Home assignment 1 (ETS061) - *Simulation*

Niclas Lövdahl <dic13nlo@student.lu.se>

Task 1

```
---------- Interarrival time 1 ----------
Mean number of customers in queuing system 2: 7.423
Number of customers arrived in queuing system 1: 5078
Number of rejected customers in queuing system 1: 2674
Propability of rejection in queuing system 1: 0.5265852697912564
---------- Interarrival time 2 ----------
Mean number of customers in queuing system 2: 4.503
Number of customers arrived in queuing system 1: 2661
Number of rejected customers in queuing system 1: 240
Propability of rejection in queuing system 1: 0.09019165727170236
---------- Interarrival time 5 ----------
Mean number of customers in queuing system 2: 0.397
Number of customers arrived in queuing system 1: 1006
Number of rejected customers in queuing system 1: 0
Propability of rejection in queuing system 1: 0.0
```

Task 2

Results from test run:

```
---------- Mean number of jobs (Prio B) ----------
Mean number of jobs in buffer: 112.558
---------- Mean number of jobs (Prio A and Exp delay) ----------
Mean number of jobs in buffer: 48.319
---------- Mean number of jobs (Prio A) ----------
Mean number of jobs in buffer: 4.17
```

Results in Matlab:

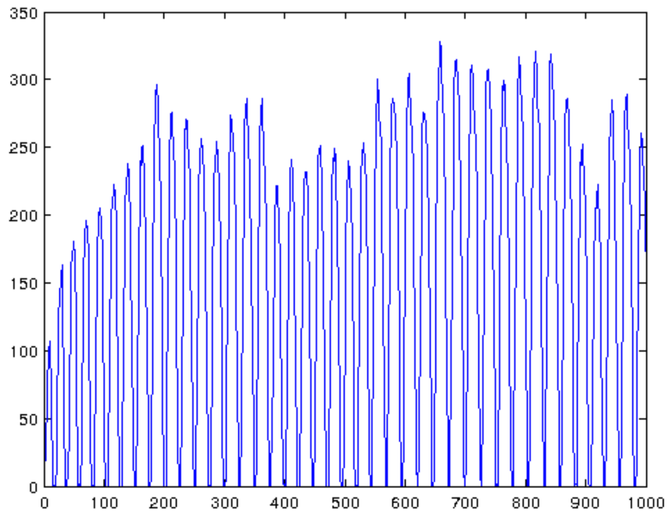

Figure 1: Number of jobs in the system (Prio B).

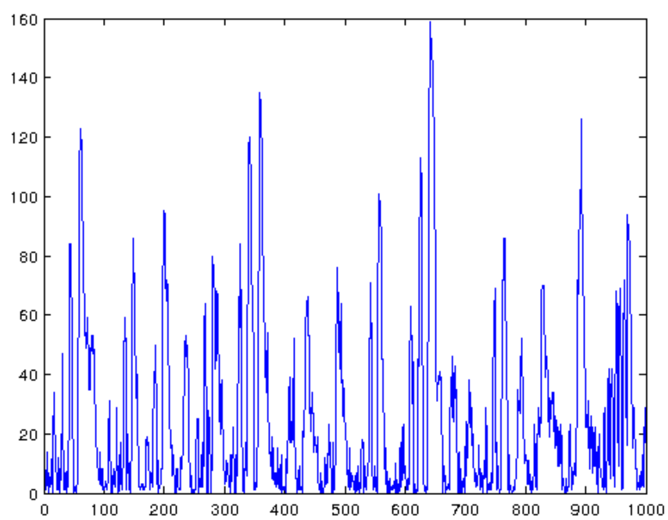

Figure 2: Number of jobs in the system (Prio B and Exp delay).

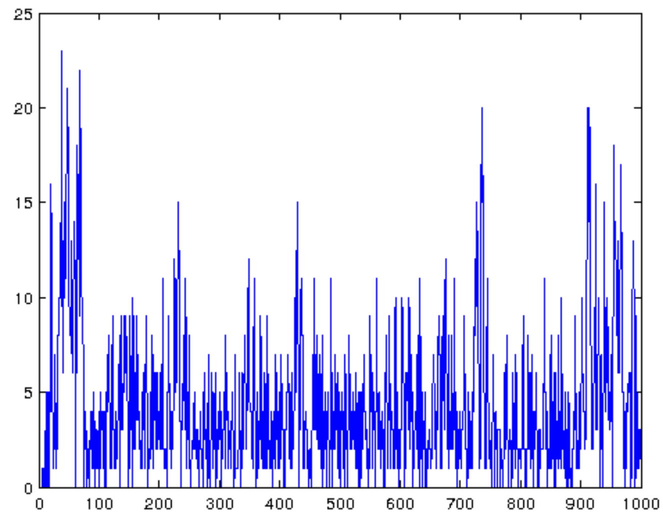

Figure 3: Number of jobs in the system (Prio A).

1. Mean number of jobs in the buffer for the system while prioritizing B is 112.558.
2. Mean number of jobs in the buffer for the system while prioritizing B and using exponential delay time is 48.319.
3. Mean number of jobs in the buffer for the system while prioritizing A is 4.17.
4. As can be seen in the figure 1 above when prioritizing jobs of type B the system produces chunks of B jobs in the queue. Even when the delay time is exponentially distributed the problem occurs occasionally. As can be seen in figure 3 it is favourably to prioritize jobs of type A since it produces a more steady and mixed flow of jobs in the system.

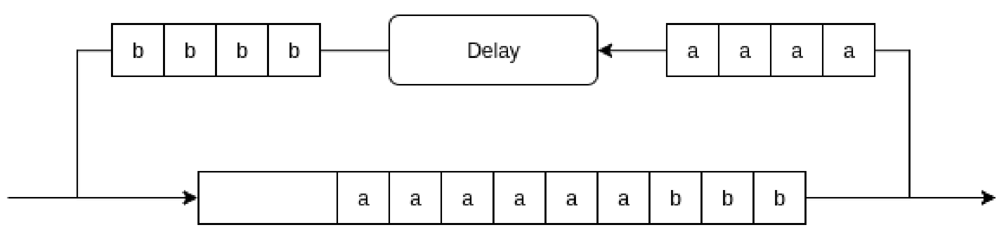

Figure 4: Chunks of jobs in the system (Prio B).

Task 3

Results from test run:

```
---------- Interarrival time 2.0 ----------
Mean number of customers in queuing system: 2.089
Mean time a customer spends in the queuing network: 4.656490627160915
---------- Interarrival time 1.5 ----------
Mean number of customers in queuing system: 4.171
Mean time a customer spends in the queuing network: 6.952222401389203
---------- Interarrival time 1.1 ----------
Mean number of customers in queuing system: 19.607
Mean time a customer spends in the queuing network: 22.191403929784833
```

Comparing with formulas given:

$$N = \frac{2}{x-1}$$

$$T = \frac{2x}{x-1}$$

Interarrival	2	1.5	1.1
N (Formula)	2	4	20
N (Test run)	2.089	4.171	19.607
T (Formula)	4	6	22
T (Test run)	4.656490627160915	6.952222401389203	22.191403929784833

Table 1: Comparison between formula and test run.

