Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

Programmazione Concorrente e Distribuita

Progettazione di programmi concorrenti

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

Progettazione di programmi concorrenti

- Viene proposta una semplice metodologia di progettazione per applicazioni concorrenti e distribuite
- Useremo la metodologia per progettare e realizzare una soluzione al classico problema dei produttori e consumatore
  ... e per altri esempi

Perché ci serve una metodologia di progettazione?

- Perché i problemi concorrenti e distribuiti sono complessi da capire e da gestire.
- Ci serve una guida che proponga un approccio sistematico al problema.

In cosa consiste la metodologia di progettazione?

- Riguarda la creazione di collezioni di oggetti che si coordinano tra loro per risolvere un problema.
- Il problema viene scomposto in due tipi di oggetti: attivi e passivi.
- Gli oggetti attivi implementano il codice sequenziale che viene eseguito da ogni thread.
- Gli oggetti passivi
  - Reagiscono alle richieste degli oggetti attivi,
  - Contengono elementi di controllo degli oggetti attivi (semafori, metodi synchronized, ecc.)
    - Per essere thread-safe, coordinare e sincronizzare thread, ecc.

Il metodo passo-passo

1. Scrivere una breve descrizione del problema da risolvere.
   - Tipicamente testuale
2. Descrivere il sistema con UML
3. Implementare gli oggetti attivi come thread Java.
4. Implementare gli oggetti passivi come monitor.
5. Scrivere un oggetto di controllo che crea le istanze di tutti gli oggetti.

1. Specifiche testuali del problema

- Esempio: il problema Produttore-Consumatore
  - Un produttore di dati memorizza i dati prodotti in un buffer di capienza limitata.
  - Il produttore crea dati consistenti in numeri interi e li memorizza nel buffer.
  - Il consumatore preleva da buffer i numeri interi e li visualizza sul dispositivo di output.
    - Quando un dato viene prelevato dal consumatore viene cancellato dal buffer e si crea un posto vuoto per un ulteriore dato
  - Produttore e consumatore operano senza soluzione di continuità
  - Il buffer ha la capacità di memorizzare N elementi

1. Specifiche testuali del problema

- Esempio: il problema Produttore-Consumatore
  - Se il produttore produce un dato e il buffer è pieno, non potrà depositarlo immediatamente, ma dovrà aspettare che nel buffer ci sia almeno uno spazio libero.
    - Lo spazio libero si crea solo se il consumatore consuma un dato.
  - Se il consumatore è pronto a recuperare un dato ma il buffer è vuoto, deve aspettare che nel buffer ci sia almeno un dato.
    - Nel buffer compare un dato solo se il produttore ve lo deposita.

2. Modelliamo le specifiche

- Le descrizioni testuali sono passibili di essere male interpretate.
- Creiamo dei modelli
  - Più precisi
  - Più facili da leggere e da comprendere
  - Più facili da analizzare (vogliamo che le specifiche siano complete, consistenti, non ridondanti, ecc.)
- Usiamo UML

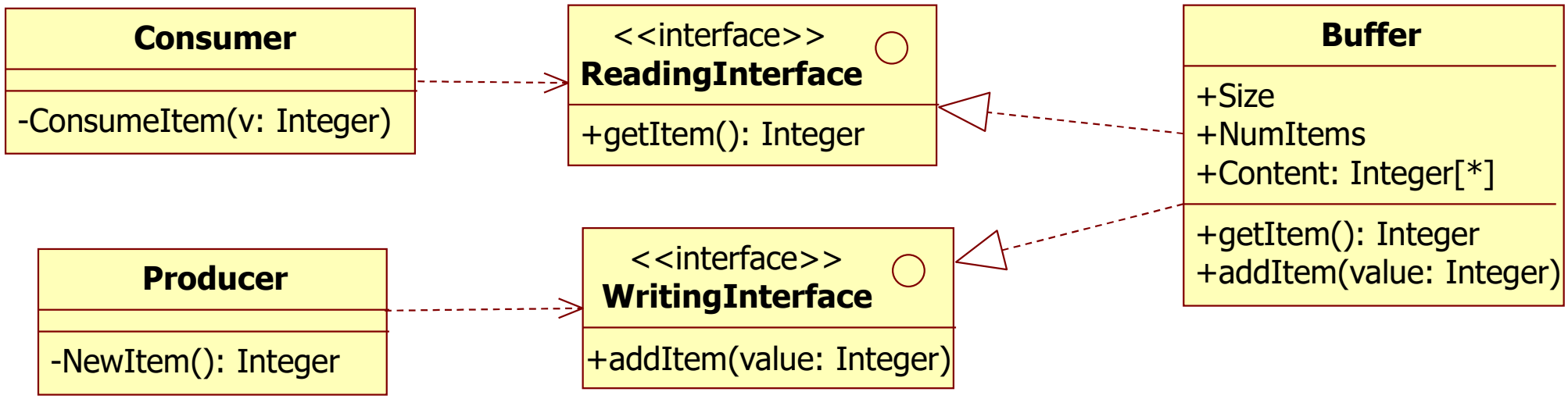
Il modello strutturale

- Rappresentiamo gli elementi che compongono il problema e che ne sono protagonisti.
- Usiamo il class diagram di UML
  - i sostantivi nella descrizione del problema rappresentano gli elementi del problema e il loro contenuto informativo (classi e attributi)
  - i verbi rappresentano le azioni e/o metodi
- NB: si potrebbe usare anche un component diagram

