Azzolini Riccardo

# Progettazione di programmi concorrenti

## 1 Metodologia di progettazione

I problemi concorrenti e distribuiti sono spesso abbastanza complessi da capire a gestire. Serve allora un approccio sistematico per descrivere e ragionare su tali problemi, che faciliti la progettazione e la realizzazione di una soluzione.

La progettazione riguarda la creazione di collezioni di oggetti che si coordinano tra loro per risolvere un problema. Il problema può essere scomposto in oggetti di due tipi:

- gli **oggetti attivi**, che rappresentano il comportamento dei thread (ovvero conterranno il codice che verrà eseguito da ciascun thread);
- gli **oggetti passivi**, che:
  - reagiscono alle richieste degli oggetti attivi (e non hanno, invece, un comportamento proprio: in assenza di richieste, rimangono completamente inattivi);
  - contengono elementi di controllo degli oggetti attivi (semafori, metodi synchronized, ecc.), per essere thread-safe, coordinare e sincronizzare thread, ecc.

La progettazione basata su questa suddivisione comporta i seguenti passi:

1. scrivere una breve descrizione, tipicamente testuale e informale, del problema da risolvere;
2. descrivere il sistema con UML, identificando gli oggetti attivi, quelli passivi e le loro interazioni;
3. implementare gli oggetti attivi come thread Java;
4. implementare gli oggetti passivi come monitor;
5. scrivere un oggetti di controllo (il "main") che crea le istanze di tutti gli altri oggetti.

## 2 Esempio di specifiche testuali

Come esempio, si considera il problema del produttore-consumatore, le cui specifiche (già viste in precedenza) sono le seguenti:

- Il produttore crea dati consistenti in numeri interi (in questo esempio di problema; in generale, può produrre dati di un tipo qualsiasi), e li memorizza in un buffer di capienza limitata.
- Il consumatore preleva dal buffer i numeri interi, e (in questo esempio) li visualizza sul dispositivo di output.
- Quando il consumatore preleva un dato, questo viene cancellato dal buffer, creando un posto vuoto per un ulteriore dato.
- Produttore e consumatore operano senza soluzione di continuità, cioè eseguono cicli di produzione e consumo potenzialmente infiniti.
- Il buffer la capacità di memorizzare $N$ elementi.
- Se il produttore produce un dato e il buffer è pieno, non può depositarlo immediatamente, ma deve aspettare che nel buffer ci sia almeno uno spazio libero (che si creerà quando il consumatore consumerà 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 (che dovrà essere depositato dal produttore).

## 3 Modellazione delle specifiche con UML

Le descrizioni testuali possono essere male interpretate. Perciò, è opportuno creare dei modelli UML, che risultano più precisi, più facili da leggere e comprendere, e più facili da analizzare (per verificare che le specifiche siano complete, consistenti, non ridondanti, ecc.).

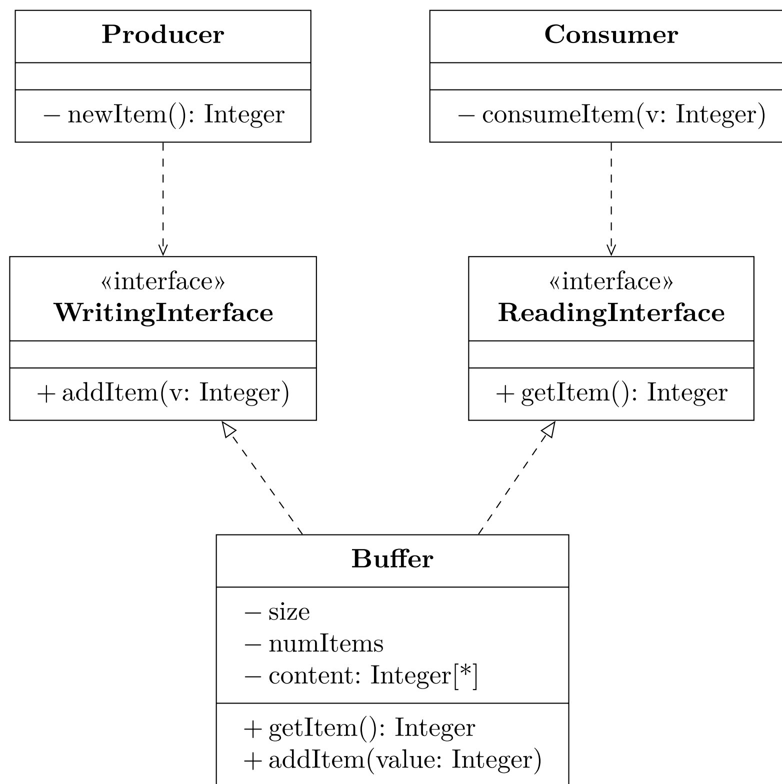
- Per il modello strutturale, che rappresenta gli elementi che compongono il problema, si usa un class diagram.[1] Indicativamente, nel testo della descrizione del problema:
  - i sostantivi rappresentano gli elementi del problema (classi) e il loro contenuto informativo (attributi);
  - i verbi rappresentano le azioni e/o i metodi.

