Teoria współbieżności

Sprawozdanie

Laboratorium 7

Joanna Bryk

1. Wstęp

   Celem laboratorium było zapoznanie się z jednym z wzorców projektowych dla programowania współbieżnego jakim jest wzorzec Active Object. Ideą tego wzorca jest oddzielenie procesu wywołania metody od procesu jej wykonania. Wywołanie odbywa się w wątku klienta, wykonanie w wątku pracownika. Nauka tego wzorca polegała na zaimplementowaniu rozwiązania problemu producent – konsument z ograniczonym buforem z jego wykorzystaniem.

2. Implementacja

   Implementację zaplanowałam zgodnie ze wskazówkami:

- Pracownik powinien implementować samą kolejkę (bufor) oraz dodatkowe metody (*czyPusty* etc.), które pomogą w implementacji strażników. W klasie tej powinna być tylko funkcjonalność, ale nie logika związana z synchronizacją.

  Właśnie w ten sposób została zaimplementowana klasa *Servant*. Bufor jest tablicą liczb całkowitych. Wartość zero traktuję jako pustą komórkę. W klasie tej znajdują się funkcję takie jak *add(), remove(), isEmpty(), isFull()*.

- Dla każdej metody aktywnego obiektu powinna być specjalizacja klasy *MethodRequest*. W tej klasie m.in. zaimplementowana jest metoda *guard(),* która oblicza spełnienie warunków synchronizacji (korzystając z metod dostarczonych przez *Pracownika*).

  Stworzyłam interfejs *IMethodRequest* oraz dwie klasy go implementujące - *MethodRequestAdd* (do dodawania elementów) i *MethodRequestRemove* (do usuwania). Obie te metody implementują metody *call()* i *guard()*. W moim programie *Pracownik* występuje pod nazwą *Servant*.

- Proxy wykonuje się w wątku klienta, który wywołuje metodę. Tworzenie *Method request* i kolejkowanie jej w *Activation queue* odbywa się również w wątku klienta. *Servant* i *Scheduler* wykonują się w osobnym wątku.

Dodatkowo zgodnie z założeniami wzorca utworzone zostały klasy *Future* i *Proxy*. Pierwsza z nich implementuje metody *changeVal()*, *getVal()*. Jest to zmienna, do której zapisywany będzie wynik wykonywania metody. Zwracana jest ona wątkowi wywołującemu.

Z klasy tej korzystam w implementacji metod klasy *Proxy*. *Proxy* jest dostępnym publicznie interfejsem, który służy do wywoływania metod przez klientów.

