Höhere Programmierkonzepte – Testklausur

Prof. Dr. Nikolaus Wulff

Januar 2017

1 Ein Google-Map Algorithmus *(5 Punkte)*

```
1
2  typedef void* X;
3  typedef void* Y;
4
5  void map(unsigned int n      /* tuple length          */,
6           X x                 /* X tuple pointer       */,
7           size_t sizeof_X     /* sizeof(x[0])          */,
8           Y y                 /* Y tuple pointer       */,
9           size_t sizeof_Y     /* sizeof(y[0])          */,
10          void (*f)(X , Y )   /* f: X -> Y fct-pointer */
11         );
```

Listing 1: Protypdeklartion für den Map-Algorithmus in C.

Zur Verarbeitung von großen Datenmengen (*BigData*) hat Google den sogenannten MapReduce-Algorithmus patentieren lassen. Diese Aufgabe beinhaltet in einer vereinfachten Form den Map-Teil. Zu einem gegebenen Eingabetupel $[x_1, \ldots, x_n] \in X^n$ berechnet der Map-Algorithmus komponentenweise für eine beliebige unäre Funktion $f : X \mapsto Y$ das Ausgabetupel $[y_1, \ldots, y_n] \in Y^n$ als: $[y_1, \ldots, y_n] := \text{map}(f)([x_1, \ldots, x_n]) \equiv [f(x_1), \ldots, f(x_n)]$.
Hierbei sind X und Y beliebige Datentypen, so dass sich dieser Algorithmus recht einfach mit Java Generics implementieren lässt – aber auch einer C Implementierung mit geeignet allgemein definierten Datentypen steht nichts im Wege. Das Listing (1) zeigt die allgemeine Deklartion eines solchen Map-Algorithmus. Die Elemente aus X und Y sind als strukturlose `void*` Zeiger modelliert worden. Um eine sinnvolle Schleife über alle Tupelelemente ausführen zu können wird daher die jeweilige Größe eines der Elemente im zugehörigen `sizeof_` Parameter an die Funktion übergeben, ebenfalls die Länge der Tupel, sowie die Tupel selber, welche vom Aufrufenden vorher geeignet alloziert sein müssen.

Aufgabe

1. Implementieren Sie den Map Algorithmus in C passend zur Prototypdelaration (1). Beachten Sie insbesondere den richtigen Einsatz von Zeigern und wie sich `void*`-Zeiger inkrementieren lassen.

2. Geben Sie eine generische Java Schnittstelle für die unäre Funktion f an.

3. Geben Sie eine generische Implementierung des Map Algorithmus in Java an unter Einbeziehung der Schnittstelle aus Teilaufgabe 2). Beachten Sie, dass sich die Signatur der Map-Methode in Java erheblich von der C Deklaration unterscheiden wird, da z.B. die Tupel-Länge nicht mehr erforderlich ist (Java Arrays kennen ihre Länge) etc.

2 Produzenten-Konsumer Problem

```
1  /**
2   * RingBuffer for items of type V.
3   */
4  public interface Buffer<V> {
5      /**
6       * Put an item of type V into the buffer.
7       * The put operation blocks if the buffer is full.
8       * @param item to store
9       * @throws InterruptedException if blocking thread is interrupted
10      */
11     void put(V item) throws InterruptedException;
12     /**
13      * Get an item of type V from the buffer.
14      * The take operation blocks if the buffer is empty.
15      * @return item to retrieve
16      * @throws InterruptedException if blocking thread is interrupted
17      */
18     V take() throws InterruptedException;
19 }
```

Listing 2: Schnittstelle eines generischen zirkularen Puffers.

Produzenten erzeugen quasi gleichzeitig Daten, die von Konsumenten verarbeitet werden sollen. Die Weitergabe der Daten von den Produzenten an die Konsumenten erfolgt mittels eines Puffers, der maximal k Daten vom Typ V speichern kann. Falls der Puffer voll ist sollen die Produzenten warten – d.h. `put` blockiert -, falls der Puffer leer ist sollen die Konsumenten warten – d.h. `take` blockiert.

Aufgabe

- Entwickeln Sie eine thread-sichere `Buffer` Implementierung der Schnittstelle (2) als generische Klasse `RingBuffer`.

Tip

Die Aufgabe hat Ähnlichkeit mit dem `ResourcePool`, nur dass jetzt die Produzenten und Konsumenten vollkommen unabhängig voneinander agieren und die Objekte der `put`-Operation nicht vorher mit `take` entnommen wurden. Es reicht daher ein einziges Synchronisierungsobjekt wie im `ResourcePool` nicht aus. Es werden daher zwei Semaphore oder zwei Lock-Objekte mit wait-notify Semantik benötigt.