[1] In alternativa, si potrebbe usare un component diagram.

- Il comportamento degli oggetti attivi è modellabile adeguatamente con dei sequence diagram.
- Il comportamento degli oggetti passivi può essere modellato con degli state diagram.

### 3.1 Esempio di class diagram

Nell'esempio del produttore-consumatore, alcuni dei sostantivi presenti nella descrizione sono "produttore", "dati", "buffer", "capienza", "consumatore", mentre alcuni dei verbi sono "memorizza", "preleva", "cancellato". Si potrebbe allora realizzare il seguente class diagram:

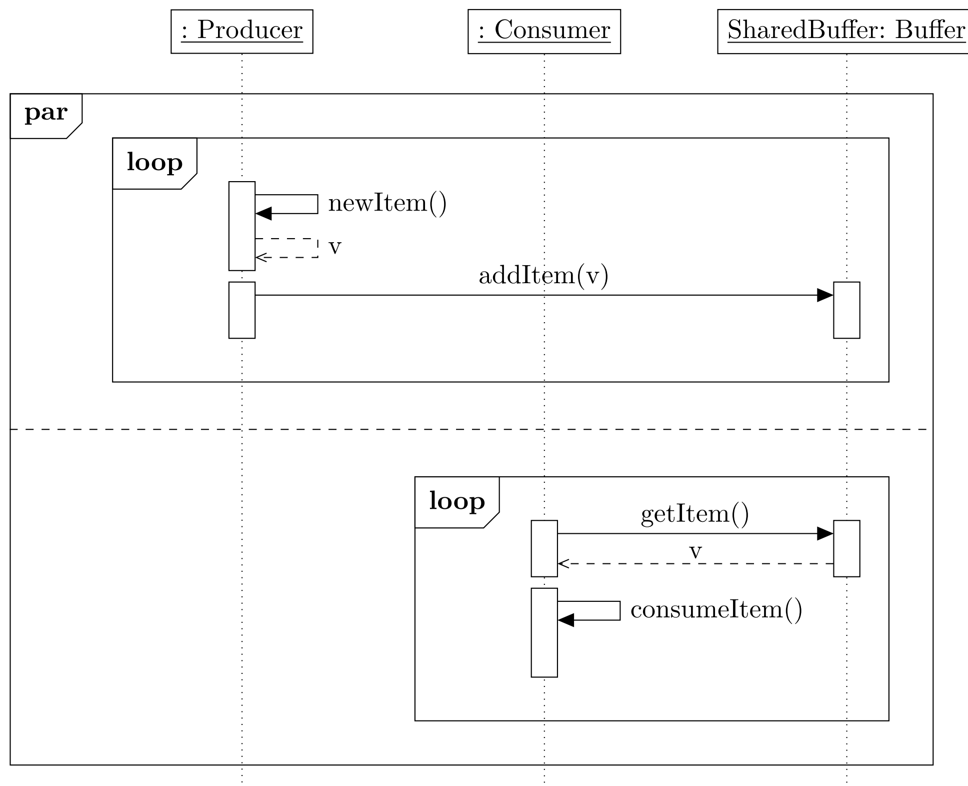



- Il `Buffer` ha attributi corrispondenti alla sua capienza (`size`), al numero di elementi contenuti (`numItems`), e ai contenuti veri e propri (`content`, che è un array di `Integer`), e supporta le operazioni di prelievo (`getItem`) e memorizzazione (`addItem`) di un elemento.
- Le interfacce `ReadingInterface` e `WritingInterface` "evidenziano" le operazioni supportate da `Buffer`; separare queste interfacce permette di mostrare quali siano le operazioni utilizzate dai diversi oggetti attivi.

