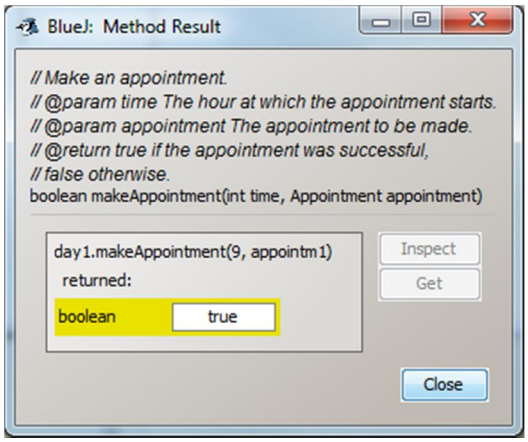
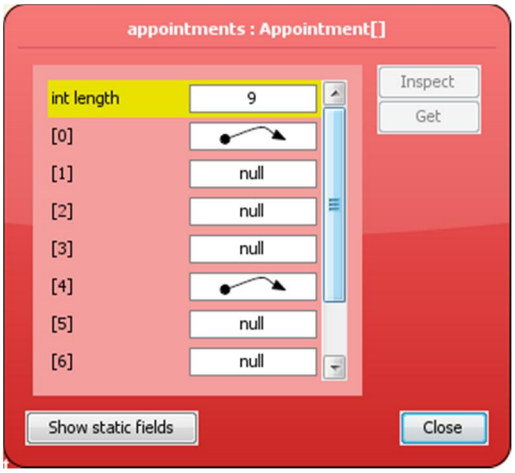
Kapitel 6.3.1

1. zu bearbeitende Aufgaben: 6.1 bis 6.5

6.1:

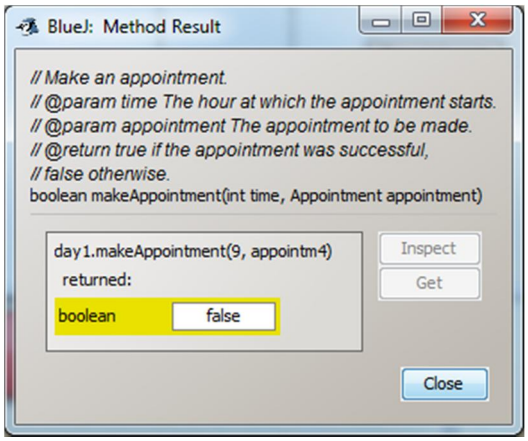


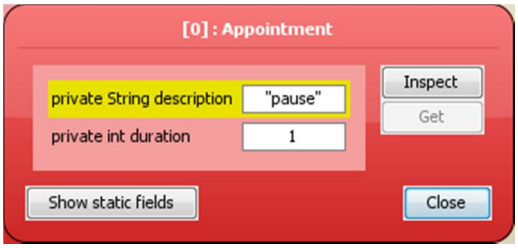


6.2:

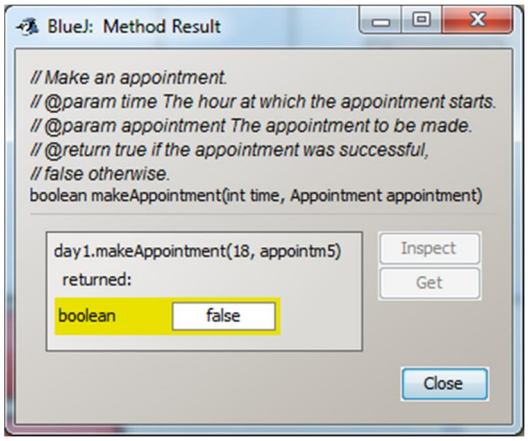
```
=== Day 23 ===
9: pause
10:
11:
12:
13: lunch
14:
15:
16:
17: meeting
```

