31 Kobolde

Eine Basisklasse bietet elementare Funktionalität, die in abgeleiteten Klassen erweitert oder angepasst werden kann. Die damit aufgebaute Ableitungshierarchie ist allerdings statisch im Quelltext fixiert und kann zur Laufzeit nicht mehr beeinflusst werden. Darüber hinaus legt der Quelltext bereits die zur Verfügung stehenden Kombinationen von Erweiterungen fest. Der Anwender kann diese Kombinationen benutzen, aber keine neuen erschaffen.

In dieser Aufgabe wird gezeigt, wie die Restriktionen der Erweiterung durch Ableitung überwunden werden können. Zunächst wird anhand eines Beispiels das Problem veranschaulicht und dann ein Lösungsweg erarbeitet, der die erwünschte Flexibilität erlaubt.

31.1 Zahlen-Kobolde

Kobolde haben vor allem Unsinn im Kopf. In dieser Aufgabe werden Kobolde modelliert, die Zahlen, die ihnen anvertraut werden, manchmal manipulieren und dann verändert wieder herausrücken. Definieren Sie zunächst eine Klasse `Kobold` mit dem Namen des Kobolds. Der Name eines Kobolds ändert sich nicht.

Die Neigung eines Kobolds zum Unsinnmachen drückt sich in der Methode `messUp` aus. Der Anwender überlässt dem Kobold als Argument der Methode eine ganze Zahl und erwartet sie als Ergebnis wieder zurück.[1] Wie bei Kobolden nicht anders zu erwarten, stimmt das Ergebnis aber nicht unbedingt mit dem Argument überein, sondern liegt etwas höher.

Der Hang zum Schabernack ist bei Kobolden unterschiedlich stark ausgeprägt. Jeder Kobold erhält daher im Konstruktor einen unveränderlichen Schabernack-Faktor, der die maximale Erhöhung der anvertrauten Zahlen festlegt. Dabei sind nur Werte zwischen 0 und 10 erlaubt. Unerlaubte Angaben werden kommentarlos durch den nächstgelegenen zulässigen Grenzwert ersetzt. Ein Kobold mit dem Schabernack-Faktor 0 ist vollkommen harmlos, weil er jede anvertraute Zahl brav wieder abgibt. Dagegen muss bei einem Kobold mit dem Schabernack-Faktor 10 damit gerechnet werden, dass die abgelieferte Zahl um bis zu 10 höher liegt als die ursprünglich anvertraute Zahl.

[1] Wozu der Anwender den Kobold überhaupt braucht, bleibt hier im Dunkeln :-)

Geben Sie der Klasse einen Konstruktor, Getter und die Methode `messUp`. Dazu kommt eine `toString`-Methode, die einen einfachen Umgang mit der Klasse zur Entwicklungszeit erlaubt.

Die folgende `main`-Methode erzeugt zwei Kobolde und übergibt ihnen ein paar Zahlen:

```
public static void main(String... args) {
    koboldTest(new Kobold("Max", 1));
    koboldTest(new Kobold("Moritz", 2000));
}

static void koboldTest(Kobold k) {
    for(int i = 0; i < 10; i++)
        System.out.printf("%s.messUp(%d) --> %d%n", );
}
```

Lösung

Die Klasse `Kobold` kann geradlinig definiert werden. Die Objektvariablen `name` und `max` legen den Namen und den Schabernack-Faktor fest. Sie werden im Konstruktor initialisiert:

```
public class Kobold {
    private final String name;
    private final int max;

    public Kobold(String n, int mx) {
        name = n;
        max = mx;
    }
    ...
```

Die Begrenzung auf einen Schabernack-Faktor zwischen 0 und 10 wird direkt im Konstruktor fixiert:

```
    public Kobold(String n, int mx) {
        name = n;
        if(mx < 0)
            mx = 0;
        else if(mx > 10)
            mx = 10;
        max = mx;
    }
    ...
```

Die Alternative lässt sich durch einen einzigen Ausdruck ersetzen, der das Gleiche leistet:[2]

```
public Kobold(String n, int mx) {
    name = n;
    // mx auf Werte [0, ..., 10] begrenzen
    max = Math.max(0, Math.min(10, mx));
}
...
```