- Gli oggetti attivi sono `Producer` e `Consumer`; essi utilizzano, rispettivamente, l'interfaccia di scrittura e quella di lettura.
- Internamente, il `Consumer` ha un'operazione `consumeItem` (privata, perché non "interessa" agli altri oggetti), che rappresenta il consumo dell'elemento letto dal buffer (in questo esempio, ciò corrisponde alla visualizzazione sul dispositivo di output). Analogamente, il `Producer` ha un'operazione privata `newItem`, che produce "in qualche modo" (non di interesse degli altri oggetti) un intero da inserire nel buffer.

### 3.2 Esempio di sequence diagram

Sempre in riferimento al produttore-consumatore, il seguente sequence diagram modella le interazioni tra gli oggetti attivi (istanze di) `Producer` e `Consumer` e l'oggetto passivo `SharedBuffer` (istanza della classe `Buffer`):




- Il combined fragment *par* indica che le sue due sezioni sono svolte in parallelo. Infatti, la prima descrive il comportamento del produttore, mentre la seconda riguarda il consumatore, e i due funzionano appunto in parallelo.

- Il produttore esegue un ciclo potenzialmente infinito (indicato da un combined fragment *loop* senza condizioni) nel quale “chiede a sé stesso” di produrre un nuovo elemento, “riceve da sé stesso” tale elemento (indicato con “v”), e richiede al buffer di aggiungerlo.
- Analogamente, il consumatore, in un ciclo potenzialmente infinito, chiede al buffer di leggere un valore, lo riceve, e lo consuma.

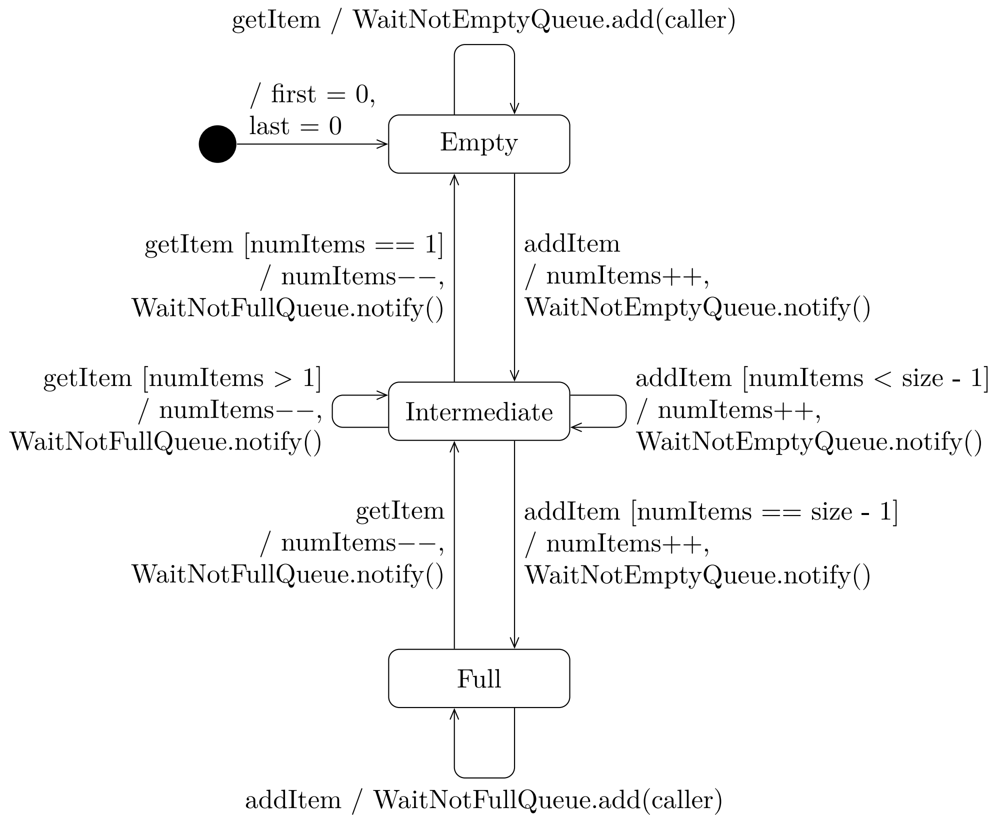
### 3.3 Esempio di state diagram

Una versione semplificata del comportamento del buffer è rappresentata da questo state diagram:

Empty
getItem [numItems == 1] / numItems−−
addItem / numItems++
getItem [numItems > 1] / numItems−−
Intermediate
addItem [numItems < size - 1] / numItems++
getItem / numItems−−
addItem [numItems == size - 1] / numItems++
Full

Qui, non sono considerati i casi in cui il buffer riceve una richiesta di lettura quando è vuoto, oppure una richiesta di scrittura quando è pieno.