3 Generische Prototyp Fabrik *(5 Punkte)*

```
1  /**
2   * Marker interface for IPrototype classes with default constructor.
3   */
4  interface IPrototype {  }
5  /**
6   * Factory to create prototype instances.
7   */
8  public class Factory {
9      /**
10      * Create a list  filled with "n" fresh instances of the given prototype.
11      */
12     public List  create(IPrototype prototype, int n) throws Exception {
13         ArrayList list  = new ArrayList();
14         for(int j=0; j<n; j++) list.add(create(prototype));
15         return list;
16     }
17    /**
18     * Create a new instance of the given prototype.
19     */
20     public IPrototype  create(IPrototype  prototype) throws Exception {
21         Class type = prototype.getClass();
22         return create(type);
23     }
24     /**
25      * Create a new instance of the given type.
26      */
27      public IPrototype  create(Class type) throws Exception {
28          IPrototype  obj = (IPrototype ) type.newInstance();
29          return obj;
30      }
31 }
```

Listing 3: Implementierung einer Fabrik zur Erzeugung von Prototyp Instanzen.

Im Listing (3) wird die Implementierung einer Prototyp Fabrik gezeigt. Diese Fabrik erzeugt beliebig viele neue Instanzen eines an die `create` Methode übergebenen Objekts. Das Objekt muß das Markerinterface `IPrototype` erweitern und einen public Konstruktor ohne Argumente besitzen, damit die Fabrik weitere Instanzen dieser Klasse erzeugen kann.
Ob es sich um Instanzen von `IPrototyp` handelt, wird in den ersten beiden `create` Methoden durch die Signatur erzwungen und vom Compiler überprüft. In der dritten `create` Methode ist diese Bedingung nicht mehr erkennbar und der Compiler kann nicht sicherstellen, dass das übergebene Class Objekt auf implementierende Klassen vom Typ `IPrototyp` abzielt. Diese Implementierung einer Fabrik entspricht dem Programmierstil vor der Einführung von ***Java Generics*** und garantiert noch keine Typsicherheit. Dies ist auch in dem illustrierenden Anwendungsbeispiel (4) erkennbar: Eine einzige Fabrik ist in der Lage beliebig viele Instanzen von beliebig vielen unterschiedlichen `IPrototyp` Klassen zu erzeugen. Im Beispiel sind dies die Klassen `Foo` und `Bar`, der Typ der `Foo` Instanz wird per Cast erzwungen, dass sich in der Liste `Bar` Objekte befinden sollen ist nicht mehr ersichtlich.

Das die Fabrik noch nicht mittels Generics arbeitet ist an dem Cast des Rückgabewertes der Fabrik und innerhalb der Implementierung zu erkennen. Seit der Einführung der Java Generics gehören solche Casts und Inkonsistenzen der Vergangenheit an.

```
1  class Bar implements IPrototype {
2      public Bar() {}
3  }
4  class Foo implements IPrototype {
5      public Foo() {}
6  }
7
8  public class Example {
9      public static void main(String[] args) throws Exception {
10         Factory factory  = new Factory();
11
12         Foo prototype1 = new Foo();
13         Bar prototype2 = new Bar();
14
15         Foo  inst = (Foo) factory.create(prototype1);  // one Foo instance
16         List list = factory.create(prototype2,10);     // ten Bars in List
17
18         // do something with Foo and Bar ...
19     }
20 }
```

Listing 4: Verwendung der Prototyp Fabrik.

Aufgabe

Die Prototyp Fabrik soll mit Java Generics realisiert werden. Hierzu müssen sowohl die `IPrototyp` Schnittstelle, als auch die Signaturen der Fabrik Methoden neu definiert und ausimplementiert werden.

1. *(1 Punkt)*
   Definieren Sie das Prototyp Markerinterface mit Hilfe von Generics typsicher als `IPrototyp<...>` neu, so dass der Datentyp und die Vererbungshierarchie für den Compiler überprüfbar ist.

2. *(3 Punkte)*
   Schreiben Sie eine neue Implementierung der Fabrik, passend zur neuen `IPrototyp` Schnittstelle.

3. *(1 Punkt)*
   Schreiben Sie das Beispiel (4) angepasst für Generics neu.

Tip

Die Methoden selber sind nur 2 – 3 Zeiler, sie brauchen daher die Kommentare des Quelltexts nicht mit abschreiben. Die Klausur soll nicht in Schreibarbeit ausarten. Achten Sie bei der Definition der neuen `IPrototyp<...>` Schnittstelle darauf, dass der generische Typ auch selber die `IPrototyp` Schnittstelle erweitert, damit so etwas wie `IPrototyp<String>` oder `IPrototyp<Object>` nicht möglich ist und bereits vom Compiler als Fehler verworfen wird.