Task 4

4.1

Figure 5: Transient time is about 100 time units.

4.2

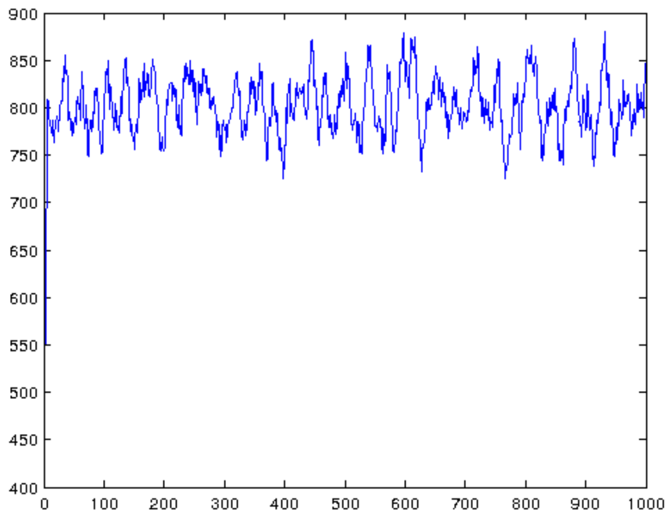

Figure 6: Transient time is almost non-existing.

4.3

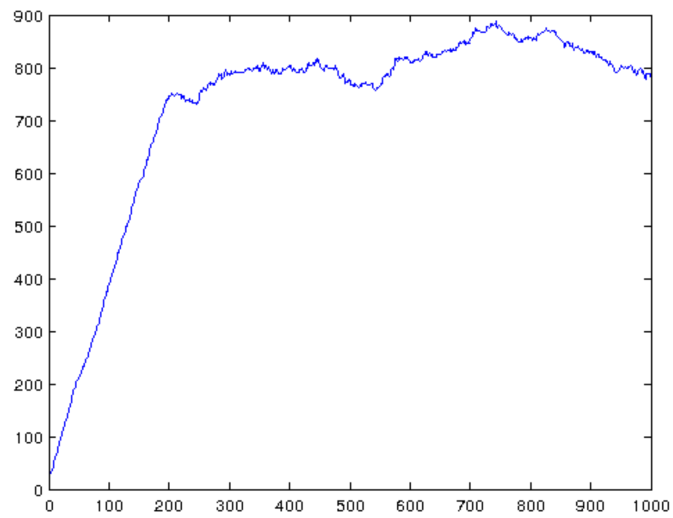

Figure 7: Transient time is about 200 time units.

4.4

Result from corr.m in Matlab.

```
0.0250 Lower confidence point = 39.67811
0.0250 Upper confidence point = 41.02589
```

4.5

Result from corr.m in Matlab.

```
0.0250 Lower confidence point = 39.23957
0.0250 Upper confidence point = 40.60693
```

4.6

Result from corr.m in Matlab.

```
0.0250 Lower confidence point = 39.67514
0.0250 Upper confidence point = 40.24886
```

As can be seen in table 2 below the accuracy of the measurements does vary according to time between measurements and number of measurements. The results shows that $T = 4$ and $M = 4000$ produces the most accurate result since the length of the confidence interval is significantly smaller.

Parameters	T = 4, M = 1000	T = 1, M = 4000	T = 4, M = 4000
Length of CI	1.34778	1.36736	0.57372

Table 2: Length of the confidence interval with different T- and M value.

Task 5

5.2

Results from test run:

```
---------- Interarrival time 0.12 (Random) ----------
Mean number of customers in queuing system: 22.881781989543388
Result using littles theorem: 22.863226163334648

---------- Interarrival time 0.12 (Round robin) ----------
Mean number of customers in queuing system: 14.036988937103434
Result using littles theorem: 14.027187736544954

---------- Interarrival time 0.12 (Prio) ----------
Mean number of customers in queuing system: 6.782706851481578
Result using littles theorem: 6.787253767981347
```

5.3

Results from test run:

```
---------- Interarrival time 0.11 (Random) ----------
Mean number of customers in queuing system: 45.77589027747728
Result using littles theorem: 45.72839158597921

---------- Interarrival time 0.15 (Random) ----------
Mean number of customers in queuing system: 9.453522278368288
Result using littles theorem: 9.425402876928635

---------- Interarrival time 2.0 (Random) ----------
Mean number of customers in queuing system: 0.25857633554463066
Result using littles theorem: 0.2563572742172415

---------- Interarrival time 0.11 (Round robin) ----------
Mean number of customers in queuing system: 27.18996214009714
Result using littles theorem: 27.237344227803
```

```
---------- Interarrival time 0.15 (Round robin) ----------
Mean number of customers in queuing system: 5.953953100808574
Result using littles theorem: 5.968672286270355

---------- Interarrival time 2.0 (Round robin) ----------
Mean number of customers in queuing system: 0.25051197864119723
Result using littles theorem: 0.24926439916630877

---------- Interarrival time 0.11 (Prio) ----------
Mean number of customers in queuing system: 10.660348481354804
Result using littles theorem: 10.665867173801626

---------- Interarrival time 0.15 (Prio) ----------
Mean number of customers in queuing system: 3.9523908478553613
Result using littles theorem: 3.95045834381455

---------- Interarrival time 2.0 (Prio) ----------
Mean number of customers in queuing system: 0.2518284584841326
Result using littles theorem: 0.24989277595252055
```

iii. is the best algorithm.

Task 6

Result from test run:

```
Average time when his work will have finished: 18:29
Average time from the arrival of a prescription until it has been filled
    in: 59.880891719258635 minutes
```

a Average time his work will have finished is about 18.29.

b Average time from the arrival of a prescription until it has been filled in is about 60 minutes.

Task 7

Results from test run:

```
Mean time until the system breaks down: 3.706584313852778
```