Per completare il diagramma, si aggiungono agli stati Empty e Full, rispettivamente, delle transizioni corrispondenti agli eventi (metodi) `getItem` e `addItem`. Come azione di tali transizioni, il thread chiamante (il produttore o il consumatore) viene messo in attesa (nel diagramma, ciò è indicato tramite l’inserimento del thread in un’apposita coda, mentre in Java diventerà una chiamata `wait`). Inoltre, è necessario aggiungere a alle altre transizioni le corrispondenti azioni di sblocco (`notify`) di eventuali thread bloccati.




*Osservazione*: Intuitivamente, si potrebbe pensare di mettere le `nofity` solo sulle transizioni che passano da Empty / Full a Intermediate. Così, però, se ci fossero più produttori o più consumatori in attesa, solo uno di essi verrebbe svegliato, perché poi il buffer si troverebbe nello stato Intermediate, e non eseguirebbe più alcuna `notify`. Invece, è necessario che ogni aggiunta di un elemento possa svegliare un consumatore, e che ogni estrazione di un elemento possa svegliare un produttore, quindi le `notify` vanno messe "ovunque".

# 4 Esempio di implementazione

Nell'esempio del produttore-consumatore, i due oggetti attivi, da implementare come thread, sono il produttore e il consumatore:

```
public class Producer extends Thread {
    private Buffer sharedBuffer;

    public Producer(Buffer sharedBuffer) {
        this.sharedBuffer = sharedBuffer;
```

```java
    }

    private int produceItem() {
        // Here v is produced
        return v;
    }

    public void run() {
        while (true) {
            int v = produceItem();
            try {
                sharedBuffer.addItem(v);
            } catch (InterruptedException e) { break; }
        }
    }
}

public class Consumer extends Thread {
    private Buffer sharedBuffer;

    public Consumer(Buffer sharedBuffer) {
        this.sharedBuffer = sharedBuffer;
    }

    private void consumeItem(int v) {
        System.out.println(v);
    }

    public void run() {
        while (true) {
            int v;
            try {
                v = sharedBuffer.getItem();
            } catch (InterruptedException e) { break; }
            consumeItem(v);
        }
    }
}
```

*Osservazione*: I corpi dei metodi `run` di tali oggetti sono traduzioni pressoché dirette delle parti corrispondenti del sequence diagram (se non per la gestione delle eccezioni).

Si implementa poi l'oggetto passivo buffer, sotto forma di monitor (usando metodi synchronized per evitare le race condition, e `wait` / `notify` per bloccare e sbloccare i thread

quando necessario), in base alle indicazioni contenute nello state diagram:

```
public class Buffer {
    private final int size;
    private int numItems = 0;
    private int[] content;
    private int first = 0;
    private int last = 0;

    public Buffer(int size) {
        this.size = size;
        this.content = new int[size];
    }

    public synchronized int getItem() throws InterruptedException {
        while (numItems == 0) {
            wait();
        }
        numItems--;
        int v = content[first];
        first = (first + 1) % size;
        notify();
        return v;
    }

    public synchronized void addItem(int v) throws InterruptedException {
        while (numItems == size) {
            wait();
        }
        content[last] = v;
        last = (last + 1) % size;
        numItems++;
        notify();
    }
}
```

Infine, si implementa l'oggetto di controllo, cioè (in questo caso) il main, che:

1. crea il buffer condiviso;
2. crea i produttori e i consumatori (ad esempio, due per tipo), passando a tutti un riferimento al buffer condiviso;
3. lancia i thread produttori e consumatori.

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

# 5 Altro esempio

Si vuole modellare una stazione di servizio, per calcolare il tempo medio necessario a un automobilista per ottenere il carburante.

