 Die Lösung aller Probleme?

Nun haben wir es fast geschafft: wir haben sowohl ein Modell für ein Schachbrett als auch ein Framework für die Lösung von Problemen geschaffen. Wir müssen lediglich noch beides miteinander verbinden – und somit stehen wir also kurz davor, unser Springerproblem zu lösen!
Nun mag der aufmerksame Leser an dieser Stelle sich vielleicht fragen: wäre das Ganze nicht vielleicht ein wenig einfacher gegangen: die Antwort lautet: *Ja*. Auf den letzten Seiten dieses Kapitels haben wir viele Schritte getan, die zur Lösung *eines* speziellen Problems nicht unbedingt nötig gewesen wären. Warum modellieren wir Schachfiguren wie den Bauern, wenn wir es nur mit dem Pferd zu tun haben? Warum machen wir uns die Mühe, unser Problem so weit zu abstrahieren, dass wir einen allgemeinen Problemlöser verfassen können?
Eine der Aufgaben dieses Buchs ist es, Ihnen einen Einblick in die kommerzielle Seite des Programmierens zu geben. Hierzu zählen neue Klassen wie etwa für die Programmierung grafischer Oberflächen oder Anwendungen im Netzwerk. Kommerziell zu Programmieren bedeutet aber auch, einen gewissen Anspruch bezüglich der Wirtschaftlichkeit zu erfüllen.
Stellen wir uns also vor, wir wären eine kleine Softwarefirma. Wäre unser bisheriges Vorgehen wirtschaftlich gewesen? Die Frage lässt sich nicht so einfach beantworten. Angenommen, wir hätten die Lösung des Springer-Problems als Auftrag erhalten und könnten davon ausgehen, dass wir nie wieder etwas ähnliches machen werden. In einem solchen Fall sollte man eine Lösung suchen, die die Spezifikation des Auftraggebers erfüllt, aber uns so wenig wir möglich Aufwand

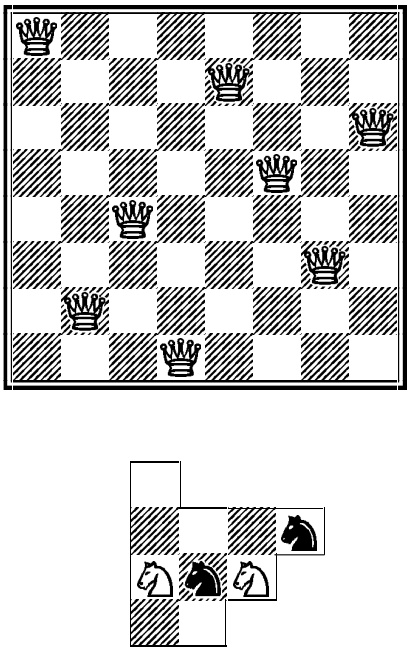

Abbildung 1.12: Springer- und Damenproblem

kostet. Jede Arbeitsstunde eines Softwareentwicklers ist teuer!
Gehen wir aber auf der anderen Seite einmal davon aus, dass sich unsere Firma auf die Lösung von „Problemen aller Art“ spezialisiert. Noch während Sie als Entwickler an der Lösung arbeiten, zieht der Vertrieb eine Vielzahl von ähnlichen Aufträgen an Land. In diesem Falle ist es ein Gewinn, von Anfang an etwas allgemeiner entwickelt zu haben. Haben Sie beim ersten Projekt vielleicht noch einen Verlust gemacht, ersparen Sie sich in den Folgeprojekten einen Großteil des Entwiclungsaufwandes. Wir können billiger produzieren und schaffen uns somit einen Wettbewerbsvorteil.
Um diesen Umstand zu verdeutlichen, ändern wir (mehr oder weniger) spontan das Ziel dieses Kapitels. Wir werden statt einem einfach *zwei* schachähnliche Probleme lösen.

1.3.1 Aufgabenstellung

Abbildung 1.12 zeigt die zu lösenden Aufgabenstellungen. Wir beschäftigen uns zum einen mit dem bereits bekannten Springerproblem. Auf einem beschnittenen Schachbrett befinden sich zwei schwarze und zwei weiße Pferde, die ihre Positionen in weniger als 40 Zügen tauschen müssen.
Zuvor wollen wir uns aber um das Achtdamenproblem kümmern, das Ihnen