Code

Task 1

```
// As the name indicates this class contains the definition of an event.
    next is needed to
// build a linked list which is used by the EventListClass. It would
    have been just as easy
// to use a priority list which sorts events using eventTime.

class Event {
   public double eventTime;
   public int eventType;
   public Event next;
}

public class EventListClass {

   private Event list, last; // Used to build a linked list

   EventListClass() {
      list = new Event();
      last = new Event();
      list.next = last;
   }

   // The method insertEvent creates a new event, and searches the list
       of
   // events for the
   // right place to put the new event.

   public void InsertEvent(int type, double TimeOfEvent) {
      Event dummy, predummy;
      Event newEvent = new Event();
      newEvent.eventType = type;
      newEvent.eventTime = TimeOfEvent;
      predummy = list;
      dummy = list.next;
      while ((dummy.eventTime < newEvent.eventTime) & (dummy != last)) {
         predummy = dummy;
         dummy = dummy.next;
      }
      predummy.next = newEvent;
      newEvent.next = dummy;
   }

   // The following method removes and returns the first event in the
       list.
   // That is the
```

```java
   // event with the smallest time stamp, i.e. the next thing that shall
       take
   // place.

   public Event fetchEvent() {
      Event dummy;
      dummy = list.next;
      list.next = dummy.next;
      dummy.next = null;
      return dummy;
   }
}

public class GlobalSimulation{

   // This class contains the definition of the events that shall take
       place in the
   // simulation. It also contains the global time, the event list and
       also a method
   // for insertion of events in the event list. That is just for making
       the code in
   // MainSimulation.java and State.java simpler (no dot notation is
       needed).

   public static final int ARRIVAL = 1, READY = 2, MEASURE = 3, ARRIVAL2
       = 4, READY2 = 5; // The events, add or remove if needed!
   public static double time = 0; // The global time variable
   public static EventListClass eventList = new EventListClass(); // The
       event list used in the program
   public static void insertEvent(int type, double TimeOfEvent){ // Just
       to be able to skip dot notation
      eventList.InsertEvent(type, TimeOfEvent);
   }
}

import java.io.*;

public class MainSimulation extends GlobalSimulation {

   public static void simulate(int interarrival) {
      Event actEvent;
      State actState = new State(); // The state that shoud be used
      actState.interarrival = interarrival; // Interarrival set to State
                                 // class.
      // Some events must be put in the event list at the beginning
      insertEvent(ARRIVAL, 0);
      insertEvent(MEASURE, 5);

      // The main simulation loop
      while (actState.noMeasurements < 1000) {
```

```
      actEvent = eventList.fetchEvent();
      time = actEvent.eventTime;
      actState.treatEvent(actEvent);
    }

    // Printing the result of the simulation, in this case a mean value
    System.out.println("---------- Interarrival time " + interarrival
        + " ----------");
    System.out.println("Mean number of customers in queuing system 2: "
         + 1.0 * actState.accumulated / actState.noMeasurements);
    System.out.println("Number of customers arrived in queuing system
        1: " + actState.numberOfArrivalsQueue1);
    System.out.println("Number of rejected customers in queuing system
        1: " + actState.numberOfRejectedQueue1);
    System.out.println("Propability of rejection in queuing system 1: "
         + 1.0 * actState.numberOfRejectedQueue1 /
             actState.numberOfArrivalsQueue1);
  }

  public static void main(String[] args) throws IOException {
    simulate(1);
    simulate(2);
    simulate(5);
  }
}

import java.util.*;

class State extends GlobalSimulation {

  // Here follows the state variables and other variables that might be
      needed
  // e.g. for measurements
  public int numberInQueue1 = 0, numberOfRejectedQueue1 = 0,
      numberOfArrivalsQueue1 = 0, numberInQueue2 = 0,
       accumulated = 0, noMeasurements = 0;

  Random slump = new Random(); // This is just a random number generator

  public int interarrival = 0;

  // The following method is called by the main program each time a new
      event
  // has been fetched
  // from the event list in the main loop.
  public void treatEvent(Event x) {
    switch (x.eventType) {
    case ARRIVAL:
      arrival();
      break;
```

```
   case READY:
      ready();
      break;
   case ARRIVAL2:
      arrival2();
      break;
   case READY2:
      ready2();
      break;
   case MEASURE:
      measure();
      break;
   }
}

// The following methods defines what should be done when an event
    takes
// place. This could
// have been placed in the case in treatEvent, but often it is
    simpler to
// write a method if
// things are getting more complicated than this.

private void arrival() {
   numberOfArrivalsQueue1++;
   if (numberInQueue1 == 0)
      insertEvent(READY, time + expDist(2.1));
   insertEvent(ARRIVAL, time + interarrival);
   if (numberInQueue1 < 10) {
      numberInQueue1++;
   } else {
      numberOfRejectedQueue1++;
   }
}

private void ready() {
   numberInQueue1--;
   insertEvent(ARRIVAL2, time);
   if (numberInQueue1 > 0)
      insertEvent(READY, time + expDist(2.1));
}

private void arrival2() {
   if (numberInQueue2 == 0)
      insertEvent(READY2, time + 2);
   numberInQueue2++;
}

private void ready2() {
   numberInQueue2--;
```

```
      if (numberInQueue2 > 0)
         insertEvent(READY2, time + 2);
   }

   private void measure() {
      accumulated = accumulated + numberInQueue2;
      noMeasurements++;
      insertEvent(MEASURE, time + expDist(5));
   }

   private double expDist(double mean) {
      return -(mean) * Math.log(slump.nextDouble());
   }
}
```

