CPDS-Conc Lab 2
FSP Models & Java Programs

Authors: Jorge Castro and Joaquim Gabarro

Important: Students should have a `Java` running environment in their laptops.

Goal: Move from models to programs, that is from `FSP` models to `Java` programs.

Basic material:

- Course slides.
- Basic reading: Chapters 4 and 5 of the book: *Concurrency, State Models & Java Programs*, Jeff Magee and Jeff Kramer, Wiley, Second Edition, 2006 (M&K for short).

2.1 Training Exercises

The main goal of this part is consists on moving from `FSP` models to `Java` programs.

1. (M&K 5.4) *The Dining Savages*: A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty, in which case he waits for the pot to be filled. If the pot is empty, the cook refills the pot with M servings. The behaviour of the savages and cook are described by:

```
SAVAGE = (getserving -> SAVAGE).
COOK = (fillpot -> COOK).
```

   - Model the behavior of the pot as an `FSP` process. To do that complete the following snipped code

```
const M = 5

SAVAGE = (getserving -> SAVAGE).

COOK   = (fillpot -> COOK).
```

```
POT              = SERVINGS[0],
SERVINGS[i:0..M] = (when (i==0) fillpot     -> ...
                   |when (i>0 )  getserving -> ...
                   ).

||BANKET = (SAVAGE || COOK || POT).
```

- Run the `Banket` application. To do that, start filling the following code:

```
public class Banket {

    public static void main(String args[]) {
        Pot pot = new Pot(5);
        Thread a = new Savage(pot);
        a.setName("alice");
        Thread b = new ... (pot);
        b.setName("bob");
        Thread c = new Cook(...);
        c.setName("...");
        a.start();
        b. ...;
        c. ...;
    }
}
```

  taking into account that a possible printout can start like:

```
alice is hungry and would like to eat
cook would like to fill the pot
bob is hungry and would like to eat
cook fills the pot
servings in the pot: 5
cook would like to fill the pot
bob is served
servings in the pot: 4
bob is hungry and would like to eat
alice is served
servings in the pot: 3
alice is hungry and would like to eat
alice is served
servings in the pot: 2
cook has to wait
alice is hungry and would like to eat
bob is served
....
```

- Fill in the Java monitor `Pot`. Pay attention to comments and fill in those that are incomplete. We turn the `N` of the FSP schema into a `capacity` attribute. This will be fixed to `5` when the constructor is called.

```
public class Pot {

    private int servings = 0;
    private int capacity;
```

```
    public Pot(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void getserving() throws .... {
        // Condition synchronization: at least one serving available,
        // otherwise, go to the Wait Set til the cook fill the pot
        while (servings == 0) {
           System.out.println(Thread.currentThread().getName() + " has to wait");
           ...;
        }
        --servings;
        System.out.println(Thread.currentThread().getName() + " ....");
        // when necessary, wake up threads  in Wait Set in order to asure
        // a runnable cook
        if (servings == 0) ...;
        print_servings();
    }

    public synchronized void fillpot() throws .... {
        //Condition synchronization: .....
        //....
        while (servings > 0) {
           System.out.println(Thread.currentThread().getName() + "....");
           ....;
        }
        servings = capacity;
        System.out.println(servings);
        // wake up threads in Wait Set in order to asure....
        print_servings();
        notifyAll();
    }

    //only for trace purposes
    public synchronized void print_servings() {
        System.out.println("servings in the pot: " + servings);
    }

}
```

- Implement the `Savage` Thread filling the following snipped code:

```
public class Savage extends Thread {

    Pot pot;

    public Savage(Pot pot) {
        this.pot = ...;
    }

    public void run() {
        while (true) {
             System.out.println(Thread.currentThread().getName() + " is hungry ... ");
            try {
               Thread.sleep(200);
                pot. ...;
```

```
            }
            catch(InterruptedException e) {};
        }
    }
}
```

- Implement the `Cook` Thread filling the code:

```
public class Cook extends Thread {

    Pot pot;

    public Cook(pot) {
        ...;
    }

    public void run() {
        while (true) {
            System.out.println(...);
            try {
               Thread.sleep(200);
                ...
            }
            catch(...) {};
        }
    }
}
```

2. *BadBanketOne*. What happen if we replace the *while* loops into *if* conditionals in the `Pot` instructions? Let us rename `BadPot` the resulting class:

```
public class BadPot {

    private int servings = 0;
    private int capacity;

    public BadPot(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void getserving() throws InterruptedException {
        if (servings == 0) {
         ...
        }
       ...
    }

    public synchronized void fillpot() throws InterruptedException {
        if (servings > 0){
         ...
        }
       ...
    }
```

```
    public synchronized void print_servings() {
        ...
    }
}
```

