# Laboratorium 7 - Active Object

Piotr Karamon

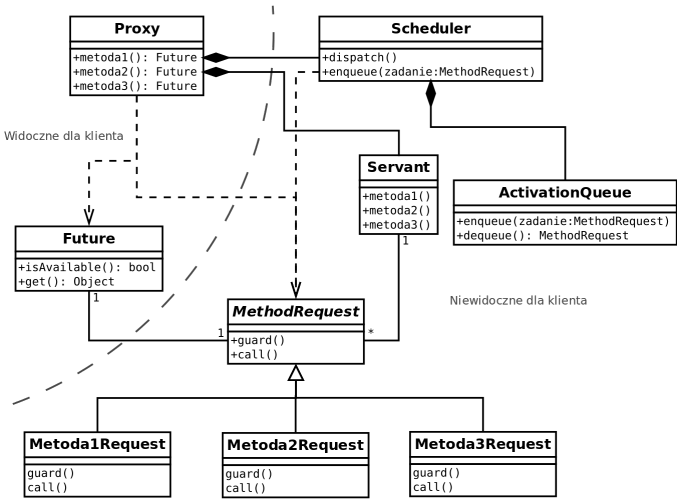
25.11.2024r.

## Treści zadań

Zaimplementować bufor jako aktywny obiekt (Producenci-Konsumenci) Wskazówki:

- 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.
- 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).
- Proxy wykonuje się w wątku klienta, który wywołuje metodę. Tworzenie `MethodRequest` i kolejkowanie jej w Activation queue odbywa się również w wątku klienta. `Servant` i `Scheduler` wykonują się w osobnym (oba w tym samym) wątku.

## Rozwiązanie




**Rysunek 1:** Schemat wzroca Active Object

Wzorzec Active Object oddziela wykonywanie długotrwałych operacji od głównego wątku, umożliwiając asynchroniczne przetwarzanie zadań. Zadania są umieszczane w kolejce i wykonywane przez osobny wątek

(serwant). Klient otrzymuje wynik operacji za pomocą obiektu Future, który zostaje "ukończony" po zakończeniu zadania. Celem laboratorium było zaimplementowanie bufora, używając właśnie tego wzorca. W tym celu będziemy implementować kolejne klasy z diagramu, lub też czasami używać gotowych klas z biblioteki standardowej.

### Klasa `Buffer` (Pracownik/Servant)

Klasa ta jest odpowiedzialna za logikę bufora, czyli dodawanie i pobieranie elementów z kolejki. Również znajdują się w niej metody umożliwiające sprawdzenie czy dana operacja jest możliwa.

```
class Buffer {
    private final int capacity;
    private final List<Integer> buffer = new LinkedList<>();

    public Buffer(int capacity) {
        this.capacity = capacity;
    }

    public void put(int value) {
        buffer.add(value);
    }

    public int get() {
        return buffer.removeFirst();
    }

    public boolean isEmpty() {
        return buffer.isEmpty();
    }

    public boolean isFull() {
        return buffer.size() == capacity;
    }
}
```

#### `MethodRequest`

`MethodRequest` reprezentuje pojedyncze zadanie do wykonania, przechowując informacje o metodzie i jej parametrach. Jest umieszczany w kolejce i przetwarzany przez pracownika. Poza metodą `call()` mamy również metodę `guard()` która sprawdza czy operacja może być wykonana. Używamy tutaj `CompletableFuture`, aby poinformować klienta o zakończeniu zadania. Użycie gotowej klasy upraszcza implementację oraz dostarcza użytkownikowi mnóstwo wygodnych metod do obsługi wyniku. Klasa `MethodRequest` jest generyczna ze względu na typ zawarty w `CompletableFuture`.