Task 2

```
// As the name indicates this class contains the definition of an event.
    next is needed to
// build a linked list which is used by the EventListClass. It would
    have been just as easy
// to use a priority list which sorts events using eventTime.

class Event {
   public double eventTime;
   public int eventType;
   public Event next;
}

public class EventListClass {

   private Event list, last; // Used to build a linked list

   EventListClass() {
      list = new Event();
      last = new Event();
      list.next = last;
   }

   // The method insertEvent creates a new event, and searches the list
       of
   // events for the
   // right place to put the new event.

   public void InsertEvent(int type, double TimeOfEvent) {
      Event dummy, predummy;
      Event newEvent = new Event();
      newEvent.eventType = type;
```

```
    newEvent.eventTime = TimeOfEvent;
    predummy = list;
    dummy = list.next;
    while ((dummy.eventTime < newEvent.eventTime) & (dummy != last)) {
      predummy = dummy;
      dummy = dummy.next;
    }
    predummy.next = newEvent;
    newEvent.next = dummy;
  }

  // The following method removes and returns the first event in the
      list.
  // That is the
  // event with the smallest time stamp, i.e. the next thing that shall
      take
  // place.

  public Event fetchEvent() {
    Event dummy;
    dummy = list.next;
    list.next = dummy.next;
    dummy.next = null;
    return dummy;
  }
}

public class GlobalSimulation{

  // This class contains the definition of the events that shall take
      place in the
  // simulation. It also contains the global time, the event list and
      also a method
  // for insertion of events in the event list. That is just for making
      the code in
  // MainSimulation.java and State.java simpler (no dot notation is
      needed).

  public static final int ARRIVALA = 1, ARRIVALB = 2, READYA = 3,
      READYB = 4, ARRIVALAEXP = 5, ARRIVALBEXP = 6, READYAEXP = 7,
      READYBEXP = 8, MEASURE = 9; // The events, add or remove if
      needed!
  public static double time = 0; // The global time variable
  public static EventListClass eventList = new EventListClass(); // The
      event list used in the program
  public static void insertEvent(int type, double TimeOfEvent){ // Just
      to be able to skip dot notation
    eventList.InsertEvent(type, TimeOfEvent);
  }
}
```

```java
import java.io.*;

public class MainSimulation extends GlobalSimulation {

  public static final int PRIOB = 1, EXP = 2, PRIOA = 3;

  public static void simulate(int type, String fileName) throws
      FileNotFoundException, UnsupportedEncodingException {
    Event actEvent;
    // The state that shoud be used
    // Some events must be put in the event list at the beginning
    String typeString = "";

    State actState = new State();

    if (type == PRIOB) {
      actState.prio = true;
      insertEvent(ARRIVALA, 0);
      insertEvent(MEASURE, 5);
      typeString = "(Prio B)";
    } else if (type == EXP) {
      actState.prio = true;
      insertEvent(ARRIVALAEXP, 0);
      insertEvent(MEASURE, 5);
      typeString = "(Prio A and Exp delay)";
    } else if (type == PRIOA) {
      actState.prio = false;
      insertEvent(ARRIVALA, 0);
      insertEvent(MEASURE, 0);
      typeString = "(Prio A)";
    }

    // The main simulation loop
    while (actState.noMeasurements < 1000) {
      actEvent = eventList.fetchEvent();
      time = actEvent.eventTime;
      actState.treatEvent(actEvent);
    }

    File file = new File("res/" + fileName);
    file.getParentFile().mkdirs();
    PrintWriter pw = new PrintWriter(file, "UTF-8");

    for (String s : actState.noCustomers) {
      pw.println(s);
    }

    pw.close();
```

```java
    // Printing the result of the simulation, in this case a mean value
    System.out.println("---------- Mean number of jobs " + typeString
        + " ----------");
    System.out.println("Mean number of jobs in buffer: " + 1.0 *
        actState.accumulated / actState.noMeasurements);
  }

  public static void main(String[] args) throws IOException {
    simulate(PRIOB, "priob.txt");
    simulate(EXP, "priobexp.txt");
    simulate(PRIOA, "prioa.txt");
  }
}

import java.util.*;

class State extends GlobalSimulation {

  // Here follows the state variables and other variables that might be
      needed
  // e.g. for measurements
  public int numberInQueue = 0, accumulated = 0, noMeasurements = 0,
      noOfA = 0, noOfB = 0;
  public boolean prio;

  Random slump = new Random(); // This is just a random number generator
  public ArrayList<String> noCustomers = new ArrayList<String>();

  // The following method is called by the main program each time a new
      event
  // has been fetched
  // from the event list in the main loop.
  public void treatEvent(Event x) {
    switch (x.eventType) {
    case ARRIVALA:
      arrivalA();
      break;
    case READYA:
      readyA();
      break;
    case ARRIVALB:
      arrivalB();
      break;
    case READYB:
      readyB();
      break;
    case MEASURE:
      measure();
      break;
    case ARRIVALAEXP:
```

```java
            arrivalAExp();
            break;
        case READYAEXP:
            readyAExp();
            break;
        case READYBEXP:
            readyBExp();
            break;
        }
    }

    // The following methods defines what should be done when an event
        takes
    // place. This could
    // have been placed in the case in treatEvent, but often it is
        simpler to
    // write a method if
    // things are getting more complicated than this.

    private void arrivalA() {
        if (noOfA + noOfB == 0) {
            insertEvent(READYA, time + 0.002);
        }
        noOfA++;
        insertEvent(ARRIVALA, time + expDist((double) 1 / 150));
    }

    private void readyA() {
        noOfA--;
        insertEvent(ARRIVALB, time + 1.0);
        checkQueue();
    }

    private void arrivalB() {
        if (noOfA + noOfB == 0) {
            insertEvent(READYB, time + 0.004);
        }
        noOfB++;
    }

    private void readyB() {
        noOfB--;
        checkQueue();
    }

    private void measure() {
        accumulated = accumulated + noOfA + noOfB;
        noMeasurements++;
        noCustomers.add(Integer.toString(noOfA + noOfB));
        insertEvent(MEASURE, time + 0.1);
```

```java
}

private void checkQueue() {
   if (prio) {
      if (noOfB > 0) {
         insertEvent(READYB, time + 0.004);
      } else if (noOfA > 0) {
         insertEvent(READYA, time + 0.002);
      }
   } else {
      if (noOfA > 0) {
         insertEvent(READYA, time + 0.002);
      } else if (noOfB > 0) {
         insertEvent(READYB, time + 0.004);
      }
   }
}

private void arrivalAExp() {
   if (noOfA + noOfB == 0) {
      insertEvent(READYAEXP, time + 0.002);
   }
   noOfA++;
   insertEvent(ARRIVALAEXP, time + expDist((double) 1 / 150));
}

private void readyAExp() {
   noOfA--;
   insertEvent(ARRIVALB, time + expDist(1.0));
   checkQueueExp();
}

private void readyBExp() {
   noOfB--;
   checkQueueExp();
}

private void checkQueueExp() {
   if (prio) {
      if (noOfB > 0) {
         insertEvent(READYBEXP, time + 0.004);
      } else if (noOfA > 0) {
         insertEvent(READYAEXP, time + 0.002);
      }
   } else {
      if (noOfA > 0) {
         insertEvent(READYAEXP, time + 0.002);
      } else if (noOfB > 0) {
         insertEvent(READYBEXP, time + 0.004);
      }
```

```
        }
    }

    private double expDist(double mean) {
        return -(mean) * Math.log(slump.nextDouble());
    }

}
```