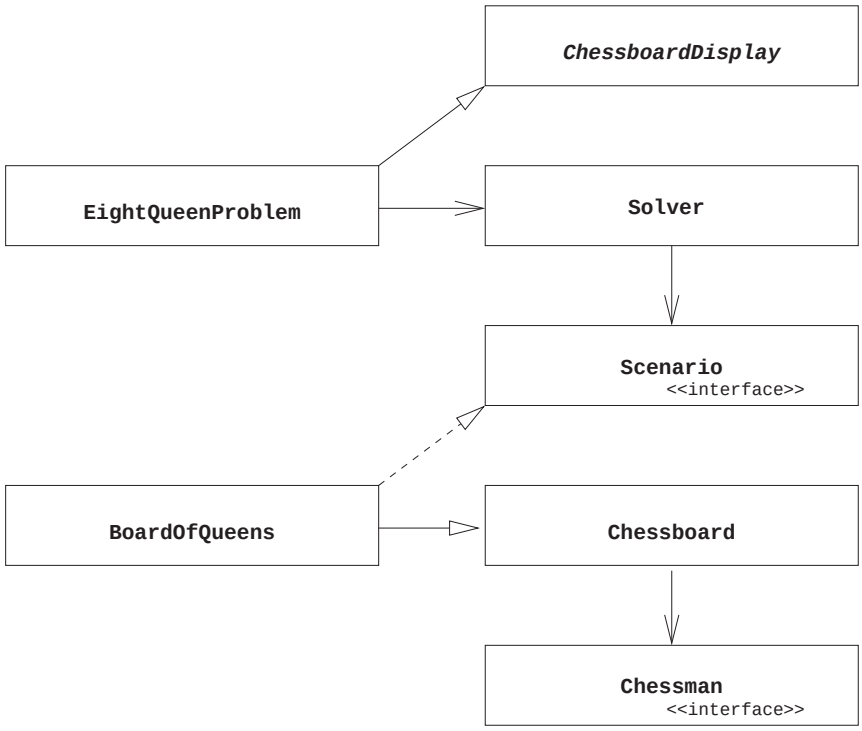

Abbildung 1.13: Die Lösung des Achtdamen-Problems im Design

aus dem ersten Teil dieses Buchs noch bekannt sein dürfte. Auf einem normalen Schachbrett sind acht Damen so zu positionieren, dass keine von ihnen eine andere schlagen kann. Eine mögliche Lösung ist in Abbildung 1.12 dargestellt. Wir wollen nun auch alle anderen möglichen Abbildungen finden.

1.3.2 Designphase

Wenn man unsere bereits vorhandenen Klassen zugrunde legt, ist das Design recht einfach und im Grunde identisch für beide Problemstellungen. Wir gehen deshalb an dieser Stelle lediglich auf das Achtdamen-Problem ein.
Für die Darstellung der Lösung auf dem Bildschirm benötigen wir einen Mechanismus, den wir dank unserer Klasse `ChessboardDisplay` bereits im ersten Teil dieses Kapitels realisiert haben. Wir definieren eine Klasse `EightQueenProblem`, die sich von dem abstrakten `ChessboardDisplay` ableitet. Da diese Klasse die Lösung eines Problems visualisieren soll, benötigt sie des weiteren ein `Solver`-Objekt, das die Lösung beinhaltet. Abbildung 1.13 stellt diesen Sachverhalt durch eine Assoziation dar.
Damit unser `Solver` eine Lösung des Problemes finden kann, benötigt er ein

`Scenario` als Ausgangssituation. Wir definieren eine Klasse `BoardOfQueens`, die das Interface implementiert. Als Superklasse wählen wir die Klasse `Chessboard`, die bereits das Modell eines Schachbrettes darstellt. Auf dem Brett unserer Klasse werden sich verschiedene Schachfiguren (`Chessman`) tummeln, die wir in der Implementierungsphase genauer betrachten werden. Es liegt jedoch relativ nahe, die Klasse `Queen` oder eine an sie angelehnte Subklasse zu verwenden.
Abbildung 1.13 stellt die Designidee und die an ihr beteiligten Klassen dar. Sie sehen, dass wir den Großteil der Realisierung bereits in den ersten Abschnitten erledigt haben. Es fehlen lediglich die Klassen `EightQueenProblem` und `BoardOfQueens`, die wir in der Implementierungsphase realisieren werden.
Um das Springerproblem zu lösen, werden wir denselben Ansatz wählen. Ersetzen Sie in Gedanken einfach die Klasse `BoardOfQueens` durch eine Klasse `BoardOfKnights` und das `EightQueenProblem` durch das `KnightSwitchProblem`.