Die Methode `messUp` erwartet ein ganzzahliges Argument und soll dieses zufällig nach oben verschieben. Hier leistet die Bibliotheksmethode `double Math.random()` gute Dienste, die einen Zufallswert zwischen 0 (einschließlich) und 1 (ausschließlich) liefert. Wenn ein solcher Zufallswert mit dem Schabernack-Faktor multipliziert wird, ergibt sich eine zufällige Verschiebung im gewünschten Bereich. Das Ergebnis hat allerdings zunächst noch den Typ `double`, muss also mit einem *Typecast* wieder in eine ganze Zahl konvertiert werden. Der *Typecast* schneidet aber lediglich alle Nachkommastellen ab, was hier nicht den gewünschten Effekt erzielt. Ein Schabernack-Faktor von 1 wird zum Beispiel immer nur zu Verschiebungen von 0 führen, also überhaupt keine Wirkung haben, weil jede `double`-Verschiebung zwischen 0 und 1 durch den *Typecast* zu 0 wird. Das lässt sich ausgleichen, wenn der Schabernack-Faktor grundsätzlich um 1 erhöht wird:[3]

```
public int messUp(int n) {
    return (int)(n + (max + 1)*Math.random());
}
...
```

Die Methode `toString` liefert eine lesbare Darstellung, die einen Kobold identifiziert:

```
public String toString() {
    return String.format("Kobold \"%s\" (max=%d)", name, max);
}
...
```

Die verbleibenden Getter sind sehr einfach und werden hier nicht abgedruckt.

31.2 Zähe Kobolde

Es stellt sich heraus, dass es noch eine andere Sorte von Kobolden gibt, die zwar auch Unsinn treibt, aber auf andere Art: Sie verändern den übergebenen Wert

[2] Darüber hinaus wäre sicher ein Test auf `null` für den Namen sinnvoll. Der Kürze wegen wird diese Absicherung hier nicht implementiert.

[3] Die konkrete Arithmetik, nach der Kobolde ihren Unfug umsetzen, ist nebensächlich.

nicht, sondern halten ihn für eine Weile zurück und rücken ihn erst mit Verzögerung wieder heraus.

Definieren Sie eine neue Kobold-Klasse `SlowKobold`, die die gleichen Eigenschaften wie die Kobolde aus der vorhergehenden Teilaufgabe aufweist. Allerdings wird ein übergebener Wert in der Methode `messUp` für eine bestimmte Anzahl Sekunden zurückgehalten und erst dann wieder unverändert herausgegeben. Benutzen Sie dazu die Methode `Thread.sleep(long millis)`, die den Programmablauf für die gegebene Anzahl Millisekunden anhält.[4] Die Verzögerung in Sekunden wird im Konstruktor übergeben und darf nur zwischen 0 und 10 liegen.

Die folgende veränderte `main`-Methode testet zähe Kobolde:

```
public static void main(String... args) {
    koboldTest(new SlowKobold("Pumuckl", 1));
    koboldTest(new SlowKobold("BlauerKlabauter", 5));
}
```

Lösung

Die Klasse `SlowKobold` ähnelt der Klasse `Kobold` weitgehend. Sie kann daher abgeleitet werden, wobei lediglich die Methode `messUp` ersetzt wird. Die Verzögerung wird im Konstruktor initialisiert, der Name an den Basisklassenkonstruktor übergeben.

```
public class SlowKobold extends Kobold {
    private final int delay;

    public SlowKobold(String n, int d) {
        super(n, 0);
        delay = Math.max(0, Math.min(10, d));
    }

    public int messUp(int n) {
        try {
            Thread.sleep(1000*delay);
        }
        catch(InterruptedException ex) {}
        return n;
    }
}
```

[4] Die Methode wirft eine `InterruptedException`, die hier ignoriert werden kann.

31.3 Kobold-Verhalten

Obwohl die Lösungen der beiden vorhergehenden Teilaufgaben jede für sich ihren Anforderungen genügen, sind sie dennoch nicht sehr flexibel. Wollte man beispielsweise eine dritte Sorte von Kobolden definieren, die überlassene Zahlen verändert *und* verzögert, so müsste man die beiden `messUp`-Methoden aufrufen. Mit der bisher entwickelten Struktur gelingt das nicht ohne Weiteres.