Task 3

```
// As the name indicates this class contains the definition of an event.
     next is needed to
// build a linked list which is used by the EventListClass. It would
     have been just as easy
// to use a priority list which sorts events using eventTime.

class Event {
    public double eventTime;
    public int eventType;
    public Event next;
}

public class EventListClass {

    private Event list, last; // Used to build a linked list

    EventListClass() {
        list = new Event();
        last = new Event();
        list.next = last;
    }

    // The method insertEvent creates a new event, and searches the list
         of
    // events for the
    // right place to put the new event.

    public void InsertEvent(int type, double TimeOfEvent) {
        Event dummy, predummy;
        Event newEvent = new Event();
        newEvent.eventType = type;
        newEvent.eventTime = TimeOfEvent;
        predummy = list;
        dummy = list.next;
        while ((dummy.eventTime < newEvent.eventTime) & (dummy != last)) {
            predummy = dummy;
            dummy = dummy.next;
```

```java
      }
      predummy.next = newEvent;
      newEvent.next = dummy;
   }

   // The following method removes and returns the first event in the
       list.
   // That is the
   // event with the smallest time stamp, i.e. the next thing that shall
       take
   // place.

   public Event fetchEvent() {
      Event dummy;
      dummy = list.next;
      list.next = dummy.next;
      dummy.next = null;
      return dummy;
   }
}

public class GlobalSimulation{

   // This class contains the definition of the events that shall take
       place in the
   // simulation. It also contains the global time, the event list and
       also a method
   // for insertion of events in the event list. That is just for making
       the code in
   // MainSimulation.java and State.java simpler (no dot notation is
       needed).

   public static final int ARRIVAL = 1, READY = 2, MEASURE = 3, ARRIVAL2
       = 4, READY2 = 5; // The events, add or remove if needed!
   public static double time = 0; // The global time variable
   public static EventListClass eventList = new EventListClass(); // The
       event list used in the program
   public static void insertEvent(int type, double TimeOfEvent){ // Just
       to be able to skip dot notation
      eventList.InsertEvent(type, TimeOfEvent);
   }
}

import java.io.*;

public class MainSimulation extends GlobalSimulation {

   public static void simulate(double interarrival) {
      Event actEvent;
      State actState = new State(); // The state that shoud be used
```

```java
      actState.interarrival = interarrival; // Interarrival set to State
                                 // class.
      // Some events must be put in the event list at the beginning
      insertEvent(ARRIVAL, 0);
      insertEvent(MEASURE, 5);

      // The main simulation loop
      while (actState.noMeasurements < 1000) {
         actEvent = eventList.fetchEvent();
         time = actEvent.eventTime;
         actState.treatEvent(actEvent);
      }

      // Printing the result of the simulation, in this case a mean value
      System.out.println("---------- Interarrival time " + interarrival
          + " ----------");
      System.out.println(
           "Mean number of customers in queuing system: " + 1.0 *
                actState.accumulated / actState.noMeasurements);
      System.out.println(
           "Mean time a customer spends in the queuing network: " +
                (actState.timeSpent / actState.noOfReady));

   }

   public static void main(String[] args) throws IOException {
      simulate(2);
      simulate(1.5);
      simulate(1.1);
   }
}

import java.util.*;

class State extends GlobalSimulation {

   // Here follows the state variables and other variables that might be
       needed
   // e.g. for measurements
   public int numberInQueue1 = 0, numberOfRejectedQueue1 = 0,
       numberOfArrivalsQueue1 = 0, numberInQueue2 = 0,
        accumulated = 0, noMeasurements = 0;

   Random slump = new Random(); // This is just a random number generator

   public double interarrival = 0;
   public double timeSpent = 0;
   public double noOfReady = 0;
   public LinkedList<Double> arrivals = new LinkedList<Double>();
```

```java
// The following method is called by the main program each time a new
    event
// has been fetched
// from the event list in the main loop.
public void treatEvent(Event x) {
   switch (x.eventType) {
   case ARRIVAL:
      arrival();
      break;
   case READY:
      ready();
      break;
   case ARRIVAL2:
      arrival2();
      break;
   case READY2:
      ready2();
      break;
   case MEASURE:
      measure();
      break;
   }
}

// The following methods defines what should be done when an event
    takes
// place. This could
// have been placed in the case in treatEvent, but often it is
    simpler to
// write a method if
// things are getting more complicated than this.

private void arrival() {
   numberOfArrivalsQueue1++;
   arrivals.addLast(new Double(time));
   if (numberInQueue1 == 0) {
      insertEvent(READY, time + expDist(1));
   }
   insertEvent(ARRIVAL, time + expDist(interarrival));
   numberInQueue1++;
}

private void ready() {
   numberInQueue1--;
   insertEvent(ARRIVAL2, time);
   if (numberInQueue1 > 0) {
      insertEvent(READY, time + expDist(1));
   }

}
```

```
  private void arrival2() {
    if (numberInQueue2 == 0) {
      insertEvent(READY2, time + expDist(1));
    }
    numberInQueue2++;
  }

  private void ready2() {
    numberInQueue2--;
    timeSpent += time - arrivals.poll().doubleValue();
    noOfReady++;
    if (numberInQueue2 > 0) {
      double temp = expDist(1);
      timeSpent += temp;
      insertEvent(READY2, time + temp);
    }

  }

  private void measure() {
    accumulated = accumulated + numberInQueue1 + numberInQueue2;
    noMeasurements++;
    insertEvent(MEASURE, time + expDist(5));
  }

  private double expDist(double mean) {
    return -(mean) * Math.log(slump.nextDouble());
  }
}
```

Task 4

```
// As the name indicates this class contains the definition of an event.
    next is needed to
// build a linked list which is used by the EventListClass. It would
    have been just as easy
// to use a priority list which sorts events using eventTime.

class Event {
  public double eventTime;
  public int eventType;
  public Event next;
}

public class EventListClass {

  private Event list, last; // Used to build a linked list
```

```
    EventListClass() {
        list = new Event();
        last = new Event();
        list.next = last;
    }

    // The method insertEvent creates a new event, and searches the list
        of
    // events for the
    // right place to put the new event.

    public void InsertEvent(int type, double TimeOfEvent) {
        Event dummy, predummy;
        Event newEvent = new Event();
        newEvent.eventType = type;
        newEvent.eventTime = TimeOfEvent;
        predummy = list;
        dummy = list.next;
        while ((dummy.eventTime < newEvent.eventTime) & (dummy != last)) {
            predummy = dummy;
            dummy = dummy.next;
        }
        predummy.next = newEvent;
        newEvent.next = dummy;
    }

    // The following method removes and returns the first event in the
        list.
    // That is the
    // event with the smallest time stamp, i.e. the next thing that shall
        take
    // place.

    public Event fetchEvent() {
        Event dummy;
        dummy = list.next;
        list.next = dummy.next;
        dummy.next = null;
        return dummy;
    }
}

public class GlobalSimulation{

    // This class contains the definition of the events that shall take
        place in the
    // simulation. It also contains the global time, the event list and
        also a method
```

```java
  // for insertion of events in the event list. That is just for making
      the code in
  // MainSimulation.java and State.java simpler (no dot notation is
      needed).

  public static final int ARRIVAL = 1, READY = 2, MEASURE = 3; // The
      events, add or remove if needed!
  public static double time = 0; // The global time variable
  public static EventListClass eventList = new EventListClass(); // The
      event list used in the program
  public static void insertEvent(int type, double TimeOfEvent){ // Just
      to be able to skip dot notation
    eventList.InsertEvent(type, TimeOfEvent);
  }
}

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;

public class MainSimulation extends GlobalSimulation {

  public static void simulate(int noServers, int serviceTime, double
      interarrival, double timeMeasurements,
       int totalNoMeasurements, String fileName) throws
          FileNotFoundException, UnsupportedEncodingException {

    Event actEvent;
    State actState = new State(); // The state that shoud be used
    // Some events must be put in the event list at the beginning
    insertEvent(ARRIVAL, 0);
    insertEvent(MEASURE, 5);

    // Variables declared in actstate
    actState.noServers = noServers;
    actState.serviceTime = serviceTime;
    actState.interarrival = interarrival;
    actState.timeMeasurements = timeMeasurements;

    // The main simulation loop
    while (actState.noMeasurements < totalNoMeasurements) {
      actEvent = eventList.fetchEvent();
      time = actEvent.eventTime;
      actState.treatEvent(actEvent);
    }

    // Creating file in res directory
    File file = new File("res/" + fileName);
    file.getParentFile().mkdirs();
    PrintWriter pw = new PrintWriter(file, "UTF-8");
```

```java
    for (String s : actState.noCustomers) {
      System.out.println(s);
      pw.println(s);
    }

    pw.close();
  }

  public static void main(String[] args) throws IOException {
    // simulate(1000, 100, (double) 1 / 8, 1, 1000, "4.1.txt");
    // simulate(1000, 10, (double) 1 / 80, 1, 1000, "4.2.txt");
    // simulate(1000, 200, (double) 1 / 4, 1, 1000, "4.3.txt");
    // simulate(100, 10, (double) 1 / 4, 4, 1000, "4.4.txt");
    // simulate(100, 10, (double) 1 / 4, 1, 4000, "4.5.txt");
    // simulate(100, 10, (double) 1 / 4, 4, 4000, "4.6.txt");
  }
}

import java.util.*;

class State extends GlobalSimulation {

  // Here follows the state variables and other variables that might be
      needed
  // e.g. for measurements
  public int serversOccupied = 0, accumulated = 0, noMeasurements = 0,
      noServers, serviceTime;
  public double interarrival, timeMeasurements;

  // List holding results from measurements
  public ArrayList<String> noCustomers = new ArrayList<String>();

  Random slump = new Random(); // This is just a random number generator

  // The following method is called by the main program each time a new
      event
  // has been fetched
  // from the event list in the main loop.
  public void treatEvent(Event x) {
    switch (x.eventType) {
    case ARRIVAL:
      arrival();
      break;
    case READY:
      ready();
      break;
    case MEASURE:
      measure();
      break;
```

```
    }
  }

  // The following methods defines what should be done when an event
      takes
  // place. This could
  // have been placed in the case in treatEvent, but often it is
      simpler to
  // write a method if
  // things are getting more complicated than this.

  private void arrival() {
    if (serversOccupied < noServers) {
      serversOccupied++;
      insertEvent(READY, time + serviceTime);
    }
    insertEvent(ARRIVAL, time + expDist(interarrival));
  }

  private void ready() {
    serversOccupied--;
  }

  private void measure() {
    noMeasurements++;
    noCustomers.add(Integer.toString(serversOccupied));
    insertEvent(MEASURE, time + timeMeasurements);
  }

  private double expDist(double mean) {
    return -(mean) * Math.log(slump.nextDouble());
  }
}
```