4 Test einer generischen Operation *(5 Punkte)*

```
1 interface  Operation<A, R, E extends Exception> {
2       R operate(A...args)  throws E;
3 }
```

Listing 5: Generische Operation-Schnittstelle.

Die Schnittstelle (5) definiert eine generische Operation, mit ihren Argument(en) vom Typ **A** als *Varargs*, einem Ergebnistyp **R** und einer möglichen geworfenen Ausnahme vom Typ **E**.
Passend zur Schnittstelle (5) ist der generische JUnit Test (6) entwickelt worden. Überprüft werden das Verhalten bei richtigen, bei illegalen und bei fehlenden Argument(en). Die drei finalen Testmethoden definieren die minimalen Anforderung zur Überprüfung einer `Operation`-Implementierung.
Konkrete Testklassen müssen die drei abstrakten `create`-Methoden zum `setUp` des Tests und zwei Validierungsmethoden für das Ergebnis und für geworfene Ausnahmen zur Verfügung stellen. Die Validierungsmethoden liefern ein `true` zurück, falls das Ergebnis bzw. die Ausnahme der Testerwartung entsprechen, andernfalls `false`. Ein Entwickler kann sich unter Verwendung dieses Basistests auf weitere fachliche Tests konzentrieren.

Aufgabe

1. *(2 Punkte)*
   Implementieren sie die Klasse `SquareRoot`, die obige `Operation` Schnittstelle implementiert. `SquareRoot` liefert die Wurzel einer reelen, positiven Zahl (Typ **Double** in Groß!) und wirft eine `IllegalArgumentException` falls das Argument negativ ist oder nicht genau ein Double als *Vararg* übergeben wird.

2. *(2 Punkte)*
   Schreiben Sie die Klasse `SquareRootTest` und implementieren Sie die fehlenden abstrakten Methoden als Erweiterung von `OperationTest`, so dass die Klasse `SquareRoot` sinnvoll getestet wird.

3. *(1 Punkt)*
   Es gibt mehr als nur eine Möglichkeit illegale Argumente an die `Operation` zu übergeben. Es ist daher noch eine weitere Testmethode notwendig, so dass Sie sowohl den Fall eines negativen Arguments, als auch den Fall von mehr als einem *Varargs*-Argument testen können. Sehen Sie daher noch eine weitere Testmethode vor, die den jeweils fehlenden dieser zwei Fälle zusätzlich überprüft.

Tip

Damit Sie nicht seitenlange JUnit Klassen schreiben müssen bietet Ihnen das Listing (6) eine Vorlage und Anleitung, für Ihre eigene Testklasse. Sie müssen nur die fünf abstrakten Methoden und eine zusätzliche Testmethode implementieren. Sie müssen `OperationTest` nicht abschreiben, sondern nur sinnvoll erweitern.

```
1  public abstract class OperationTest<A,R,E extends Exception> {
2    protected Operation<A,R,E> undertest;
3    protected A[] validArgs;
4    protected A[] invalidArgs;
5
6    /** Helper method to validate the return of the test.        */
7    protected abstract boolean validateReturn(R r);
8    /** Helper method to validate an exception of the test.      */
9    protected abstract boolean validateError(Exception e);
10   /** Helper method to create the operation under test.        */
11   protected abstract Operation<A,R,E> createOperation();
12   /** Helper method to create valid test arguments.            */
13   protected abstract A[] createArgs();
14   /** Helper method to create invalid test arguments.          */
15   protected abstract A[] createInvalidArgs();
16
17     /**
18      * Setup of the test case(s).
19      */
20     @Before public final void setUp() throws Exception {
21         undertest = createOperation();
22         invalidArgs = createInvalidArgs();
23         validArgs = createArgs();
24     }
25     /**
26      * Test with legal arguments, validating the return value.
27      */
28     @Test public final void testValidArgs() throws Exception {
29         R retValue = undertest.operate(validArgs);
30         assertTrue("wrong result:"+retValue, validateReturn(retValue));
31     }
32     /**
33      * Test with illegal arguments, validating the exception.
34      */
35     @Test public final void testInvalidArgs() throws Exception {
36         try {
37             undertest.operate(invalidArgs);
38             fail ("invalid arguments not detected");
39         } catch(Exception error) {
40             assertTrue("wrong error:"+error, validateError(error));
41         }
42     }
43     /**
44      * Test for a NullPointer as argument.
45      */
46     @Test(expected=NullPointerException.class)
47     public final void testNullArgs() throws Exception {
48         A[] nullArgs = null;
49         undertest.operate(nullArgs);
50     }
51 }
```

Listing 6: Generischer JUnit Test für eine Operation.