1.3.3 Implementierung

1.3.3.1 Die Klasse `BoardOfQueens`

Wir beginnen mit der Klasse `BoardOfQueens`, die sich wie angesprochen von der Klasse `Chessboard` und dem Interface `Scenario` ableitet:

```
package de.uni.karlsruhe.aifb.prog2.solutions;

import de.uni.karlsruhe.aifb.prog2.chess.Chessboard;
import de.uni.karlsruhe.aifb.prog2.chess.Chessman;
import de.uni.karlsruhe.aifb.prog2.chess.Queen;

/** Die Klasse BoardOfQueens repraesentiert eine (Teil-)
 *  Loesung des Achtdamen-Problems.
 */
public class BoardOfQueens extends Chessboard implements Scenario {
```

Im ersten Schritt machen wir uns Gedanken über die Schachfiguren, die auf dem Brett beheimatet sind. Acht Damen gilt es zu platzieren. Da wir nur zwei Farben zur Verfügung haben, verwenden wir für alle die Farbe Weiß. Jede unserer Damen muss in der Lage sein, eine andere Dame zu schlagen – obwohl diese dieselbe Farbe hat. Wir definieren aus diesem Grund eine eigene Klasse `MyQueen`, in der wir die ursprüngliche `canTake`-Methode unserer Schachfigur überschreiben:

```
  /** Die Damen-Figur, die wir in unserem Spiel benutzen. */
  private static class MyQueen extends Queen {

    /** Konstruktor */
    public MyQueen() {
      super(false);
    }

    /** Testet, ob diese Figur eine andere Figur theoretisch
```

```
     * zu schlagen in der Lage ist. Im Gegensatz zur normalen
     * Dame kann diese Dame auch Figurn der eigenen
     * Farbe schlagen.
     **/
    public boolean canTake(Chessman chessman) {
      return true;
    }
  }
```

Wir instantiieren diese neu definierte Klasse und halten ein Damen-Objekt in einer entsprechenden Konstante. Dies wird unser `Chessman` sein, der in dem Problem zum Einsatz kommt:

```
/** Dies Konstante verweist auf die Schachfigur, die
 *  auf diesem Brett verwendet wird
 */
public final static Chessman QUEEN = new MyQueen();
```

Kümmern wir uns nun um unser spezielles Problem. Welche Attribute könnte unsere Klasse benötigen? Die Schachfiguren selbst werden durch die Superklasse `Chessboard` verwaltet. Wir wollen uns jedoch auch die Position der letzten Figur merken, die wir auf dem Schachbrett positioniert haben. Aus diesem Grund definieren wir zwei Variablen `row` und `column`:

```
/** Diese Variable besagt, in welche Spalte
 *  beim Konstruktionsvorgang die letzte Dame eingefuegt wurde.
 *  -1 steht hierbei fuer ein leeres Brett.
 */
private int row = -1;

/** Diese Variable besagt, in welche Zeile beim
 *  Konstruktionsvorgang die letzte Dame eingefuegt wurde.
 *  -1 steht hierbei fuer ein leeres Brett.
 */
private int column = -1;
```

Eine weitere Variable `canTake` wird besagen, ob die zuletzt eingefügte Dame eine der anderen Spielfiguren schlagen kann:

```
/** Kann die beim Konstruktionsvorgang eingefuegte Dame eine
 *  andere schlagen?
 */
private boolean canTake = false;
```

Mit diesen Informationen können wir nun beispielsweise herausfinden, ob unser aktuelles `Scenario` die Lösung des Problems ist. Dies ist nämlich genau dann der Fall, wenn wir nacheinander in jede Reihe eine Dame gesetzt haben (also `row==7` ist) und `canTake==false` gilt:

```
/** Stellt dieses Szenario bereits die Problemloesung dar? */
public boolean isSolution() {
  return row == 7 && !canTake;
}
```

Für die Berechnung des Wertes von `canTake` definieren wir uns eine gleichnamige Hilfsmethode. Wir lassen uns alle möglichen Züge ausgeben und überprüfen,

ob auf einem dieser Felder eine Dame steht:

```
/** Pruefe, ob die zuletzt eingefuegte Dame
 *  eine andere Dame schlagen kann
 */
private boolean canTake() {
  // Haben wir ein leeres Brett?
  if (row == -1 || column == -1)
    return false;
  // Andernfalls berechne alle moeglichen Zuege.
  boolean moves[][] = QUEEN.getDestinations(this,row,column);
  for (int i = 0; i < moves.length; i++)
    for (int j = 0; j < moves[i].length; j++)
      if (moves[i][j] && getContent(i,j) != null)
        return true;
  // Wenn wir den Check ueberstanden haben,
  // gibt es keine Schlagmoeglichkeiten
  return false;
}
```