Ważnym elementem programu jest także *ActivationQueue* – bufor dla obiektów *MethodRequest* (wywołań metod), które utworzone są w *Proxy* a pobierane przez wątek *Scheduler*.

```
public class Proxy {
    private Scheduler scheduler;
    private Servant buffer; //servant

    public Proxy(int size) {
        this.scheduler = new Scheduler();
        this.buffer = new Servant(size);
        this.scheduler.start();
    }

    public Future add(int val) {
        Future future = new Future();
        IMethodRequest request_add = new MethodRequestAdd(val, future,
this.buffer);
        System.out.println("REQUEST ADD " + val);
        scheduler.enqueue(request_add);
        return future;
    }

    public Future get() {
        Future future = new Future();
        IMethodRequest request_remove = new MethodRequestRemove(future,
this.buffer);
        System.out.println("REQUEST GET ");
        scheduler.enqueue(request_remove);
        return future;
    }

    public Scheduler getScheduler(){
        return scheduler;
    }

}
```

```
public class Servant {

    private int size;
    private final int[] buffer;

    public Servant(int size) {
        this.size = size;
        this.buffer = new int[size];
```

```
        for (int i = 0; i < this.size; i++)
            this.buffer[i] = 0; //traktuje zero jako pusta komórka
    }

    public void add(int val) {
        int index = findEmpty();
        this.buffer[index] = val;
    }

    public int remove() {
        int index = findElement();
        int value = buffer[index];
        buffer[index] = 0;
        return value;
    }

    public boolean isEmpty() {
        for (int i = 0; i < this.size; i++) {
            if (this.buffer[i] != 0)
                return false;
        }
        return true;
    }

    public boolean isFull() {
        for (int i = 0; i < this.size; i++) {
            if (this.buffer[i] == 0)
                return false;
        }
        return true;
    }

    public int findEmpty() {
        for (int i = 0; i < this.size; i++) {
            if (buffer[i] == 0) {
                return i;
            }
        }
        throw new IllegalStateException("error");
    }

    public int findElement() {
        for (int i = 0; i < this.size; i++) {
            if (buffer[i] != 0) {
                return i;
            }
        }
        throw new IllegalStateException("error");
    }
}
```

```
public class Scheduler extends Thread {
    private final LinkedBlockingQueue<IMethodRequest> activationQueue;

    public Scheduler() {
        activationQueue = new LinkedBlockingQueue<>();
    }

    public void enqueue(IMethodRequest r) {
```

```
        activationQueue.add(r);
    }

    @Override
    public void run() {
        System.out.println("Hello scheduler here!");
        while (true) {
            IMethodRequest r;
            try {
                r = activationQueue.take();
                if (r.guard()) {
                    r.call();
                    activationQueue.remove(r);
                } else {
                    activationQueue.put(r);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public interface IMethodRequest {
    void call();
    boolean guard();
}
```

```
public class MethodRequestRemove implements IMethodRequest {
    private Future future;
    private Servant servant;


    public MethodRequestRemove(Future future, Servant servant) {
        this.future = future;
        this.servant = servant;
    }

    @Override
    public void call() {
        int val = this.servant.remove();
        future.changeVal(val);
    }

    @Override
    public boolean guard() {
        return !this.servant.isEmpty();
    }
}
```

```
public class MethodRequestAdd implements IMethodRequest {
    private Future future;
    private Servant servant;
    private int value;

    public MethodRequestAdd(int val, Future future, Servant servant) {
        this.value = val;
        this.future = future;
```

```
        this.servant = servant;
    }

    @Override
    public void call() {
        this.servant.add(this.value);
        this.future.changeVal(this.value);
    }

    @Override
    public boolean guard() {
        return !this.servant.isFull();
    }
}
```

```
public class Future {

    private int val = 0;
    Semaphore sem = new Semaphore(0);

    public void changeVal(int val) {
        this.val = val;
        sem.release();
    }

    public int getVal() {
        try {
            sem.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return this.val;
    }
}
```

```
class Consumer extends Thread {
    private Proxy proxy;
    private int ID;

    public Consumer(int ID, Proxy proxy) {
        this.ID = ID;
        this.proxy = proxy;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            Future get = this.proxy.get();
            System.out.println("consumer " + ID + " got value " +
get.getVal());
        }
        try {
            this.proxy.getScheduler().join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class Producer extends Thread {
    private Proxy proxy;
    private int ID;

    public Producer(int ID, Proxy proxy) {
        this.ID = ID;
        this.proxy = proxy;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            Future adding = this.proxy.add(ID);
            System.out.println("Producer added val " +  adding.getVal() + "
" + i);
        }
        try {
            this.proxy.getScheduler().join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3. Podsumowanie

   Zaimplementowany przeze mnie wzorzec pozwala na zrealizowane założonego zadania – oddziela proces wywoływania metody od jej wykonania. Jest to ciekawe i użyteczne rozwiązanie, ponieważ z perspektywy klienta widoczny jest jedynie interfejs Proxy, cała logika, mechanizmy współbieżne są wykonywane w oddzielnych częściach programu, niejako ukrytych przed klientem. Odseparowanie to sprawia, że program zdaje się działać bardziej intuicyjnie. Dodatkowo kolejność wykonywania metod może być inna niż kolejność wywoływania.

4. Bibliografia

- https://home.agh.edu.pl/~funika/tw/lab7/
- https://pl.wikipedia.org/wiki/Active_object
- https://www.dre.vanderbilt.edu/~schmidt/PDF/Active-Objects.pdf
- Android and Java Concurrency: The Active Object Pattern (Part 1), Douglas C. Schmidt, Vanderbilt University, Nashville, Tennessee, USA https://www.youtube.com/watch?v=Cd8t2u5Qmvc