Analisi del testo

- Un **produttore** di dati **memorizza** i **dati** prodotti in un **buffer** di **capienza** limitata.
- Il produttore crea dati consistenti in numeri interi e li memorizza nel buffer.
- Il **consumatore preleva** da buffer i numeri interi e li visualizza sul dispositivo di output.
  - Quando un dato viene prelevato dal consumatore viene **cancellato** dal buffer e si crea un posto vuoto per un ulteriore dato
- Produttore e consumatore operano senza soluzione di continuità
- Il buffer ha la capacità di memorizzare N elementi

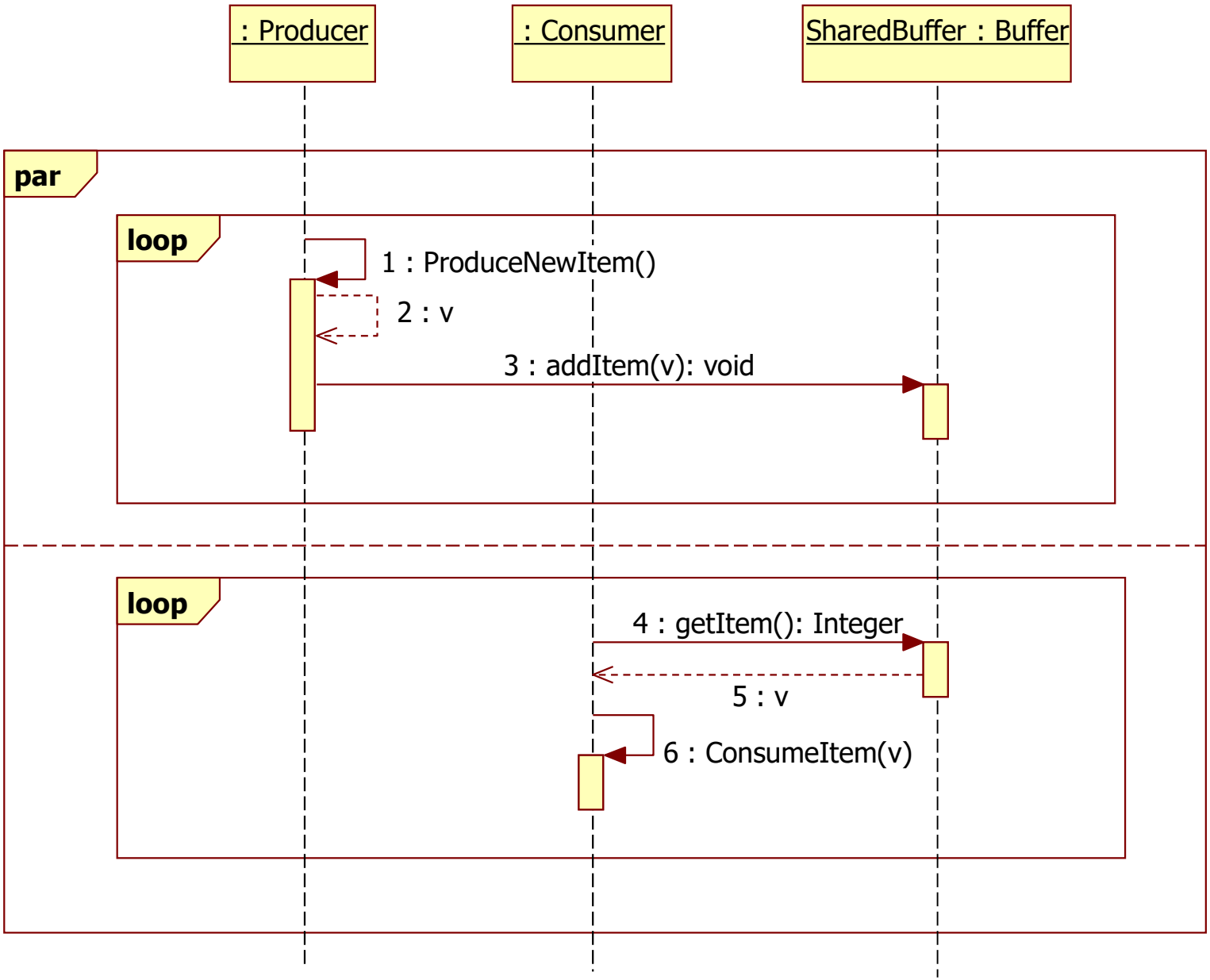
Class diagram



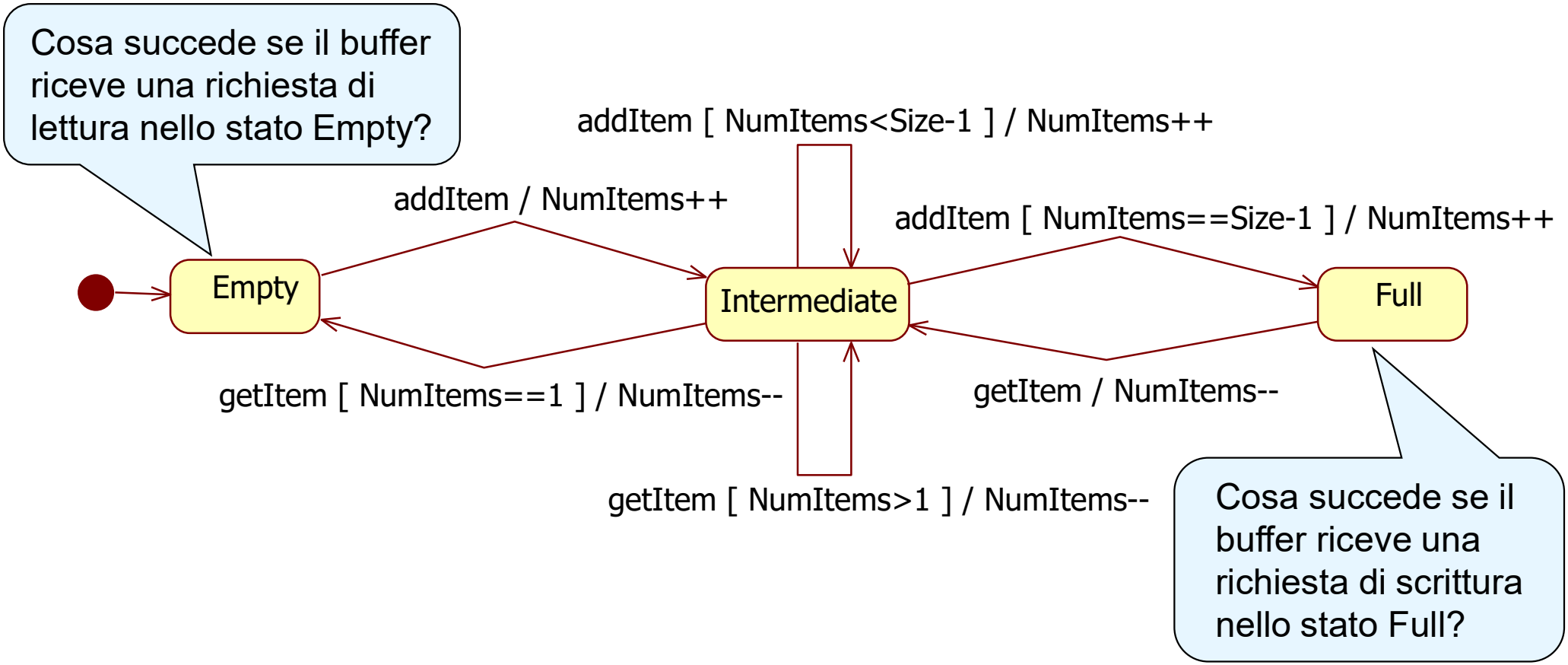
Comportamento degli elementi

- Oggetti passivi
  - Modellabili adeguatamente con un diagramma di stato
- Oggetti attivi
  - Modellabili adeguatamente con un sequence diagram