- Il tempo viene calcolato a partire da quando il conducente comincia l'attesa a una pompa di distribuzione, e termina quando l'automobile lascia la stazione di servizio.
- La simulazione inizia con 50 vetture che arrivano alla stazione di servizio in modo casuale, con ritardi che vanno da 0 a 30 UT (Unità di Tempo[2]).
- Ciascun conducente si comporta come segue:

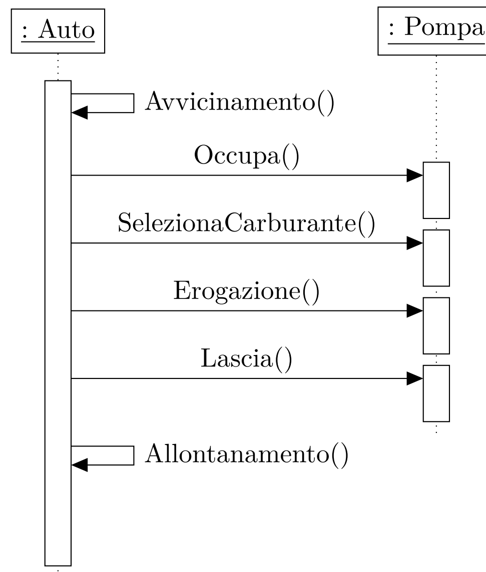
  1. Verifica se la pompa sia libera.
     - Se non è libera, aspetta.
     - Se invece è libera, avvicina la macchina fino alla pompa, spegne il motore, scende, seleziona il carburante desiderato e inserisce l'erogatore. Queste operazioni richiedono 1 UT.
  2. Riempie il serbatoio. Ciò richiede un tempo compreso tra 1 e 3 UT, a seconda di quanto sia il carburante prelevato.
  3. Ripone l'erogatore, sale in macchina e si allontana dalla pompa, lasciandola libera. Quest'operazione richiede 1 UT.

  *Osservazione*: Di fatto, per la simulazione, sapere esattamente cosa faccia il conducente non importa. L'unico dato rilevante è che, quando un conducente trova una pompa libera, la terrà poi occupata per un tempo complessivo compreso tra 3 e 5 UT.

[2] Quanto duri effettivamente un'unità di tempo è irrilevante.

## 5.1 Sequence diagram: auto

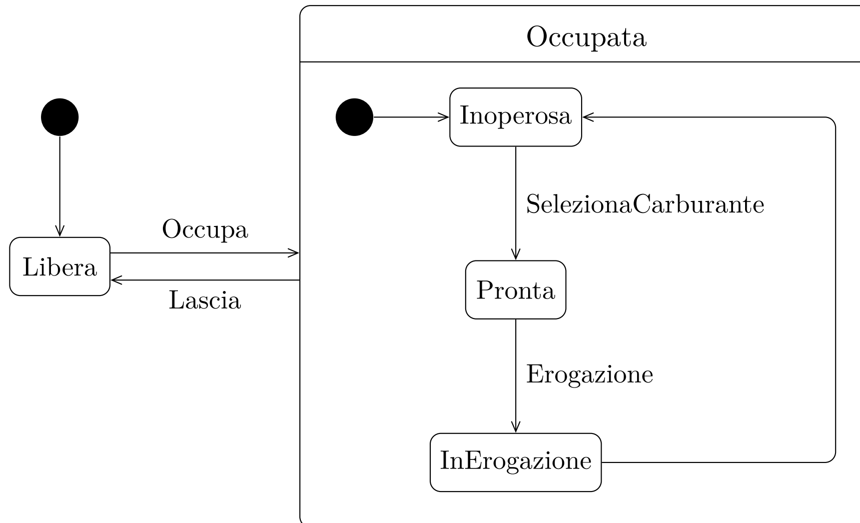
Ciascun'auto è un oggetto attivo, il cui comportamento viene quindi descritto dal seguente sequence diagram:




Le operazioni di avvicinamento e allontanamento sono svolte autonomamente dall'automobile (che le "richiede a sé stessa"), mentre le altre richiedono un'interazione con la pompa.

## 5.2 State diagram: pompa

Il comportamento della pompa, in quanto oggetto passivo, è descritto mediante uno state diagram:




Qui, per semplicità, è indicato che la pompa può essere lasciata (cioè si può uscire dallo stato “Occupata”) in qualsiasi momento. Nella realtà, ciò non avverrà durante il rifornimento.