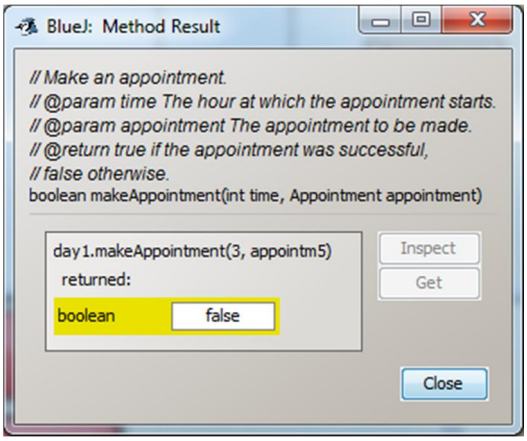
6.3:



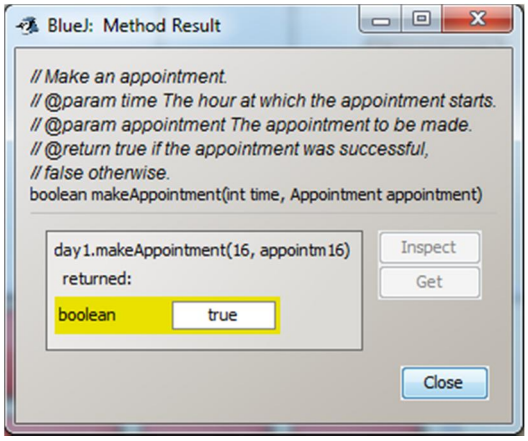


6.4:





6.5:




```
=== Day 23 ===
9: pause
10: 10 Uhr
11: 11 Uhr
12: Mittag
13: lunch
14: 14 Uhr
15: 15 Uhr
16: 16 Uhr
17: meeting
```

Kapitel 6.4.1

2. zu bearbeitende Aufgaben: 6.10 bis 6.13

6.10:

```
/**
     * Check empty-booking is not permitted.
     */
    public void testtoLargeValue()
    {
        // Set up the day with three legitimate appointments.
        makeThreeAppointments();
        Appointment badAppointment = new Appointment("Error", 1);
        day.makeAppointment(9999, badAppointment);

        // Show that the badAppointment has not been made.
        day.showAppointments();
    }
```

6.11:

```
/**
 * Write a description of class TwoHourTests here.
 *
 * @author Felix Rohrer
 * @version 1.0
 */
public class TwoHourTests
{
    // The Day object being tested.
    private Day day;

    /**
     * Constructor for objects of class OneHourTests
     */
    public TwoHourTests()
    {
        // Create a Day object that can be used in testing.
        // Individual methods might choose to create
        // their own instances.
        day = new Day(1);
    }

    /**
     * Test basic functionality by booking at either end
     * of a day, and in the middle.
     */
    public void makeThreeAppointments()
    {
        // Start with a fresh Day object.
        day = new Day(1);
        // Create three one-hour appointments.
```

```
        Appointment first = new Appointment("Java lecture", 2);
        Appointment second = new Appointment("Java class", 2);
        Appointment third = new Appointment("Meet John", 2);

        // Make each appointment at a different time.
        day.makeAppointment(9, first);
        day.makeAppointment(13, second);
        day.makeAppointment(17, third);

        day.showAppointments();
    }

    /**
     * Check that double-booking is not permitted.
     */
    public void testDoubleBooking()
    {
        // Set up the day with three legitimate appointments.
        makeThreeAppointments();
        Appointment badAppointment = new Appointment("Error", 2);
        day.makeAppointment(9, badAppointment);

        // Show that the badAppointment has not been made.
        day.showAppointments();
    }

    /**
     * Check empty-booking is not permitted.
     */
    public void testtoLargeValue()
    {
        // Set up the day with three legitimate appointments.
        makeThreeAppointments();
        Appointment badAppointment = new Appointment("Error", 2);
        day.makeAppointment(9999, badAppointment);

        // Show that the badAppointment has not been made.
        day.showAppointments();
    }

    /**
     * Test basic functionality by filling a complete
     * day with appointments.
     */
    public void fillTheDay()
    {
        // Start with a fresh Day object.
        day = new Day(1);
        for(int time = Day.START_OF_DAY;
                    time <= Day.FINAL_APPOINTMENT_TIME;
                        time++) {
            day.makeAppointment(time,
                                new Appointment("Test " + time, 1));
        }

        day.showAppointments();
    }
}
```