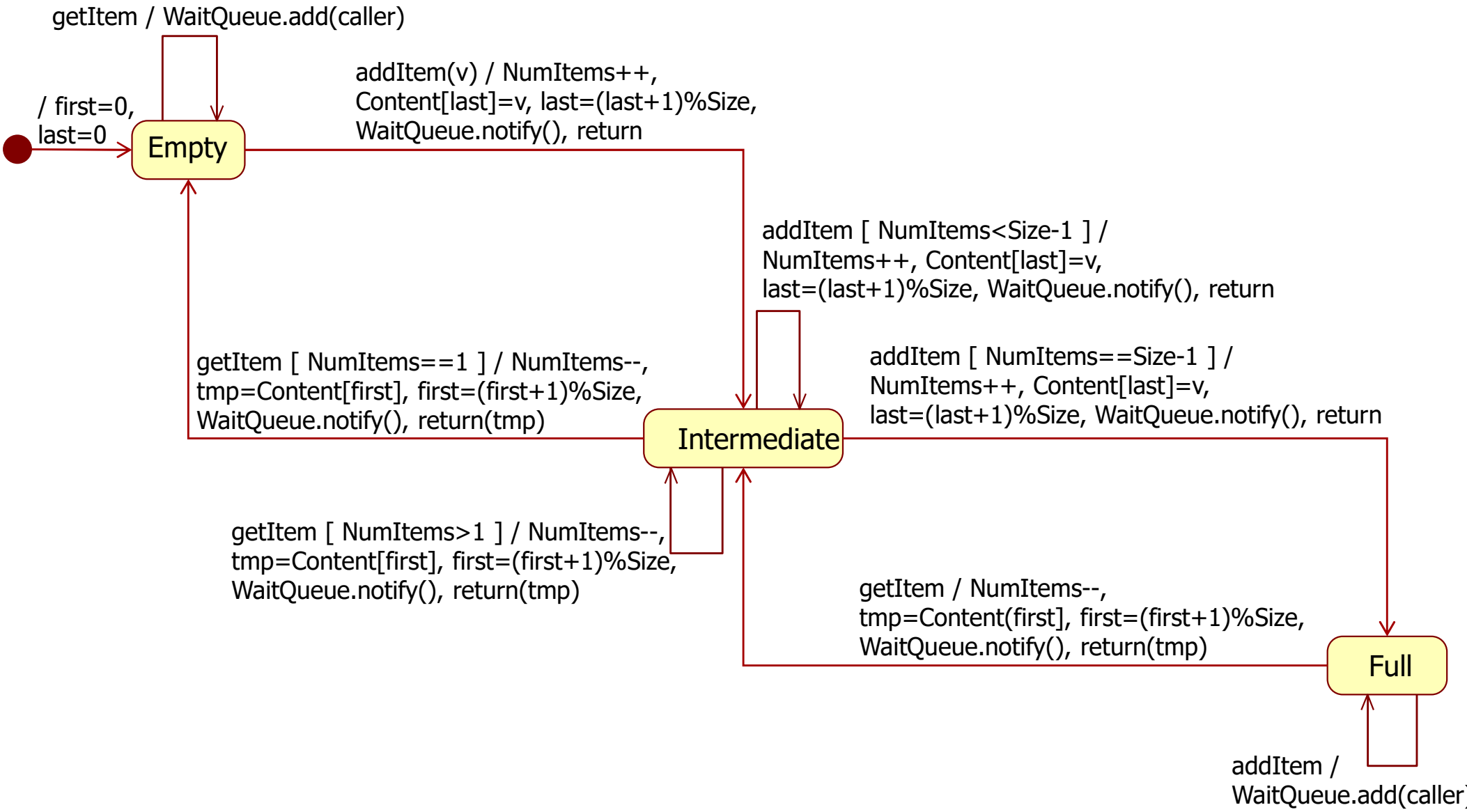
Sequence diagram



State diagram del buffer (avente capienza > 1)