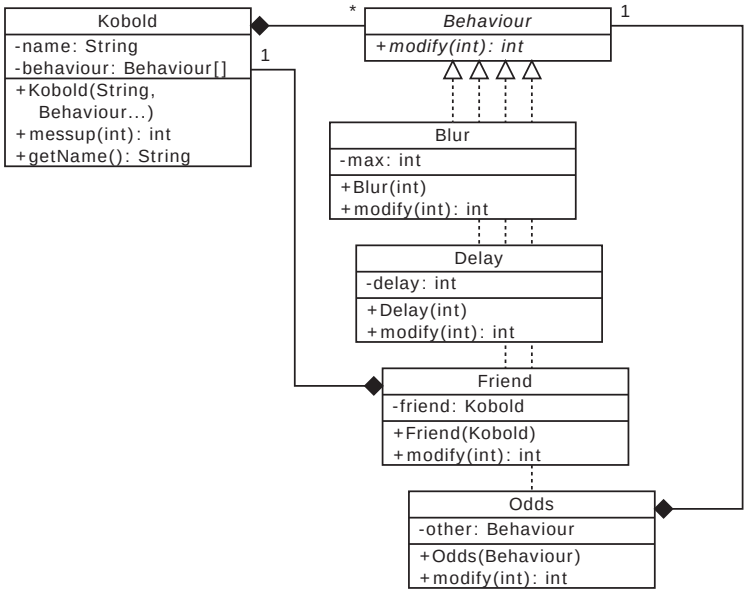
Stattdessen kann man die Implementierung konkreter Verhaltensweisen ganz von der `Kobold`-Klasse trennen und in eigene Klassen auslagern. Ein `Kobold`-Objekt kann beim Erzeugen mit beliebigen Verhaltensweisen ausgestattet werden, die nacheinander zur Wirkung kommen, wenn der Kobold seinen Unfug treibt.

Definieren Sie ein Interface `Behaviour` mit einer einzigen Methode `modify`, die das betreffende Verhalten auslöst. Definieren Sie weiter eine Reihe konkreter Verhaltensweisen als Klassen, die alle das Interface `Behaviour` implementieren:

`Blur`	Verschiebt den überlassenen Wert zufällig nach oben bis zu einem Maximalwert, der im Konstruktor festgelegt wird. Das ist das gleiche Verhalten, das die Kobolde in der ersten Teilaufgabe zeigen.
`Delay`	Verzögert die Rückgabe des überlassenen Werts um eine Anzahl Sekunden, die im Konstruktor festgelegt wird. Das ist das Verhalten der Kobolde in der zweiten Teilaufgabe.
`Friend`	Erklärt einen anderen Kobold zum Freund des Kobolds mit diesem Verhalten. Der überlassene Wert wird vom Freund nach dessen Vorstellungen verändert.
`Odds`	Erwartet ein anderes Verhalten und verändert überlassene *ungerade* Werte mit diesem Verhalten. Gerade Werte werden dagegen immer ohne Störung zurückgeliefert.

Der `Kobold`-Konstruktor akzeptiert eine Liste von einzelnen Verhaltensobjekten, die alle zusammen das Benehmen des Kobolds ergeben.[5] Das folgende UML-Klassendiagramm zeigt die beteiligten Typen:

[5] Wahlweise könnten die Verhaltensweisen eines Kobolds zur Laufzeit mit passenden Methoden ergänzt und gelöscht werden. Diese Variation wird hier nicht umgesetzt, weil sie wenig Neues zur Struktur der Lösung beiträgt.



Lösung

Das Interface `Behaviour` ist recht kurz:

```
public interface Behaviour {
    int modify(int n);
}
```

Die `Kobold`-Klasse kennt nun kein bestimmtes Verhalten mehr, sondern erwartet beim Konstruktoraufruf eine Liste von Verhaltensobjekten:[6]

```
public class Kobold {
    private final String name;
    private final Behaviour[] behaviour;

    public Kobold(String n, Behaviour... bs) {
        name = n;
        behaviour = bs.clone();
    }
    ...
```

[6]Hier lauert ein Problem mit der Datenkapselung, das in der Lösung von Aufgabe 27.1 (Seite 150) diskutiert wird. In der Objektvariablen `behaviour` sollte sicherheitshalber eine *Kopie* des *Vararg*-Parameters abgespeichert werden statt des Parameters selbst. Genau genommen reicht auch die flache Kopie nicht aus, die mit `clone` erzeugt wird. Der Aufrufer des Konstruktors könnte immer noch ein *veränderliches* Verhalten übergeben und dieses manipulieren, nachdem der Kobold erzeugt wurde. Dieses Problem führt aber am Gegenstand dieser Aufgabe vorbei und soll hier nicht weiter verfolgt werden.

