Anhang B (Online)
Spielende Hamster

In den Kapiteln dieses Buches haben Sie alle wichtigen Konzepte der objektorientierten Programmierung kennen gelernt. In diesem Anhangskapitel werden die Konzepte nun an einem vertiefenden, etwas komplexeren Beispiel wiederholt.

Das Beispiel stammt aus der Welt der Spiele-Programmierung. Wir werden den Hamstern das Spielen beibringen. Nach diesem Kapitel werden die Hamster in der Lage sein, gegen uns Menschen oder gegen andere Hamster so genannte *Zwei-Spieler-Strategiespiele* wie Schach, Mühle, Dame, 4-Gewinnt oder Reversi zu spielen, und die Hamster werden so gut sein, dass wir als Menschen kaum noch eine Chance haben, gegen sie zu gewinnen. Sie als Programmierer werden lernen, wie man derartige Spielstrategien programmiert.

Das Spiel, das wir uns als Beispiel ausgesucht haben, ist ein afrikanisches Brettspiel namens *Kalah*. Die Spielregeln werden in Abschnitt 1 erläutert. In Abschnitt 2 wird dann zunächst ein Programm entwickelt, das es zwei menschlichen Spielern ermöglicht, gegeneinander Kalah zu spielen. Das Hamster-Territorium ist dabei das Spielfeld und die Hamster führen die ihnen von den Spielern mitgeteilten Spielzüge aus. Abschnitt 3 stellt anschließend einen allgemeingültigen Algorithmus zur Ermittlung guter Spielzüge für Zwei-Spieler-Strategiespiele vor. Dieser Algorithmus wird in Abschnitt 4 an das Kalah-Spiel angepasst. Konkret wird in Abschnitt 4 das in Abschnitt 2 entwickelte Programm so erweitert, dass anschließend Menschen gegen Menschen, Menschen gegen Hamster und auch Hamster gegen Hamster Kalah spielen können. Zum Testen, ob Sie die Thematik dieses Kapitels verstanden haben, werden Ihnen in Abschnitt 5 einige Spiele vorgestellt, die Sie selbst programmieren können.

B.1 Das Spiel Kalah

Kalah ist ein afrikanisches Brettspiel für zwei Spieler. Es besteht aus einem Brett mit zwei Reihen von je sechs Mulden, in denen anfangs je sechs Steine liegen, und zwei weiteren Mulden, die *Kalah* heißen und anfangs leer sind. Spieler A gehören die oberen Löcher und die linke Kalah, Spieler B die unteren Löcher und die rechte Kalah (siehe auch Abbildung B.1).

Ziel der beiden Spieler ist es, im Laufe eines Spiels möglichst viele Steine in die eigene Kalah zu bringen. Spieler A beginnt das Spiel. Ein Spielzug läuft dabei wie folgt ab:

- Der Spieler wählt eine seiner Mulden (nicht seine Kalah) aus, die nicht leer ist, entleert sie und verteilt auf die entgegen dem Uhrzeigersinn folgenden Mulden je einen Stein, wobei er die Kalah des Gegners auslässt.
- Landet der letzte Stein in einer eigenen leeren Mulde (keine Kalah) und ist die gegenüberliegende Mulde des Gegners nicht leer, so kommen der Stein sowie alle Steine der gegenüberliegenden Mulde in die eigene Kalah.
- Landet der letzte Stein in der eigenen Kalah, so muss der Spieler noch einmal ziehen. Ansonsten ist der Gegner an der Reihe.

Das Spiel ist beendet, wenn ein Spieler an der Reihe ist, aber keinen erlaubten Zug mehr ausführen kann, d.h. alle seine Mulden leer sind. Der andere Spieler darf dann alle Steine, die noch in seinen Mulden liegen, in die

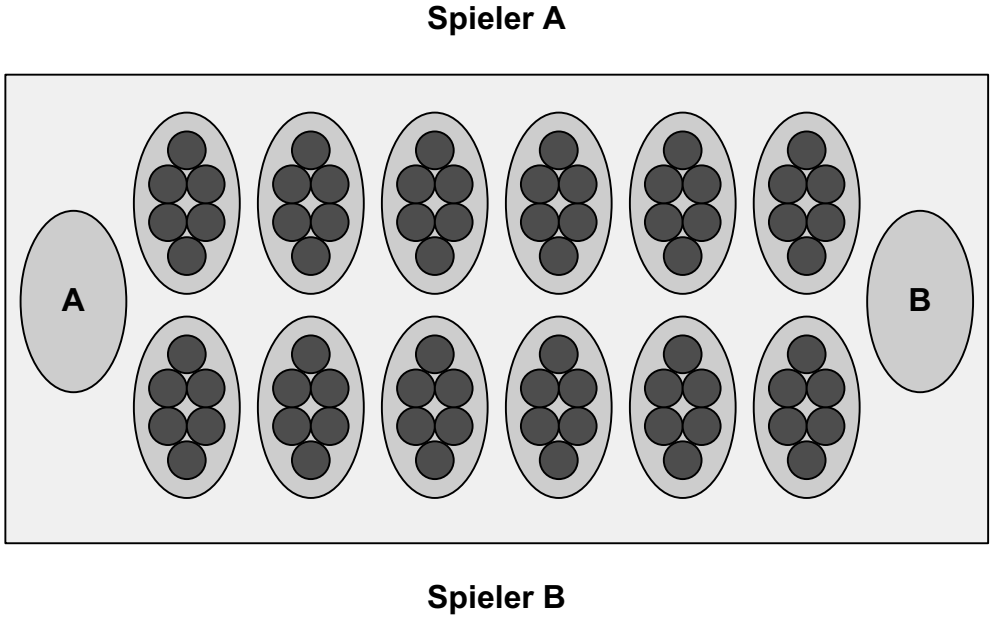

Abbildung B.1: Kalah-Spielbrett

eigene Kalah legen. Gewonnen hat der Spieler, der anschließend mehr Steine in seiner Kalah liegen hat. Bei Gleichstand endet das Spiel mit einem Unentschieden.

Bevor Sie sich ans Programmieren machen, spielen Sie das Kalah-Spiel doch mal zunächst gegen einen Freund oder eine Freundin. Anstelle von Steinen kann man gut auch Gummibärchen nehmen. Am Ende des Spiels darf dann jeder Spieler die Gummibärchen in seiner Kalah essen.

B.2 Menschen-Kalah

In diesem Abschnitt wird ein Programm entwickelt, in dem die Hamster zwei menschlichen Spielern helfen, gegeneinander das Kalah-Spiel zu spielen. Gespielt wird dabei auf einem genügend großen mauerlosen Hamster-Territorium. Gespielt wird auch nicht mit Steinen, sondern natürlich mit „leckeren“ Körnern. Abbildung B.2 skizziert das Hamster-Territorium, bevor das eigentliche Spielen beginnt.

Zunächst benötigen wir eine Klasse `Spielbrett`, die ein Kalah-Spielbrett repräsentiert. Die interne Datenstruktur dieser Klasse bildet ein `int`-Array `mulden`, in dem die Körneranzahl pro Mulde (inklusive der beiden Kalahs) gespeichert wird. Abbildung B.2 deutet an, welche Kachel des Territoriums welcher Array-Komponente entspricht. Die Klasse `Spielbrett` stellt eine Menge an Methoden zur Verfügung, um den aktuellen Zustand eines Spielbretts abfragen zu können.

Von besonderer Bedeutung ist jedoch die Methode `fuehreSpielzugAus`. Dieser wird als zweiter Parameter ein Objekt einer Klasse `Spielzug` übergeben, die Kalah-Spielzüge repräsentiert. Ein Spielzug besteht dabei aus der Nummer einer Mulde, wobei die Nummerierung dem Index der Array-Komponenten entspricht, d.h. Spieler A besitzt die Mulden mit den Nummer 0 bis 5, Spieler B die Mulden mit den Nummern 7 bis 12, Kalah A hat die Nummer 6 und Kalah B die Nummer 13. Die Methode `fuehreSpielzugAus` führt entsprechend der Kalah-Spielregeln den ihr übergebenen Spielzug auf der internen Datenstruktur aus.

