Laborator 7

1) Exemplu de program AspectJ

```
// Sample.java package test;
public class Sample
{
public static void sendMessage(String message)
        {System.out.println(message);}
public static void main(String[] args)
         { sendMessage("message to send");
        }
}

// SampleAspect.aj package test;
public aspect SampleAspect
{
pointcut messageSending() : call (* Sample.sendMessage(..));
        before() : messageSending()
                {System.out.println("a message is about to be send");}
}
```

Ieşirea în urma executiei acestui program este:

a message is about to be send message to send

2. Exemplu caching pentru un algoritm de calcul factorial

```
// TestFactorial.java package test;
public class TestFactorial
{
public static void main(String[] args)
    {
    System.out.println("Result: " + factorial(5) + "\n");
    System.out.println("Result: " + factorial(10) + "\n");
    System.out.println("Result: " + factorial(15) + "\n");
    System.out.println("Result: " + factorial(15) + "\n");
    }
 public static long factorial(int n)
     {
    if (n == 0)
            {return 1;}
    else
            {return n * factorial(n - 1);}
    }
 }

// OptimizeFactorialAspect.aj package test;
import java.util.*;
public aspect OptimizeFactorialAspect
{
pointcut factorialOperation(int n) : call(long *.factorial(int)) && args(n);
pointcut topLevelFactorialOperation(int n) : factorialOperation(n) && !cflowbelow(factorialOperation(int));
private Map<Integer, Long> factorialCache = new HashMap<Integer, Long>();
        before(int n) : topLevelFactorialOperation(n)
                {System.out.println("Seeking factorial for " + n);}
        long around(int n) : factorialOperation(n)
                {
                Object cachedValue = factorialCache.get(n);
                if ( cachedValue != null)
                        {System.out.println("Found cached value for " + n + ": "
                            + cachedValue); return ((Long) cachedValue).longValue();}
                return proceed(n);
                }
        after(int n) returning(long result) : topLevelFactorialOperation(n)
                { factorialCache.put(n, result);}
```

....

3. Sa reanalizam problema hotelului. Din punct de vedere al componentelor (clase, module) în care se poate diviza sistemul ce trebui implementat, distingem următoarele (nivel minimal):
   - Customer şi HotelStaff: actorii sistemului.
   - Room şi Reservation: datele cu care aceştia operează.
   - ReserveRoom, CheckInCustomer, CheckOutCustomer: operaţiile ce pot fi efectuate de către actori, ale căror rezultate sunt vizibile în stările datelor.

Pentru început stabilim clasele de bază, care conţin informaţii ce nu sunt legate în mod explicit de o anumită operaţie.

```
package hotel; public class Room
{
private int nr;
private RoomCategory categ;
public int getNr()
        { return nr;}
public RoomCategory getCateg()
        { return categ;}
public Room(int nr, RoomCategory categ)
        {
        this.nr = nr;
        this.categ = categ;
        }
}
```

Clasa Room conţine un număr de ordine prin care se distinge de celelalte, precum şi un tip de categorie, prin care se stabileşte preţul unitar. Categoriile se pot defini de exemplu prin intermediul unui tip enum RoomCategory:

```
package hotel;
public enum RoomCategory {
A(10),B(20),C(30),D(40);
private int price;
public int price()
        { return price;}
private RoomCategory(int price) { this.price = price;}
}
```

Definim apoi tipul Reservation, prin care se reţin rezervările făcute de diferiti Customeri. Elementele esenţiale sunt camera rezervată şi clientul care face rezervarea:

```
package hotel;
public class Reservation
{
private Room room;
private Customer customer;
public Room getRoom()
        { return room;}
public Customer getCustomer()
        { return customer;}
public Reservation(Room room, Customer customer)
        {
        this.room = room;
        this.customer = customer;
        }
}
```

Observăm că am folosit deja tipul Customer. Acesta este conceput să reţină informaţiile despre un client ce doreşte să facă rezervări.
Într-un sistem complex, acesta ar trebui să aibă un cont pentru autentificare şi drepturi pentru a efectua anumite operaţii. Pentru test, vom reţine deocamdată un id(entificator) pentru a putea distinge diferiţi clienţi:

```
package hotel;
public class Customer
{
private int id;
public int getId()
        { return id;}
public Customer(int id)
{ this.id = id;}
}
```