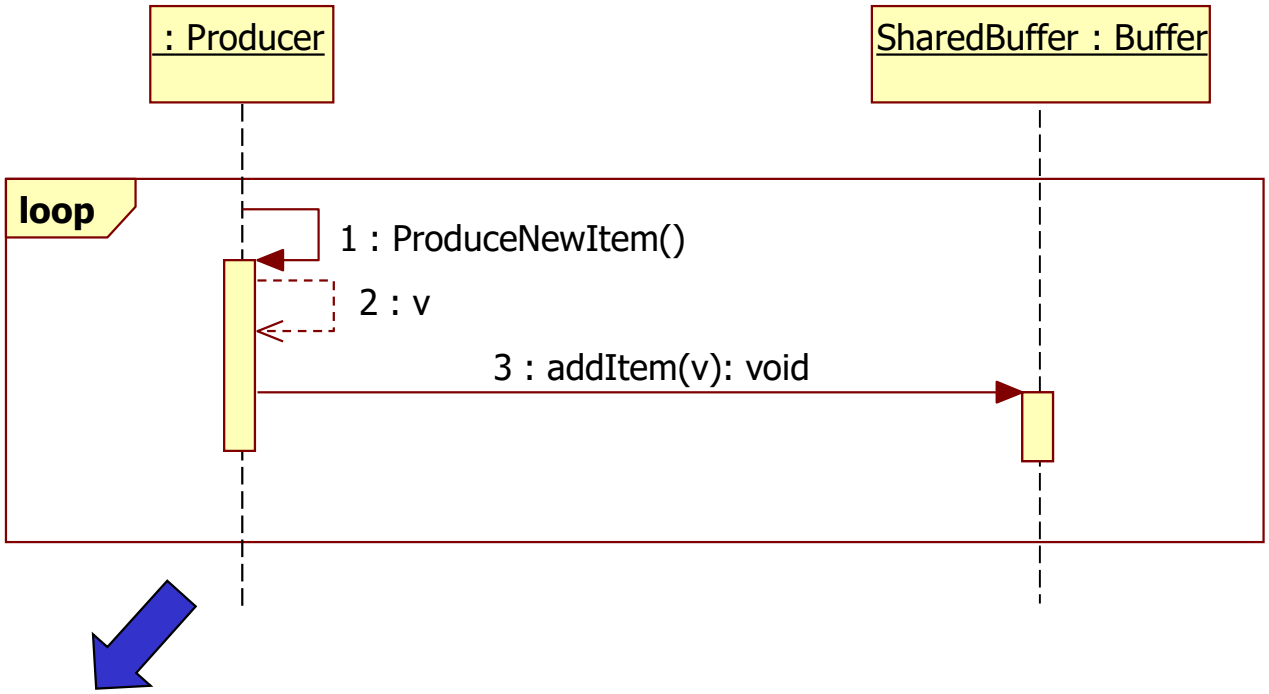


State diagram del Buffer thread-aware



3. Implementare gli oggetti attivi come thread Java

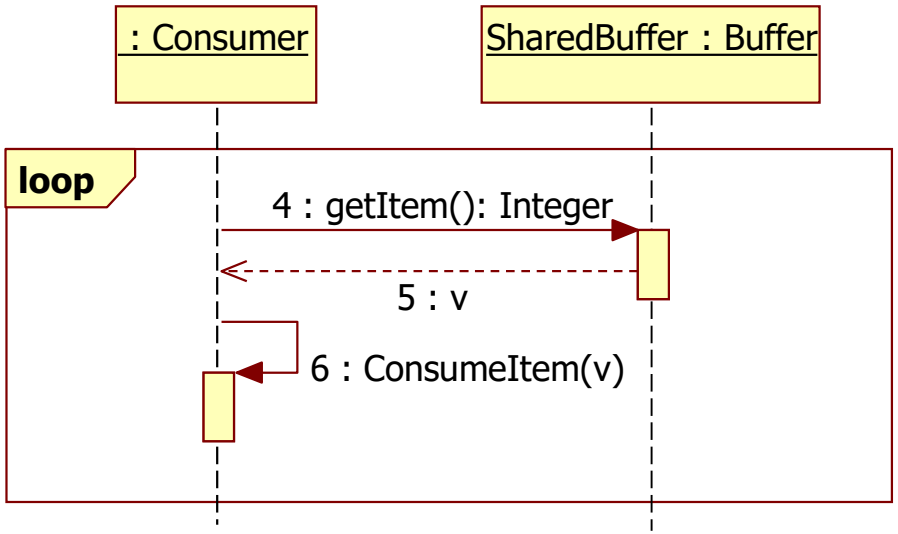
- Nel nostro caso, abbiamo due oggetti attivi: produttore e consumatore.
- Produttore:




```
while(true){
    v=newItem();
    sharedBuffer.addItem(v);
}
```

3. Implementare gli oggetti attivi come thread Java

- Nel nostro caso, abbiamo due oggetti attivi: produttore e consumatore.



- Consumatore:

```
while(true){
    v=sharedBuffer.getItem();
    consumeItem(v);
}
```

Codice

```
public class Consumer extends Thread {
        Buffer sharedBuffer;
        int v;
        public Consumer(Buffer b){
            this.sharedBuffer=b;
        }
        private void consumeItem(int v) {
           // here v is used
        }
        public void run(){
          while(true){
              v=sharedBuffer.getItem();
              consumeItem(v);
          }
        }
}
```

Codice

```
public class Producer extends Thread {
        Buffer sharedBuffer;
        int v;
        public Producer(Buffer b){
            this.sharedBuffer=b;
        }
        private int produceItem() {
           // here v is produced
           return v;
        }
        public void run(){
         while(true){
              v=produceItem();
              sharedBuffer.addItem(v);
           }
        }
}
```

5. Implementare gli oggetti passivi come monitor

```
public class Buffer{
        static int Size;
        private int numItems;
        private int[] Content;
        private int first, last;
        public Buffer(int bufsize) {
                Size=bufsize;
                Content=new int[Size];
                first=0;
                last=0;
                numItems=0;
        }
```

5. Implementare gli oggetti passivi come monitor

```
public synchronized int getItem(){
      int tmp;
      while(numItems==0){
        try { wait(); }
        catch (InterruptedException e) {}
      }
      numItems--;
      tmp=Content[first];
      first=(first+1)%Size;
      notify();
      return tmp;
}
```

5. Implementare gli oggetti passivi come monitor

```
public synchronized void addItem(int v){
      while(numItems==Size){
        try { wait(); }
        catch (InterruptedException e) {}
      }
      Content[last]=v;
      last=(last+1)%Size;
      numItems++;
      notify();
 }
}
```

5. Scrivere un oggetto di controllo che crea le istanze di tutti gli oggetti

- Dobbiamo:
  - Creare il buffer condiviso
  - Creare il produttore, passandogli un riferimento al buffer condiviso
  - Creare il consumatore, passandogli un riferimento al buffer condiviso
  - Lanciare produttore e consumatore

Codice

```
public class ProdCons {
      public static void main(String[] args) {
          Buffer buf=new Buffer(4);
          new Producer(buf).start();
          new Consumer(buf).start();
          new Producer(buf).start();
          new Consumer(buf).start();
      }
}
```