Diese Verhaltensobjekte werden in der Methode `messUp` nacheinander aufgerufen. Die Konstruktion funktioniert auch dann, wenn überhaupt kein Verhalten übergeben wird. Das `Behaviour`-Array bleibt dann leer und die Schleife wird nicht durchlaufen. Dieser Kobold unternimmt nichts und liefert überlassene Werte sofort und unverändert zurück.

```
    public int messUp(int n) {
        for(Behaviour b: behaviour)
            n = b.modify(n);
        return n;
    }

    // weitere Methoden wie oben ...
```

Die konkreten Verhaltensklassen implementieren das Interface `Behaviour`. `Blur` ist ein Extrakt des betreffenden Teiles der ursprünglichen `Kobold`-Klasse:

```
public class Blur implements Behaviour {
    private final int max;

    public Blur(int mx) {
        max = Math.max(0, Math.min(10, mx));
    }

    public int modify(int n) {
        return (int)(n + (max + 1)*Math.random());
    }
}
```

`Delay` enthält den entsprechenden Auszug aus der Klasse `SlowKobold`:

```
public class Delay implements Behaviour {
    private final int delay;

    public Delay(int d) {
        delay = Math.max(0, Math.min(10, d));
    }

    public int modify(int n) {
        try {
            Thread.sleep(1000*delay);
        }
        catch(InterruptedException ex) {}
        return n;
    }
}
```

Die Klasse `Friend` referenziert einen anderen Kobold als Freund des Kobolds, dem dieses Verhalten zugesprochen wird:

```
public class Friend implements Behaviour {
    private final Kobold friend;

    public Friend(Kobold f) {
        friend = f;
    }

    public int modify(int n) {
        return friend.messUp(n);
    }
}
```

Hier lauern zwei Fallen: Zum einen muss mit einem Argument `null` als Freund gerechnet werden. Im einfachsten Fall wird dieser Wert abgelehnt:

```
    public Friend(Kobold f) {
        if(f == null)
            throw new NullPointerException("Freund existiert nicht!");
        friend = f;
    }
    ...
```

Zum anderen würde ein narzistischer Kobold, der sich selbst zum Freund hat, ein Problem aufwerfen: Der Aufruf von `messUp` würde endlos im Kreis laufen und schließlich zum Abbruch durch Endlosrekursion führen. Entsprechendes würde drohen, wenn sich zwei oder mehr Kobolde gegenseitig beziehungsweise reihum zu Freunden erklären. Allerdings lässt sich eine solche zyklische Abhängigkeit überhaupt nicht konstruieren, weil Kobolde nur zeitlich einer nach dem anderen erzeugt werden können. Anders ausgedrückt, ein Kobold muss bereits existieren, bevor er zum Freund erklärt werden kann.

Die letzte Klasse, `Odds`, beschreibt ein Art »Meta-Verhalten«. Sie repräsentiert kein eigenständiges Verhalten, sondern entscheidet über die Wirkung eines anderen Verhaltens.[7]

[7] Auch hier müsste ein Schutz gegen `null` vorgesehen werden.

```
public class Odds implements Behaviour {
    private final Behaviour other;

    public Odds(Behaviour o) {
        other = o;
    }

    public int modify(int n) {
        return n%2 == 0? n: other.modify(n);
    }
}
```

Mit diesen Klassen können fast beliebige Kobold-Arten definiert werden:

```
public static void main(String... args) {
    // siehe Teilaufgabe 1
    koboldTest(new Kobold("Max", new Blur(1)));
    koboldTest(new Kobold("Moritz", new Blur(2000)));

    // siehe Teilaufgabe 2
    koboldTest(new Kobold("Pumuckl", new Delay(1)));
    koboldTest(new Kobold("BlauerKlabauter", new Delay(5)));

    koboldTest(new Kobold("Max+Pumuckl", new Blur(1), new Delay(1)));

    Kobold max = new Kobold("Max", new Blur(1));
    Kobold pumuckl = new Kobold("Pumuckl", new Delay(1));
    koboldTest(new Kobold("Freak",
                          new Odds(new Delay(5)),
                          new Friend(max),
                          new Friend(pumuckl)));
}
```