Task 5

```
import java.util.*;

//It inherits Proc so that we can use time and the signal names without
    dot notation

class Gen extends Proc {

  public QS Q1, Q2, Q3, Q4, Q5;
  public ArrayList<QS> queueList = new ArrayList<QS>();

  private int roundInt = 1;
```

```java
// The random number generator is started:
Random slump = new Random();

// There are two parameters:
public Proc sendTo; // Where to send customers
public double lambda; // How many to generate per second

// What to do when a signal arrives
public void TreatSignal(Signal x) {
   switch (x.signalType) {
   case READYRANDOM:
      readyRandom();
      break;
   case READYROUND:
      readyRound();
      break;
   case READYPRIO:
      readyPrio();
      break;
   }
}

// Sends incoming arrivals to dispatcher to random server.
private void readyRandom() {
   switch (slump.nextInt(5) + 1) {
   case 1:
      sendTo = Q1;
      break;
   case 2:
      sendTo = Q2;
      break;
   case 3:
      sendTo = Q3;
      break;
   case 4:
      sendTo = Q4;
      break;
   case 5:
      sendTo = Q5;
      break;
   }

   SignalList.SendSignal(ARRIVAL, sendTo, time);
   SignalList.SendSignal(READYRANDOM, this, time + (2.0 / lambda) *
       slump.nextDouble());
}

private void readyRound() {
   switch (roundInt) {
   case 1:
```

```
            sendTo = Q1;
            break;
        case 2:
            sendTo = Q2;
            break;
        case 3:
            sendTo = Q3;
            break;
        case 4:
            sendTo = Q4;
            break;
        case 5:
            sendTo = Q5;
            break;
        }
        roundInt++;

        if (roundInt == 6) {
            roundInt = 1;
        }

        SignalList.SendSignal(ARRIVAL, sendTo, time);
        SignalList.SendSignal(READYROUND, this, time + (2.0 / lambda) *
             slump.nextDouble());
    }

    private void readyPrio() {
        int temp = Integer.MAX_VALUE;
        QS tempQueue = new QS();
        for (QS qs : queueList) {
            if (qs.numberInQueue < temp) {
                temp = qs.numberInQueue;
                tempQueue = qs;
            }
        }
        sendTo = tempQueue;

        SignalList.SendSignal(ARRIVAL, sendTo, time);
        SignalList.SendSignal(READYPRIO, this, time + (2.0 / lambda) *
             slump.nextDouble());
    }
}

public class Global {
    public static final int ARRIVAL = 1, READY = 2, READYRANDOM = 3,
         READYROUND = 4, READYPRIO = 5, MEASURE = 6;
    public static double time = 0;
    public static double timeSpent;
    public static double noOfReady;
}
```

```java
import java.io.*;

//It inherits Proc so that we can use time and the signal names without
    dot notation

public class MainSimulation extends Global {

   public static void simulate(int type, double interarrival) {
      // The signal list is started and actSignal is declaree. actSignal
          is
      // the latest signal that has been fetched from the
      // signal list in the main loop below.

      Signal actSignal;
      new SignalList();

      // Here process instances are created (two queues and one
          generator) and
      // their parameters are given values.
      QS Q1, Q2, Q3, Q4, Q5;

      Q1 = new QS();
      Q2 = new QS();
      Q3 = new QS();
      Q4 = new QS();
      Q5 = new QS();
      Q1.sendTo = null;
      Q2.sendTo = null;
      Q3.sendTo = null;
      Q4.sendTo = null;
      Q5.sendTo = null;

      Gen Generator = new Gen();
      Generator.lambda = (double) 1 / interarrival; // Uniform 0.12
      Generator.Q1 = Q1;
      Generator.Q2 = Q2;
      Generator.Q3 = Q3;
      Generator.Q4 = Q4;
      Generator.Q5 = Q5;

      // To start the simulation the first signals are put in the signal
          list

      String stringType = "";

      if (type == READYRANDOM) {
         stringType = "(Random)";
         SignalList.SendSignal(READYRANDOM, Generator, time);
         SignalList.SendSignal(MEASURE, Q1, time);
```

```
    SignalList.SendSignal(MEASURE, Q2, time);
    SignalList.SendSignal(MEASURE, Q3, time);
    SignalList.SendSignal(MEASURE, Q4, time);
    SignalList.SendSignal(MEASURE, Q5, time);
} else if (type == READYROUND) {
    stringType = "(Round robin)";
    SignalList.SendSignal(READYROUND, Generator, time);
    SignalList.SendSignal(MEASURE, Q1, time);
    SignalList.SendSignal(MEASURE, Q2, time);
    SignalList.SendSignal(MEASURE, Q3, time);
    SignalList.SendSignal(MEASURE, Q4, time);
    SignalList.SendSignal(MEASURE, Q5, time);
} else if (type == READYPRIO) {
    stringType = "(Prio)";
    SignalList.SendSignal(READYPRIO, Generator, time);
    SignalList.SendSignal(MEASURE, Q1, time);
    SignalList.SendSignal(MEASURE, Q2, time);
    SignalList.SendSignal(MEASURE, Q3, time);
    SignalList.SendSignal(MEASURE, Q4, time);
    SignalList.SendSignal(MEASURE, Q5, time);
    Generator.queueList.add(Q1);
    Generator.queueList.add(Q2);
    Generator.queueList.add(Q3);
    Generator.queueList.add(Q4);
    Generator.queueList.add(Q5);
}

// This is the main loop

while (time < 100000) {
    actSignal = SignalList.FetchSignal();
    time = actSignal.arrivalTime;
    actSignal.destination.TreatSignal(actSignal);
}

// Finally the result of the simulation is printed below:

double totalNoCustomers = (1.0 * Q1.accumulated /
    Q1.noMeasurements)
     + (1.0 * Q2.accumulated / Q2.noMeasurements) + (1.0 *
         Q3.accumulated / Q3.noMeasurements)
     + (1.0 * Q4.accumulated / Q4.noMeasurements) + (1.0 *
         Q5.accumulated / Q5.noMeasurements);

System.out.println("---------- Interarrival time " + interarrival
    + " " + stringType + " ----------");
System.out.println("Mean number of customers in queuing system: "
    + totalNoCustomers);
System.out.println("Result using littles theorem: " + (1 /
    interarrival) * (timeSpent / noOfReady));
```

```
  }

  public static void main(String[] args) throws IOException {
    simulate(READYRANDOM, 0.11); // change values.
  }
}

// This is an abstract class which all classes that are used for
    defining real
// process types inherit. The puropse is to make sure that they all
    define the
// method treatSignal, which is needed in the main program.

public abstract class Proc extends Global {
  public abstract void TreatSignal(Signal x);
}

import java.util.*;

// This class defines a simple queuing system with one server. It
    inherits Proc so that we can use time and the
// signal names without dot notation
class QS extends Proc {
  public int numberInQueue = 0, accumulated, noMeasurements;
  public Proc sendTo;
  Random slump = new Random();
  public LinkedList<Double> arrivals = new LinkedList<Double>();

  public void TreatSignal(Signal x) {
    switch (x.signalType) {
    case ARRIVAL:
      arrival();
      break;

    case READY:
      ready();
      break;

    case MEASURE:
      measure();
      break;
    }
  }

  private void arrival() {
    numberInQueue++;
    arrivals.addLast(time);
    if (numberInQueue == 1) {
      SignalList.SendSignal(READY, this, time + expDist(0.5));
```

```java
        }
    }

    private void ready() {
        numberInQueue--;
        timeSpent += time - arrivals.poll().doubleValue();
        noOfReady++;
        if (sendTo != null) {
            SignalList.SendSignal(ARRIVAL, sendTo, time);
        }
        if (numberInQueue > 0) {
            SignalList.SendSignal(READY, this, time + expDist(0.5));
        }
    }

    private void measure() {
        noMeasurements++;
        accumulated = accumulated + numberInQueue;
        SignalList.SendSignal(MEASURE, this, time + 2 *
              slump.nextDouble());
    }

    private double expDist(double mean) {
        return -(mean) * Math.log(slump.nextDouble());
    }
}

// This class defines a signal. What can be seen here is a mainimum. If
    one wants to add more
// information just do it here.

class Signal {
    public Proc destination;
    public double arrivalTime;
    public int signalType;
    public Signal next;
}

// This class defines the signal list. If one wants to send more
    information than here,
// one can add the extra information in the Signal class and write an
    extra sendSignal method
// with more parameters.

public class SignalList {
    private static Signal list, last;

    SignalList() {
        list = new Signal();
        last = new Signal();
```

```
    list.next = last;
  }

  public static void SendSignal(int type, Proc dest, double arrtime) {
    Signal dummy, predummy;
    Signal newSignal = new Signal();
    newSignal.signalType = type;
    newSignal.destination = dest;
    newSignal.arrivalTime = arrtime;
    predummy = list;
    dummy = list.next;
    while ((dummy.arrivalTime < newSignal.arrivalTime) & (dummy !=
        last)) {
      predummy = dummy;
      dummy = dummy.next;
    }
    predummy.next = newSignal;
    newSignal.next = dummy;
  }

  public static Signal FetchSignal() {
    Signal dummy;
    dummy = list.next;
    list.next = dummy.next;
    dummy.next = null;
    return dummy;
  }
}
```