Diverse elemente de logică şi control le vom implementa într-o clasă comună: StaffHandler. Printre altele, aceasta are rolul de a ţine evidenţa camerelor ce există în sistem precum şi a rezervărilor în aşteptare. Tot pentru test vom face şi o iniţializare a camerlor.

```
package hotel; import java.util.*;
public class StaffHandler
{
private List<Room> rooms = new ArrayList<Room>();
private  List<Reservation> reservations = new ArrayList<Reservation>();
public StaffHandler()
        { testInits();}
public List<Room> getRooms()
        { return rooms;}
public List<Reservation> getReservations()
        { return reservations;}
private void testInits()
        {
        for (int i = 0; i < 5; i++)
        {
        rooms.add(new Room(10 + i, RoomCategory.A));
        rooms.add(new Room(20 + i, RoomCategory.B));
        rooms.add(new Room(30 + i, RoomCategory.C));
        rooms.add(new Room(40 + i, RoomCategory.D));
...
```

Până în acest moment avem clase ce reprezintă datele şi controlul asupra acestor date, fără a implementa în mod explicit vreuna dintre operaţiile specifice sistemului.
Pentru implementarea acestora vom folosi aspectele, pentru ca tot codul relativ la fiecare operaţie să fie separat într-un modul propriu.

```
package hotel;
public aspect ReserveRoomAspect
{private boolean Room.available;
public void Room.setAvailability(boolean value)
        { available = value;}
public boolean Room.isAvailable()
        { return available;}
private Room StaffHandler.getAvailableRoom(RoomCategory categ)
        {
        for(Room room : getRooms())
                {
                if ( room.isAvailable() && room.getCateg() == categ)
                { return room;}
        }
return null;
}
public void StaffHandler.makeReservation(Customer customer, RoomCategory categ)
        {
        Room foundRoom = getAvailableRoom(categ);
        if (foundRoom != null)
                {
                Reservation res = new Reservation(foundRoom, customer);
                getReservations().add(res);
                foundRoom.setAvailability(false);
                }
        }
}
```

Rezervarea unei camere de către un client se face prin intermediul aspectului ReserveRoomAspect. Mai întâi observăm că se folosesc declaraţii intertip facilitate de sintaxa AspectJ.
Astfel, modificările necesare în clasa Room legate de crearea unei rezervări au fost create în acest aspect.
Aşadar clasei Room i se mai adaugă un parametru: available, printr-o variabilă privată şi metodele publice de acces la aceasta.
Să presupunem pentru simplitate că un client se poate caza prin acceptarea de către personalul hotelului a rezervării create de el în prealabil. Pentru aceasta, o cameră, pe lângă faptul că este ocupată, trebuie să mai reţină şi clientul care s-a cazat în acea cameră. Este necesară astfel inserarea unei noi proprietăţi clasei Room:

```
package hotel;
public aspect CheckInCustomer
{
private Customer Room.checkBy = null;
public Customer Room.getCheckBy()
        { return checkBy;}
public void Room.setCheckBy(Customer value)
        { checkBy = value;}
public void Room.uncheck()
        { setCheckBy(null);}
public void StaffHandler.makeCheckIn(Reservation res)
        { res.getRoom().setCheckBy(res.getCustomer());
        //consume reservation getReservations().remove(res);
        }
}
```

Operaţia în sine este de asemenea introdusă în clasa de control şi constă în eliminarea rezervării si atribuirea camerei respective clientul care se cazează. La decazare această legătură se rupe, iar camera se setează ca fiind liberă.

```
package hotel;
public aspect CheckOutCustomer
{
public void StaffHandler.makeChekcOut(Room room)
        {
        room.uncheck();
        room.setAvailability(true);
        }
}
```

Aceste clase (inclusiv tipurile enum şi aspect) definesc un schelet de bază al sistemului de management al rezervărilor la un hotel. Pentru a testa aceste clase de control si date, este nevoie de o interfaţă utilizator.

Tema 1. Creaţi o aplicaţie de test care să permită simularea celor 3 operaţii de bază implementate de către sistem.Identificaţi elementele cheie ce trebuie extinse pentru ca aplicaţia sa fie cât mai realistă (ex. la rezervare clientul alege şi perioada, după care se calculează un preţ pe acea perioadă) (50 min).
Tema 2. Odată realizată această implementare, gândiţi o modalitate de a extinde o funcţionalitate: de ex, atunci când nu se găsesc camere de tipul dorit libere, să se realizeze o listă de aşteptare. Această extra funcţionalitate trebuie implementată în mod identic cum au fost implementate cele de bază, adică fără a modifica codul deja existent (inclusiv aspectele).
(50 min)

Tema acasa: Creati un aspect peste proiectul cu navele care sa permita modificarea comportamentului unei nave (de exemplu inainte de a trage un foc(cand este apelat focul) sare inainte cu o cuanta (spre adversar) si dup ace a tras face un salt in lateral cu o cianta spatiala)