Um unser Interface `Scenario` zu erfüllen, müssen wir lediglich noch die Methode `getNextSteps()` realisieren. Bevor wir dies tun, spendieren wir unserer Klasse aber noch zwei Konstruktoren:

- Ein argumentloser Konstruktor, der ein leeres Schachbrett erzeugt. Dieser Konstruktor kann später von unserer `main`-Methode verwendet werden, um die Ausgangssituation für das Spielgeschehen zu generieren.
- Ein Konstruktor, der aus einem alten Spielbrett `oldboard` und einer Spaltenangabe `column` ein neues Schachbrett erzeugt. Hierbei wird in die nächste freie Zeile unter der mit `column` gegebenen Spalte eine Dame gesetzt. Anschließend wird mit Hilfe der `canTake()`-Methode überprüft, ob eine Dame geschlagen werden kann.

```
/** Konstruktor; erzeugt ein leeres Schachbrett */
public BoardOfQueens() {
  super(8,8);
}

/** Erzeugt ein neues Schachbrett aus einem alten.
 *  Fuegt in die naechste zu besetzende Spalte unter der
 *  gegebenen Zeilennummer eine neue Dame ein.
 *  @param oldboard das Board, von dem ausgegangen wird.
 *  @param column die Spalte, in der die Dame eingefuegt
 *    werden soll.
 */
public BoardOfQueens(BoardOfQueens oldboard, int column) {
  // Kopiere den Inhalt
  super(8,8);
  for (int i = 0; i < 8; i++)
    for (int j = 0; j < 8; j++)
      content[i][j] = oldboard.content[i][j];
  // Setze Zeile und Spalte
  row = oldboard.row + 1;
```

```
    this.column = column;
    // Setze eine neue Dame in die gegebene Zeile
    setContent(row,column,QUEEN);
    // Pruefe, ob geschlagen werden kann
    canTake = canTake();
  }
```

Nun aber zur Methode `getNextSteps()`. Wir gehen von einem Brett aus, in dem wir zuletzt in Zeile `row` eine Dame platziert haben. Wir erzeugen nun mit Hilfe des oben definierten Konstruktors acht neue `Scenario`-Objekte, indem wir in einer Schleife die möglichen Spaltenindizes durchgehen. Wir zählen nebenbei die Zahl der Konstellationen, in denen keine Dame geschlagen werden kann (Variable `matches`) und liefern eben diese Objekte in einem Array zurück:

```
  /** Liefert alle Szenarien, zu denen man sich von diesem
    * Szenario aus bewegen kann.
    **/
  public Scenario[] getNextSteps() {
    // Falls wir am Ende angekommen sind,
    // koennen wir keine Felder mehr erzeugen
    if (row == 7)
      return new Scenario[0];
    // Wir speichern die moeglichen naechsten
    // Schritte in einem Feld
    BoardOfQueens[] next = new BoardOfQueens[8];
    // Wir belegen das Feld mit jeder
    // moeglichen naechsten Kombination.
    // Dann zaehlen wir die wirklich moeglichen
    // naechsten Schritte.
    int matches = 0;
    for (int i = 0; i < 8; i++) {
      next[i] = new BoardOfQueens(this,i);
      if (!next[i].canTake)
        matches++;
    }
    // Kopiere die Treffer in ein kleineres Array
    BoardOfQueens[] res = new BoardOfQueens[matches];
    matches = 0;
    for (int i = 0; i < 8; i++)
      if (!next[i].canTake) {
        res[matches] = next[i];
        matches++;
      }
    // Gib das Ergebnis zurueck
    return res;
  }
```

Somit ist unsere Klasse `BoardOfQueens` komplett.

1.3.3.2 Die Klasse `BoardOfKnights`

Die Hauptschwierigkeit bei der Klasse `BoardOfKnights` scheint es zu sein, ein beschnittenes Schachbrett zu modellieren. Tatsächlich ist die Lösung gar nicht so schwer: Das Schachbrett zu beschneiden bedeutet doch nur, gewisse Felder für

die Pferde nicht zugänglich zu machen. Dies können wir auch erledigen, indem wir sie durch andere Figuren blockieren. Wir definieren zu diesem Zweck eine unverrückbare Schachfigur, die wir auf den Namen `Block` taufen:

```
/** Innere Klasse; repraesentiert ein blockiertes Feld */
private static class Block extends LinearChessman {

  /** Konstruktor */
  public Block() {
    super(true,'X',false,false,0);
  }

  /** Testet, ob diese Figur eine andere Figur
    * theoretisch zu schlagen
    * in der Lage ist. Ein Block kann natuerlich
    * keine andere Figur schlagen.
    **/
  public boolean canTake(Chessman chessman) {
    return false;
  }
}
```

