

Musterlösung Übung 10

Software-Tests (Modultests, Blackbox-Test, Akzeptanztest)

1. Bauen Sie in die rekursive Implementierung des einfachen `int`-basierten Fakultät-Berechnungsprogramms aus der Vorlesung ein Testmodul ein, das für Eingabewerte von -1 bis +1000 den berechneten Funktionswert auf Plausibilität überprüft, indem einmal mit dem „Vorgängerwert“ und einmal mit dem „korrekten Wert“ (unter Zuhilfenahme einer `BigInteger`-Variante) verglichen wird. Protokollieren Sie die Ausgaben (Erwartungswert, berechneter Wert, OK oder ERROR).

Siehe Verzeichnis [uebung10-musterloesung](#), die erzeugte Ausgabe enthält ein typisches OK/FEHLER-Protokoll für eine solche Testroutine. Besonders interessant ist hier der Wert für `berechne(13)`, bei dem das Ergebnis zwar noch „richtig aussieht“, weil es größer ist als sein Vorgänger, tatsächlich aber bereits vom mit `BigInteger` errechneten Ergebnis abweicht.

Code:

```
import java.math.*;

public class Fakultaet {

    // Das ist die eigentliche Berechnungsroutine
    static int berechne(int i) {
        if(i <= 1) return 1;
        return i * berechne(i-1);
    }

    // Das ist die "genaue" Version
    static BigInteger fakultaet_2(int i) {
        if (i <= 1) return BigInteger.ONE;
        return BigInteger.valueOf(i).multiply(fakultaet_2(i-1));
    }

    static void test(int von, int bis) {
        int vorgaenger = -1;
        for(int i=von; i<=bis; i++) {
            boolean fehler = false;
            int wert = berechne(i);
            BigInteger erwartet = fakultaet_2(i);
            if(vorgaenger > wert) {
                System.out.println("FEHLER: berechne(" + i + ")=" + wert +
                    " < vorgaenger=" + vorgaenger);
                fehler = true;
            }
            vorgaenger = wert;
            if(!BigInteger.valueOf(wert).equals(erwartet)) {

```

```

        System.out.println("FEHLER: berechne(" + i + ")=" + wert +
            " Erwartet: " + erwartet);
        fehler = true;
    }
    if(!fehler) System.out.println("OK: berechne(" + i + ")=" + wert);
}
}

public static void main(String[] args) {
    int von = -1, bis = 1000;
    System.out.println("Lasse Testschleife für berechne(i) laufen " +
        "von i=" + von + " bis i=" + bis);

    test(von,bis);
}
}

```

Protokoll (Ausschnitt):

```

Lasse Testschleife für berechne(i) laufen von i=-1 bis i=1000
OK: berechne(-1)=1
OK: berechne(0)=1
OK: berechne(1)=1
OK: berechne(2)=2
OK: berechne(3)=6
OK: berechne(4)=24
OK: berechne(5)=120
OK: berechne(6)=720
OK: berechne(7)=5040
OK: berechne(8)=40320
OK: berechne(9)=362880
OK: berechne(10)=3628800
OK: berechne(11)=39916800
OK: berechne(12)=479001600
FEHLER: berechne(13)=1932053504 Erwartet: 6227020800
FEHLER: berechne(14)=1278945280 < vorgaenger=1932053504
FEHLER: berechne(14)=1278945280 Erwartet: 87178291200
FEHLER: berechne(15)=2004310016 Erwartet: 1307674368000
FEHLER: berechne(16)=2004189184 < vorgaenger=2004310016
FEHLER: berechne(16)=2004189184 Erwartet: 20922789888000

```

2. Versuchen Sie, anhand eines **Blackbox** Tests mit verschiedenen Eingabe-Zeichenketten herauszufinden, was die statische Klassenmethode `public static String geheimnis(String eingabe)` aus der nur als Compilat vorliegenden Datei **Geheim.class** eigentlich tut. Sie können hierzu, um nicht für jeden Test neu compilieren zu müssen, auch wieder die Klasse `Eingabe.java` zuhulfe nehmen, wie in diesem Code-Fragment gezeigt:

```

public class BlackboxTest {
    public static void main(String[] args) {
        for(;;) { // Endlosschleife
            String s1 = Eingabe.readString("Eingabe: ");
            String s2 = Geheim.geheimnis(s1);
            System.out.println("Ausgabe: " + s2);
        }
    }
}

```

Beim **Reverse Engineering** versucht man, im Gegensatz zum Disassemblieren, nicht den Original-Code wiederherzustellen, sondern einen neuen Code bzw. zunächst einen abstrakten Algorithmus zu entwickeln, der **das gleiche tut wie der Originalcode** (also im Definitionsbereich identische Ergebnisse liefert), obwohl er völlig unterschiedlich aufgebaut sein kann.