Altro esempio

Simulazione di una stazione di servizio

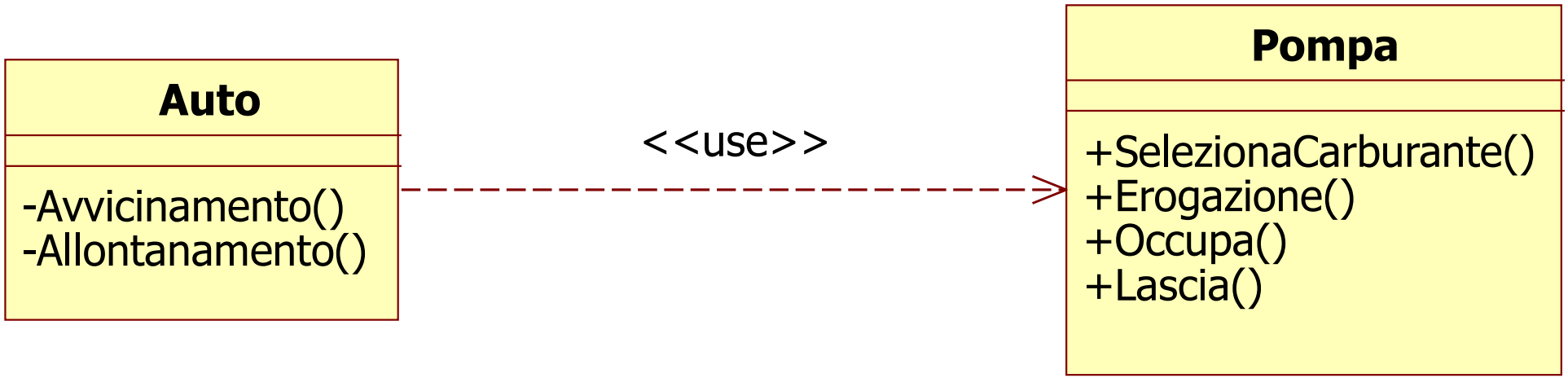
Definizione del problema

- Si vuole modellare una stazione di servizio per calcolare il tempo medio necessario ad un automobilista per ottenere il carburante.
- Il tempo viene calcolato da quando il conducente comincia l'attesa ad una pompa di distribuzione e termina quando l'automobile lascia la stazione di servizio
- La simulazione inizia con 50 vetture in arrivo alla stazione di servizio in modo casuale con ritardi che vanno da 0 a 30 UT (Unità di Tempo).

Comportamento dei conducenti

- Ciascun conducente si comporta come segue:
  - Il conducente verifica se la pompa è libera.
    - Se non è libera aspetta
    - Se è libera avvicina la macchina fino alla pompa, spegne il motore, scende, seleziona il carburante desiderato e inserisce l'erogatore.
      - Queste operazioni richiedono 1 UT
  - Riempie il serbatoio
    - Questa operazione richiede un tempo compreso tra 1 e 3 UT (dipende da quanto carburante si preleva)
  - Ripone l'erogatore, sale in macchina e si allontana dalla pompa, lasciandola libera.
    - Questa operazione richiede 1 UT