Ferner definieren wir, ähnlich wie beim Achtdamenproblem, eine spezielle Springer-Klasse namens `MyKnight`. Dieser Springer ist nicht in der Lage, andere Schachfiguren zu schlagen:

```
/** Innere Klasse; repraesentiert das Pferd */
private static class MyKnight extends Knight {

  /** Konstruktor */
  public MyKnight(boolean isBlack)
  {
    super(isBlack);
  }

  /** Testet, ob diese Figur eine andere Figur
    * theoretisch zu schlagen in der Lage ist. Im Gegensatz
    * zum normalen Springer kann dieser keine
    * wie auch immer gearteten Figuren schlagen.
    **/
  public boolean canTake(Chessman chessman) {
    return false;
  }
}
```

Mit Hilfe dieser inneren Klassen definieren wir nun Konstanten, die die verschiedenen verwendeten Figuren repräsentieren:

```
/** Dies Konstante steht fuer den schwarzen Springer */
public final static Chessman BLACK = new MyKnight(true);

/** Diese Konstante steht fuer den weissen Springer */
public final static Chessman WHITE = new MyKnight(false);

/** Diese Konstante steht fuer ein blockiertes Feld */
public final static Chessman BLOCK = new Block();
```

Wir wollen nun im ersten Schritt ein leeres, beschnittenes Schachbrett erstellen. In einem Feld merken wir uns, welche Stellen des Schachbretts mit blockierenden Figuren belegt werden müssen:

```
/** Der Startaufbau des Feldes (ohne Springer)
 *  false steht fuer ein leeres Feld,
 *  true fuer ein blockiertes.
 */
public final static boolean[][] BOARD_AT_START =
{
  {false,true,true,true},
  {false,false,false,false},
  {false,false,false,true},
  {false,false,true,true}
};
```

Anschließend definieren wir einen Konstruktor, der ein Brett der Höhe und Breite 4 erzeugt und dies mit den notwendigen Figuren belegt:

```
/** Konstruktor. Erzeugt ein leeres Brett, ohne Springer */
public BoardOfKnights() {
  super(4,4);
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      if (BOARD_AT_START[i][j])
        this.setContent(i,j,BLOCK);
}
```

Wir wollen nun ein Schachbrett mit der Ausgangsposition der Springer erzeugen können. In entsprechenden Konstanten speichern wir die Position der schwarzen und weißen Pferde ab:

```
/** Die Anfangs-Positionen der schwarzen Springer */
public final static int[][] BLACK_POS = {{1,3},{2,1}};

/** Die Anfangs-Positionen der weissen Springer */
public final static int[][] WHITE_POS = {{2,0},{2,2}};
```

Anschliessend definieren wir eine Methode `createInitial()`, die diese Felder ausliest und anhand der gespeicherten Positionen die Pferde an die richtige Stelle setzt:

```
/** Erzeugt die Anfangskonstellation des Problems. */
public static BoardOfKnights createInitial() {
  BoardOfKnights res = new BoardOfKnights();
  res.setContent(WHITE_POS[0][0],WHITE_POS[0][1],WHITE);
  res.setContent(WHITE_POS[1][0],WHITE_POS[1][1],WHITE);
  res.setContent(BLACK_POS[0][0],BLACK_POS[0][1],BLACK);
  res.setContent(BLACK_POS[1][0],BLACK_POS[1][1],BLACK);
  return res;
}
```

Die konstanten Positionsangaben werden wir übrigens auch nutzen, um ein Brett daraufhin zu überprüfen, ob eine Lösung vorliegt:

```
/** Stellt das Scenario bereits die Loesung dar ? */
```

```
public boolean isSolution() {
  Chessman black1 = getContent(WHITE_POS[0][0],WHITE_POS[0][1]);
  Chessman black2 = getContent(WHITE_POS[1][0],WHITE_POS[1][1]);
  Chessman white1 = getContent(BLACK_POS[0][0],BLACK_POS[0][1]);
  Chessman white2 = getContent(BLACK_POS[1][0],BLACK_POS[1][1]);
  if (black1 == null || black2 == null ||
      white1 == null || white2 == null)
    return false;
  return
    black1.equals(BLACK) &&
    black2.equals(BLACK) &&
    white1.equals(WHITE) &&
    white2.equals(WHITE);
}
```