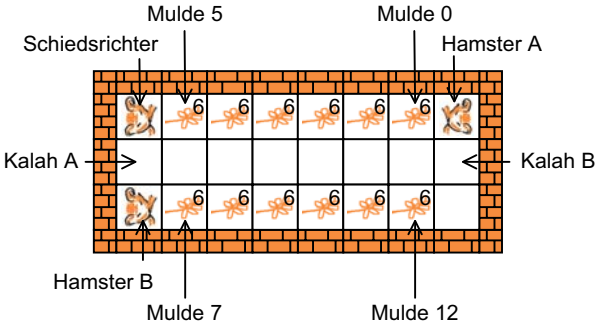

Abbildung B.2: Kalah-Hamster-Territorium

```
package kalah;

// repraesentiert ein Kalah-Spielbrett
public class Spielbrett {

    // Konstanten
    public final static int ANZAHL_MULDEN = 14;

    public final static int ANZAHL_KOERNER_PRO_MULDE = 6;

    public final static int ANZAHL_MULDEN_PRO_SPIELER =
        (Spielbrett.ANZAHL_MULDEN - 2) / 2;

    public final static int KALAH_A = // Nummer von Kalah A
    Spielbrett.ANZAHL_MULDEN_PRO_SPIELER;

    public final static int KALAH_B = // Nummer von Kalah B
    Spielbrett.KALAH_A + Spielbrett.ANZAHL_MULDEN_PRO_SPIELER
            + 1;

    public final static int ANZAHL_KOERNER = 2
            * Spielbrett.ANZAHL_MULDEN_PRO_SPIELER
            * Spielbrett.ANZAHL_KOERNER_PRO_MULDE;

    // Attribute
    protected int[] mulden;

    // repraesentiert die Mulden; der gespeicherte Wert gibt
    // jeweils an, wie viele Koerner sich in der Mulde befinden
    // Spieler A gehoren die Mulden 0 - 5, Spieler B die Mulden 7
    // - 12; Kalah A ist Mulde 6, Kalah B ist Mulde 13

    // initialisiert ein Anfangsspielbrett
    public Spielbrett() {
        this.mulden = new int[Spielbrett.ANZAHL_MULDEN];
        for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
            this.mulden[i] = Spielbrett.ANZAHL_KOERNER_PRO_MULDE;
        }
        this.mulden[Spielbrett.KALAH_A] = 0;
        for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
            this.mulden[Spielbrett.KALAH_A + 1 + i] =
                Spielbrett.ANZAHL_KOERNER_PRO_MULDE;
```

```
        }
        this.mulden[Spielbrett.KALAH_B] = 0;
    }

    // Copy-Konstruktor; erzeugt eine Kopie des uebergebenen
    // Spielbrettes
    public Spielbrett(Spielbrett brett) {
        this.mulden = new int[brett.mulden.length];
        for (int i = 0; i < this.mulden.length; i++) {
            this.mulden[i] = brett.mulden[i];
        }
    }

    // ueberprueft zwei Spielbretter auf Gleichheit
    public boolean equals(Object obj) {
        Spielbrett brett = (Spielbrett) obj;
        if (this.mulden.length != brett.mulden.length) {
            return false;
        }
        for (int i = 0; i < this.mulden.length; i++) {
            if (this.mulden[i] != brett.mulden[i]) {
                return false;
            }
        }
        return true;
    }

    // liefert die Anzahl an Koernern der Mulde mit der
    // angegebenen Nummer
    public int liefereAnzahlKoerner(int muldenNummer) {
        return this.mulden[muldenNummer];
    }

    // liefert die Anzahl an Koernern in Kalah A
    public int liefereAnzahlKoernerInKalahA() {
        return this.liefereAnzahlKoerner(Spielbrett.KALAH_A);
    }

    // liefert die Anzahl an Koernern in Kalah B
    public int liefereAnzahlKoernerInKalahB() {
        return this.liefereAnzahlKoerner(Spielbrett.KALAH_B);
    }

    // liefert die Anzahl an Koernern im angegebenen Kalah
    public int liefereAnzahlKoernerInKalah(boolean kalahA) {
        if (kalahA) {
            return this.liefereAnzahlKoernerInKalahA();
        } else {
            return this.liefereAnzahlKoernerInKalahB();
        }
    }

    // liefert die Gesamtanzahl an Koernern in den Mulden von
    // Spieler A
    public int liefereAnzahlKoernerVonSpielerA() {
        int anzahl = 0;
        for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
            anzahl = anzahl + this.mulden[i];
        }
        return anzahl;
```

```
}

// liefert die Gesamtanzahl an Koernern in den Mulden von
// Spieler B
public int liefereAnzahlKoernerVonSpielerB() {
    int anzahl = 0;
    for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
        anzahl = anzahl
                + this.mulden[Spielbrett.KALAH_A + 1 + i];
    }
    return anzahl;
}

// liefert die Gesamtanzahl an Koernern in den Mulden des
// angegebenen Spielers
public int liefereAnzahlKoernerVonSpieler(boolean spielerA) {
    if (spielerA) {
        return this.liefereAnzahlKoernerVonSpielerA();
    } else {
        return this.liefereAnzahlKoernerVonSpielerB();
    }
}

// fuehrt den angegebenen Spielzug des angegebenen Spielers
// auf dem Spielbrett aus;
// Bedingung: Der Spielzug ist korrekt;
// liefert die Nummer der Mulde, in die das letzte Korn
// gelegt wurde
public int fuehreSpielzugAus(boolean istSpielerA,
        Spielzug zug) {
    int nummer = zug.liefereMulde();
    int anzahlKoerner = this.mulden[nummer];

    // die betroffene Mulde wird geleert
    this.mulden[nummer] = 0;

    // die Koerner werden gegen den Uhrzeigersinn verteilt
    int i = 1;
    while (i <= anzahlKoerner) {
        nummer = (nummer + 1) % Spielbrett.ANZAHL_MULDEN;
        if (!((istSpielerA && nummer == Spielbrett.KALAH_B) ||
                (!istSpielerA && nummer == Spielbrett.KALAH_A))
        // die Kalah des Gegners wird uebergangen
        ) {
            this.mulden[nummer] = this.mulden[nummer] + 1;
            i = i + 1;
        }
    }

    // entsprechend den Regeln muessen evtl. einige Koerner
    // in ein Kalah uebertragen werden
    if (nummer != Spielbrett.KALAH_A
            && nummer != Spielbrett.KALAH_B
            && this.mulden[nummer] == 1
            && this.mulden[Spielbrett
                    .gegenueberMulde(nummer)] > 0) {
        if (istSpielerA && 0 <= nummer
                && nummer < Spielbrett.KALAH_A) {
            this.mulden[Spielbrett.KALAH_A] =
                this.mulden[Spielbrett.KALAH_A]
```

```
                        + this.mulden[nummer]
                        + this.mulden[Spielbrett
                                .gegenueberMulde(nummer)];
                this.mulden[nummer] = 0;
                this.mulden[Spielbrett.gegenueberMulde(nummer)] = 0;
            } else if (!istSpielerA
                    && Spielbrett.KALAH_A < nummer
                    && nummer < Spielbrett.KALAH_B) {
                this.mulden[Spielbrett.KALAH_B] =
                    this.mulden[Spielbrett.KALAH_B]
                        + this.mulden[nummer]
                        + this.mulden[Spielbrett
                                .gegenueberMulde(nummer)];
                this.mulden[nummer] = 0;
                this.mulden[Spielbrett.gegenueberMulde(nummer)] = 0;
            }
        }

        return nummer;
    }

    // packt nach Spielende uebrig gebliebene Koerner ins
    // Kalah des entsprechenden Spielers
    public void spielBeenden() {
        // zunaechst Spieler A betrachten
        int anzahl = 0;
        for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
            anzahl = anzahl + this.mulden[i];
            this.mulden[i] = 0;
        }
        this.mulden[Spielbrett.KALAH_A] = this.mulden[Spielbrett.KALAH_A]
                + anzahl;

        // jetzt Spieler B
        anzahl = 0;
        for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
            anzahl = anzahl
                    + this.mulden[Spielbrett.KALAH_A + 1 + i];
            this.mulden[Spielbrett.KALAH_A + 1 + i] = 0;
        }
        this.mulden[Spielbrett.KALAH_B] = this.mulden[Spielbrett.KALAH_B]
                + anzahl;
    }

    // liefert die Nummer der gegenueber liegenden Mulde
    public static int gegenueberMulde(int nummer) {
        return Spielbrett.ANZAHL_MULDEN - 2 - nummer;
    }
}
```

```
package kalah;

// repraesentiert einen Spielzug beim Kalah-Spiel
public class Spielzug {

    private int mulde; // Nummer einer Mulde

    // initialisiert ein Spielzug-Objekt
    public Spielzug(int muldeNummer) {
        this.mulde = muldeNummer;
```

```
    }

    // liefert die Muldennummer
    public int liefereMulde() {
        return this.mulde;
    }
}
```

Die beiden Klassen `Spielbrett` und `Spielzug` werden, wie auch die im Folgenden beschriebenen Klassen, in ein Paket namens `kalah` gepackt.

Beachten Sie weiterhin die beiden Konstanten `ANZAHL_MULDEN` und `ANZAHL_KOERNER_PRO_MULDE` der Klasse `Spielbrett`, die die Werte 14 bzw. 6 repräsentieren. Diese Konstanten werden konsequent in allen Klassen genutzt, so dass Sie durch Änderung der Konstanten, ohne weitere Änderungen am Sourcecode vornehmen zu müssen, Kalah auch mal bspw. mit 16 Mulden oder 8 Körnern pro Mulde spielen können.

Weiterhin gibt es eine Klasse `Spielregeln`. Objekte dieser Klasse erlauben über entsprechende Methoden, Spielzüge bzw. den Spielzustand zu überprüfen. Grundlage ist dabei ein Spielbrett, das einem Spielregel-Objekt im Konstruktor übergeben wird.

```
package kalah;

// repraesentiert die Spielregeln des Kalah-Spiels
public class Spielregeln {

    private Spielbrett brett; // das zu kontrollierende Brett

    // initialisiert die Regeln; uebergeben wird das zu
    // kontrollierende Spielbrett
    public Spielregeln(Spielbrett brett) {
        this.brett = brett;
    }

    // kontrolliert, ob der angegebene Spielzug des angegebenen
    // Spielers korrekt ist
    public boolean istSpielzugOk(boolean istSpielerA,
            Spielzug zug) {
        if (zug == null) {
            return false;
        }

        // es muss eine Mulde sein, die dem Spieler gehoert
        int nummer = zug.liefereMulde();
        if (istSpielerA) {
            if (!(0 <= nummer && nummer < Spielbrett.KALAH_A)) {
                return false;
            }
        } else {
            if (!(Spielbrett.KALAH_A < nummer &&
                    nummer < Spielbrett.KALAH_B)) {
                return false;
            }
        }

        // die Mulde darf nicht leer sein
        if (this.brett.liefereAnzahlKoerner(nummer) == 0) {
            return false;
        }

        // Spielzug ist ok
```

```
        return true;
    }

    // liefert die Information, ob A als naechster ziehen muss;
    // uebergeben wird die Information, ob A als letzter gezogen
    // hat, sowie die Nummer der Mulde, in der der letzte Zug
    // geendet hat
    public boolean istAAmZug(boolean aLetzterSpieler, int mulde) {
        if (aLetzterSpieler) {
            return mulde == Spielbrett.KALAH_A;
        } else {
            return mulde != Spielbrett.KALAH_B;
        }
    }

    // ueberprueft; ob das Spiel beendet ist; uebergeben wird die
    // Information, ob A am Zug ist
    public boolean istSpielBeendet(boolean istA) {
        // das Spiel ist beendet, wenn alle Mulden des Spielers
        // leer
        // sind
        return this.brett.liefereAnzahlKoernerVonSpieler(istA) == 0;
    }

    // ueberprueft auf Unentschieden;
    // Bedingung: Spiel ist beendet und die uebrig gebliebenen
    // Koerner wurden in das entsprechende Kalah gepackt
    public boolean istUnentschieden() {
        return this.brett.liefereAnzahlKoernerInKalahA() == this.brett
                .liefereAnzahlKoernerInKalahB();
    }

    // liefert true, wenn Spieler A Sieger ist;
    // liefert false, wenn Spieler B Sieger ist;
    // Bedingung: Spiel ist beendet und nicht unentschieden
    public boolean istSpielerASieger() {
        return this.brett.liefereAnzahlKoernerInKalahA() > this.brett
                .liefereAnzahlKoernerInKalahB();
    }

    // liefert das Spielbrett
    public Spielbrett getBrett() {
        return this.brett;
    }
}
```

Zur Beschreibung der Spieler definieren wir ein Interface namens `Spieler`. Dieses beinhaltet insgesamt fünf Methoden. Von besonderem Interesse ist die Methode `naechsterSpielzug`, bei deren Ausführung der Spieler den nächsten Spielzug ermitteln, im Territorium ausführen und liefern soll.

```
package kalah;

// repraesentiert einen Kalah-Spieler
public interface Spieler {

    // Ueberpruefung, welcher Spieler es ist
    public boolean istSpielerA();

    // liefert A oder B, je nachdem, welcher Spielertyp es ist
    public String liefereSpielerTyp();
```

```
    // ermittelt den naechsten Spielzug, fuehrt ihn aus und
    // liefert ihn
    public Spielzug naechsterSpielzug();

    // sammelt uebrig gebliebene Koerner ein und packt sie ins
    // eigene Kalah
    public void sammleResteUndBringSieZumKalah();

    // beliebige Aktion, die ein Spieler nach Ende des Spiels
    // durchfuehrt; uebergeben wird die Information, ob der
    // Spieler gewonnen hat
    public void spielBeenden(boolean istSieger);
}
```

