DETI

Problem of the Sleeping Barbers

Analysis of concurrent solutions – 1

António Rui Borges

Departamento de Electrónica, Telecomunicações e Informática

Summary

Classes of solutions
Static solution
- *State diagrams*
- *Life cycles*
- *General characterization of the* `BarberShop` *data type*
- *Implementation* (*monitors / semaphores*)

DETI

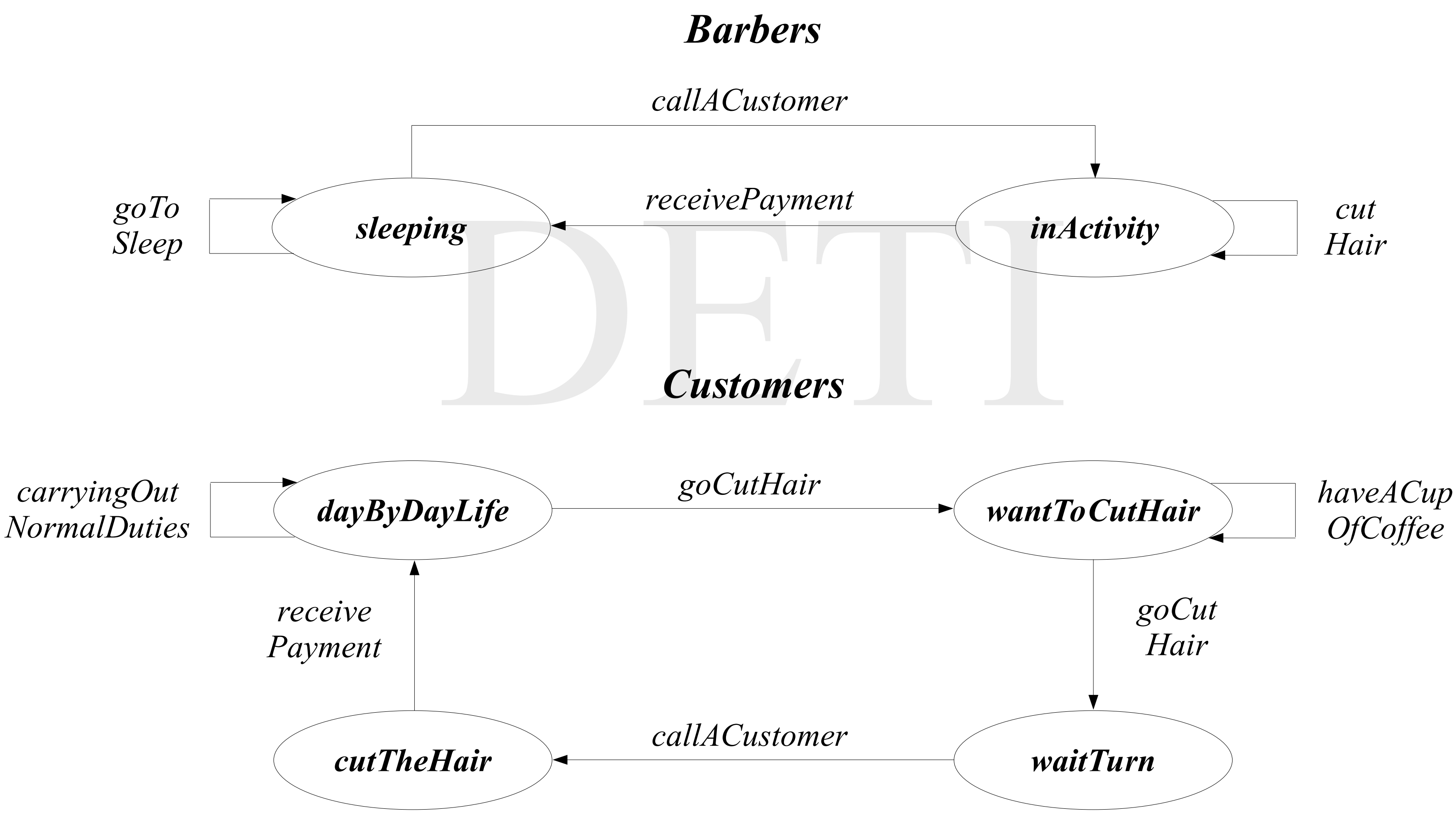
Classes of solutions

There are essentially two different classes of solutions

- *static solutions* – the barber and customer *threads* are all started at the beginning of the simulation and will be kept alive while it does not end
- *dynamic solutions* – only the customer *threads* are started at the beginning of the simulation, the barber *threads* will only be started when a barber is waken up by the arrival of a customer and will be kept alive only while they are required (there are still customers waiting in the barber shop to have their hair cut).