Nun fehlt uns zu unserem Glück eigentlich nur noch die Methode `getNextSteps()`. Um alle möglichen nächsten Schritte zu erfragen, suchen wir unser Schachbrett nach Springer-Figuren ab. Für jede Springer-Figur lassen wir uns durch die Methode `getDestinations()` alle Zugmöglichkeiten nennen. Für jede dieser Möglichkeiten erstellen wir dann ein neues Schachbrett:

```
/** Liefert alle Szenarien, zu denen man sich von diesem
  * Szenario aus bewegen kann.
  **/
public Scenario[] getNextSteps() {
  // Theoretisch kann jeder Springer in
  // bis zu acht Positionen wandern.
  // Wir reservieren einfach einmal genug Platz :-)
  Scenario[] next = new Scenario[32];
  int counter = 0;
  // Nun wandern wir ueber das Schachbrett
  // und suchen unsere Springer
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++) {
      Chessman current = getContent(i,j);
      if (current != null && current instanceof MyKnight) {
        boolean[][] destinations
          = current.getDestinations(this,i,j);
        for (int x = 0; x < 4; x++)
          for (int y = 0; y < 4; y++)
            if (destinations[x][y]) {
              BoardOfKnights board = (BoardOfKnights) clone();
              board.move(i,j,x,y);
              next[counter] = board;
              counter++;
            }
      }
    }
  // Jetzt kopieren wir alles in ein auf
  // die richtige Groesse zugeschnittenes Feld
  Scenario[] res = new Scenario[counter];
  System.arraycopy(next,0,res,0,counter);
  return res;
}
```

Um ein neues Schachbrett zu erzeugen, haben wir mit Hilfe der Methode `clone()` eine exakte Kopie des alten Brettes erzeugt. Die Methode `clone()` ist Bestandteil der Klasse `java.lang.Object` und muss von uns für unsere Klasse natürlich maßgeschneidert werden:

```
/** Klont dieses Objekt */
public Object clone() {
  BoardOfKnights res = new BoardOfKnights();
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      res.setContent(i,j,getContent(i,j));
  return res;
}
```

Unsere Klasse kann somit Objekte **klonen**. Gemäß Konvention müssen wir somit auch das Interface `java.lang.Cloneable` implementieren:

```
public class BoardOfKnights extends Chessboard
  implements Scenario, Cloneable{
```

Mit diesem letzten Schritt ist unser `BoardOfKnights` endgültig fertig. Wir haben somit die `Scenario`-Objekte für Springer- und Achtdamenproblem geschaffen, so dass wir unseren Problemlöser hiermit füttern können. Es fehlt uns also nur noch die Visualisierung auf dem Bildschirm.

1.3.4 Die Klassen `EightQueenProblem` und `KnightSwitchProblem`

Kommen wir nun zur grafischen Darstellung unserer Probleme. Wir vereinbaren eine Klasse `EightQueenProblem`, die die Lösung unseres Achtdamenproblems darstellen wird. Wir vereinbaren ferner eine Instanzvariable `solver`, in der wir die Lösung des Problems hinterlegen:

```
package de.uni.karlsruhe.aifb.prog2.solutions;

import de.uni.karlsruhe.aifb.prog2.chess.ChessboardDisplay;

import Prog1Tools.GameEngine;

/** Dieses Programm stellt alle moeglichen Loesungen
 *  des Achtdamen-Problems dar.
 */
public class EightQueenProblem extends ChessboardDisplay{

  /** Der Solver, der die Loesungen beinhaltet. */
  private Solver solver;
```

In unserem Konstruktor setzen wir den Inhalt der Variablen `solver`. Ferner setzen wir das darzustellende Feld (die Variable `board` der Superklasse) auf die erste gefundene Lösung und weisen den `Solver` an, bereits nach der nächsten Lösung zu suchen. Da wir diesen Mechanismus später wieder brauchen werden, lagern wir ihn in eine Hilfsmethode aus:

```
/** Konstruktor. Der uebergebene Solver soll
 *  alle Loesungen des Problems beinhalten.
 */
public EightQueenProblem(Solver solver) {
  // Rufe den Konstruktor der Superklasse
  // mit der Start-Situation aus
  super((BoardOfQueens) solver.get(0));
  // Speichere den Solver ab
  this.solver = solver;
  // Nun schalte zur naechsten Loesung
  toNext();
}

/** Schalte zur naechsten Loesung */
private void toNext() {
  if (solver.isSolution())
    chessboard =
      (BoardOfQueens) solver.get(solver.getSteps());
  solver.toNext();
}
```

Wenn der Bentzer auf den Feuer-Knopf drücken sollte, schalten zur nächsten Lösung. Die Methode `firePressed()` ruft also ebenfalls die `toNext()`-Methode auf:

```
/** Signalisert, dass der Feuer-Button gedrueckt wurde.
  **/
public void firePressed() {
  toNext();
}
```

