Anhang C (Online)
Typänderung des Standard-Hamsters

Sicher hat es Sie bisher auch schon oft geärgert, dass der Standard-Hamster immer vom Typ `Hamster` und nicht vom Typ einer erweiterten Hamster-Klasse ist und man für ihn keine neu definierten Befehle aufrufen kann. Wir haben uns daher bisher immer mit einem Vertretungshamster begnügt. Im Folgenden wird demonstriert, wie Sie dem Standard-Hamster durch Anwendung des Singleton-Musters (siehe Kapitel 14.4.2.1) andere Typen verpassen können.

C.1 Klasse `SHamster`

Zunächst wird eine erweiterte Hamster-Klasse `SHamster` definiert. Diese enthält einen als `protected` deklarierten Konstruktor, der einem internen Hamster-Attribut `sHamster` den Standard-Hamster zuweist. Alle anderen Konstruktoren weisen dem Attribut den Wert `null` zu. Ansonsten überschreibt die Klasse `SHamster` alle Methoden der Klasse `Hamster`, und zwar auf immer die gleiche Art und Weise: Wenn das Attribut `sHamster` den Wert `null` enthält, wird die entsprechende geerbte Methode aufgerufen. Andernfalls wird der Methodenaufruf an den Standard-Hamster delegiert.

```
public class SHamster extends Hamster {

    private Hamster sHamster;

    protected SHamster(boolean dummy) {
        this.sHamster = Hamster.getStandardHamster();
    }

    public SHamster() {
        super();
        this.sHamster = null;
    }

    public SHamster(int reihe, int spalte, int blickrichtung,
            int anzahlKoerner) {
        super(reihe, spalte, blickrichtung, anzahlKoerner);
        this.sHamster = null;
    }

    public SHamster(Hamster hamster) {
        super(hamster.getReihe(), hamster.getSpalte(), hamster
                .getBlickrichtung(), hamster.getAnzahlKoerner());
        this.sHamster = null;
    }

    public void init(int reihe, int spalte, int blickrichtung,
            int anzahlKoerner) {
        if (this.sHamster == null) {
            super.init(reihe, spalte, blickrichtung,
                    anzahlKoerner);
```

```
        } else {
            this.sHamster.init(reihe, spalte, blickrichtung,
                    anzahlKoerner);
        }
    }

    public void vor() {
        if (this.sHamster == null) {
            super.vor();
        } else {
            this.sHamster.vor();
        }
    }

    public void linksUm() {
        if (this.sHamster == null) {
            super.linksUm();
        } else {
            this.sHamster.linksUm();
        }
    }

    public void gib() {
        if (this.sHamster == null) {
            super.gib();
        } else {
            this.sHamster.gib();
        }
    }

    public void nimm() {
        if (this.sHamster == null) {
            super.nimm();
        } else {
            this.sHamster.nimm();
        }
    }

    public boolean vornFrei() {
        if (this.sHamster == null) {
            return super.vornFrei();
        } else {
            return this.sHamster.vornFrei();
        }
    }

    public boolean maulLeer() {
        if (this.sHamster == null) {
            return super.maulLeer();
        } else {
            return this.sHamster.maulLeer();
        }
    }

    public boolean kornDa() {
        if (this.sHamster == null) {
            return super.kornDa();
        } else {
            return this.sHamster.kornDa();
        }
```

```
    }

    public void schreib(String zeichenkette) {
        if (this.sHamster == null) {
            super.schreib(zeichenkette);
        } else {
            this.sHamster.schreib(zeichenkette);
        }
    }

    public String liesZeichenkette(String aufforderung) {
        if (this.sHamster == null) {
            return super.liesZeichenkette(aufforderung);
        } else {
            return this.sHamster.liesZeichenkette(aufforderung);
        }
    }

    public int liesZahl(String aufforderung) {
        if (this.sHamster == null) {
            return super.liesZahl(aufforderung);
        } else {
            return this.sHamster.liesZahl(aufforderung);
        }
    }

    public int getReihe() {
        if (this.sHamster == null) {
            return super.getReihe();
        } else {
            return this.sHamster.getReihe();
        }
    }

    public int getSpalte() {
        if (this.sHamster == null) {
            return super.getSpalte();
        } else {
            return this.sHamster.getSpalte();
        }
    }

    public int getBlickrichtung() {
        if (this.sHamster == null) {
            return super.getBlickrichtung();
        } else {
            return this.sHamster.getBlickrichtung();
        }
    }

    public int getAnzahlKoerner() {
        if (this.sHamster == null) {
            return super.getAnzahlKoerner();
        } else {
            return this.sHamster.getAnzahlKoerner();
        }
    }

    protected Object clone() {
        if (this.sHamster == null) {
```

```
            return new SHamster(this);
        } else {
            return new SHamster(this.sHamster);
        }
    }

    public boolean equals(Object hamster) {
        if (this.sHamster == null) {
            return super.equals(hamster);
        } else {
            return this.sHamster.equals(hamster);
        }
    }

    public String toString() {
        if (this.sHamster == null) {
            return super.toString();
        } else {
            return this.sHamster.toString();
        }
    }
}
```

C.2 Ableiten von der Klasse SHamster

Wenn Sie nun eine erweiterte Hamster-Klasse definieren wollen, von deren Typ (auch) der Standard-Hamster sein soll, müssen Sie diese von der Klasse `SHamster` ableiten. Diese Klasse kann wie üblich neue Attribute und Methoden definieren. Zusätzlich muss sie jedoch analog zum Singleton-Muster klassenintern über einen `protected`-Konstruktor, der den `protected`-Konstruktor der Oberklasse `SHamster` aufruft, ein Objekt der neuen Klasse erzeugen und über eine get-Klassenmethode nach außen liefern. Das Singleton-Muster wird hier nicht in seiner reinen Form angewendet. Über andere Konstruktoren als den `protected`-Konstruktor ist es wohl noch möglich, weitere Objekte der Klasse zu erzeugen. Die folgende Klasse `DrehHamster` demonstriert das Prinzip:

```
public class DrehHamster extends SHamster {

    // Singleton-Muster

    private static DrehHamster standardHamster = new DrehHamster(
            true);

    protected DrehHamster(boolean dummy) {
        super(dummy);
    }

    public static DrehHamster getStandardHamsterAlsDrehHamster() {
        return standardHamster;
    }

    // normale Konstruktoren und Methoden

    public DrehHamster(int r, int s, int b, int k) {
        super(r, s, b, k);
    }

    public DrehHamster(Hamster ham) {
```

```
        super(ham);
    }

    public void kehrt() {
        this.linksUm();
        this.linksUm();
    }

    public void rechtsUm() {
        this.kehrt();
        this.linksUm();
    }

}
```

C.3 Standard-Hamster mit anderem Typ

Von der Klasse `DrehHamster` können nun zum einen normale Hamster-Objekte erzeugt werden. Zum anderen ist es jedoch auch möglich, den Standard-Hamster als Objekt des neuen Typs zu betrachten und für ihn Methoden der Klasse `DrehHamster` aufzurufen:

```
void main() {
    // neuer normaler DrehHamster
    DrehHamster willi = new DrehHamster(1, 2, Hamster.OST, 3);
    while (willi.vornFrei()) {
        willi.vor();
    }
    willi.rechtsUm();

    // Standard-Hamster als DrehHamster
    DrehHamster paul = DrehHamster
            .getStandardHamsterAlsDrehHamster();
    while (paul.vornFrei()) {
        paul.vor();
    }
    paul.rechtsUm();
}
```