Task 6

```
// As the name indicates this class contains the definition of an event.
    next is needed to
// build a linked list which is used by the EventListClass. It would
    have been just as easy
// to use a priority list which sorts events using eventTime.

class Event {
  public double eventTime;
  public int eventType;
  public Event next;
}

public class EventListClass {

  private Event list, last; // Used to build a linked list

  EventListClass() {
```

```
      list = new Event();
      last = new Event();
      list.next = last;
   }

   // The method insertEvent creates a new event, and searches the list
       of
   // events for the
   // right place to put the new event.

   public void InsertEvent(int type, double TimeOfEvent) {
      Event dummy, predummy;
      Event newEvent = new Event();
      newEvent.eventType = type;
      newEvent.eventTime = TimeOfEvent;
      predummy = list;
      dummy = list.next;
      while ((dummy.eventTime < newEvent.eventTime) & (dummy != last)) {
         predummy = dummy;
         dummy = dummy.next;
      }
      predummy.next = newEvent;
      newEvent.next = dummy;
   }

   // The following method removes and returns the first event in the
       list.
   // That is the
   // event with the smallest time stamp, i.e. the next thing that shall
       take
   // place.

   public Event fetchEvent() {
      Event dummy;
      dummy = list.next;
      list.next = dummy.next;
      dummy.next = null;
      return dummy;
   }
}

public class GlobalSimulation{

   // This class contains the definition of the events that shall take
       place in the
   // simulation. It also contains the global time, the event list and
       also a method
   // for insertion of events in the event list. That is just for making
       the code in
```

```java
  // MainSimulation.java and State.java simpler (no dot notation is
      needed).

  public static final int ARRIVAL = 1, READY = 2, CLOSE = 3; // The
      events, add or remove if needed!
  public static double time = 0; // The global time variable
  public static EventListClass eventList = new EventListClass(); // The
      event list used in the program
  public static void insertEvent(int type, double TimeOfEvent){ // Just
      to be able to skip dot notation
    eventList.InsertEvent(type, TimeOfEvent);
  }
}

import java.io.*;

public class MainSimulation extends GlobalSimulation {

  public static void main(String[] args) throws IOException {
    Event actEvent;
    State actState; // The state that shoud be used
    // Some events must be put in the event list at the beginning
    double timeSpent = 0;
    double noOfReady = 0;
    double overtime = 0;

    for (int i = 0; i < 1000; i++) {
      actState = new State();
      insertEvent(ARRIVAL, 0);
      // The main simulation loop
      while (!actState.close) {
        actEvent = eventList.fetchEvent();
        time = actEvent.eventTime;
        actState.treatEvent(actEvent);
      }
      timeSpent += actState.timeSpent;
      noOfReady += actState.noOfReady;
      overtime += time - 480;
    }

    // Printing the result of the simulation, in this case a mean value
    int hours = (int) (17 + ((overtime / 1000) / 60));
    int minutes = ((int) (overtime / 1000) % 60) + 1; // Rounding up.

    System.out.println("Average time when his work will have finished:
        " + hours + ":" + minutes);
    System.out.println("Average time from the arrival of a
        prescription until it has been filled in: "
         + timeSpent / noOfReady + " minutes");
  }
```

```
}

import java.util.*;

class State extends GlobalSimulation {

   // Here follows the state variables and other variables that might be
       needed
   // e.g. for measurements
   public int numberInQueue = 0, accumulated = 0, noMeasurements = 0;

   Random slump = new Random(); // This is just a random number generator
   public boolean close = false;
   public LinkedList<Double> arrivals = new LinkedList<Double>();
   public double timeSpent = 0;
   public double noOfReady = 0;

   // The following method is called by the main program each time a new
       event
   // has been fetched
   // from the event list in the main loop.
   public void treatEvent(Event x) {
      switch (x.eventType) {
      case ARRIVAL:
         arrival();
         break;
      case READY:
         ready();
         break;
      case CLOSE:
         close();
         break;
      }
   }

   // The following methods defines what should be done when an event
       takes
   // place. This could
   // have been placed in the case in treatEvent, but often it is
       simpler to
   // write a method if
   // things are getting more complicated than this.

   private void arrival() {
      if (numberInQueue == 0)
         insertEvent(READY, time + (10 * slump.nextDouble() + 10));
      numberInQueue++;
      arrivals.addLast(new Double(time));
      if (time < 480) {
         insertEvent(ARRIVAL, time + expDist(15));
```

```
      } else {
         insertEvent(CLOSE, time);
      }

   }

   private void ready() {
      numberInQueue--;
      timeSpent += time - arrivals.poll().doubleValue();
      noOfReady++;
      if (numberInQueue > 0)
         insertEvent(READY, time + (10 * slump.nextDouble() + 10));
   }

   private void close() {
      if (numberInQueue == 0) {
         close = true;
      } else {
         insertEvent(CLOSE, time + 1); // Try to close again in 1 minute.
      }
   }

   private double expDist(double mean) {
      return -(mean) * Math.log(slump.nextDouble());
   }
}
```

