SPRAWOZDANIE VII

TEORIA WSPÓŁBIEŻNOŚCI

Wzorzec projektowy

Active Object w problemie producentów i konsumentów

DAWID BIAŁKA

DATA LABORATORIUM 17.11.2020

DATA ODDANIA 24.11.2020

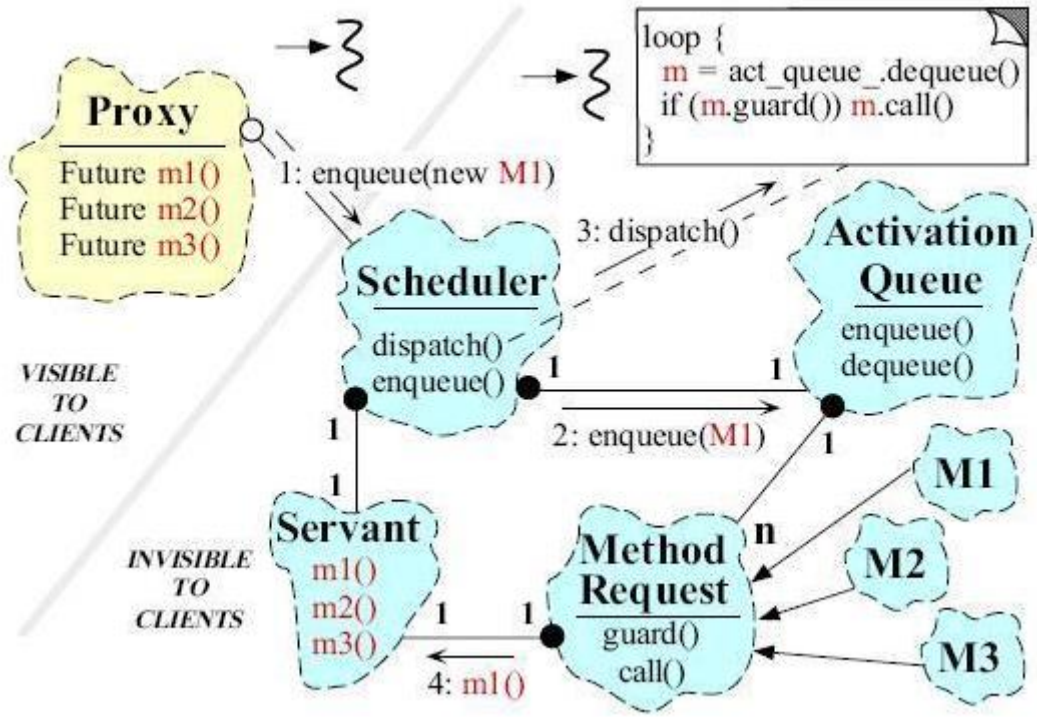
Zadania:

Zaimplementować bufor jako aktywny obiekt (Producenci-Konsumenci)

Wskazówki:

1. 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.
2. 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).
3. 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 (oba w tym samym) wątku.

Koncepcja



Ogólna idea wzorca: mamy bufor implementowany przez klasę Servant, chcemy uzyskać współbieżny dostęp do tego bufora i wykonywanych na nim operacji przez klientów (Producenci i Konsumenci). Tworzymy klasę Proxy która ma dokładnie te same metody co Servant ale zapewnia dostęp współbieżny do bufora Servant'a. Klienci korzystają z Proxy wykonując te same metody, co wykonywaliby na Servant'cie. Proxy tworzą żądanie wykonania danej metody (dodania lub zdjęcia elementu z bufora) i wysyłają ją do Schedulera (osobny wątek), który umieszcza je w swojej kolejce. Scheduler cały czas pobiera żądania metod z swojej kolejki i próbuje je wykonać (sprawdza, czy np. bufor nie jest pusty, aby móc pobrać element z niego.) Jeśli nie udaje się jej wykonać to umieszcza ją z powrotem w kolejce.