Klasa abstrakcyjna `MethodRequest`:

```
abstract class MethodRequest<T> {
    protected final Buffer buffer;
    protected final CompletableFuture<T> future = new CompletableFuture<>();

    public MethodRequest(Buffer buffer) {
        this.buffer = buffer;
    }

    public abstract void execute();
```

```
    public abstract boolean guard();
}
```

### PutRequest

```
class PutRequest extends MethodRequest<Void> {
    private final int value;

    public PutRequest(Buffer buffer, int value) {
        super(buffer);
        this.value = value;
    }

    @Override
    public void execute() {
        System.out.println("PUT REQUEST " + value);
        buffer.put(value);
        future.complete(null);
    }

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

### GetRequest

```
class GetRequest extends MethodRequest<Integer> {
    public GetRequest(Buffer buffer) {
        super(buffer);
    }

    @Override
    public void execute() {
        System.out.println("GET REQUEST");
        future.complete(buffer.get());
    }

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

### Scheduler

`Scheduler` zajmuje się wrzucaniem obiektów `MethodRequest` do kolejki. Odpowiada on również za pobieranie tych obiektów z kolejki i uruchamianie ich. Zamiast dedykowanej klasy na kolejkę, używamy gotowej klasy `ConcurrentLinkedQueue`. Kod klasy `Scheduler`, jest bardzo prosty. Metoda `run()` jest wykonywana w osobnym wątku od klienta i zarówno pobiera zadania z kolejki, jak i je wykonuje. Gdy zadanie nie może być wykonane, jest ono ponownie dodawane do kolejki.

```
class Scheduler {
    private final Queue<MethodRequest<?>> requests = new ConcurrentLinkedQueue<>();

    public void enqueue(MethodRequest<?> request) {
        requests.add(request);
    }

    public void run() {
        while (true) {
            MethodRequest<?> request = requests.poll();
            if (request != null) {
                if (request.guard()) {
                    request.execute();
                } else {
                    requests.add(request);
                }
            }
        }
    }
}
```

### BufferProxy

`BufferProxy` jest klasą, która jest używana przez klienta. Jej interfejs jest bardzo podobny do interfejsu klasy `Buffer`, ale zamiast wyników bezpośrednich `void` / `int` zwraca obiekty `CompletableFuture`, przez to używanie takiej klasy jest bardzo intuicyjne. Klasa `BufferProxy` uruchamiania wątek scheduler'a, oraz tworzy odpowiednie obiekty `MethodRequest` i przekazuje je do `Scheduler`.

```
class BufferProxy {
    private final Scheduler scheduler = new Scheduler();
    private final Buffer buffer;

    public BufferProxy(int capacity) {
        this.buffer = new Buffer(capacity);

        var thread = new Thread(scheduler::run);
        thread.setDaemon(true);
        thread.start();
    }

    public CompletableFuture<Void> put(int value) {
        var request = new PutRequest(buffer, value);
        scheduler.enqueue(request);
        return request.future;
    }

    public CompletableFuture<Integer> get() {
        var request = new GetRequest(buffer);
        scheduler.enqueue(request);
        return request.future;
    }
}
```

## Przykład użycia

Implementujemy dwie klasy `Producer` i `Consumer`. Wykonują swoje operacje w pętli, ale po każdej operacji **nie** czekają na zakończenie operacji. Po wykoaniu całej pętli, wątki czekają aż wszystkie ich wysłane zadania zostały zakończone.

### Producer

```
class Producer implements Runnable {
    private final int toProduce;
    private final BufferProxy buffer;
    private final int id;

    public Producer(int id, int toProduce, BufferProxy buffer) {
        this.id = id;
        this.toProduce = toProduce;
        this.buffer = buffer;
    }

    @Override
    public void run() {
        List<CompletableFuture<Void>> futures = new ArrayList<>();
        for (int i = 0; i < toProduce; i++) {
            futures.add(buffer.put(i));

            try {
                Thread.sleep(30);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.printf("Producer %d finished sending requests%n", id);
        CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();
        System.out.printf("Producer %d: all actions have been done%n", id);
    }
}
```

### Consumer

```
class Consumer implements Runnable {
    private final int toConsume;
    private final BufferProxy buffer;
    private final int id;

    public Consumer(int id, int toConsume, BufferProxy buffer) {
        this.id = id;
        this.toConsume = toConsume;
        this.buffer = buffer;
    }

    @Override
    public void run() {
        List<CompletableFuture<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < toConsume; i++) {
            futures.add(buffer.get());
```

```
            try {
                Thread.sleep(30);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.printf("Consumer %d finished sending requests%n", id);

        var results = futures.stream()
                .map(CompletableFuture::join)
                .map(String::valueOf)
                .collect(Collectors.joining(", "));
        System.out.printf("Consumer %d: received values: %s%n", id, results);

        System.out.printf("Consumer %d: all actions have been done%n", id);
    }

}
```