Implementiert wird dieses Interface von der erweiterten Hamster-Klasse `MenschHamster`, die von einer erweiterten Hamster-Klasse `AllroundHamster` abgeleitet ist. Die Methode `naechsterSpielzug` ist dabei so implementiert, dass ein Hamster dieser Klasse den menschlichen Spieler nach dem nächsten Spielzug – eine Muldennummer – fragt. Anschließend überprüft er mit Hilfe der Klasse `Spielregeln`, ob der Spielzug korrekt ist und führt ihn auf dem Territorium aus.

```
package kalah;

public class AllroundHamster extends Hamster {

    // initialisiert einen neuen AllroundHamster mit den
    // uebergebenen Werten
    AllroundHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    // initialisiert einen neuen AllroundHamster mit den
    // Attributwerten eines bereits existierenden Hamsters
    AllroundHamster(Hamster existierenderHamster) {
        super(existierenderHamster);
    }

    // der Hamster dreht sich um 180 Grad
    void kehrt() {
        this.linksUm();
        this.linksUm();
    }

    // der Hamster dreht sich nach rechts
    void rechtsUm() {
        this.kehrt();
        this.linksUm();
    }

    // der Hamster laeuft "anzahl" Schritte, maximal jedoch
    // bis zur naechsten Mauer; geliefert wird die tatsaechliche
    // Anzahl gelaufener Schritte
    int vor(int anzahl) {
        int schritte = 0;
        while (this.vornFrei() && anzahl > 0) {
            this.vor();
            schritte = schritte + 1;
            anzahl = anzahl - 1;
        }
        return schritte;
```

```
}

// der Hamster legt "anzahl" Koerner ab, maximal jedoch
// so viele, wie er im Maul hat; geliefert wird die
// tatsaechliche Anzahl abgelegter Koerner
int gib(int anzahl) {
    int abgelegteKoerner = 0;
    while (!this.maulLeer() && anzahl > 0) {
        this.gib();
        abgelegteKoerner = abgelegteKoerner + 1;
        anzahl = anzahl - 1;
    }
    return abgelegteKoerner;
}

// der Hamster frisst "anzahl" Koerner, maximal jedoch
// so viele, wie auf der aktuellen Kachel liegen; geliefert
// wird die tatsaechliche Anzahl gefressener Koerner
int nimm(int anzahl) {
    int gefresseneKoerner = 0;
    while (this.kornDa() && anzahl > 0) {
        this.nimm();
        gefresseneKoerner = gefresseneKoerner + 1;
        anzahl = anzahl - 1;
    }
    return gefresseneKoerner;
}

// der Hamster legt alle Koerner, die er im Maul hat,
// auf der aktuellen Kachel ab; geliefert wird die Anzahl
// abgelegter Koerner
int gibAlle() {
    int abgelegteKoerner = 0;
    while (!this.maulLeer()) {
        this.gib();
        abgelegteKoerner = abgelegteKoerner + 1;
    }
    return abgelegteKoerner;
}

// der Hamster frisst alle Koerner auf der aktuellen Kachel;
// geliefert wird die Anzahl gefressener Koerner
int nimmAlle() {
    int gefresseneKoerner = 0;
    while (this.kornDa()) {
        this.nimm();
        gefresseneKoerner = gefresseneKoerner + 1;
    }
    return gefresseneKoerner;
}

// der Hamster laeuft bis zur naechsten Mauer;
// geliefert wird die Anzahl ausgefuehrter Schritte
int laufeZurWand() {
    int schritte = 0;
    while (this.vornFrei()) {
        this.vor();
        schritte = schritte + 1;
    }
    return schritte;
```

```
}

// der Hamster testet, ob links von ihm die Kachel frei ist
boolean linksFrei() {
    this.linksUm();
    boolean frei = this.vornFrei();
    this.rechtsUm();
    return frei;
}

// der Hamster testet, ob rechts von ihm die Kachel frei ist
boolean rechtsFrei() {
    this.rechtsUm();
    boolean frei = this.vornFrei();
    this.linksUm();
    return frei;
}

// der Hamster testet, ob hinter ihm die Kachel frei ist
boolean hintenFrei() {
    this.kehrt();
    boolean frei = this.vornFrei();
    this.kehrt();
    return frei;
}

// der Hamster dreht sich so lange um, bis er in die
// uebergebene Blickrichtung schaut
void setzeBlickrichtung(int richtung) {
    while (this.getBlickrichtung() != richtung) {
        this.linksUm();
    }
}

// der Hamster laeuft zur Kachel (reihe/spalte);
// Voraussetzung: die Kachel existiert und es befinden sich
// keine Mauern im Territorium bzw. auf dem gewaehlten Weg
void gotoKachel(int reihe, int spalte) {
    // zunaechst begibt sich der Hamster in die entsprechende
    // Reihe
    if (reihe > this.getReihe()) {
        this.setzeBlickrichtung(Hamster.SUED);
    } else {
        this.setzeBlickrichtung(Hamster.NORD);
    }
    while (reihe != this.getReihe()) {
        this.vor();
    }

    // nun begibt sich der Hamster in die entsprechende
    // Spalte
    if (spalte > this.getSpalte()) {
        this.setzeBlickrichtung(Hamster.OST);
    } else {
        this.setzeBlickrichtung(Hamster.WEST);
    }
    while (spalte != this.getSpalte()) {
        this.vor();
    }
}
```

```
}
```

```
package kalah;

// repraesentiert einen von einem Menschen gesteuerten
// Kalah-spielenden Hamster
public class MenschHamster extends AllroundHamster implements
        Spieler {
    // Attribute
    protected boolean istSpielerA; // Spieler A oder B?

    protected Spielregeln regeln; // die Spielregeln

    // initialisiert einen SpielHamster; uebergeben werden die
    // Spielregeln sowie die Information, ob der Hamster
    // Spieler A oder B ist,
    public MenschHamster(boolean istSpielerA, Spielregeln regeln) {
        // interne Initialisierungen
        super(0, 0, Hamster.OST, 0);
        this.istSpielerA = istSpielerA;
        this.regeln = regeln;

        // in Ausgangsposition begeben
        this.begibDichAufAusgangsKachel();
    }

    // Ueberpruefung, welcher Spieler es ist
    public boolean istSpielerA() {
        return this.istSpielerA;
    }

    // liefert A oder B, je nachdem, welcher Spielertyp es ist
    public String liefereSpielerTyp() {
        if (this.istSpielerA) {
            return "A";
        } else {
            return "B";
        }
    }

    // ermittelt den naechsten Spielzug, fuehrt ihn aus und
    // liefert ihn
    public Spielzug naechsterSpielzug() {
        // naechsten Spielzug beim Benutzer erfragen
        int mulde = this.liesZahl("Spieler "
                + this.liefereSpielerTyp() + ": Welche Mulde?");
        Spielzug zug = new Spielzug(mulde);
        boolean ok = this.regeln.istSpielzugOk(this.istSpielerA,
                zug);
        while (!ok) {
            mulde = this.liesZahl("Fehler!\n Spieler "
                    + this.liefereSpielerTyp()
                    + ": Bitte korrekte Mulde angeben!");
            zug = new Spielzug(mulde);
            ok = this.regeln
                    .istSpielzugOk(this.istSpielerA, zug);
        }

        // anschliessend wird der Spielzug im Hamster-Territorium
        // ausgefuehrt
```

```
        this.fuehreSpielzugAus(zug);

        // der ausgefuehrte Spielzug wird geliefert
        return zug;
    }

    // sammelt uebrig gebliebene Koerner ein und packt sie ins
    // eigene Kalah
    public void sammleResteUndBringSieZumKalah() {
        this.vor();
        int anzahl = 0;
        for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
            anzahl = anzahl + this.nimmAlle();
            this.vor();
        }
        this.linksUm();
        this.vor();
        this.gibAlle();
    }

    // frisst zur Belohung alle Koerner seiner Kalah
    public void spielBeenden(boolean istSieger) {
        // sitzt bereits im Kalah
        this.nimmAlle(); // auch wenn er nicht gewonnen hat
    }

    // interne Methoden

    // der Spielzug wird im Hamster-Territorium ausgefuehrt
    protected void fuehreSpielzugAus(Spielzug zug) {
        this.gotoMulde(zug.liefereMulde());
        int anzahl = this.nimmAlle();
        this.verteileKoerner(anzahl, zug.liefereMulde());
        this.begibDichAufAusgangsKachel();
    }

    // laeuft zur angegebenen Mulde
    protected void gotoMulde(int muldenNummer) {
        if (this.istSpielerA) {
            this.vor(muldenNummer + 1);
        } else {
            this.vor(muldenNummer
                    - Spielbrett.ANZAHL_MULDEN_PRO_SPIELER);
        }
    }

    // verteilt die aufgenommenen Koerner gemaess der
    // Kalah-Spielregeln
    protected void verteileKoerner(int anzahl, int aktuelleMulde) {
        // zunaechst entgegen dem Uhrzeigersinn jeweils ein Korn
        // pro Mulde ablegen
        while (anzahl > 0) {
            if (aktuelleMulde == Spielbrett.KALAH_A - 1
                    || aktuelleMulde == Spielbrett.KALAH_B - 1) {
                this.vor();
                this.linksUm();
                this.vor();
                if (this.istSpielerA
                        && aktuelleMulde == Spielbrett.KALAH_A - 1
                        || !this.istSpielerA
```

```
                    && aktuelleMulde == Spielbrett.KALAH_B - 1) {
                // Kalah des Gegners auslassen
                this.gib();
                anzahl = anzahl - 1;
            }
        } else if (aktuelleMulde == Spielbrett.KALAH_A
                || aktuelleMulde == Spielbrett.KALAH_B) {
            this.vor();
            this.linksUm();
            this.vor();
            this.gib();
            anzahl = anzahl - 1;
        } else {
            this.vor();
            this.gib();
            anzahl = anzahl - 1;
        }
        aktuelleMulde = (aktuelleMulde + 1)
                % Spielbrett.ANZAHL_MULDEN;
    }

    // nun ueberpruefen, ob evtl. einige Koerner ins eigene
    // Kalah abgelegt werden duerfen
    if (this.istSpielerA
            && 0 <= aktuelleMulde
            && aktuelleMulde < Spielbrett.KALAH_A
            && Territorium.getAnzahlKoerner(this.getReihe(),
                    this.getSpalte()) == 1
            && Territorium.getAnzahlKoerner(
                    this.getReihe() + 2, this.getSpalte()) > 0) {
        this.nimm(); // das eine Korn der aktuellen Mulde
        // fressen
        this.linksUm();
        this.vor();
        this.vor();
        // gegenueberliegende Mulde leer fressen
        this.nimmAlle();
        this.gotoKalahA();
        this.gibAlle(); // in eigenes Kalah ablegen
    } else if (!this.istSpielerA
            && Spielbrett.KALAH_A < aktuelleMulde
            && aktuelleMulde < Spielbrett.KALAH_B
            && Territorium.getAnzahlKoerner(this.getReihe(),
                    this.getSpalte()) == 1
            && Territorium.getAnzahlKoerner(
                    this.getReihe() - 2, this.getSpalte()) > 0) {
        this.nimm(); // das eine Korn der aktuellen Mulde
        // fressen
        this.linksUm();
        this.vor();
        this.vor();
        // gegenueberliegende Mulde leer fressen
        this.nimmAlle();
        this.gotoKalahB();
        this.gibAlle(); // in eigenes Kalah ablegen
    }
}

// in Ausgangsposition begeben
protected void begibDichAufAusgangsKachel() {
```

```
        if (this.istSpielerA) {
            this.gotoKachel(0,
                    Spielbrett.ANZAHL_MULDEN_PRO_SPIELER + 1);
            this.setzeBlickrichtung(Hamster.WEST);
        } else {
            this.gotoKachel(2, 0);
            this.setzeBlickrichtung(Hamster.OST);
        }
    }

    // nach Kalah A laufen
    protected void gotoKalahA() {
        this.gotoKachel(1, 0);
    }

    // nach Kalah B laufen
    protected void gotoKalahB() {
        this.gotoKachel(1,
                Spielbrett.ANZAHL_MULDEN_PRO_SPIELER + 1);
    }
}
```