Class diagram

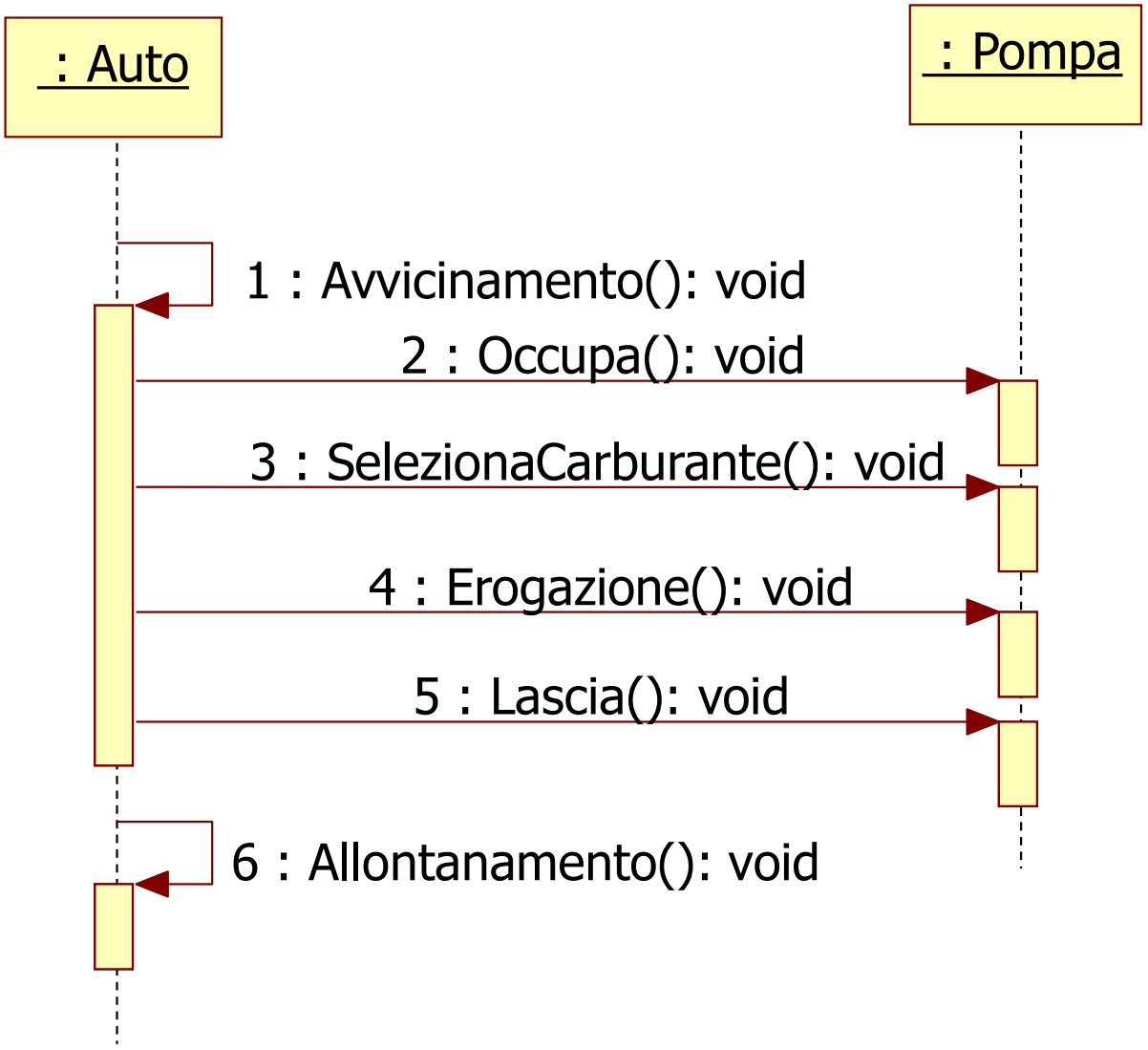


Cosa serve per la simulazione

- Di fatto, sapere esattamente cosa faccia il conducente è irrilevante.
- Quello che interessa è che una volta che la pompa si libera, resta occupata per un tempo complessivo compreso tra 3 e 5 UT.