6.12:

```
    /**
     * Test two booking during the same time (overlap)
     */
    public void testOverlapBooking()
    {
        // Start with a fresh Day object.
        day = new Day(1);
        // Create three one-hour appointments.
        Appointment first = new Appointment("3h Meeting", 3);
        Appointment second = new Appointment("1h Lunch", 1);

        // Make each appointment at a different time.
        day.makeAppointment(11, first);
        day.makeAppointment(13, second);

        day.showAppointments();

    }
```

```
=== Day 1 ===
9:
10:
11: 3h Meeting
12: 3h Meeting
13: 3h Meeting
14:
15:
16:
17:
```

6.13:

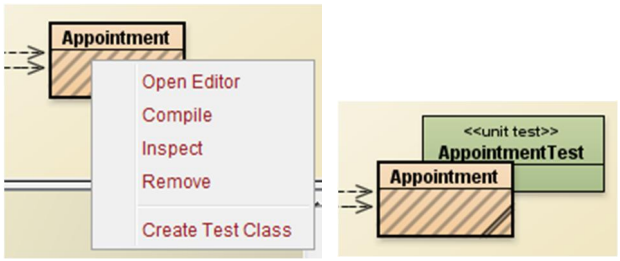
Die Tests müssen manuell ausgeführt werden und sind dadurch aufwendig und werden nicht immer durchgeführt.

In grossen, aufwendigen Projekten wird dies viel zu viel Zeit in Anspruch nehmen. Die Ergebnisse manuell zu überprüfen ist fehleranfällig.

Kapitel 6.4.2

3. zu bearbeitende Aufgaben: 6.14 bis 6.15

6.14:



6.15:

```
setUp()
tearDown()
```

Kapitel 6.4.3

4. zu bearbeitende Aufgaben: 6.16 bis 6.18

6.16:

```
/**
 * Check if findSpace returns 10 if a day already has a single one-hour
 * appointment at 9 a.m.
 */
public void testFindSpace10()
{
    Day day1 = new Day(1);
    Appointment appointm1 = new Appointment("Java lecture", 1);
    assertEquals(true, day1.makeAppointment(9, appointm1));

    Appointment appointm2 = new Appointment("test appointment", 1);
    assertEquals(10, day1.findSpace(appointm2));
}
```

6.17:

```
/**
 * Check if findSpace() returns -1 if the day is already full
 */
public void testFindSpaceFullDay()
{
    // Start with a fresh Day object.
    Day day1 = new Day(1);
    for(int time = Day.START_OF_DAY;
                time <= Day.FINAL_APPOINTMENT_TIME;
                    time++) {
        day1.makeAppointment(time, new Appointment("Test " + time, 1));
    }

    // test findSpace()
    Appointment appointm1 = new Appointment("1h Test", 1);
    assertEquals(-1, day1.findSpace(appointm1));
}
```

6.18:

```
/**
 * The test class AppointmentTest.
 *
 * @author  Felix Rohrer
 * @version 23.11.2011
 */
public class AppointmentTest extends junit.framework.TestCase
{
    /**
     * Default constructor for test class AppointmentTest
     */
    public AppointmentTest()
    {
    }

    /**
     * Sets up the test fixture.
     *
     * Called before every test case method.
     */
    public void setUp()
    {
    }

    /**
     * Tears down the test fixture.
     *
     * Called after every test case method.
     */
    public void tearDown()
    {
    }

    /**
     * Test Appointment description and duration
     */
    public void testAppointment()
    {
        Appointment appointm1 = new Appointment("3h Test", 3);
        assertEquals("3h Test", appointm1.getDescription());
        assertEquals(3, appointm1.getDuration());
    }

}
```

Kapitel 6.4.4

5. zu bearbeitende Aufgaben: 6.20

6.20:

Kapitel 6.7

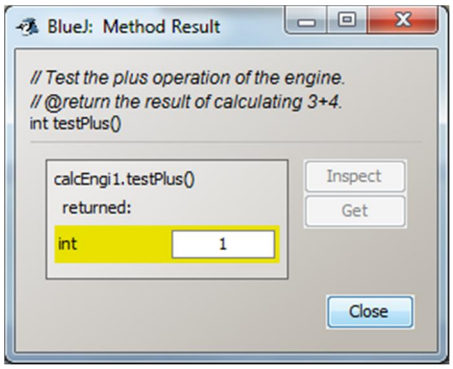
6. zu bearbeitende Aufgaben: 6.21 bis 6.23