Schließlich gibt es eine erweiterte Hamster-Klasse `SchiedsrichterHamster`. Ein Hamster dieser Klasse stellt den regelkonformen Ablauf eines Kalah-Spiels sicher. In seiner Methode `initialisiereSpiel` baut er auf dem Territorium das anfängliche Spielfeld auf und erzeugt als Spieler zwei Mensch-Hamster. Intern sind einem Schiedsrichter-Hamster die Spielregeln und ein internes Objekt der Klasse `Spielbrett` zugeordnet, das den aktuellen Spielzustand repräsentiert. Über die Methode `istSpielzugOk` kontrolliert der Schiedsrichter-Hamster, ob die von den spielenden Hamstern auf dem Territorium durchgeführten Aktionen regelkonform waren. Dies überprüft er zum einen auf seinem internen Spielbrett-Objekt, und zum anderen, indem er das Territorium abläuft und jeweils die Körner auf den Mulden-Kacheln zählt. Des Weiteren kann der Schiedsrichter-Hamster feststellen, ob ein Spiel beendet ist, und er kann den Sieger liefern.

```
package kalah;

// repraesentiert einen Hamster als Schiedsrichter des
// Kalah-Spiels
public class SchiedsrichterHamster extends AllroundHamster {
    // Attribute
    private Spielbrett brett; // internes Spielbrett

    private Spielregeln regeln; // Spielregeln

    private Spieler spielerA, spielerB; // die Spieler

    // Hilfsattribute
    private Spieler aktuellerSpieler;

    private int letzteMulde;

    // Konstruktor
    public SchiedsrichterHamster() {
        super(0, 0, Hamster.OST, Spielbrett.ANZAHL_KOERNER);
    }

    // bereitet das Spiel vor
    public void initialisiereSpiel() {
        this.brett = new Spielbrett();
        this.regeln = new Spielregeln(this.brett);
        this.aktuellerSpieler = null;
```

```
        this.erstelleSpielfeld();
        this.spielerA = erzeugeSpielerA(this.regeln);
        this.spielerB = erzeugeSpielerB(this.regeln);
    }

    // liefert Spieler A
    public Spieler getSpielerA() {
        return this.spielerA;
    }

    // liefert Spieler B
    public Spieler getSpielerB() {
        return this.spielerB;
    }

    // ueberprueft den Spielzug eines Spielers
    public boolean istSpielzugOk(Spieler spieler, Spielzug zug) {
        // merkt sich den aktuellen Spieler und Spielzug
        this.aktuellerSpieler = spieler;

        // ueberprueft den Spielzug auf dem internen Brett
        if (!this.regeln.istSpielzugOk(spieler.istSpielerA(),
                zug)) {
            return false;
        }

        // fuehrt den Spielzug auf dem internen Brett aus;
        // merkt sich die Mulde, in die das letzte Korn gelegt
        // wurde
        this.letzteMulde = this.brett.fuehreSpielzugAus(spieler
                .istSpielerA(), zug);

        // nun ueberprueft der Schiedsrichter-Hamster noch, ob
        // auch das Hamster-Territorium gemaess der Regeln in
        // Ordnung ist
        return this.kontrolliereTerritorium();
    }

    // ueberprueft, ob ein Spiel beendet ist
    public boolean istSpielBeendet() {
        boolean ende = this.regeln.istSpielBeendet(this
                .werIstAmZug().istSpielerA());
        if (ende) {
            // auf dem internen Brett wird der Endzustand
            // hergestellt
            this.brett.spielBeenden();
        }
        return ende;
    }

    // ueberprueft Endzustand von Spieler A
    public boolean istEndzustandVonSpielerAOk() {
        this.vor(Spielbrett.ANZAHL_MULDEN_PRO_SPIELER);
        this.kehrt();
        for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
            if (Territorium.getAnzahlKoerner(this.getReihe(),
                    this.getSpalte()) != 0) {
                return false;
            }
            this.vor();
```

```
        }
        this.linksUm();
        this.vor();
        if (Territorium.getAnzahlKoerner(this.getReihe(), this
                .getSpalte()) != this.brett
                .liefereAnzahlKoerner(Spielbrett.KALAH_A)) {
            return false;
        }
        this.gehZumAusgangspunkt();
        return true;
    }

    // ueberprueft Endzustand von Spieler B
    public boolean istEndzustandVonSpielerBOk() {
        this.rechtsUm();
        this.vor(2);
        this.linksUm();
        this.vor(1);
        for (int i = 0; i < Spielbrett.ANZAHL_MULDEN_PRO_SPIELER; i++) {
            if (Territorium.getAnzahlKoerner(this.getReihe(),
                    this.getSpalte()) != 0) {
                return false;
            }
            this.vor();
        }
        this.linksUm();
        this.vor();
        if (Territorium.getAnzahlKoerner(this.getReihe(), this
                .getSpalte()) != this.brett
                .liefereAnzahlKoerner(Spielbrett.KALAH_B)) {
            return false;
        }
        this.gehZumAusgangspunkt();
        return true;
    }

    // liefert den Sieger eines beendeten Spieles;
    // liefert null, wenn das Spiel mit einem Unentschieden
    // endete
    public Spieler liefereSieger() {
        if (this.regeln.istUnentschieden()) {
            return null;
        } else if (this.regeln.istSpielerASieger()) {
            return this.spielerA;
        } else {
            return this.spielerB;
        }
    }

    // liefert den naechsten Spieler, der am Zug ist
    public Spieler werIstAmZug() {
        // es beginnt Spieler A
        if (this.aktuellerSpieler == null) {
            return this.spielerA;
        }

        if (this.aktuellerSpieler.istSpielerA()
                && this.letzteMulde == Spielbrett.KALAH_A
                || !this.aktuellerSpieler.istSpielerA()
                && this.letzteMulde == Spielbrett.KALAH_B) {
```

```
            // die letzte Kugel landete im eigenen Kalah, d.h.
            // der Spieler muss nochmal ziehen
            return this.aktuellerSpieler;
        } else {
            // der Gegner ist an der Reihe
            return this.liefereGegner(this.aktuellerSpieler);
        }
    }

    // liefert den gegnerischen Spieler
    public Spieler liefereGegner(Spieler s) {
        if (s == this.spielerA) {
            return spielerB;
        } else {
            return spielerA;
        }
    }

    // liefert die Punkte von Spieler A
    public int getPunkteA() {
        return this.brett.liefereAnzahlKoernerInKalahA();
    }

    // liefert die Punkte von Spieler B
    public int getPunkteB() {
        return this.brett.liefereAnzahlKoernerInKalahB();
    }

    // interne Methoden

    // laeuft zur Kachel (0/0) und schaut nach Osten
    protected void gehZumAusgangspunkt() {
        this.gotoKachel(0, 0);
        this.setzeBlickrichtung(Hamster.OST);
    }

    // der Schiedsrichter-Hamster ueberprueft, ob auch
    // das Hamster-Territorium gemaess der Regeln in Ordnung ist;
    // d.h. ob der Zustand des Territoriums und des internen
    // Brettes identisch sind
    protected boolean kontrolliereTerritorium() {
        this.vor();
        for (int i = Spielbrett.ANZAHL_MULDEN_PRO_SPIELER - 1; i >= 0; i--) {
            if (this.brett.liefereAnzahlKoerner(i) != Territorium
                    .getAnzahlKoerner(this.getReihe(), this
                            .getSpalte())) {
                return false;
            }
            this.vor();
        }
        this.rechtsUm();
        this.vor();
        if (this.brett.liefereAnzahlKoerner(Spielbrett.KALAH_B) !=
                Territorium
                .getAnzahlKoerner(this.getReihe(), this
                        .getSpalte())) {
            return false;
        }
        this.vor();
        this.rechtsUm();
```

```
        this.vor();
        for (int i = Spielbrett.KALAH_B - 1; i >= Spielbrett.KALAH_A + 1;
                i--) {
            if (this.brett.liefereAnzahlKoerner(i) != Territorium
                    .getAnzahlKoerner(this.getReihe(), this
                            .getSpalte())) {
                return false;
            }
            this.vor();
        }
        this.rechtsUm();
        this.vor();
        if (this.brett.liefereAnzahlKoerner(Spielbrett.KALAH_A) != Territorium
                .getAnzahlKoerner(this.getReihe(), this
                        .getSpalte())) {
            return false;
        }

        // und zurueck zum Ausgangspunkt
        this.vor();
        this.rechtsUm();

        return true;
    }

    // interne Methoden

    // baut das Spielfeld auf
    protected void erstelleSpielfeld() {
        // sorgt dafuer, dass in den Mulden die exakte Anzahl an
        // Koernern liegt
        this.vor();
        for (int i = Spielbrett.ANZAHL_MULDEN_PRO_SPIELER - 1; i >= 0; i--) {
            int anzahl = Territorium.getAnzahlKoerner(this
                    .getReihe(), this.getSpalte());
            // zu viele Koerner?
            while (anzahl > Spielbrett.ANZAHL_KOERNER_PRO_MULDE) {
                this.nimm();
                anzahl = anzahl - 1;
            }
            // zu wenige Koerner?
            while (anzahl < Spielbrett.ANZAHL_KOERNER_PRO_MULDE) {
                this.gib();
                anzahl = anzahl + 1;
            }
            this.vor();
        }
        this.rechtsUm();
        this.vor();

        // Kalah B leer machen
        this.nimmAlle();

        this.vor();
        this.rechtsUm();
        this.vor();
        for (int i = Spielbrett.KALAH_B - 1; i >= Spielbrett.KALAH_A + 1;
                i--) {
            int anzahl = Territorium.getAnzahlKoerner(this
                    .getReihe(), this.getSpalte());
```

```
            // zu viele Koerner?
            while (anzahl > Spielbrett.ANZAHL_KOERNER_PRO_MULDE) {
                this.nimm();
                anzahl = anzahl - 1;
            }
            // zu wenige Koerner?
            while (anzahl < Spielbrett.ANZAHL_KOERNER_PRO_MULDE) {
                this.gib();
                anzahl = anzahl + 1;
            }
            this.vor();
        }
        this.rechtsUm();
        this.vor();

        // Kalah A leer machen
        this.nimmAlle();

        // und zurueck zum Ausgangspunkt
        this.vor();
        this.rechtsUm();
    }

    // erzeugt und liefert Spieler A
    protected Spieler erzeugeSpielerA(Spielregeln regeln) {
        return new MenschHamster(true, regeln);
    }

    // erzeugt und liefert Spieler B
    protected Spieler erzeugeSpielerB(Spielregeln regeln) {
        return new MenschHamster(false, regeln);
    }
}
```

Ein Kalah-Spiel selbst wird durch die Klasse `KalahSpiel` repräsentiert. Die Durchführung eines Spiels ist in der Methode `spielen` implementiert. Über einen Schiedsrichter-Hamster wird jeweils ermittelt, welcher Spieler als nächstes an der Reihe ist. Für diesen wird die Methode `naechsterSpielzug` aufgerufen. Anschließend kontrolliert der Schiedsrichter-Hamster, ob der Spielzug in Ordnung war und ob das Spiel beendet ist und leitet entsprechende Aktionen ein.

```
package kalah;

public class KalahSpiel {

    SchiedsrichterHamster schiedsrichter;

    // Konstruktor
    public KalahSpiel(SchiedsrichterHamster schiedsrichter) {
        this.schiedsrichter = schiedsrichter;

    }

    // Durchfuehrung eines Spiels
    public void spielen() {
        // Spielvorbereitung
        this.schiedsrichter.initialisiereSpiel();

        // es wird gemaess der Spielregeln gezogen;
        // nach jedem Spielzug kontrolliert der Schiedsrichter
        Spieler aktuellerSpieler = this.schiedsrichter
```

```
                .werIstAmZug();
        while (true) {
            Spielzug zug;
            try {
                zug = aktuellerSpieler.naechsterSpielzug();
            } catch (Exception exc) {
                // wenn waehrend der Ausfuehrung der Methode ein
                // Fehler auftritt, hat der Spieler verloren
                this.schiedsrichter
                        .schreib("Spieler "
                                + aktuellerSpieler
                                        .liefereSpielerTyp()
                                + " hat die folgende Exception erzeugt\n"
                                + " und damit verloren: "
                                + exc.toString());
                return; // Spielende
            }

            if (!this.schiedsrichter.istSpielzugOk(
                    aktuellerSpieler, zug)) {
                this.schiedsrichter.schreib("Spieler "
                        + aktuellerSpieler.liefereSpielerTyp()
                        + " hat sich nicht an die\n"
                        + "Spielregeln gehalten und damit\n"
                        + "verloren!");
                return; // Spielende
            }

            if (this.schiedsrichter.istSpielBeendet()) {
                // die verbliebenen Koerner muessen noch in die
                // Kalahs gepackt werden
                Spieler spielerA = this.schiedsrichter
                        .getSpielerA();
                spielerA.sammleResteUndBringSieZumKalah();
                if (!this.schiedsrichter
                        .istEndzustandVonSpielerAOk()) {
                    this.schiedsrichter.schreib("Spieler A"
                            + " hat sich nicht an die\n"
                            + "Spielregeln gehalten und damit\n"
                            + "verloren!");
                    return; // Spielende
                }

                Spieler spielerB = this.schiedsrichter
                        .getSpielerB();
                spielerB.sammleResteUndBringSieZumKalah();
                if (!this.schiedsrichter
                        .istEndzustandVonSpielerBOk()) {
                    this.schiedsrichter.schreib("Spieler B"
                            + " hat sich nicht an die\n"
                            + "Spielregeln gehalten und damit\n"
                            + "verloren!");
                    return; // Spielende
                }

                // Siegerehrung
                Spieler sieger = this.schiedsrichter
                        .liefereSieger();
                if (sieger == null) {
                    this.schiedsrichter
```

```
                        .schreib("Unentschieden!");
                    // zur Belohnung duerfen die Hamster noch die
                    // Koerner in ihren Kalahs fressen
                    aktuellerSpieler.spielBeenden(false);
                    Spieler gegner = this.schiedsrichter
                            .liefereGegner(aktuellerSpieler);
                    gegner.spielBeenden(false);
                } else {
                    int punkteA = this.schiedsrichter
                            .getPunkteA();
                    int punkteB = this.schiedsrichter
                            .getPunkteB();
                    this.schiedsrichter.schreib("Endstand: "
                            + punkteA + " : " + punkteB + "\n"
                            + "Spieler "
                            + sieger.liefereSpielerTyp()
                            + " hat gewonnen!");
                    // zur Belohnung duerfen die Hamster noch die
                    // Koerner in ihren Kalahs fressen
                    sieger.spielBeenden(true);
                    Spieler gegner = this.schiedsrichter
                            .liefereGegner(sieger);
                    gegner.spielBeenden(false);
                }

                return; // Spielende
            }

            // naechste Runde
            aktuellerSpieler = schiedsrichter.werIstAmZug();
        }
    }

}
```