Tworzymy następujące klasy:

- **Main** – klasa główna tworząca Proxy, Schedulera, Servanta i wszystkie wątki Producentów i Konsumentów
- **IMethodRequest** – interfejs żądania metody, metody: call() i guard()
- **AddRequest** – implementacja interfejsu IMethodRequest odpowiadająca u Servanta metodzie add()
- **RemoveRequest** - implementacja interfejsu IMethodRequest odpowiadająca u Servanta metodzie remove()
- **Servant** – zwykła klasa z buforem oraz metodami add() i remove() z bufora oraz metody sprawdzające możliwość wykonania danej operacji
- **Future** – klasa składającą „obietnicę" wykonania danej metody. Z obiektu Future pobieramy wynik, gdy Servant zakończy wykonywanie danej metody.
- **Proxy** – klasa odpowiadająca klasie Servant, która ma te same metody dostępu do bufora co Servant (Servant dodatkowo ma metody do sprawdzania możliwości wykonania danej operacji). Proxy tworzy żądania metod i wysyła je do Schedulera jednocześnie zwracając obiekt Future.
- **Scheduler –** klasa wykonująca się w osobnym wątku i odpowiada za przechowywanie w kolejce żądań metod, pobierania ich i próbie ich wykonania.
- **Producer –** klasa wykonująca się w osobnym wątku, klient, który chce dodać element do bufora
- **Consumer -** klasa wykonująca się w osobnym wątku, klient, który chce pobrać element z bufora

Implementacja i wyniki

```
public class Main {

    public static void main(String[] args) {
        int numOfConsumers = 7;
        int numoOfProducers = 4;

        Scheduler scheduler = new Scheduler();
        scheduler.start();
        Servant servant = new Servant(10);
        Proxy proxy = new Proxy(scheduler, servant);

        ArrayList<Producer> producers = new ArrayList<>();
        ArrayList<Consumer> consumers = new ArrayList<>();

        for (int i = 0; i < numoOfProducers; i++) {
            producers.add(new Producer(i, proxy));
        }

        for (int i = 0; i < numOfConsumers; i++) {
            consumers.add(new Consumer(i, proxy));
        }

        for (int i = 0; i < numoOfProducers; i++) {
            producers.get(i).start();
        }

        for (int i = 0; i < numOfConsumers; i++) {
            consumers.get(i).start();
        }

        try {
            for (int i = 0; i < numoOfProducers; i++) {
                producers.get(i).join();
            }

            for (int i = 0; i < numOfConsumers; i++) {
                consumers.get(i).join();
            }

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

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

```
public class AddRequest implements IMethodRequest {
    private Future future;
    private Servant servant;
    private Object object;

    public AddRequest(Future future, Servant servant, Object object) {
        this.future = future;
        this.servant = servant;
        this.object = object;
    }

    public void call() {
        servant.add(object);
        future.setResult(object);
    }
    public boolean guard() {
        return !servant.isFull();
    }
}
```

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

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

    public void call() {
        future.setResult(servant.remove());
    }
    public boolean guard() {
        return !servant.isEmpty();
    }
}
```

```java
public class Servant {
    private int bufSize;
    private Queue<Object> buffer;

    public Servant(int bufSize){
        this.bufSize = bufSize;
        this.buffer = new LinkedList<Object>();
    }
    public void add(Object object) {
        if(!isFull()){
            this.buffer.add(object);
        }
    }

    public Object remove() {
        if(isEmpty()){
            return null;
        }
        else {
            return buffer.remove();
        }
    }
    public boolean isFull() {
        return buffer.size() == bufSize;
    }
    public boolean isEmpty() {
        return buffer.isEmpty();
    }

}
```

```java
public class Future {
    private Object object;

    public boolean isAvaiable() {
        return object!=null;
    }

    public void setResult(Object object) {
        this.object = object;
    }

    public Object getResult() {
        return object;
    }
}
```

```
public class Proxy {
    Scheduler scheduler;
    Servant servant;

    public Proxy(Scheduler scheduler, Servant servant){
        this.scheduler = scheduler;
        this.servant = servant;
    }
    public Future add(Object object) {
        Future future = new Future();
        IMethodRequest addRequest = new AddRequest(future, servant, object);
        scheduler.enqueue(addRequest);
        return future;
    }

    public Future remove() {
        Future future = new Future();
        IMethodRequest removeRequest = new RemoveRequest(future, servant);
        scheduler.enqueue(removeRequest);
        return future;
    }
}
```