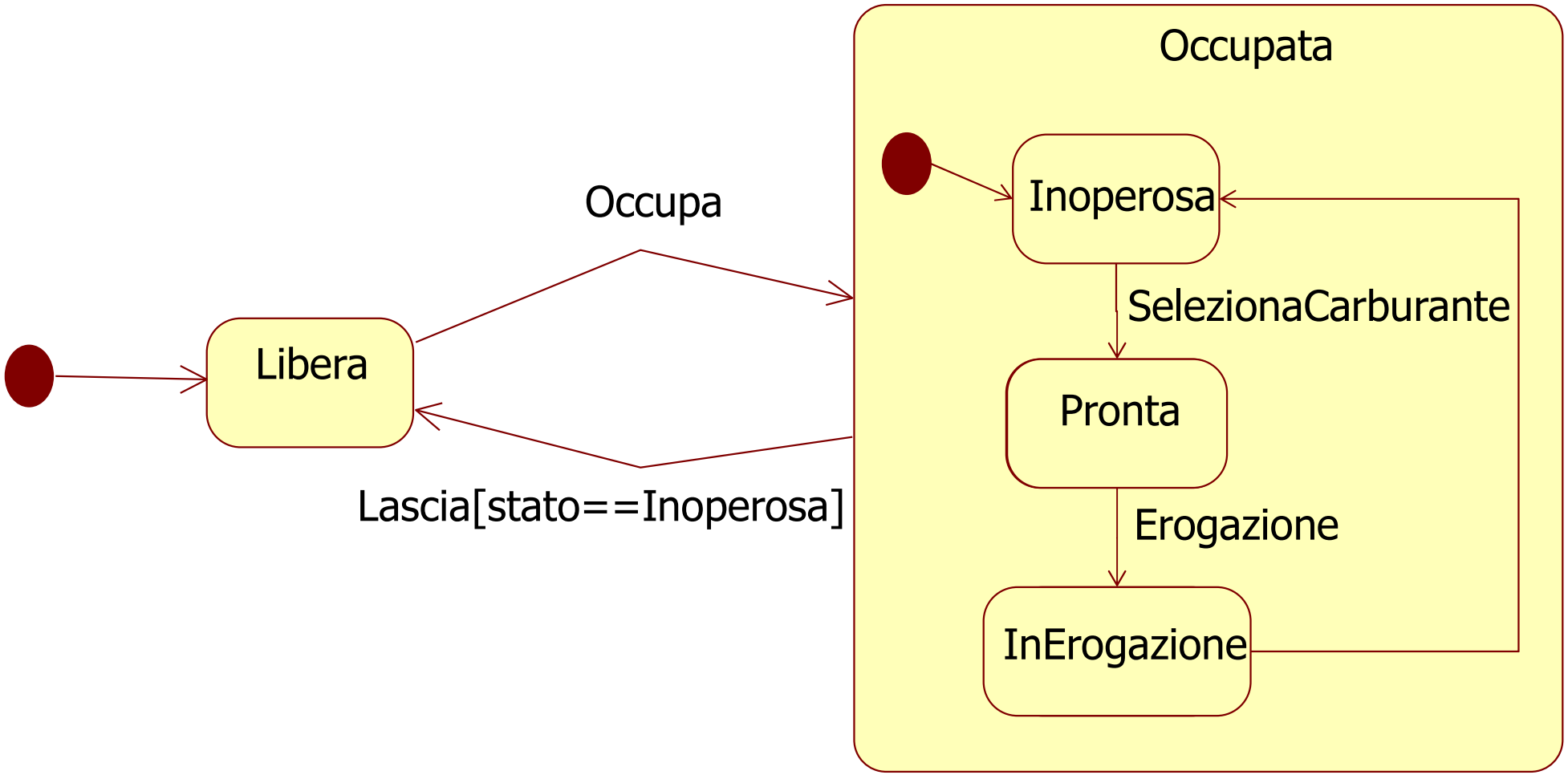
Sequence diagram

- Comportamento dell'auto



State diagram

- Comportamento della pompa



Composizione del sistema

- Un oggetto passivo: la pompa
- Un insieme di oggetti attivi: le auto

Class Auto

```
import java.util.*;
import java.util.concurrent.ThreadLocalRandom;

class Auto implements Runnable {
   private static int tempoTotale = 0,
                      numAuto = 0;
   private int numCliente;
   private Pompa pompaUsata;
   public Auto(int n, Pompa p) {
      this. pompaUsata = p;
      this. numCliente = n;
   }
   public static float calcMedia () {
     return tempoTotale/ numAuto;
   }
```

Class Auto

```
private void avvicinamento() {
  final int MAXtime= 3000;
  try {
   Thread.sleep(ThreadLocalRandom.current().nextInt(1,MAXtime));
  } catch (InterruptedException e) { }
}
private void preparazione() {
  try {
    Thread.sleep(100);
  } catch (InterruptedException e) { }
}
private void allontanamento() {
  try {
    Thread.sleep(40);
  } catch (InterruptedException e) { }
}
```

Class Auto

```
public void run () {
  long tempoInizio, tempoFine;
  avvicinamento();
  System.out.println("Auto " + numCliente+ " arriva a pompa");
  tempoInizio = (new Date()).getTime();
  pompaUsata.occupa();
  System.out.println("Auto "+ numCliente+ " in rifornimento");
  preparazione();
  pompaUsata.eroga();
  System.out.println ("Auto "+ numCliente+ " lascia pompa");
  allontanamento();
  pompaUsata.lascia();
  tempoFine = (new Date()).getTime();
  System.out.println("Tempo auto "+ numCliente+ " = "+
                     (tempoFine-tempoInizio));
  synchronized(Auto.class) {
    tempoTotale+=(tempoFine-tempoInizio);
    numAuto++;
  }
}
}
```

Class Pompa

```
class Pompa{
  private static final int LIBERA = 0 ;
  private static final int OCCUPATA = 1 ;
  private int state = LIBERA;
  public Pompa(){
    state=LIBERA;
  }
  synchronized public void occupa() {
    try {
      while(state != LIBERA)
        wait();
      state = OCCUPATA;
    } catch(InterruptedException e) {}
  }
```

Class Pompa

```
  synchronized public void eroga() {
    try {
      while(state != OCCUPATA)
        wait();
      Thread.sleep(ThreadLocalRandom.current().nextInt(300, 500));
    } catch(InterruptedException e) { }
  }
  synchronized public void lascia() {
    try {
      while(state != OCCUPATA)
        wait();
      state = LIBERA;
      notifyAll();
    } catch(InterruptedException e) {}
  }
}
```

main

```
public class StazioneServizio{
  static final int QUANTE_AUTO = 10;
  public static void main(String args[]) {
    Thread autoThreads[] = new Thread[QUANTE_AUTO];
    try {
      Pompa laPompa = new Pompa();
      for (int i=0; i< QUANTE_AUTO; i++) {
        Auto auto = new Auto(i, laPompa);
        (autoThreads[i] = new Thread(auto)).start();
      }
      for(int i=0; i< QUANTE_AUTO; i++) {
        autoThreads[i].join();
      }
    } catch(InterruptedException e) {}
    System.out.println("Tempo medio rifornimento = "+
                       Auto.calcMedia());
  }
}
```