Das Hauptprogramm ist relativ kurz. Es wird ein Schiedsrichter-Hamster erzeugt, ein Kalah-Spiel initiiert und anschließend durch Aufruf der Methode `spielen` gestartet.

```
import kalah.*;

// Hauptprogramm
void main() {
    SchiedsrichterHamster schiedsrichter = new SchiedsrichterHamster();
    KalahSpiel spiel = new KalahSpiel(schiedsrichter);
    spiel.spielen();
}
```

Bevor Sie weiterlesen, starten Sie doch zunächst einmal den Hamster-Simulator und probieren die eben vorgestellte Implementierung des Kalah-Spiels aus.

B.3 Programmierung von Spielstrategien

Bevor im nächsten Abschnitt das Kalah-Spiel so erweitert wird, dass Menschen auch gegen „intelligente“ Hamster spielen können, wird in diesem Abschnitt zunächst ein allgemeingültiger Algorithmus für die Ermittlung optimaler Spielzüge bei Zwei-Spieler-Strategiespielen vorgestellt. Zwei-Spieler-Strategiespiele, wie Schach, Mühle, Dame, Reversi oder Kalah, sind Spiele, bei denen zwei Spieler in der Regel abwechselnd

Spielzüge ausführen, wissen, welche Züge bereits gemacht wurden und welche möglich sind, und bei denen der Zufall keine Rolle spielt. Mensch-Ärgere-Dich-Nicht und Backgammon gehören also bspw. nicht zu dieser Klasse von Spielen.

B.3.1 Programmierung von Spielstrategien

Stellen wir uns einmal eine gewisse Spielsituation vor. Das Spielbrett hat einen bestimmten Zustand inne und dem Spieler, der an der Reihe ist, steht eine bestimmte Anzahl an Zugmöglichkeiten offen. Graphisch lässt sich das durch so genannte *Spielbäume* veranschaulichen, die aus Knoten (Kreise) und Kanten (Linien) bestehen. Knoten repräsentieren das Spielbrett in einem bestimmten Zustand. Kanten repräsentieren Spielzüge und verbinden immer zwei Knoten miteinander, nämlich die beiden Knoten, die das Spielbrett vor und nach dem Zug repräsentieren. Von besonderer Bedeutung sind das anfängliche Spielbrett sowie Spielbretter, für die es keine Folgezüge mehr gibt, weil das Spiel beendet ist. Das Anfangs-Spielbrett wird auch *Wurzel* des Spielbaumes und die End-Spielbretter werden *Blätter* genannt. Abbildung B.3 skizziert einen solchen Spielbaum. Anders als in der Natur befindet sich die Wurzel dabei ganz oben und die Blätter ganz unten.

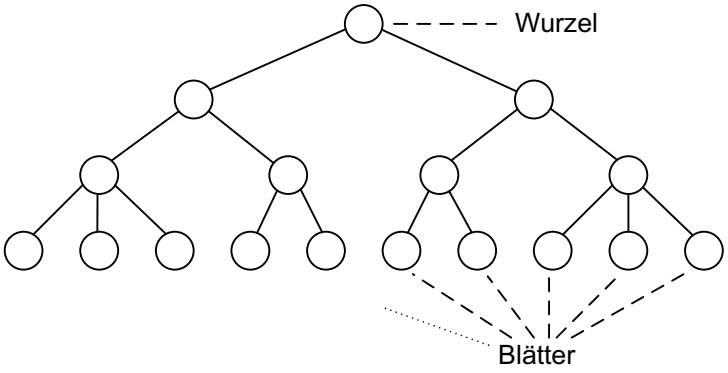

Abbildung B.3: Spielbaum

Im Prinzip kann man für alle Zwei-Spieler-Strategiespiele vollständige Spielbäume erstellen, bei denen die Wurzel den Anfangszustand des Spielbrettes und die Blätter die möglichen Endzustände repräsentieren. Da aus den Endzuständen abzulesen ist, wer das Spiel gewonnen hat, könnte damit ermittelt werden, welcher Spieler mit welchen Spielzügen das Spiel gewinnen würde.

Allerdings sind die meisten Zwei-Spieler-Strategiespiele so komplex, dass ein vollständiger Spielbaum aus einer fast unvorstellbar großen Menge an Knoten besteht; für Schach bspw. 10^{120} Knoten. Selbst sehr schnelle Computer schaffen es daher nicht, vollständige Spielbäume zu berechnen.

Aus diesem Grund wird ein derartiger Spielbaum, ausgehend von einem bestimmten Spielzustand, immer nur bis zu einer bestimmten Tiefe – *Suchtiefe* genannt – aufgebaut. Eine Suchtiefe von 5 bedeutet dabei bspw., dass der Spielbaum aus 6 Ebenen besteht, die Blätter also alle möglichen Spielzustände repräsentieren, die in 5 Spielzügen erreichbar sind[1]. Wenn es nun gelingen würde zu bewerten, ob bestimmte Spielzustände besser für Spieler A oder für Spieler B sind und wie gut sie verglichen mit anderen sind, könnte man anhand eines solchen Spielbaumes trotzdem optimale Spielzüge ermitteln.

[1] d.h. man kann prinzipiell 5 Spielzüge im Voraus denken

B.3.2 Bewertung von Spielzuständen

Für die Bewertung einer Spielsituation, d.h. im Allgemeinen eines Spielbrettes in einem bestimmten Zustand, bedient man sich einer so genannten *Bewertungsfunktion*. Diese liefert ganzzahlige Werte. Die Werte sind dabei umso größer, je besser Spieler A steht, und umso kleiner, je besser Spieler B steht. Am Ende eines Spiels sollte die Bewertungsfunktion also theoretisch ∞ liefern, wenn Spieler A gewonnen hat, und -∞, wenn Spieler B gewonnen hat. Leider ist es im Allgemeinen nicht so einfach möglich, von dem Zustand eines Spielbrettes abzulesen, ob die Spielsituation besser für Spieler A oder für Spieler B ist und wie gut sie verglichen mit anderen Spielzuständen ist. Das Auffinden einer guten Bewertungsfunktion ist daher die eigentliche Schwierigkeit bei der Implementierung von Programmen, die Zwei-Spieler-Strategiespiele spielen können.

B.3.3 Der Minimax-Algorithmus

Der so genannte *Minimax-Algorithmus* ermittelt den in einer bestimmten Spielsituation besten[2] Spielzug für den Spieler, der an der Reihe ist. Dazu baut er ausgehend von der aktuellen Spielsituation einen Spielbaum bis zu einer bestimmten Ebene auf und ordnet jedem Blatt mit Hilfe einer Bewertungsfunktion einen Zahlenwert zu.

Schauen Sie sich dazu Abbildung B.4 an, die den Minimax-Algorithmus an einem Beispiel verdeutlicht. Der Spielbaum hat eine Suchtiefe von 3. Die Blätter sind mit den Zahlenwerten versehen, die die Bewertungsfunktion liefert. In dem Beispiel ziehen die beiden Spieler immer abwechselnd.

Der Minimax-Algorithmus analysiert nun den Spielbaum von unten nach oben. Im Beispiel in Abbildung B.4 ist auf der Ebene über den Blättern Spieler A am Zug. Er entscheidet sich natürlich jeweils für den Zug, der zu dem Blatt mit der höchsten Bewertung führt, denn die deutet ja auf den für ihn besseren Spielzustand hin. Der Minimax-Algorithmus vermerkt diesen Wert an dem jeweiligen Knoten.

Auf der Ebene darüber ist Spieler B am Zug. Da für Spieler B Spielzustände mit möglichst kleinen Werten günstig sind, wird er sich jeweils für einen Spielzug entscheiden, der zu dem Knoten mit dem niedrigsten Wert führt. Entsprechend notiert sich der Minimax-Algorithmus diesen Wert an dem jeweiligen Knoten.

Auf der Wurzelebene, die die aktuelle Spielsituation repräsentiert, ist Spieler A an der Reihe, der sich letztendlich für den Spielzug entscheidet, der zur Spielsituation mit der höchster Bewertung führt (die in Abbildung B.4 fett eingezeichnete Linie).

Insgesamt lässt sich also festhalten: Der Minimax-Algorithmus baut ausgehend von der aktuellen Spielsituation einen Spielbaum bis zu einer vorgegebenen Suchtiefe auf und bewertet die Blätter. Anschließend traversiert er den Baum von unten nach oben. Immer wenn Spieler A an der Reihe ist, entscheidet er sich für den Zug, der zum Spielzustand mit der höchsten Bewertung führt (Maximum). Wenn Spieler B am Zug ist, wird der Zug mit der kleinsten Bewertung genommen (Minimum)[3]. Die entsprechende Bewertung wird dabei von Knotenebene zu Knotenebene mitgeführt. Am Wurzelknoten angelangt steht dann fest, welchen Spielzug der Spieler, der an der Reihe ist, ausführen sollte. Der dem Wurzelknoten zugeordnete Wert entspricht der Bewertung der Spielsituation, die der Spieler, der an der Reihe ist, im schlechtesten Fall ereichen kann. Dieser Fall tritt ein, wenn der Gegner dieselbe Bewertungsfunktion benutzt und bei der Zugauswahl keine Fehler macht.

Unter der Voraussetzung, dass die Bewertungsfunktion gut ist, lässt sich feststellen: Je größer die Suchtiefe ist, umso bessere Züge können ermittelt werden. Allerdings wächst die für die Suche benötigte Zeit auch (enorm!) an.