## 5.3 Implementazione

Per prima cosa, si definisce la classe corrispondente all’oggetto attivo auto, che deve realizzare un thread (in questo caso, si sceglie di fare ciò mediante l’implementazione dell’interfaccia `Runnable`):

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

public class Auto implements Runnable {
    private static final int UNITA_TEMPO = 100;
    private static final int MAX_ATTESA = 30 * UNITA_TEMPO;
    private static int tempoTotale = 0;
    private static int numAuto = 0;

    private int numCliente;
    private Pompa pompaUsata;

    public Auto(int numCliente, Pompa pompaUsata) {
        this.numCliente = numCliente;
        this.pompaUsata = pompaUsata;
    }
```

```java
public static double tempoMedio() {
    return tempoTotale / numAuto;
}

private int tempoAttesa() {
    return ThreadLocalRandom.current().nextInt(1, MAX_ATTESA);
}

private int tempoErogazione() {
    // Per semplicità, il tempo "di erogazione" comprende anche
    // quello richiesto dalle altre operazioni svolte dal conducente.
    return ThreadLocalRandom.current().nextInt(
        3 * UNITA_TEMPO,
        5 * UNITA_TEMPO
    );
}

public void run() {
    try {
        Thread.sleep(tempoAttesa()); // Ritardo di arrivo
    } catch (InterruptedException e) { return; }
    System.out.println("Auto " + numCliente + " arriva alla pompa");
    long tempoInizio = new Date().getTime();
    try {
        pompaUsata.occupa();
    } catch (InterruptedException e) { return; }

    System.out.println("Auto " + numCliente + " in rifornimento");
    try {
        pompaUsata.eroga(tempoErogazione());
    } catch (InterruptedException e) {
        // Anche in caso di interrupt,
        // prima di terminare bisogna liberare la pompa,
        // altrimenti rimarrebbe occupata per sempre.
    }

    System.out.println("Auto " + numCliente + " lascia la pompa");
    try {
        pompaUsata.lascia();
    } catch (InterruptedException e) {}
    long tempoFine = new Date().getTime();
    long tempoImpiegato = tempoFine - tempoInizio;
    System.out.println(
        "Tempo auto " + numCliente + " = " + tempoImpiegato
```

```
        );
        synchronized (Auto.class) {
            tempoTotale += tempoImpiegato;
            numAuto++;
        }
    }
}
```

Questa classe contiene alcune informazioni `static` (a livello di classe, condivise da tutte le istanze), usate per calcolare il tempo impiegato in media da un'automobile (mediante il metodo `tempoMedio()`, anch'esso statico). Per evitare race condition, tali informazioni devono essere aggiornate in un blocco synchronized *sulla classe* `Auto` (dato che si tratta, appunto, di dati appartenenti alla classe, e non alla singola istanza).

Si implementa poi l'oggetto passivo pompa, che gestisce tutti gli aspetti della sincronizzazione:

```
public class Pompa {
    private enum Stato {
        LIBERA, OCCUPATA
    }

    private Stato stato = Stato.LIBERA;

    public synchronized void occupa() throws InterruptedException {
        while (stato != Stato.LIBERA) {
            wait();
        }
        stato = Stato.OCCUPATA;
    }

    public synchronized void eroga(int attesa)
            throws InterruptedException {
        while (stato != Stato.OCCUPATA) {
            wait();
        }
        Thread.sleep(attesa); // Tempo necessario all'erogazione
    }

    public synchronized void lascia() throws InterruptedException {
        while (stato != Stato.OCCUPATA) {
            wait();
        }
        stato = Stato.LIBERA;
        notifyAll();
```

```
    }
}
```

In quest'implementazione, i metodi `eroga` e `lascia` controllano / aspettano che la pompa sia effettivamente occupata prima di eseguire. In realtà, per via del comportamento delle automobili, che prima occupano la pompa, poi erogano, e infine lasciano, tale condizione sarà sempre verificata, quindi non sarà mai necessario aspettare.

Infine, il main (l'oggetto di controllo) crea gli oggetti pompa e auto, avvia i thread corrispondenti alle auto, aspetta che questi terminino, e visualizza il tempo medio di rifornimento.

```
public class StazioneServizio {
    private static final int QUANTE_AUTO = 50;

    public static void main(String[] args) throws InterruptedException {
        Pompa pompa = new Pompa();
        Thread[] threadAuto = new Thread[QUANTE_AUTO];
        for (int i = 0; i < QUANTE_AUTO; i++) {
            Auto auto = new Auto(i, pompa);
            threadAuto[i] = new Thread(auto);
            threadAuto[i].start();
        }

        for (Thread t : threadAuto) {
            t.join();
        }
        System.out.println(
            "Tempo medio rifornimento = " + Auto.tempoMedio()
        );
    }
}
```