Rewrite `Savage` and `Cook` to deal with `BadPot` as follows:

```
public class Savage extends Thread{
      BadPot pot;

      public Savage(BadPot pot) {
            ...
      }

      public void run() {
         while (true) {
                  ...
         }
      }
}
```

and

```
public class Cook extends Thread{
    BadPot pot;

    public Cook(BadPot pot) {
        ...
    }

    public void run() {
        while (true) {
           ...
        }
    }
}
```

Following a `BadBanketOne` with eight savages:

```
public class BadBanketOne {
 public static void main(String args[])  throws InterruptedException{
        BadPot pot = new BadPot(5);
        Thread s1 = new Savage(pot); s1.setName("alice");
        Thread s2 = new Savage(pot); s2.setName("bob");
        Thread  s3= new Savage(pot); s3.setName("peter");
        Thread  s4= new Savage(pot); s4.setName("xana");
        Thread s5 = new Savage(pot); s5.setName("tom");
        Thread s6 = new Savage(pot); s6.setName("jerry");
        Thread  s7= new Savage(pot); s7.setName("kim");
        Thread  s8= new Savage(pot); s8.setName("berta");
        Thread c = new Cook(pot); c.setName("cook");
        s1.start(); s2.start(); s3.start(); s4.start();
        s5.start(); s6.start(); s7.start(); s8.start();
        c.start();
    }
}
```

The following printout give us a misbehavior (we have negative servings)

```
alice is hungry and would like to eat
peter is hungry and would like to eat
jerry is hungry and would like to eat
xana is hungry and would like to eat
bob is hungry and would like to eat
tom is hungry and would like to eat
berta is hungry and would like to eat
kim is hungry and would like to eat
cook would like to fill the pot
tom has to wait
xana has to wait
cook fills the pot
servings in the pot: 5
cook would like to fill the pot
xana is served
servings in the pot: 4
...
cook fills the pot
servings in the pot: 5
cook would like to fill the pot
tom is served
servings in the pot: 4
tom is hungry and would like to eat
xana is served
...
peter is served
servings in the pot: 0
peter is hungry and would like to eat
jerry is served
servings in the pot: -1
jerry is hungry and would like to eat
bob is served
servings in the pot: -2
bob is hungry and would like to eat
kim is served
servings in the pot: -3
...
```

Questions. Why this misbehavior occur? Why having a big number of savages help us to find the misbehavior?

3. *BadBanketTwo.* We can try to program the idea of "do not wait, just go for a walk". Following there is a wrong implementation following this idea. Let us start with the class `BadBanketTwo`

```
public class BadBanketTwo {
   public static void main(String args[])  throws InterruptedException{
        BadPotTwo pot = new BadPotTwo(5);
       ...
        s8.start();
        c.start();
    }
}
```

Note that in this case the `BadPotTwo` below is not a monitor because some methods are not synchronized. Moreover we consider a `if`$(\cdots)\{\cdots\}$`else`$\{\cdots\}$ approach.

```
public class BadPotTwo {

    private int servings = 0;
    private int capacity;

    public BadPotTwo(int capacity) {
        ....
    }

    public void getserving() throws InterruptedException {
        if (servings == 0) {
            System.out.println(Thread.currentThread().getName() + " go walk");
        }
        else {
            Thread.sleep(200);
            --servings;
            System.out.println(Thread.currentThread().getName()+ " is served");
            print_servings();
        }
    }

    public  void fillpot() throws InterruptedException {
        if (servings > 0){
             ...
        }
        else {
            ...
        }
    }

    public synchronized void print_servings() {
       ...
    }
}
```

Classes `Savage` and `Cook` are reshaped to deal with `BadPotTwo`.

A possible misbehavior is:

```
...
cook would like to fill the pot
alice is served
kim is served
servings in the pot: 0
kim is hungry and would like to eat
jerry is served
servings in the pot: 0
jerry is hungry and would like to eat
peter is served
servings in the pot: 0
peter is hungry and would like to eat
bob is served
```

```
servings in the pot: 0
bob is hungry and would like to eat
servings in the pot: 0
alice is hungry and would like to eat
xana is served
servings in the pot: -2
xana is hungry and would like to eat
tom is served
servings in the pot: -2
tom is hungry and would like to eat
berta is served
servings in the pot: -3
berta is hungry and would like to eat
cook fills the pot
servings in the pot: 5
cook would like to fill the pot
cook go walk
cook would like to fill the pot
berta is served
...
```

Question Explain the origin of the misbehavior. Why the instruction `Thread.sleep(200)` in `BadPotTwo` potentially helps to find the misbehavior.