Im Folgenden wird der Minimax-Algorithmus in einer Pseudo-Programmiersprache skizziert:

[2] in Abhängigkeit von der Güte der Bewertungsfunktion
[3] daher der Name *Minimax*

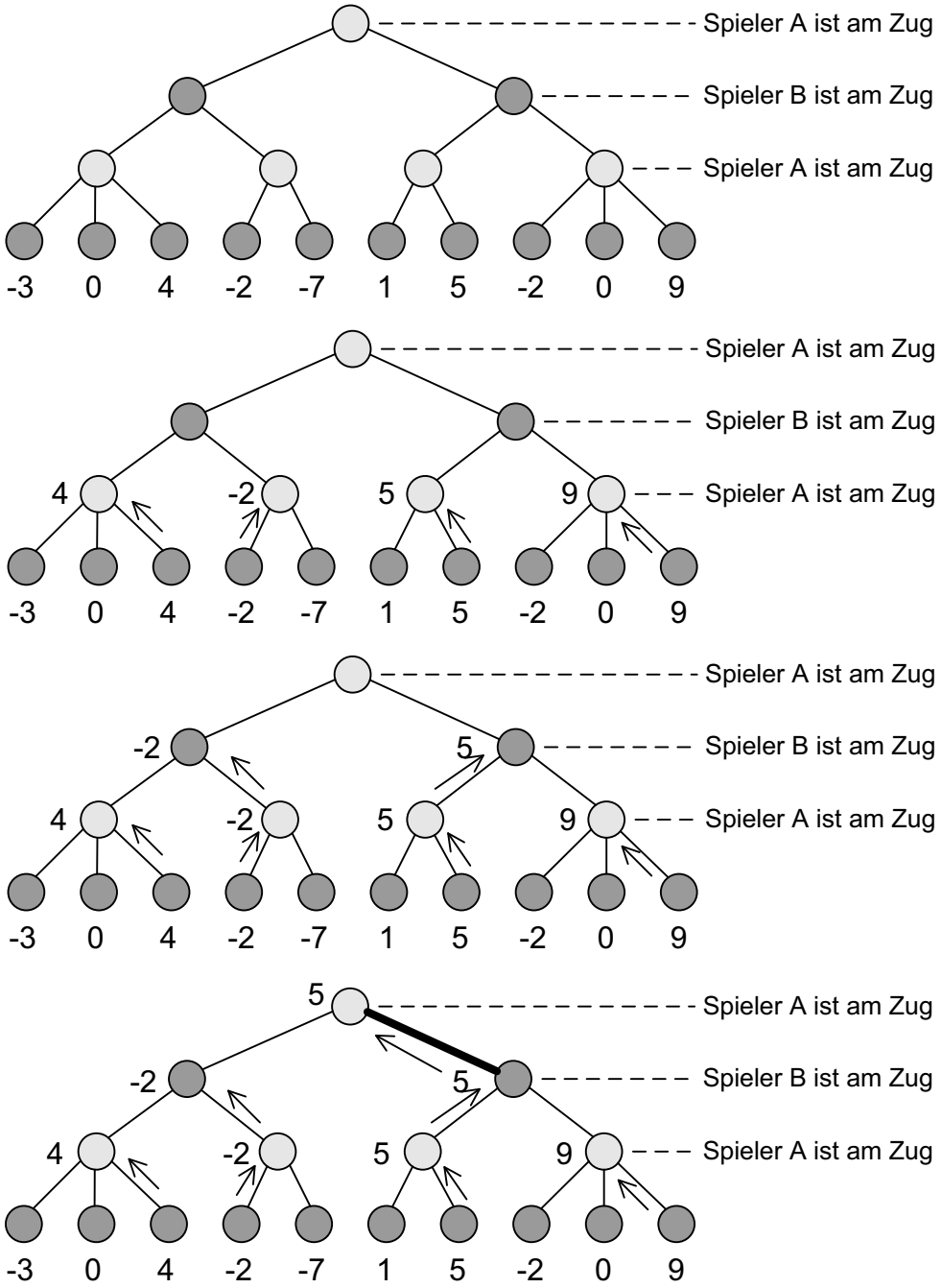

Abbildung B.4: Minimax-Algorithmus

```
void main() {
    ...
    Spielzug zug;
```

```
    int wert;
    if (SpielerAIstAmZug()) {
        (zug, wert) = maxWert(suchtiefe);
    } else {
        (zug, wert) = minWert(suchtiefe);
    }
    zug.ausfuehren();
    ...
}

(Spielzug, int) maxWert(int resttiefe) {
    Spielzug besterZug = null;
    int hoechsterWert = -unendlich;
    Spielzug zuege = ermittleAlleFolgeZuege();
    for (int z = 0; z < zuege.length; z++) {
        zuege[z].simulieren();
        Spielzug zug;
        int zugWert;
        if (resttiefe <= 1 || keineFolgeZuegeMehrMoeglich()) {
            zug = zuege[z];
            zugWert = bewertungsfunktion();
        } else if (SpielerAIstAmZug()) {
           (zug, zugWert) = maxWert(resttiefe-1);
        } else {
           (zug, zugWert) = minWert(resttiefe-1);
        }
        zuege[z].simulationRueckgaengigMachen();
        if (zugWert > hoechsterWert) {
            hoechsterWert = zugWert;
            besterZug = zug;
        }
    }
    return (besterZug, hoechsterWert);
}

(Spielzug, int) minWert(int resttiefe) {
    Spielzug besterZug = null;
    int niedrigsterWert = unendlich;
    Spielzug zuege = ermittleAlleFolgeZuege();
    for (int z = 0; z < zuege.length; z++) {
        zuege[z].simulieren();
        Spielzug zug;
        int zugWert;
        if (resttiefe <= 1 || keineFolgeZuegeMehrMoeglich()) {
            zug = zuege[z];
            zugWert = bewertungsfunktion();
        } else if (SpielerAIstAmZug()) {
           (zug, zugWert) = maxWert(resttiefe-1);
        } else {
           (zug, zugWert) = minWert(resttiefe-1);
        }
        zuege[z].simulationRueckgaengigMachen();
        if (zugWert < niedrigsterWert) {
            niedrigsterWert = zugWert;
            besterZug = zug;
        }
    }
    return (besterZug, niedrigsterWert);
}
```

B.4 Hamster-Kalah

Jetzt sind wir soweit, dass wir den Hamstern Strategien zum Spielen von Zwei-Spieler-Strategiespielen beibringen können. Wir nutzen natürlich den Minimax-Algorithmus. Wir führen dazu ein paar neue Klassen ein, die wir alle in ein Paket namens `hamsterkalah` packen.

Zunächst leiten wir von der Klasse `Spielbrett` eine Klasse `WertSpielbrett` ab, die in der Methode `bewerteStellung` eine Bewertungsfunktion implementiert. Diese Bewertungsfunktion ist sehr einfach gehalten. Sie geht davon aus, dass der Spielstand für die beiden Spieler umso besser ist, je mehr Körner in ihrer Kalah liegen. Also liefert sie einfach die Differenz der jeweiligen Körneranzahl in Kalah A und Kalah B.

```
package hamsterkalah;

import kalah.*;

// repraesentiert ein Kalah-Spielbrett mit Stellungsbewerter
public class WertSpielbrett extends Spielbrett {

    // Konstanten
    public final static int MAX_WERT = Spielbrett.ANZAHL_KOERNER;

    public final static int MIN_WERT = -MAX_WERT;

    public final static int MITTEL_WERT = 0;

    // Konstruktor; erzeugt eine Kopie des uebergebenen
    // Spielbrettes
    public WertSpielbrett(Spielbrett brett) {
        super(brett);
    }

    // einfacher Stellungsbewerter; je groesser der gelieferte
    // Wert, desto besser steht Spieler A; je kleiner der
    // gelieferte Wert, desto besser steht Spieler B
    public int bewerteStellung(boolean spielBeendet) {
        if (!spielBeendet) {
            return this.liefereAnzahlKoernerInKalahA()
                    - this.liefereAnzahlKoernerInKalahB();
        }

        // Spiel ist beendet; also muessen auch noch die "Reste"
        // betrachtet werden
        int diff = (this.liefereAnzahlKoernerInKalahA() + this
                .liefereAnzahlKoernerVonSpielerA())
                - (this.liefereAnzahlKoernerInKalahB() + this
                        .liefereAnzahlKoernerVonSpielerB());
        if (diff > 0) { // Spieler A hat gewonnen
            return WertSpielbrett.MAX_WERT;
        } else if (diff < 0) { // Spieler B hat gewonnen
            return WertSpielbrett.MIN_WERT;
        } else { // Unentschieden
            return WertSpielbrett.MITTEL_WERT;
        }
    }
}
```

Als nächstes leiten wir von der Klasse `MenschHamster` eine Klasse `StrategieHamster` ab, die selbstständig spielende Hamster realisiert. Alles, was die Klasse `StrategieHamster` tut, ist die Methode `naechsterSpielzug` zu überschreiben, und zwar überschreibt sie sie durch Implementierung des Minimax-Algorithmus

für das Kalah-Spiel. Zur Simulation von Spielzügen wird eine Kopie des aktuellen Spielbrettes erzeugt und mit dieser Kopie jeweils weitergearbeitet. Weiterhin kommt noch der Zufall ins Spiel: Wenn es mehrere Spielzüge mit der besten Bewertung gibt, wird per Zufall einer von ihnen ausgewählt.