Sehen wir uns eine Beispiel-Session (nach `javac BlackboxTest.java`) an.

```
java BlackboxTest
```

```
Eingabe: a
Ausgabe: n
```

Die Eingabe eines Einzelzeichens führt offenbar (zumindest in diesem Beispiel) zur Ausgabe eines anderen Einzelzeichens.

```
Eingabe: a b c
Ausgabe: n o p
```

Die Eingabe von drei im Alphabet aufeinanderfolgenden Zeichen führt hier zur Ausgabe von drei im Alphabet ebenfalls aufeinanderfolgenden Zeichen. Hier könnte man fast schon erraten, was er Algorithmus macht. Anscheinend verschiebt er jedes Zeichen um 13 Stellen nach rechts im Alphabet.

```
Eingabe: a b c x y z
Ausgabe: n o p k l m
```

Oha! Hier werden a, b, c um 13 Stellen nach rechts verschoben, aber x, y, z um 13 Stellen nach links!

```
Eingabe: l m n o
Ausgabe: y z a b
```

Die Grenze für die Entscheidung „13 nach rechts“ oder „13 nach links“ scheint in diesem Beispiel zwischen m und n zu liegen.

Eingabe: Hallo, Welt!
Ausgabe: Unyyb, Jryg!

Dies legt den Schluss nahe, dass Sonderzeichen (Leerzeichen, Satzzeichen) nicht verändert werden.

Eingabe: Unyyb, Jryg!
Ausgabe: Hallo, Welt!

Der Algorithmus ist reversibel, kann also aus einer vorigen Ausgabe die ursprüngliche Eingabe wieder berechnen.

Mit der Maßgabe, dass Groß- und Kleinschreibung keine Rolle spielt, könnte der Algorithmus also wie folgt lauten:

```
WENN der eingegebene Buchstabe zwischen A und M liegt (einschließlich),  
DANN gib den Buchstaben aus, der im Alphabet 13 Zeichen weiter liegt.  
SONST  
WENN der eingegebene Buchstabe zwischen N und Z liegt (einschließlich),  
DANN gib den Buchstaben aus, der im Alphabet 13 Zeichen davor liegt.  
SONST  
Gib den eingegebenen Buchstaben unverändert aus.
```

Entsprechende Algorithmen finden Sie unter dem Stichwort „Caesar-Verschlüsselung“ oder „rot13“ im Internet.

3. Implementieren Sie den in der vorigen Aufgabe herausgefundenen Algorithmus neu als JAVA-Klasse mit einer statischen Methode, analog der Klasse **Geheim**. Welche Methode des Software Engineering haben Sie hiermit eingesetzt?

Ein Beispiel, das mit StringBuffer arbeitet, um einfach Zeichen anhängen zu können.

```
public class Geheim {  
    public static String geheimnis(String eingabe) {  
        StringBuffer tmp = new StringBuffer();  
        for (int i=0; i<eingabe.length(); i++) {  
            int abyte = eingabe.charAt(i); // i-tes Zeichen  
            int cap = abyte & 32; // Bit32 = Groß/Kleinschreibung?  
            abyte &= ~cap; // bit-Operation, wandelt in Großbuchstaben  
            abyte = rot13(abyte); // Verschlüsselung  
            abyte |= cap; // bit-Operation, wandelt zurück, s.o.  
            tmp.append((char)abyte); // An Ausgabestring anhängen  
        }  
        return tmp.toString(); // Als String an Aufrufer übergeben  
    }  
  
    // Der rot13-Algorithmus in einfacher Form.
```

```
static int rot13(int c){
    if( (c >= 'A') && (c <= 'M') ) return c + 13;
    else if ( (c > 'M') && c <= 'Z' ) return c - 13;
    else return c;
}

}
```

4. Wie kann man als Softwareentwickler sichergehen, dass ein Softwareprodukt nicht beim Akzeptanztest durchfällt?

*Ein absolutes Kriterium existiert zwar nicht, auch wenn **man sich akribisch an die Anforderungen und Abnahmekriterien aus dem Pflichtenheft hält**. Ohne diese notwendige Voraussetzung ist es hingegen äußerst fraglich, ob der Akzeptanztest positiv ausfällt. Die Frage lässt sich also grundsätzlich mit der Einhaltung der Abnahmekriterien und Überprüfung derselben beantworten.*