Ferner überschreiben wir nun noch die Methoden `getFireLabel()` und `getGameName()`, um die Beschriftung des Fensters an unsere Wünsche anzupassen:

```
/** Gibt den Text zurueck, der aktuell auf dem
  * Feuer-Button stehen soll.
  **/
public String getFireLabel() {
  return solver.isSolution() ? "Naechste Loesung"
                         : "Keine weitere Loesung";
}

/** Gibt den Namen des Spiels als String zurueck.
  **/
public String getGameName() {
  return "AchtDamenProblem";
}
```

Zu guter Letzt definieren wir noch eine `main`-Methode, in der unsere Klasse instantiiert wird:

```
/** Hauptprogramm */
public static void main(String[] args) {
  Solver solver = new Solver(new BoardOfQueens(),8);
  new GameEngine(new EightQueenProblem(solver));
}
```

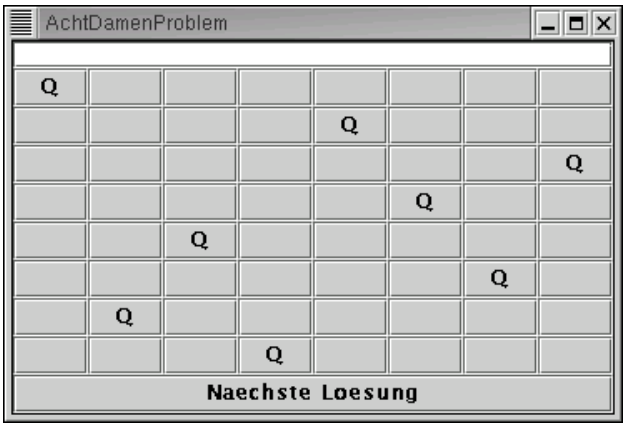

Abbildung 1.14: Eine Lösung des Achtdamenproblems

Mit Hilfe dieser Methode können wir nun sämtliche Lösungen des Problems auf dem Bildschirm anzeigen lassen. Abbildung 1.14 zeigt, wie die erste gefundene Lösung auf dem Bildschirm ausgegeben wird.
Die Definition der Klasse `KnightSwitchProblem` erfolgt analog, so dass wir an dieser Stelle den Code im Ganzen betrachten wollen.

```
1  package de.uni.karlsruhe.aifb.prog2.solutions;
2
3  import de.uni.karlsruhe.aifb.prog2.chess.ChessboardDisplay;
4
5  import Prog1Tools.GameEngine;
6
7  /** Dieses Programm berechnet eine Loesung des
8   *  im Buch beschriebenen Springerproblems
9   */
10 public class KnightSwitchProblem extends ChessboardDisplay{
11
12   /** Der Solver, der die Loesungen beinhaltet. */
13   private Solver solver;
14
15   /** Bei welchem Zug sind wir gerade? */
16   private int move;
17
18   /** Konstruktor. Der uebergebene Solver soll alle
19    *  Loesungen des Problems beinhalten.
20    */
21   public KnightSwitchProblem(Solver solver) {
22     // Rufe den Konstruktor der Superklasse
23     // mit der Start-Situation auf
24     super((BoardOfKnights) solver.get(0));
25     // Setze die Instanzvariablen
26     this.solver = solver;
27     this.move = 0;
28   }
```

```
29
30    /** Gibt den Text zurueck, der aktuell auf dem
31      * Feuer-Button stehen soll.
32      **/
33    public String getFireLabel() {
34      return (move == 0) ? "Ausgangssituation"
35        : ("Zug Nr. " + move);
36    }
37
38    /** Gibt den Namen des Spiels als String zurueck.
39      **/
40    public String getGameName() {
41      return "Springer-Problem";
42    }
43
44    /** Gibt den Text zurueck, der in der aktuellen
45      * Runde im Meldefenster stehen soll. Hier geben wir die
46      * Anzahl der Schritte
47      * zur Loesung an.
48      **/
49    public String getMessages() {
50      return solver.isSolution() ?
51        ("Problem ist in " + solver.getSteps() + " Zuegen loesbar.") :
52        "Problem konnte nicht geloest werden.";
53    }
54
55    /** Signalisert, dass der Feuer-Button gedrueckt wurde.
56      **/
57    public void firePressed() {
58      move = (move + 1) % (solver.getSteps() + 1);
59      chessboard = (BoardOfKnights) solver.get(move);
60    }
61
62    /** Hauptprogramm */
63    public static void main(String[] args) {
64      System.out.println("Finde Loesung...");
65      Solver solver = new Solver(BoardOfKnights.createInitial(),34);
66      System.out.println("Problem ist loesbar in " +
67        solver.getSteps() + " Zuegen.");
68      System.out.println("Stelle Loesung dar.");
69      new GameEngine(new KnightSwitchProblem(solver));
70    }
71
72  }
```