Da es in Java nicht möglich ist, dass eine Methode zwei Werte liefert, wird noch eine Hilfsklasse `WertSpielzug` definiert, die einen Spielzug und seine Bewertung kapselt.

```
package hamsterkalah;

import kalah.*;

// Hilfsklasse der Klasse StrategieHamster, die einen
// Spielzug plus Bewertung repraesentiert
class WertSpielzug {

    private Spielzug zug;

    private int wert;

    WertSpielzug(Spielzug z, int w) {
        this.zug = z;
        this.wert = w;
    }

    Spielzug getZug() {
        return this.zug;
    }

    int getWert() {
        return this.wert;
    }
}
```

```
package hamsterkalah;

import kalah.*;

// repraesentiert einen "intelligenten"
// Kalah-spielenden Hamster
public class StrategieHamster extends MenschHamster implements
        Spieler {
    // Attribute
    protected int spielstaerke;

    // initialisiert einen StrategieHamster; uebergeben werden
    // die Spielregeln, die Information, ob der Hamster Spieler A
    // oder B ist, sowie die Spielstaerke des Hamsters (je
    // groesser der Wert, desto besser spielt der Hamster)
    public StrategieHamster(boolean istSpielerA,
            Spielregeln regeln, int staerke) {
        super(istSpielerA, regeln);
        this.spielstaerke = staerke;
    }

    // ermittelt den naechsten Spielzug, fuehrt ihn aus und
    // liefert ihn
    public Spielzug naechsterSpielzug() {
        // ermittelt den besten Spielzug
        WertSpielzug zug = null;
        WertSpielbrett brett = new WertSpielbrett(this.regeln
```

```
                .getBrett());
        if (this.istSpielerA()) {
            zug = this.ermittleBestenAZug(this.spielstaerke,
                    brett);
        } else {
            zug = this.ermittleBestenBZug(this.spielstaerke,
                    brett);
        }

        // anschliessend wird der Spielzug im Hamster-Territorium
        // ausgefuehrt
        this.fuehreSpielzugAus(zug.getZug());

        // der ausgefuehrte Spielzug wird geliefert
        return zug.getZug();
    }

    // interne Methoden

    // ermittelt den besten Spielzug von Spieler A
    protected WertSpielzug ermittleBestenAZug(int restTiefe,
            WertSpielbrett brett) {
        Spielzug besterSpielzug = null;
        int besteBewertung = WertSpielbrett.MIN_WERT - 1;
        Spielzug[] zuege = this.ermittleFolgeZuege(true, brett);
        for (int z = 0; z < zuege.length; z++) {
            WertSpielbrett brettKopie = new WertSpielbrett(brett);
            Spielregeln regelKopie = new Spielregeln(brettKopie);
            int letzteMulde = brettKopie.fuehreSpielzugAus(true,
                    zuege[z]);
            boolean naechsterIstA = regelKopie.istAAmZug(true,
                    letzteMulde);
            boolean spielBeendet = regelKopie
                    .istSpielBeendet(naechsterIstA);
            int zugWert = 0;
            if (restTiefe <= 1 || spielBeendet) {
                zugWert = brettKopie
                        .bewerteStellung(spielBeendet);
            } else if (naechsterIstA) {
                WertSpielzug zug = ermittleBestenAZug(
                        restTiefe - 1, brettKopie);
                zugWert = zug.getWert();
            } else {
                WertSpielzug zug = ermittleBestenBZug(
                        restTiefe - 1, brettKopie);
                zugWert = zug.getWert();
            }
            int zufall = (int) (Math.random() * 2); // 0 oder 1
            if (zugWert > besteBewertung
                    || (zugWert == besteBewertung && zufall == 0)) {
                besteBewertung = zugWert;
                besterSpielzug = zuege[z];
            }
        }
        return new WertSpielzug(besterSpielzug, besteBewertung);
    }

    // ermittelt den besten Spielzug von Spieler B;
    protected WertSpielzug ermittleBestenBZug(int restTiefe,
            WertSpielbrett brett) {
```

```
        Spielzug besterSpielzug = null;
        int besteBewertung = WertSpielbrett.MAX_WERT + 1;
        Spielzug[] zuege = this.ermittleFolgeZuege(false, brett);
        for (int z = 0; z < zuege.length; z++) {
            WertSpielbrett brettKopie = new WertSpielbrett(brett);
            Spielregeln regelKopie = new Spielregeln(brettKopie);
            int letzteMulde = brettKopie.fuehreSpielzugAus(
                    false, zuege[z]);
            boolean naechsterIstA = regelKopie.istAAmZug(false,
                    letzteMulde);
            boolean spielBeendet = regelKopie
                    .istSpielBeendet(naechsterIstA);
            int zugWert = 0;
            if (restTiefe <= 1 || spielBeendet) {
                zugWert = brettKopie
                        .bewerteStellung(spielBeendet);
            } else if (naechsterIstA) {
                WertSpielzug zug = ermittleBestenAZug(
                        restTiefe - 1, brettKopie);
                zugWert = zug.getWert();
            } else {
                WertSpielzug zug = ermittleBestenBZug(
                        restTiefe - 1, brettKopie);
                zugWert = zug.getWert();
            }
            int zufall = (int) (Math.random() * 2); // 0 oder 1
            if (zugWert < besteBewertung
                    || (zugWert == besteBewertung && zufall == 0)) {
                besteBewertung = zugWert;
                besterSpielzug = zuege[z];
            }
        }
        return new WertSpielzug(besterSpielzug, besteBewertung);
    }

    // ermittelt alle moeglichen Folgezuege
    protected Spielzug[] ermittleFolgeZuege(boolean istSpielerA,
            Spielbrett brett) {
        int[] mulden = new int[Spielbrett.ANZAHL_MULDEN_PRO_SPIELER];
        int nummer = 0;
        if (istSpielerA) {
            for (int mulde = 0; mulde < Spielbrett.KALAH_A; mulde++) {
                if (brett.liefereAnzahlKoerner(mulde) > 0) {
                    mulden[nummer] = mulde;
                    nummer = nummer + 1;
                }
            }
        } else {
            for (int mulde = Spielbrett.KALAH_A + 1;
                    mulde < Spielbrett.KALAH_B;
                    mulde++) {
                if (brett.liefereAnzahlKoerner(mulde) > 0) {
                    mulden[nummer] = mulde;
                    nummer = nummer + 1;
                }
            }
        }

        Spielzug[] zuege = new Spielzug[nummer];
        for (int i = 0; i < nummer; i++) {
```

```
            zuege[i] = new Spielzug(mulden[i]);
        }
        return zuege;
    }

}
```

Von der Klasse `SchiedsrichterHamster` wird eine Klasse `AbfrageSchiedsrichterHamster` abgeleitet, die die beiden Methoden `erzeugeSpielerA` und `erzeugeSpielerB` überschreibt. Und zwar wird in den beiden Methoden jeweils beim Benutzer nachgefragt, ob der Spieler ein Mensch oder ein Hamster sein soll. Entsprechend wird ein Objekt der Klasse `MenschHamster` oder ein Objekt der Klasse `StrategieHamster` erzeugt und geliefert. Entscheidet sich der Benutzer für einen Hamster als Spieler, muss er zusätzlich noch die gewünschte Spielstärke angeben, die der Suchtiefe im Minimax-Algorithmus entspricht. Dabei gilt: Je größer der Wert für die Spielstärke ist, desto besser spielt der Hamster in der Regel, aber umso länger „überlegt“ er auch.

```
package hamsterkalah;

import kalah.*;

// repraesentiert einen Hamster als Schiedsrichter des
// Kalah-Spiels
public class AbfrageSchiedsrichterHamster extends
        SchiedsrichterHamster {
    // Konstruktor
    public AbfrageSchiedsrichterHamster() {
        super();
    }

    // erzeugt und liefert Spieler A
    public Spieler erzeugeSpielerA(Spielregeln regeln) {
        String antwort = this
                .liesZeichenkette("Spieler A: Mensch oder Hamster?");
        while (!(antwort.equals("Mensch") || antwort
                .equals("Hamster"))) {
            antwort = this.liesZeichenkette("Eingabefehler!\n"
                    + " Spieler A: Mensch oder Hamster?");
        }
        Spieler spieler;
        if (antwort.equals("Mensch")) {
            spieler = new MenschHamster(true, regeln);
        } else {
            int staerke = this.liesZahl("Staerke des Hamster?");
            spieler = new StrategieHamster(true, regeln, staerke);
        }
        return spieler;
    }

    // erzeugt und liefert Spieler B
    public Spieler erzeugeSpielerB(Spielregeln regeln) {
        String antwort = this
                .liesZeichenkette("Spieler B: Mensch oder Hamster?");
        while (!(antwort.equals("Mensch") || antwort
                .equals("Hamster"))) {
            antwort = this.liesZeichenkette("Eingabefehler!\n"
                    + " Spieler B: Mensch oder Hamster?");
        }
        Spieler spieler;
        if (antwort.equals("Mensch")) {
```

```
            spieler = new MenschHamster(false, regeln);
        } else {
            int staerke = this.liesZahl("Staerke des Hamster?");
            spieler = new StrategieHamster(false, regeln,
                    staerke);
        }
        return spieler;
    }
}
```

Das neue Hauptprogramm unterscheidet sich von dem alten nur dadurch, dass anstelle eines normalen Schiedsrichter-Hamsters ein AbfrageSchiedsrichter-Hamster erzeugt und dem Kalah-Spiel übergeben wird.

```
import kalah.*;
import hamsterkalah.*;

// Hauptprogramm
void main() {
    SchiedsrichterHamster schiedsrichter = new AbfrageSchiedsrichterHamster();
    KalahSpiel spiel = new KalahSpiel(schiedsrichter);
    spiel.spielen();
}
```

Damit sind wir fertig! Am besten, Sie starten nun den Hamster-Simulator und spielen selbst einmal gegen die Hamster oder lassen zwei unterschiedlich starke Hamster gegeneinander spielen und beobachten sie dabei.

B.5 Aufgaben

Das Entwickeln von Programmen, die Zwei-Spieler-Strategiespiele spielen können, ist eine sehr reizvolle Sache. Und es ist nicht einmal sonderlich schwierig, wie die vorangegangenen Abschnitte gezeigt haben. Ob Sie das Prinzip, insbesondere den Minimax-Algorithmus verstanden haben, können Sie sich im Folgenden selbst beweisen. Es werden Ihnen einige Spiele vorgestellt, die Sie analog zum Kalah-Spiel implementieren können.

B.5.1 Aufgabe 1

Die in der Klasse `WertSpielbrett` implementierte Bewertungsfunktion für das Kalah-Spiel ist sehr einfach. Fällt Ihnen eine bessere Bewertungsfunktion ein?

B.5.2 Aufgabe 2

Implementieren Sie analog zum Kalah-Spiel das Spiel „4-Gewinnt“ so, dass Mensch gegen Mensch, Mensch gegen Hamster und Hamster gegen Hamster gegeneinander antreten können.

4-Gewinnt ist ein Zwei-Spieler-Strategiespiel. Beide Spieler besitzen eine Menge an Scheiben. Die Scheiben von Spieler A sind rot, die von Spieler B gelb. Das Spielfeld ist ein quadratisches Feld, das aus 7*7 Kacheln besteht (7 Haufen, die maximal 7 Kacheln hoch sind). Die Spieler sind abwechselnd an der Reihe. Spieler A beginnt. Bei jedem Spielzug müssen sie eine ihrer Scheiben auf einen der Haufen legen, der noch nicht voll ist. Ziel jedes Spielers ist es, seine Scheiben so zu platzieren, dass eine Spalte, Zeile oder Diagonale zustande kommt, in der 4 seiner Scheiben aneinander liegen. Dann hat er gewonnen. Das Spiel endet unentschieden, wenn alle Kacheln belegt sind, ohne dass einer der Spieler das Ziel erreicht.

Repräsentieren Sie Spielzüge durch die Spalte des Haufens, auf den eine Scheibe gelegt werden soll, rote Scheiben durch ein einzelnes Korn und gelbe Scheiben durch zwei Körner. Abbildung B.5 skizziert ein mögliches 4-Gewinnt-Hamster-Territorium, bei dem Spieler A gewonnen hat.

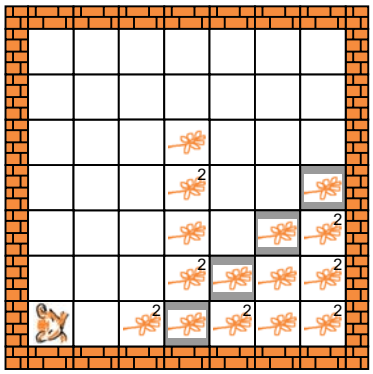

Abbildung B.5: 4-Gewinnt-Hamster-Territorium

B.5.3 Aufgabe 3

In dieser Aufgabe geht es um das Spiel „Reversi“, auch unter dem Namen „Othello“ bekannt. Es wird auf einem Spielbrett gespielt, das sich aus 8*8 einzelnen Feldern zusammensetzt. Es ist zweckmäßig, das Brett, wie bspw. beim Schach, mit Koordinaten zu versehen, um Spielzüge eindeutig angeben zu können. Als Spielfiguren werden runde Plättchen (Steine) verwendet. Diese haben je eine schwarze und eine weiße Seite. Es gibt insgesamt 64 Plättchen, also genauso viele, wie das Spielbrett Felder hat. Die Spielplättchen werden im Laufe des Spiels umgedreht (daher der Name Reversi), so dass schwarze zu weißen Steinen werden können und umgekehrt. Im Folgenden ist immer von schwarzen und weißen Steinen die Rede, was sich jeweils auf die Oberseite der Steine – also die sichtbare Farbe – bezieht.

Der Spieler, der das Spiel beginnt, wird im Folgenden Spieler A genannt. Der andere ist dementsprechend Spieler B. Spieler A bekommt die Farbe „Weiß“, Spieler B die Farbe „Schwarz“ zugeteilt. Zu Anfang des Spiels befinden sich vier Steine auf dem Spielbrett, zwei mit ihrer weißen (D4 und E5), zwei mit ihrer schwarzen Seite (E4 und D5) nach oben (siehe Abbildung B.6 (links)). Die restlichen 60 Steine liegen in einem Sammelpool neben dem Spielbrett.