DETI

Static solution: state diagrams



DETI

Static solution: life cycle of the involved entities

Barbers (*b* = 0,1,...,*M*-1)

```
forever
{ goToSleep ();      // the barber sleeps while waiting for a customer to service
  c = callACustomer ();        // the barber has waken up and calls next customer
  cutHair ();                                    // the barber cuts the customer hair
  receivePayment (c);                              // the barber finishes his service
                                                       // and receives payment for it
}
```

Customers (*c* = 0,1,...,*N*-1)

```
forever
{ carryingOutNormalDuties ();           // the customer carries out normal duties
  while (!goCutHair ())             // the customer checks if he can cut his hair
    haveACupOfCoffee ();            // if the barber shop is full, he tries later
}
```

Static solution: barber shop

Solution decomposition supposes the existence of a shared data type, called `BarberShop`, with the following organization

Internal data structure

waiting turn queue (FIFO – stores customer id and has a size *K*)

Synchronization

access with mutual exclusion to the internal data structure
barbers – single blocking point where they wait for a customer
customers – one blocking point per customer where each customer both waits his turn to cut the hair and sits on the cutting chair while having his hair cut

Operations called on it

goToSleep – called by the barber
goCutHair – called by the customer
callACustomer – called the barber
receivePayment – called the barber

DETI

Implementation of the static solution with monitors

The `BarberShop` data type becomes a monitor.

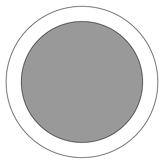
Since *condition variables* are restricted to a single one, associated to the monitor object, by Java concurrency model, it is necessary to turn the blocking conditions explicit by repetitive testing of normal variables.

Synchronization

- *access with mutual exclusion to the internal data structure* – monitor *locking* (all public methods are synchronized)
- *barbers* – single blocking point where they wait for a customer (a variable which signals the number of hair cuts requested by the customers, is increased by each customer at operation *goCutHair* and decreased by a barber at operation *sleep*, after waking up)
- *customers* – one blocking point per customer where each customer both waits his turn to cut the hair and sits on the cutting chair while having his hair cut (use is made of his state value).

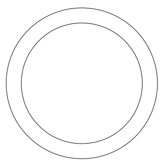
DETI

Pictographs used in the interaction diagrams

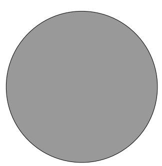


non-instantiated data type (it must be named by the data type identifier)
active entity (it is typically the application main thread)

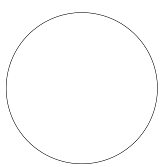
DETI



non-instantiated data type (it must be named by the data type identifier)
passive entity (it is typically a library)

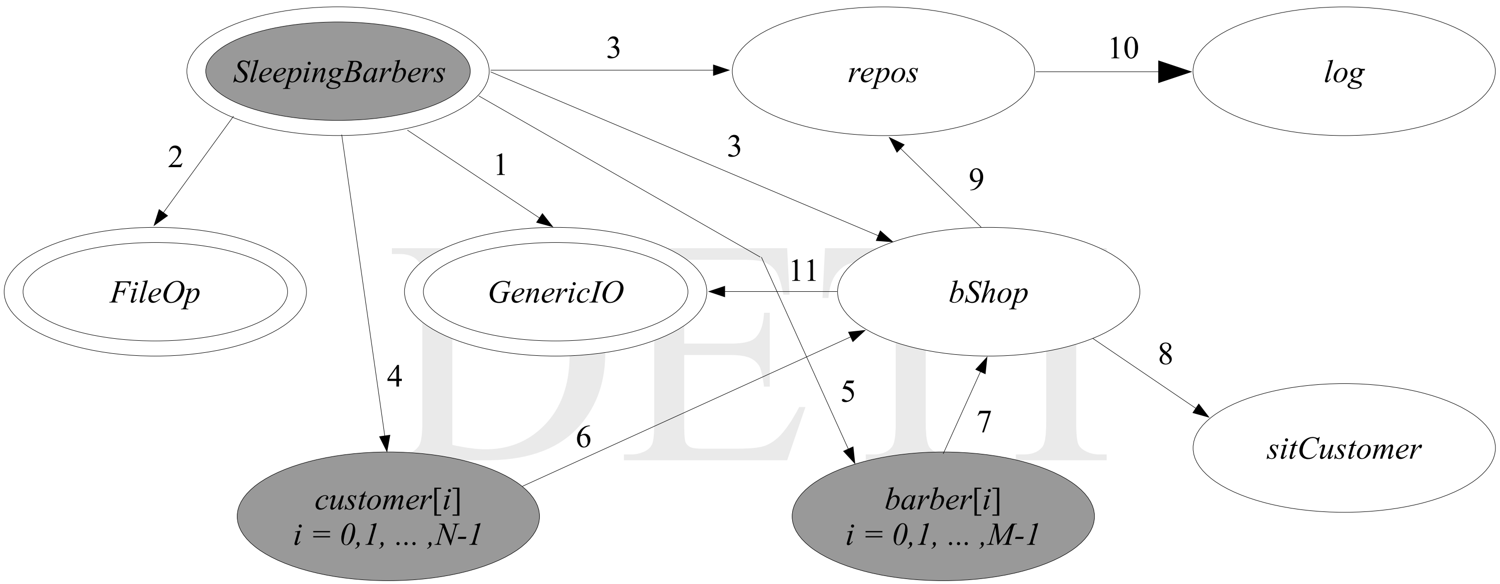


instantiated data type (it must be named by the variable identifier used on its instantiation)
active entity (it is a thread)



instantiated data type (it must be named by the variable identifier used on its instantiation)
passive entity