Abbildung 1.15 zeigt das Ergebnis unserer Arbeit. Beachten Sie, dass die Anzahl der Schritte bis zur Lösung ein wenig von dem Parameter abhängt, den Sie in Zeile 65 einstellen. Der gegebene Wert `34` ist deshalb optimal, weil wir wissen, dass das Problem in dieser Zahl von Schritten gelöst werden kann. Bei einem höheren Wert würde eventuell eine andere, etwas umständlichere Lösung gefunden.

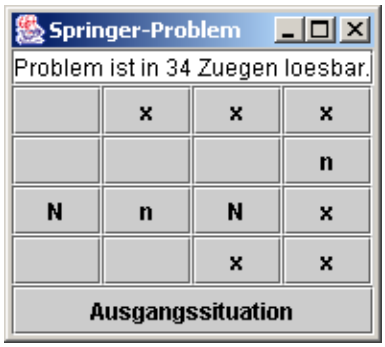

Abbildung 1.15: Die Lösung des Springer-Problems

1.3.5 Übungsaufgaben

Aufgabe 1.2

Wie lässt sich herausfinden, ob das Springer-Problem auch in weniger als 34 Zügen lösbar ist?

Aufgabe 1.3

Modifizieren Sie die Klasse `EightQueenProblem` so, dass sie nach dem Anzeigen der letzten möglichen Lösung wieder von vorne beginnt. Lesen Sie zu diesem Zweck anfangs alle möglichen Lösungen aus und speichern Sie diese in einem Feld.

1.4 Zusammenfassung

Nach diversen Seiten Code und einigen Stolpersteinen haben wir es geschafft: Nur mit den Programmierkenntnissen aus dem ersten Band haben wir das Springer-Problem gelöst!
Im Laufe dieses Entwicklungsprozesses haben wir eine der wichtigsten Fragen des ersten Bandes anhand von Beispielen wiederholt und vertieft: wie modelliert man ein Problem aus der realen Welt so, dass es sich auf dem Computer darstellen und bewältigen lässt?
Um die doch sehr mächtige Aufgabe lösen zu können, haben wir sie in drei Teilgebiete aufgeteilt:

1. Wir haben ein Schachspiel auf dem Computer realisiert und somit ein Modell für die verschiedenen Schachfiguren und ihre gegenseitige Beziehung geschaffen.

2. Wir haben auf einer abstrakten Ebene eine Methodik entwickelt, wie der Computer Denksportaufgaben lösen kann.
3. Wir haben die beiden Modelle miteinander verbunden und mit ihrer Hilfe das Springer-Problem gelöst.

Unsere Entwüfe standen hierbei unter der Idee, entwickelte Klassen wiederverwenden zu können. So konnten wir die `GameModel`-`GameEngine`-Beziehung aus dem ersten Band einsetzen, um die grafische Oberfläche zu gestalten. Und auch unsere neuen Klassen waren so allgemein, dass wir im Zuge der Entwicklung verschiedene andere Anwendungen (ein virtuelles Schachbrett, die Lösung des Achtdamen-Problems) mit denselben Basisklassen realisieren konnten. Ohne hier schon zu viel vorwegnehmen zu wollen – die Klassen `Solver` und `Scenario` werden uns auch an anderer Stelle wieder begegnen.
Im Verlauf der Implementierung haben wir aber auch feststellen müssen, wie begrenzt unser Wissen an manchen Stellen doch ist. So haben wir bei der Realisierung der `CollectionOfFailures` etwa auf sehr umständliche Art und Weise Objekte in Feldern halten müssen. Das Hinzufügen oder Löschen war hierbei mit sehr viel Programmieraufwand vebunden (und somit entsprechend fehleranfällig). Ähnliches ist uns auch bei der Realisierung der `getNextSteps()`-Methoden passiert.
Zusammengefasst lässt sich also sagen, dass wir mit unseren Kenntnissen bereits eine Vielzahl von Anforderungen an einen kommerziellen Programmierer meistern können. Es kann aber auf keinen Fall schaden, noch mehr dazuzulernen …

Ergänzung 2

Annotations in Java 5.0

Wird auf der Web-Seite bereitgestellt.