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

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

    public void enqueue(IMethodRequest request) {
        activationQueue.add(request);
    }

    public void run() {
        while (true) {
            IMethodRequest methodRequest = activationQueue.poll();
            if (methodRequest != null) {
                if (methodRequest.guard()) {
                    methodRequest.call();
                } else {
                    activationQueue.add(methodRequest);
                }
            }
        }
    }

}
```

```
public class Producer extends Thread{
    private int ID;
    private Proxy proxy;
    private static Random rand = new Random();

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

    public void run(){
        while(true){
            int tmp = rand.nextInt(150);
            Future added = proxy.add(tmp);

            while(!added.isAvaiable()){
                System.out.println("Worker " + ID + " is waiting.");
                try {
                    Thread.sleep(200);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            System.out.println("Worker " + ID
                    + " added: " + added.getResult());
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

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

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

    public void run(){
        while(true){
            Future consumed = proxy.remove();
            while(!consumed.isAvaiable()){
                System.out.println("Consument " + ID + " is waiting.");
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Consument " + ID
                    + " consumed: " + consumed.getResult());
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }


}
```

Fragment wykonywania się programu.

```
Worker 0 created 141
Worker 1 created 31
Worker 0 added: 141
Worker 1 added: 31
Worker 2 created 93
Worker 2 is waiting.
Worker 3 created 72
Worker 3 is waiting.
Consument 4 is waiting.
Consument 6 is waiting.
Consument 1 is waiting.
Consument 0 is waiting.
Consument 5 is waiting.
Consument 2 is waiting.
Consument 3 is waiting.
Worker 0 created 0
Worker 0 added: 0
Worker 1 created 120
Worker 1 is waiting.
Worker 2 added: 93
Worker 3 added: 72
Worker 1 added: 120
Worker 0 created 17
Worker 0 added: 17
Worker 2 created 47
Worker 2 added: 47
Worker 3 created 20
Worker 3 added: 20
Consument 4 consumed: 93
Consument 6 consumed: 72
Consument 0 consumed: 17
Consument 1 consumed: 0
Consument 3 consumed: 141
Consument 2 consumed: 31
Consument 5 consumed: 120
Worker 1 created 38
Worker 1 added: 38
Worker 0 created 143
Worker 0 added: 143
Worker 3 created 99
```

Wnioski

Analizując otrzymane wyniki z wykonywania się programu widzimy, że działa on poprawnie oraz również wzorzec jest prawidłowy. Na początku producenci tworzą element i wysyłają żądanie, że chcą go dodać do bufora. Później czekają na potwierdzenie umieszczenia tego elementu w buforze. Następnie konsumenci wysyłają żądanie, że chcą pobrać element z bufora i czekają, aż go dostaną. Po tym producenci znowu tworzą elementy i próbują je dodać, a po nich konsumenci uzyskują element z bufora i informują, że go konsumują i program w podobny sposób wykonuje się dalej.

Wzorzec Active Object jest jednym z popularniejszych wzorców związanych z tematyką współbieżności, który pozwala na współbieżny dostęp do danego zasobu i jednocześnie ukrywa całą logikę współbieżności przed klientem.

Bibliografia

Z. Weiss, T. Gruźlewski, Programowanie współbieżne i rozproszone. WNT, Warszawa 1993.

http://home.agh.edu.pl/~funika/tw/lab7/

https://pl.wikipedia.org/wiki/Active_object

https://www.dre.vanderbilt.edu/~schmidt/PDF/Active-Objects.pdf