Task 7

```
// As the name indicates this class contains the definition of an event.
    next is needed to
// build a linked list which is used by the EventListClass. It would
    have been just as easy
// to use a priority list which sorts events using eventTime.

class Event {
   public double eventTime;
   public int eventType;
   public Event next;
}

public class EventListClass {

   private Event list, last; // Used to build a linked list

   EventListClass() {
      list = new Event();
      last = new Event();
```

```
      list.next = last;
   }

   // The method insertEvent creates a new event, and searches the list
       of
   // events for the
   // right place to put the new event.

   public void InsertEvent(int type, double TimeOfEvent) {
      Event dummy, predummy;
      Event newEvent = new Event();
      newEvent.eventType = type;
      newEvent.eventTime = TimeOfEvent;
      predummy = list;
      dummy = list.next;
      while ((dummy.eventTime < newEvent.eventTime) & (dummy != last)) {
         predummy = dummy;
         dummy = dummy.next;
      }
      predummy.next = newEvent;
      newEvent.next = dummy;
   }

   // The following method removes and returns the first event in the
       list.
   // That is the
   // event with the smallest time stamp, i.e. the next thing that shall
       take
   // place.

   public Event fetchEvent() {
      Event dummy;
      dummy = list.next;
      list.next = dummy.next;
      dummy.next = null;
      return dummy;
   }
}

public class GlobalSimulation{

   // This class contains the definition of the events that shall take
       place in the
   // simulation. It also contains the global time, the event list and
       also a method
   // for insertion of events in the event list. That is just for making
       the code in
   // MainSimulation.java and State.java simpler (no dot notation is
       needed).
```

```java
  public static final int COMP1 = 1, COMP2 = 2, COMP3 = 3, COMP4 = 4,
      COMP5 = 5; // The events, add or remove if needed!
  public static double time = 0; // The global time variable
  public static EventListClass eventList = new EventListClass(); // The
      event list used in the program
  public static void insertEvent(int type, double TimeOfEvent){ // Just
      to be able to skip dot notation
    eventList.InsertEvent(type, TimeOfEvent);
  }
}

import java.io.*;

public class MainSimulation extends GlobalSimulation {

  public static void main(String[] args) throws IOException {
    Event actEvent;
    State actState; // The state that shoud be used
    double totalTime = 0;

    for (int i = 0; i < 1000; i++) {
      actState = new State();
      // The main simulation loop
      while (!actState.breakdown()) {
        actEvent = eventList.fetchEvent();
        time = actEvent.eventTime;
        actState.treatEvent(actEvent);
      }
      totalTime += time;
    }

    // Printing the result of the simulation, in this case a mean value
    System.out.println("Mean time until the system breaks down: " +
        totalTime / 1000);
  }
}

import java.util.*;

class State extends GlobalSimulation {

  // Here follows the state variables and other variables that might be
      needed
  // e.g. for measurements
  public int numberInQueue = 0, accumulated = 0, noMeasurements = 0;
  public boolean comp1 = true, comp2 = true, comp3 = true, comp4 =
      true, comp5 = true;

  Random slump = new Random(); // This is just a random number generator
```

```java
public State() {
   insertEvent(COMP1, slump.nextDouble() * 4 + 1);
   insertEvent(COMP2, slump.nextDouble() * 4 + 1);
   insertEvent(COMP3, slump.nextDouble() * 4 + 1);
   insertEvent(COMP4, slump.nextDouble() * 4 + 1);
   insertEvent(COMP5, slump.nextDouble() * 4 + 1);
}

// The following method is called by the main program each time a new
    event
// has been fetched
// from the event list in the main loop.
public void treatEvent(Event x) {
   switch (x.eventType) {
   case COMP1:
      comp1();
      break;
   case COMP2:
      comp2();
      break;
   case COMP3:
      comp3();
      break;
   case COMP4:
      comp4();
      break;
   case COMP5:
      comp5();
      break;
   }
}

// The following methods defines what should be done when an event
    takes
// place. This could
// have been placed in the case in treatEvent, but often it is
    simpler to
// write a method if
// things are getting more complicated than this.

private void comp1() {
   comp1 = false;
   insertEvent(COMP2, time);
   insertEvent(COMP5, time);
}

private void comp2() {
   comp2 = false;
}
```

```java
   private void comp3() {
      comp3 = false;
      insertEvent(COMP4, time);
   }

   private void comp4() {
      comp4 = false;
   }

   private void comp5() {
      comp5 = false;
   }

   public boolean breakdown() {
      if (!comp1 && !comp2 && !comp3 && !comp4 && !comp5) {
         return true;
      }
      return false;
   }
}
```