6.21:

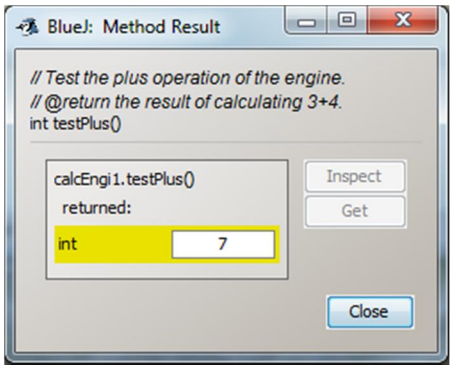
```
Testing the addition operation.
The result is: 7
Testing the subtraction operation.
The result is: 5
All tests passed.
```

⇨ *Es ist nicht bekannt was gerechnet wurde, somit ob das Ergebnis überhaupt richtig ist.*

6.22:



Es ist ein anderes Ergebnis, eigentlich wäre 3 + 4 = 7 und nicht = 1.

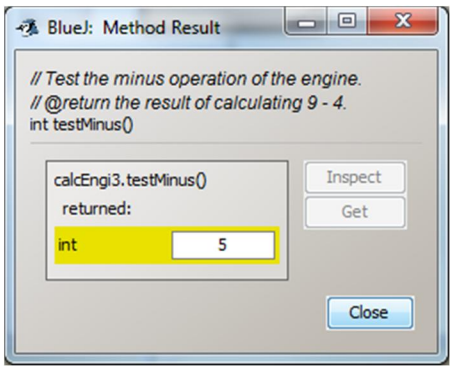


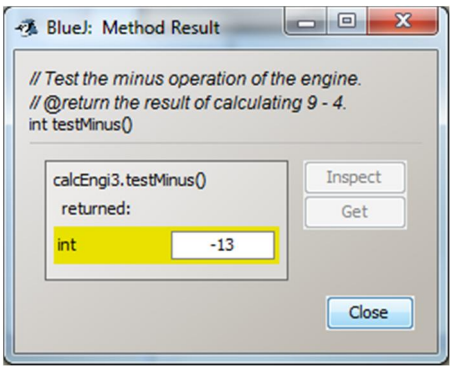
Beim zweiten Aufruf wird das richtige Resultat (7) zurück geliefert.

Es sollte immer das gleiche Resultat geben wenn man diese Methode aufruft.

Basierend dem Source-Code sollte es immer das gleiche Resultat geben.

6.23:





Nein, es gibt ebenfalls unterschiedliche Resultate zurück.

Kapitel 6.8.2

7. zu bearbeitende Aufgaben: 6.25, 6.26 und 6.28

6.25:

Method called	displayValue	leftOperand	previousOperator
initial state	*0*	*0*	‘ ‘
Clear	*0*	*0*	‘ ‘
numberPressed(3)	*3*	*0*	‘ ‘
Plus	*0*	*3*	‘ + ‘
numberPressed(4)	*4*	*3*	‘ + ‘
equals	*7*	*0*	‘ + ‘

6.26:

Nein, es wird nur überprüft ob es ein + war, alle anderen werden als – (Subtraktion) interpretiert.

6.28:

- equals() überprüft nur das + Zeichen

- clear() löscht nicht alle Felder

```
public void equals() {
  if(previousOperator == '+') {
    displayValue = leftOperand + displayValue;
  }
  if(previousOperator == '-') {
    displayValue = leftOperand - displayValue;
  }
  else {
    //do nothing
  }
  leftOperand = 0;
}

public void clear()
{
  displayValue     = 0;
  leftOperand      = 0;
  previousOperator = ' ';
}
```

Kapitel 6.9

8. zu bearbeitende Aufgaben: 6.31 bis 6.33

6.31:

Ja, jeder Schritt ist ersichtlich (aber auch „unübersichtlich“), schwierig zum lesen und Fehler finden…

6.32:

Für manuelles Debuggen ist es optimal in diesem Umfang.

6.33:

Debug Befehle müssen nachträglich wieder entfernt/auskommentiert werden.

