Repetitorium Mo14 Angabe

Hinweis: Das Implementieren und Lösen der Aufgaben ist dem Studenten überlassen. Bei diesem Repetitorium geht's hauptsächlich um die Theorie hinter den Funktionen. Verwenden Sie auch die von uns bereitgestellte Ordnerstruktur. Die Aufgaben bauen aufeinander auf. Die Lösung einer Aufgabe dient zur Lösung der nächsten Aufgabe.

Aufgabe1: *Objektorientierung*

Erstellen Sie zwei Klassen *Person* und *Dog*. Beide haben einen Namen, ein Gewicht und eine Größe. All diese Informationen sind unveränderbar. Das Gewicht und die Größe der beiden werden per Zufall entschieden und sind vom Typ float.

Person: Jede Person kann einen Hund als Haustier haben.

@ Konstruktor: drei unterschiedliche Konstruktoren.

- *Person(String name)*
- *Person(String name, Dog pet)*
- *Person(Person other)*

@ Methoden:

- *Getter-Methoden* : Um die Informationen auszugeben.
- *void setPet(Dog pet)* : Weist dieser Person ein Haustier zu.
- *boolean isTallerThan(Person other)* : Vergleicht, ob this Person größer ist als other.
- *boolean equals(Object o)* : Untersucht this und o auf Gleichheit.
- *String toString()* : Retourniert eine String Repräsentation dieses Objektes. Wenn die Person kein Haustier hat, so soll „none" ausgegeben werden. Gewicht und Höhe werden auf genau zwei Nachkommastellen formatiert.
  Format:
  "Person called <Name>, <Höhe> tall, <Gewicht> heavy and pet <Haustiername>"

Dog: Jeder Hund kann einen Besitzer haben.

@ Konstruktor:

- *Dog(String name, Person owner)*
- Die zwei anderen sind analog zu Person.

@ Methoden:

- *void setOwner(Person owner)* : Weist diesem Hund einen Besitzer zu.
- Analog zu Person nur ohne die Methode *isTallerThan.*

Aufgabe2: *Datenstrukturen*

Erstellen Sie eine Klasse *SortedList*, die *Person* Objekte aufsteigend nach deren Größe sortiert speichert. Wie üblich ist eine Liste am Anfang ihrer Erstellung leer.

@ Konstruktor:

- *SortedList()*

@ Methoden:

- *boolean isEmpty()*
- *int size()*
- *void add(Person p)* : Fügt eine *Person* der Liste hinzu. Dabei wird diese neue *Person* entsprechend ihrer Größe an der richtigen Stelle in der Liste gespeichert.
- *Person get(int index)*
- *Iterator<Person> iterator()* : Retourniert ein neues *Iterator* Objekt, um alle Elemente in der Liste zu betrachten.

Aufgabe3: *Untertypbeziehung*

Lesen Sie die Beziehung der einzelnen Objekte untereinander und erstellen Sie anschließend Klassen, die die beschriebene Hierarchie darstellen. Versuchen Sie eine sinnvolle Struktur mit wenigen Klassen und viel Wiederverwendbarkeit zu erstellen. Vergessen Sie auch nicht, dass reine Vererbung das Prinzip der Untertypbeziehung nicht widerspiegelt.

- Eine Person ist ein Wesen.
- Ein Hund ist ein Wesen.
- Tiere und Personen haben viele Gemeinsamkeiten.
- Ein Tiger, genauso wie ein Hund, ist ein Tier, aber kein Haustier.
- Personen können sich bewegen.
- Ein Hund ist ein Haustier.
- Auch Tiere können sich bewegen.

Des Weiteren erweitern Sie die Klasse *SortedList* so, dass sie generisch ist. *SortedList* soll nun in der Lage sein mit allen Arten von *vergleichbaren* Objekten umgehen zu können. Durch diese Erweiterung können Sie nun die gleiche Listenstruktur verwenden, um auch Hunde zu verwalten und nicht nur Personen. Dabei ist zu beachten, dass

- Personen weiterhin nach deren Größe und
- Hunde nach deren Gewicht

in der Liste sortiert werden.

Aufgabe4: *Dynamisches Binden*

Erweitern Sie die Klassen *Person*, *Dog* und eine weitere Klasse Ihrer Wahl, um eine Kommunikation zwischen den drei Objekten zu simulieren. Je nachdem wer mit wem interagiert, soll eine gewisse Reaktion entstehen. Eine einfache Beschreibung (System.out) der Tat reicht als Implementierung.

@ Methoden: alle drei Klassen sollen die folgenden Methoden implementieren.

- *void acts(…)*
- *void reacts(…)*

If-Abfragen sind nicht erlaubt.

Aufgabe5: *Design by Contract*

Verschönern Sie Ihren Code mithilfe von Vor- und Nachbedingungen und wenn möglich auch mit Invarianten. Achten Sie dabei auf die von Ihnen erstellte Hierarchie.

Aufgabe6: *Exceptions*

Nach dem Prinzip „Vertrauen ist gut, Kontrolle ist besser“ erstellen Sie eine Klasse *RepException* , die geworfen wird, sobald gravierende Fehler entstehen bzw. um solche zu verhindern.