Interaction diagram: static solution with monitors



1 – readlnInt, readlnChar, readlnString, writeString, writelnString
2 – exists
3 – instantiate
4 – instantiate, start, join
5 – instantiate, start, interrupt, isAlive, join
6 – goCutHair
7 – goToSleep, callACustomer, receivePayment
8 – instantiate, full, write, read
9 – setBarberState, setCustomerState, setBarberCustomerState
10 – instantiate, openForWriting, openForAppending, close, writelnString
11 – writelnString

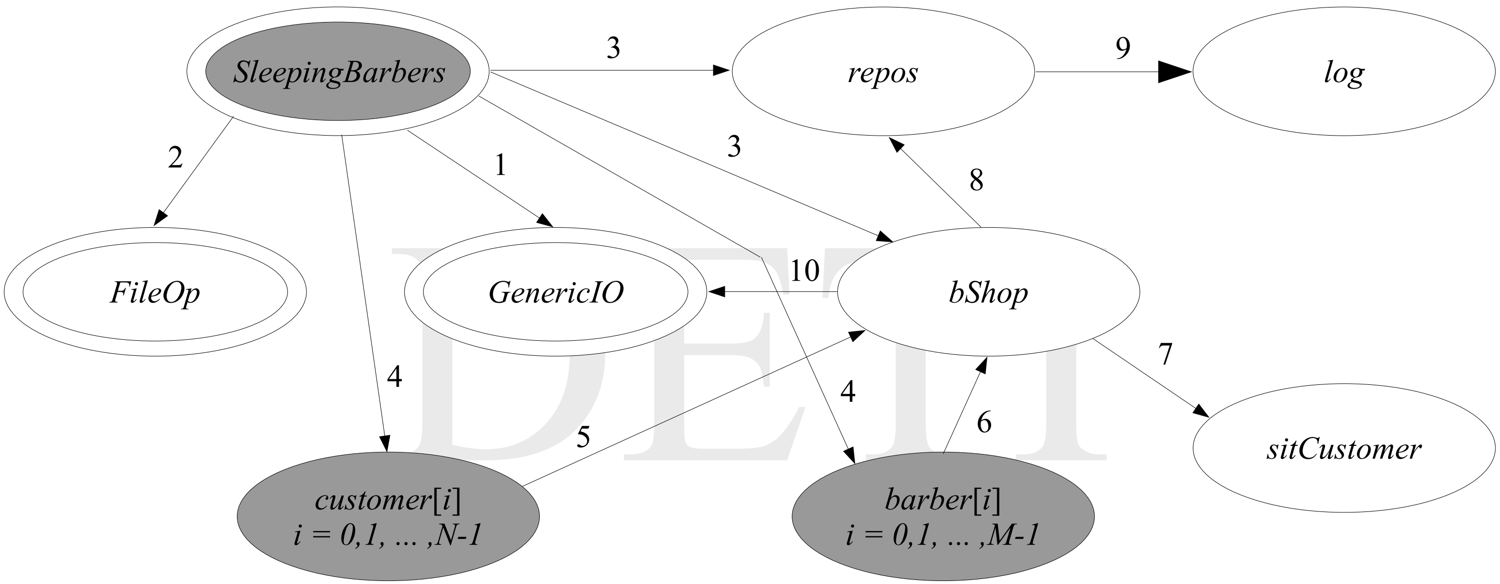
Implementation of the static solution with semaphores

The `BarberShop` data type is a conventional data type.

Synchronization

- *access with mutual exclusion to the internal data structure* – binary semaphore, initialized to *green*
- *barbers* – single blocking point where they wait for a customer (counting semaphore, initialized to *red*)
- *customers* – one blocking point per customer where each customer both waits his turn to cut the hair and sits on the cutting chair while having his hair cut (binary semaphore array, each element initialized to *red*).

Interaction diagram: static solution with semaphores



1 – readlnInt, readlnChar, readlnString, writeString, writelnString
2 – exists
3 – instantiate
4 – instantiate, start, join
5 – goCutHair
6 – goToSleep, callACustomer, receivePayment, goOn
7 – instantiate, full, write, read
8 – setBarberState, setCustomerState, setBarberCustomerState
9 – instantiate, openForWriting, openForAppending, close, writelnString
10 – writelnString

Static solution: end of operations

Although the Problem of the Sleeping Barbers assumes infinite life cycles for both the barbers and the customers, any simulation must make them finite. It is obviously trivial to make the customers life cycle finite. What is trickier is to coordinate the termination of the barbers life cycle with the customers.

Solution based on a posteriori reasoning

- *resource is made to the Java facility of interrupting blocked threads* – after termination of the customer threads, the thread which started the simulation, signals the barber threads it is time to terminate

Solution based on a priori reasoning

- *central processing of the totality of the life cycle iterations of the barber threads* – whenever a barber services a customer, the total number of life cycle iterations is decreased by one and the termination of each barber is made dependent of its presence for the continuation of operations.

Departamento de Electrónica, Telecomunicações e Informática