Debug Befehle liefern nur Informationen, sie überprüfen dabei nicht zwingend die Richtigkeit

Für „schnelles“ Debugen sind Debug-Befehle hilfreich.

Testen

9. Schreiben Sie eine Testspezifikation für ein Programm, das das Volumen V eines Quaders bei gegebenen drei Seiten a, b und c berechnet. Verwenden Sie dazu folgende Tabelle:

	Testfall	**Eingabe**	**Ausgabe**	**Bestanden?**
Normalfälle	Nr. 1	*a=1, b=2, c=3*	*V=6*	
	Nr. 2	*a=5, b=5, c=2*	*V=50*	
	Nr. 3	*a=500, b=321, c=123*	*V=19'741'500*	
Zulässige Spezialfälle	Nr. 4	*a=1, b=1, c=1*	*V=1*	
	Nr. 5	*a=5, b=5, c=5*	*V=125*	
	Nr. 6	*a=0, b=2, c=20*	*V=0 (kein Quader)*	
Unzulässige Speziallfälle	Nr. 7	*a=-5, b=10, c=5*	*V=0 (negative Zahlen)*	
	Nr. 8	*Alle Seiten sehr gross, dass es einen Overflow gibt*		
	Nr. 9	*Falls es Float sind: sehr kleine Zahlen wählen damit es gegen Null geht*		

Algorithmen

10. Implementieren Sie den Pseudocode "Türme von Hanoi" in Java.

Pseudocode:

```
moveDisks(String from, String via, String to, int n)
   Falls n == 1:
      print ("move disk from " + from + " to " + to) ;
   sonst, d.h. n > 1:
      moveDisks(from, to, via, n-1);   // A -> C -> B
      moveDisks(from, "", to, 1);      // A -> C
      moveDisks(via, from,   to, n-1); // B -> A -> C
```

Java-Implementation:

```
/**
 * Türme von Hanoi
 *
 * @author Felix Rohrer
 * @version 23.11.2011
 */
public class TowerOfHanoi
{
    // Anzahl schritte
    private int countMove;

    /**
     * Constructor for objects of class TowerOfHanoi
     */
    public TowerOfHanoi()
    {
        //nothing
    }


    /**
     * moveDisks von Quell auf den Ziel-Stapel
     * @param Quell-Stapel
     * @param Via-Stapel
     * @param Ziel-Stapel
     * @param Anzahl Scheiben die verschoben werden müssen
     */
    private void moveDisks(String from, String via, String to, int n)
    {


        // Verschieben ausführen (rekursiv)
        if (n == 1) {
            // Counter für die Anzahl Schritte erhöhen
            countMove++;
            System.out.println("Step " + countMove + ": move disk from " + from + " to " + to); //
Rekursionsbasis
        }
        else {
            // obere Scheiben auf den "via-Stapel" verschieben (A -> C -> B)
            moveDisks(from, to, via, n-1);
            // Eine Scheibe auf den Ziel-Stapel verschieben (A -> C)
            moveDisks(from, via, to, 1);
            // restliche Scheiben verschieben (B -> A -> C)
            moveDisks(via, from, to, n-1);
        }
    }

    /**
     * moveTower
     * Test für moveDisks
     * @param Anzahl Scheiben
     */
    public void moveTower(int numberDiscs)
    {
        // reset countMove
        countMove = 0;

        // move Tower from "a" to "c" via "b"
        moveDisks("a", "b", "c", numberDiscs);
    }
}
```

11. Beobachten Sie im Debugger wie die "offenen" Methoden (deren Ausführung unterbrochen ist) auf dem Stack "eingefroren" werden. Verwenden Sie dazu die im BlueJ Debugger angezeigte Aufrufkette der Methodenaufrufe (Call Graph).

```
moveTower(4)

Step 1: move disk from a to b
Step 2: move disk from a to c
Step 3: move disk from b to c
Step 4: move disk from a to b
Step 5: move disk from c to a
Step 6: move disk from c to b
Step 7: move disk from a to b
Step 8: move disk from a to c
Step 9: move disk from b to c
Step 10: move disk from b to a
Step 11: move disk from c to a
Step 12: move disk from b to c
Step 13: move disk from a to b
Step 14: move disk from a to c
Step 15: move disk from b to c
```