4. *Continuation of the exercise (M&K 3.6)* Remind the exercise developed in the previous lab.

   > A museum allows visitors to enter through the east entrance and leave through its west exit. Arrivals and departures are signaled to the museum controller by the turnstiles at the entrance and exit. At opening time, the museum director signals the controller that the museum is open and then the controller permits both arrivals and departures. At closing time, the director signals that the museum is closed, at which point only departures are permitted by the controller.

   Following a `FSP` model of the museum:

```
const N = 5
EAST = (arrive -> EAST).
WEST = (leave  -> WEST).
DIRECTOR = (open -> close -> DIRECTOR).

CONTROL        = CLOSED[0],
CLOSED[i:0..N] = (when (i==0) open -> OPENED[0]
                  |when (i>0)  leave -> CLOSED[i-1]
                  ),
OPENED[i:0..N] = (close -> CLOSED[i]
                  |when (i<N) arrive -> OPENED[i+1]
                  |when (i>0) leave  -> OPENED[i-1]
                  ).

||MUSEUM = (EAST || WEST || DIRECTOR || CONTROL).
```

Translating from `FSP` to `Java`, processes `EAST`, `WEST`, `DIRECTOR` will become threads and `CONTROL` a monitor. Let us proceed in several steps starting with the monitor.

- In order to program the *monitor* class `Control` complete the following snipped code. In the `FSP` model `N` takes value `5`. In the `Control` class we do not "bound" the `control` variable.

```
public class Control {
    int count == 0;
    boolean opened = false;

    public synchronized void arrive() throws ... {
       while (!opened) ...
       //update count
       ...
    }
    public synchronized void leave() throws ... {
       while (count<=0) ...
       // update count
       ...
       if (count==0) ....
    }
    public synchronized void open() throws ... {
        ...
    }
    public synchronized void close() throws ... {
        ....
    }
}
```

- Implement all the remaining classes and run the application

2.2 Homework

2.2.1 Exercises

1. *BanketNoWait*. It is possible to "repair" easily the `BadPotTwo` keeping the same approach. That is "just try but not wait", later you will try again. Fix and write the new code. We call such a program `BanketNoWait`. Compare the approaches between `Banket` and `BanketNoWait`, which one is better (if any)? Argue your answer.

2. (M&K 5.6) *The Saving Account Problem*: A saving account is shared by Alice and Bob. Each one may deposit or withdraw funds from the account subject to the constraint that the balance of the account must never become negative. Develop a model for the problem and from the model derive a `Java` implementation for the saving account system.

   First, fill in the snipped code to obtain the FSP model

```
const N = 10 //constant N is only for FSP modeling.
             //It should not appear in the Java impl.
```

```
PERSON = (deposit[1..N] -> PERSON
         |withdraw[1..N] -> ...
         ).

ACCOUNT = ACCOUNT[0],
ACCOUNT[balance: 0..N] = (deposit[amount:1..N] -> ...
                         |when(balance > 0) ..
                         ).

//(1) guard balance > 0 is only to avoid  empty ranges.
//    Empty ranges are not allowed by LTSA
//(2) Do not worry about errors on traces like:
//      deposit[1] deposit[N]
//    this is because FSP models can only represent **finite** state systems


||SAVING_ACCOUNT = (a:PERSON || ... || {.. , ...}::ACCOUNT).
```

Now develop the Java program for the Saving Account problem. In order to avoid the execution get stuck when the balance is not enough to satisfy withdraw orders of Alice and Bob you can assume that a third participant (perhaps a company) shares the account and only engages in the deposit action.

Java codes that stick to recommendations on Magge and Kramer book in Chapter 5 will get a higher score.

2.2.2 How to deliver the homework

Use the *Raco deliver feature* in order hand in these exercises. Each group has to submit only one file containing solutions for all exercises. The name of this file has to be composed by the lastname of the participants. Moreover, names of all students in the group have to appear at the beginning of the document. *Groups of two people are mandatory.*

Important. You should deliver a unique zip (with your names) of a folder. This folder should contain two other folders.

- The first folder contains the information related the *BanketNoWait* exercise. This folder containing both, a txt (plain text) with the comparison between `Banket` and `BanketNoWait` plus a folder with the classes corresponding to `BanketNoWait`.
- The second folder constains the solution of *Saving Account Problem.* It containins both, a txt with LST code and a folder corresponding to the Java implementation.