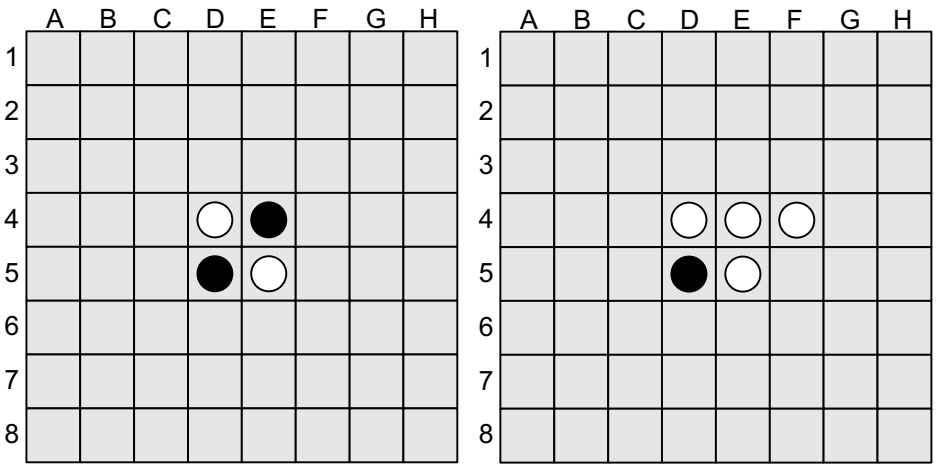

Abbildung B.6: Reversi

Spieler A führt dann seinen ersten Spielzug aus: Er nimmt einen Stein aus dem Sammelpool und legt ihn auf ein leeres Feld (selbstverständlich mit seiner Farbe „Weiß“ nach oben). Das leere Feld muss dabei an ein belegtes Feld horizontal, vertikal oder diagonal angrenzen. Außerdem muss der Spielzug zum „Schlagen“ von mindestens einem gegnerischen Stein führen. „Schlagen“ bedeutet für Spieler A: Drehe alle schwarzen Steine um, die sich horizontal, vertikal oder diagonal zwischen bereits gesetzten weißen Steinen und dem neu gesetzten (ebenfalls weißen) Stein befinden. In der Ausgangsituation kann sich Spieler A also eines der Felder E3, F4, D6 und C5 aussuchen. Setzt er den Stein bspw. auf Feld F4, muss er den schwarzen Stein auf Feld E4 umdrehen (siehe Abbildung B.6 (rechts)).

Damit ist der erste Spielzug von Spieler A beendet und Spieler B kommt zum Zug. Dieser führt nun eine identische Aktion durch, nur dass er jetzt natürlich weiße Steine schlagen muss.

Die beiden Spieler führen abwechselnd ihre Spielzüge aus. Ist es einem Spieler nicht möglich, entsprechend der Spielregeln zu ziehen, so muss er passen und der andere Spieler ist wieder am Zug.

Das Spiel ist beendet, wenn kein Spieler mehr einen Stein setzen kann. Dies ist in der Regel der Fall, wenn alle Felder besetzt sind, kann aber auch schon vorher passieren. Die schwarzen und weißen Steine auf dem Spielbrett werden gezählt. Sieger des Spiels ist der Spieler, der die größere Anzahl an Steinen auf dem Spielbrett hat.

Beachten Sie bitte:

- Ein Stein, der einmal auf dem Spielbrett liegt, wird nie mehr vom Brett genommen oder verschoben. Er wird höchstens umgedreht.
- Jeder Spielzug muss vollständig ausgeführt werden, d.h. alle Steine, die aufgrund eines neu gesetzten Steines geschlagen werden können, müssen auch umgedreht werden. Die Spieler dürfen sich nicht aussuchen, welche Steine sie umdrehen und welche nicht.
- Solange ein Spieler gegnerische Steine schlagen kann, muss er ziehen. Er darf nicht freiwillig passen.
- Der entscheidende Stein beim Umdrehen ist der neu gesetzte Stein. Zum Beispiel werden weiße Steine nicht umgedreht, die zwischen zwei schwarzen Steinen liegen, die beide schon vorher auf dem Spielbrett lagen. Es findet also keine transitive Fortsetzung beim Umdrehen von Steinen statt. Wird bspw. bei der Ausgangsposition in Abbildung B.7 (links) ein schwarzer Stein auf Feld F6 gesetzt, dann werden die beiden weißen Steine auf den Feldern D4 und E5 umgedreht und es ergibt sich die Situation im rechten Teil der Abbildung. Obwohl durch die Umdreh-Aktion nun auch der weiße Stein auf Feld D6 zwischen zwei schwarzen Steinen (E5 und C7) liegt, wird dieser nicht umgedreht, weil sich die beiden Steine schon vorher auf dem Spielbrett befanden.

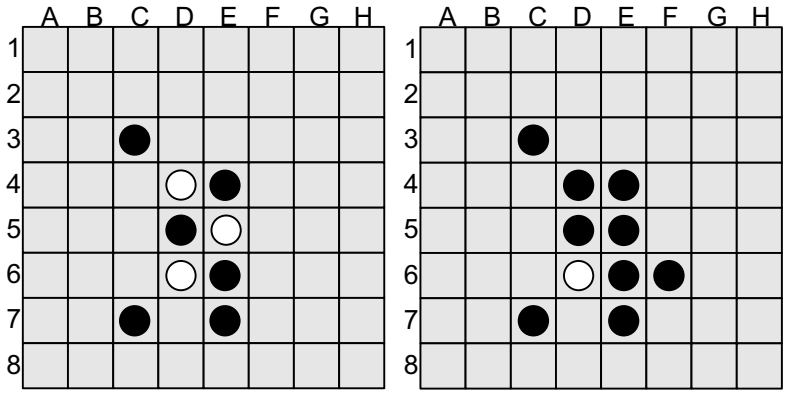

Abbildung B.7: Reversi

Implementieren Sie analog zum Kalah-Spiel das Spiel „Reversi“ so, dass es Mensch gegen Mensch, Mensch gegen Hamster und Hamster gegen Hamster gegeneinander spielen können. Repräsentieren Sie Spielzüge durch die Reihe und Spalte der entsprechenden Kachel im Hamster-Territorium, auf die eine Scheibe gelegt werden soll, schwarze Scheiben durch ein einzelnes Korn und weiße Scheiben durch zwei Körner.

B.5.4 Aufgabe 4

Natürlich könnten Sie den Hamstern auch das Schach-Spielen beibringen. Schach ist jedoch ein sehr komplexes Spiel. Das im Folgenden vorgestellte Spiel „Alapo“ ist eine einfache aber trotzdem interessante Schachvariante.

Gespielt wird Alapo auf einem Schachbrett mit allerdings nur 6x6 Feldern. Spielfiguren sind geometrische Gebilde. Es gibt große und kleine Quadrate, große und kleine Kreise sowie große und kleine Dreiecke. In Abbildung B.8 ist die Anfangsaufstellung zu sehen. Alapo ist ein Spiel für zwei Spieler. Spieler A (Weiß) spielt mit weißen Spielfiguren von unten nach oben. Spieler B (Schwarz) spielt mit schwarzen Spielfiguren von oben nach unten.

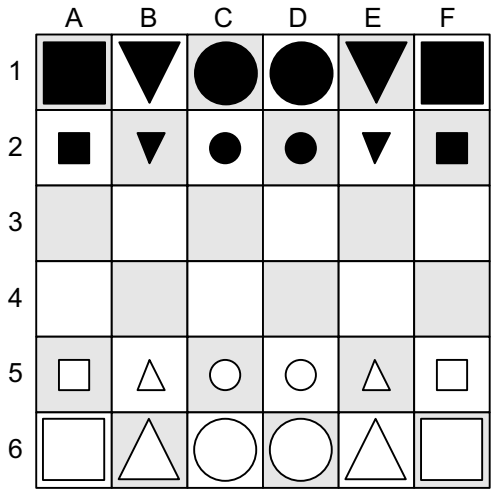

Abbildung B.8: Alapo

Es wird immer abwechselnd gezogen, wobei Spieler Weiß beginnt. Es besteht Zugzwang. Für die einzelnen Spielfiguren gelten folgende Zugregeln:

- Große Quadrate dürfen wie Türme beim Schach gezogen werden, d.h. beliebig (> 0) viele Felder horizontal oder vertikal.
- Kleine Quadrate dürfen wie große Quadrate gezogen werden, allerdings nur ein einzelnes Feld.
- Große Dreiecke dürfen wie Läufer beim Schach gezogen werden, d.h. beliebig (> 0) viele Felder in einer Diagonalen.
- Kleine Dreiecke dürfen wie große Dreiecke gezogen werden, allerdings nur ein einzelnes Feld.
- Große Kreise dürfen wie Damen beim Schach gezogen werden, d.h. beliebig (> 0) viele Felder in einer Horizontalen, Vertikalen oder Diagonalen.
- Kleine Kreise dürfen wie große Kreise gezogen werden, allerdings nur ein einzelnes Feld.

Dabei gelten grundsätzlich folgende Resultate bzw. Einschränkungen:

- Bei einem Spielzug darf nur eine einzelne eigene Spielfigur gezogen werden.
- Bei einem Spielzug dürfen keine Spielfiguren übersprungen werden.
- Es darf nicht auf ein Feld gezogen werden, auf dem bereits eine eigene Spielfigur steht.
- Wird eine Spielfigur auf ein Feld gezogen, auf dem eine Spielfigur des Gegners steht, so wird diese geschlagen, d.h. vom Spielbrett entfernt.

Ziel des Spiels ist es, mit einer eigenen Spielfigur die gegenüberliegende Grundlinie zu erreichen, und zwar ohne dass diese im nächsten Zug wieder geschlagen werden könnte.

Ein Spiel ist in den folgenden Fällen beendet:

- Ein Spieler X hat mit einer eigenen Spielfigur die gegenüberliegende Grundlinie erreicht und diese Spielfigur kann im nächsten Zug nicht direkt geschlagen werden. In diesem Fall hat Spieler X unmittelbar gewonnen.
- Ein Spieler X hat mit einer eigenen Spielfigur die gegenüber liegende Grundlinie erreicht und diese Spielfigur kann zwar im nächsten Zug direkt geschlagen werden, sein Gegner tut dies jedoch nicht. In diesem Fall hat Spieler X gewonnen, nachdem der Gegner seinen Zug ausgeführt hat.
- Das Spielbrett gelangt durch einen Spielzug in einen Zustand, den es vorher schon einmal inne hatte. In diesem Fall hat der Spieler, der den letzten Spielzug ausgeführt hat, verloren. Durch diese Regel sollen Situationen vermieden werden, bei denen beide Spieler immer hin und her ziehen und das Spiel niemals enden würde.
- Ein Spieler kann, wenn er am Zug ist, keinen legalen Spielzug mehr ausführen. In diesem Fall hat sein Gegner gewonnen.
- Ein Spieler besitzt keine eigenen Spielfiguren mehr. In diesem Fall hat der Gegner gewonnen.

Implementieren Sie analog zum Kalah-Spiel das Alapo-Spiel so, dass es Mensch gegen Mensch, Mensch gegen Hamster und Hamster gegen Hamster gegeneinander spielen können. Repräsentieren Sie dabei die Spielfiguren durch Hamster, die von entsprechenden erweiterten Hamster-Klassen erzeugt werden (`GrossQuadratHamster` und `KleinQuadratHamster`, `GrossDreieckHamster` und `KleinDreieckHamster` sowie `GrossKreisHamster` und `KleinKreisHamster`). Repräsentieren Sie Spielzüge durch die jeweilige Reihe und Spalte der beiden Kacheln im Hamster-Territorium, auf der die zu ziehende Spielfigur aktuell sitzt und auf die sie gezogen werden soll.