Bearbeitungszeitraum: eine Woche

a) In dieser Aufgabe simulieren wir einen Verpackungsprozess, der aus drei anderen Prozessen besteht:
   - einem Kartonzufuhrprozess,
   - einem Produktablageprozess und
   - einem Prozess zum Abtransport der befüllten Kartons.

   Immer dann, wenn kein Karton mehr an der Befüllposition steht, soll einer dorthin gefahren werden, anschließend mit einem Produkt befüllt und danach weggefahren werden. Im Anschluss daran beginnt der ganze Zyklus von vorn. Die drei Prozesse synchronisieren sich über 3 Signale von denen eins (`ablageposFrei`) signalisiert, dass ein neuer Karton zur (Ablage/)Befüllposition gefahren werden kann, ein zweites, das signalisiert, ob der Karton befüllt werden kann (`produktablageErlaubt`) und ein drittes, das anzeigt, ob der Karton mit einem Produkt befüllt wurde und damit weggefahren werden kann (`produktAbgelegt`). Nach dem Wegfahren des Kartons ist die Ablageposition wieder frei. Die benötigten Signale werden durch Attribute, die in einer gemeinsamen Klasse Signale deklariert sind, simuliert. Der Programmrahmen ist im Paket Aufgabe_a.rar abgespeichert.

   I) Ergänzen Sie den durch Kommentare angedeuteten Programmtext so, dass der Ablageprozess wie oben beschrieben abläuft. Setzen Sie nach Erledigung einer Teilaufgabe, wie z.B. `fahreKartonVor;` alle notwendigen Signale zurück und setzen Sie das Signal auf true, das die darauf folgende Teilaufgabe triggert. Verwenden Sie zur korrekten Koordinierung über die Signale Java's wait/notify-Mechanismus

b) Das folgende Programm beschreibt zwei Reisebüros, die parallel Tickets desselben Anbieters verkaufen.

```
class TicketAnbieter {
   private int VerfuegbareTickets;
   TicketAnbieter (int n) {
      VerfuegbareTickets = n;
   }
   boolean TicketsVerfuegbar() {
      return VerfuegbareTickets > 0;
   }
   int TicketVerkauf() {
      int nr = VerfuegbareTickets;
      VerfuegbareTickets = VerfuegbareTickets - 1;
      return nr;
   }
}

class Reisebuero extends Thread {
   private String name;
   private TicketAnbieter anbieter;
   Reisebuero(String name, TicketAnbieter anbieter) {
      this.name = name;
      this.anbieter = anbieter;
   }
   void warteAufKunde() {
      try {Thread.sleep(Math.round(1000 * Math.random()));
      } catch (InterruptedException e) {
      }
   }
   public void run() {
      warteAufKunde();
      while (anbieter.TicketsVerfuegbar()) {
         int nr = anbieter.TicketVerkauf();
         System.out.println( name + " verkauft Ticket " + nr);
         warteAufKunde();
      }
   }
}

class Test {
   public static void main(String [] argv) {
      TicketAnbieter ta = new TicketAnbieter(4);
      new Reisebuero("Reiseland", ta).start();
      new Reisebuero("Happy Travel", ta).start();
   }
}
```

I) Beschreiben Sie einen Ablauf des Programms, bei dem beide Reisebüros das Ticket mit der Nummer 3 verkaufen, bei dem also sowohl „Happy Travel verkauft Ticket 3“ als auch „Reiseland verkauft Ticket 3“ ausgegeben wird.

II) Um das in I) beschriebene Problem zu lösen, vereinbart ein Kollege von Ihnen die Methoden der Klasse TicketAnbieter als synchronized:

```
synchronized boolean TicketsVerfuegbar() { ... }
synchronized int TicketVerkauf() { ... }
```

Leider ist dies aber immer noch keine befriedigende Lösung, denn es kann passieren, dass das (nicht-existente) Ticket 0 verkauft wird, d.h. es kann z.B. „Reiseland verkauft Ticket 0“ ausgegeben werden. Beschreiben Sie einen Ablauf, der zu dieser Ausgabe führt.

III) Lösen Sie das in II) angesprochene Problem, indem Sie das Programm mit Hilfe von synchronized geeignet abändern. Dabei sollen sich die beiden Threads möglichst wenig gegenseitig behindern.

IV) Lösen Sie das in II) angesprochene Problem **ohne synchronized**, indem Sie das Programm mit Hilfe von AtomicInteger geeignet abändern. Dabei sollen sich die beiden Threads möglichst wenig gegenseitig behindern.