### Funkcja `main`

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
    BufferProxy buffer = new BufferProxy(5);

    var producers = IntStream.range(0, 3)
            .mapToObj(i -> new Producer(i, 10, buffer))
            .map(Thread::new)
            .toList();
    var consumers = IntStream.range(0, 2)
            .mapToObj(i -> new Consumer(i, 15, buffer))
            .map(Thread::new)
            .toList();

    try (var executor = Executors.newFixedThreadPool(7)) {
        producers.forEach(executor::submit);
        consumers.forEach(executor::submit);

        executor.shutdown();
        boolean terminated = executor.awaitTermination(1, TimeUnit.MINUTES);
        if (terminated) {
            System.out.println("All threads have finished");
        } else {
            System.out.println("Program has timed out");
        }
    }
}
```

**Wynik programu**

```
PUT REQUEST 0
PUT REQUEST 0
GET REQUEST
PUT REQUEST 0
GET REQUEST
PUT REQUEST 1
PUT REQUEST 1
PUT REQUEST 1
GET REQUEST
GET REQUEST
PUT REQUEST 2
GET REQUEST
GET REQUEST
PUT REQUEST 2
PUT REQUEST 2
GET REQUEST
PUT REQUEST 3
PUT REQUEST 3
GET REQUEST
PUT REQUEST 3
PUT REQUEST 4
GET REQUEST
PUT REQUEST 4
GET REQUEST
PUT REQUEST 4
GET REQUEST
PUT REQUEST 5
GET REQUEST
PUT REQUEST 5
GET REQUEST
GET REQUEST
PUT REQUEST 6
PUT REQUEST 5
GET REQUEST
PUT REQUEST 7
GET REQUEST
PUT REQUEST 7
GET REQUEST
PUT REQUEST 6
GET REQUEST
PUT REQUEST 8
```

```
GET REQUEST
PUT REQUEST 9
GET REQUEST
PUT REQUEST 9
GET REQUEST
Producer 1 finished sending requests
Producer 2 finished sending requests
Producer 0 finished sending requests
Producer 0: all actions have been done
PUT REQUEST 6
GET REQUEST
PUT REQUEST 8
GET REQUEST
PUT REQUEST 8
GET REQUEST
PUT REQUEST 7
Producer 2: all actions have been done
GET REQUEST
PUT REQUEST 9
GET REQUEST
Producer 1: all actions have been done
GET REQUEST
GET REQUEST
GET REQUEST
GET REQUEST
Consumer 0 finished sending requests
Consumer 1 finished sending requests
Consumer 1: received values: 0, 0, 1, 2, 3, 3, 4, 5, 5, 7, 6, 9, 6, 8, 7
Consumer 1: all actions have been done
Consumer 0: received values: 0, 1, 1, 2, 2, 3, 4, 4, 6, 5, 7, 8, 9, 8, 9
Consumer 0: all actions have been done
All threads have finished
```

Wszystkie wątki zakończyły swoje działanie, a wyniki są zgodne z oczekiwaniami. Występuję bardzo naturalny przeplot operacji `PUT` i `GET`.

## Wnioski

Implementacja wzorca Active Object w kontekście problemu producent-konsument pozwoliła na skuteczną separację logiki operacji (dodawania i pobierania elementów z bufora) od synchronizacji wątków. Klasy `CompletableFuture` i `BufferProxy` tworzą wygodny interfejs do interkacji z buforem.

Wynikiem zastosowania wzorca Active Object jest świetny kod z punktu widzenia klienta, który nie musi martwić się o detale współbieżności. Trzeba natomiast zwrócić uwagę na to, że wzorzec ten jest stosunkowo kosztowny w implementacji, a co za tym idzie w utrzymaniu. Dla prostych problemów może być on wysoce nadmiarowy. Trzeba też zwrócić uwagę na to dodatkowy koszt związany z tym wzorcem, który wynika m.in. z konieczności tworzenia obiektów `MethodRequest`. Dla obszarów gdzie wydajność jest

kluczowa, warto zastanowić się czy ten wzorzec jest odpowiedni.

Wzorzec Active Object może być świetnym wyborem np. gdy piszemy bibliotekę, bo dzięki niemu jej użytkownicy otrzymają wygodny interfejs oraz odsperowanie problemów związanych z wielowątkowością.

## Bibliografia


- Active Object - Wikipedia