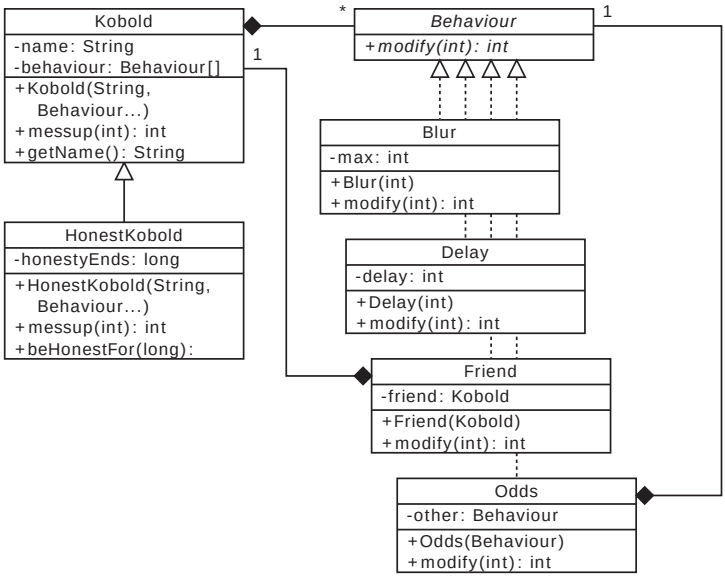
31.4 Ehrliche Kobolde

Eine neue Art von Kobolden zeigt Ansätze von Aufrichtigkeit im Gegensatz zu den Kobolden aus den vorhergehenden Teilaufgaben. Für eine gegebene Zeitspanne sieht die neue Kobold-Art von allem Unfug ab und liefert jede überlassene Zahl ohne Manipulation und unverändert wieder ab. In dieser Spanne der Aufrichtigkeit ist das »normale« Verhalten des Kobolds vorübergehend stillgelegt. Nach Ablauf der Aufrichtigkeitsdauer benimmt sich der Kobold wieder genauso wie vorher.

Die Aufrichtigkeit kann nicht als zusätzliches Verhalten implementiert werden. `Behaviour`-Objekte eines Kobolds kennen sich untereinander nicht und kön-

nen sich gegenseitig nicht beeinflussen. Genau das ist hier aber gefordert: Die Aufrichtigkeit ersetzt zeitweise alle Verhaltensweisen und steht damit auf einer anderen Ebene als die Verhaltensweisen.

Leiten Sie von `Kobold` eine neue Klasse `HonestKobold` ab. Die Entscheidung über die Aufrichtigkeit fällt im Kobold-Objekt und nicht in Verhaltensobjekten. `HonestKobold` bietet einen zusätzlichen Setter an, der eine Anzahl Millisekunden akzeptiert. Alle `messUp`-Aufrufe in der entsprechenden Zeitspanne liefern das Argument sofort und unverändert zurück. Hier sehen Sie, wie `HonestKobold` mit den übrigen Typen zusammenspielt:



Lösung

Ein `HonestKobold` führt Buch, wann die Aufrichtigkeitsspanne endet. In einer Objektvariablen wird die entsprechende Systemzeit in Millisekunden aufgezeichnet:

```
public class HonestKobold extends Kobold {
    private long honestyEnds = 0;
    ...
```

Der Konstruktor eines aufrichtigen Kobolds ruft lediglich den Basisklassenkonstruktor auf:

```
    public HonestKobold(String name, Behaviour[] behaviour) {
        super(name, behaviour);
    }
    ...
```

Die Methode `beHonestFor` akzeptiert eine Dauer der Aufrichtigkeit in Millisekunden. Aus der Systemzeit des Methodenaufrufes und dem Argument wird das Ende der Aufrichtigkeitsspanne berechnet und abgespeichert.

```
    public void beHonestFor(int millis) {
        honestyEnds = System.currentTimeMillis() + millis;
    }
    ...
```

Schließlich entscheidet sich im Aufruf von `messUp`, ob im Moment ehrlich geantwortet oder manipuliert werden soll:

```
    public int messUp(int n) {
        return System.currentTimeMillis() < honestyEnds?
            n:
            super.messUp(n);
    }
}
```