

# Teoria Współbieżności - Sprawozdanie 7

Maciej Brzeżawski

Listopad 2024

## 1 Treść zadania

### 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 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.

## 2 Implementacja zadania

### 2.1 Klasa Buffer

```
class Buffer {
    private final int capacity;
    private final Queue<String> queue;

    public Buffer(int capacity) {
        this.capacity = capacity;
        this.queue = new LinkedList<>();
    }

    public void put(String item) {
        if (queue.size() < capacity) {
            queue.add(item);
        }
    }

    public String get() {
        return queue.poll();
    }

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

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

### 2.2 MethodRequest

```
abstract class MethodRequest {
    protected final Buffer buffer;

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

    public abstract boolean guard();

    public abstract void call();
}
```

## 2.3 Klasa PutRequest

```
class PutRequest extends MethodRequest {
    private final String item;

    public PutRequest(Buffer buffer, String item) {
        super(buffer);
        this.item = item;
    }

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

    @Override
    public void call() {
        buffer.put(item);
        System.out.println("Produced: " + item);
    }
}
```

## 3 GetRequest

```
class GetRequest extends MethodRequest {

    public GetRequest(Buffer buffer) {
        super(buffer);
    }

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

    @Override
    public void call() {
        String item = buffer.get();
        System.out.println("Consumed: " + item);
    }
}
```

## 4 ActivationQueue

```
class ActivationQueue {
    private final Queue<MethodRequest> queue = new LinkedList<>();

    public synchronized void enqueue(MethodRequest request) {
        queue.add(request);
        notifyAll();
    }

    public synchronized MethodRequest dequeue() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        return queue.poll();
    }
}
```

## 5 Scheduler

```
class Scheduler extends Thread {
    private final ActivationQueue activationQueue;
    private boolean running = true;

    public Scheduler(ActivationQueue activationQueue) {
        this.activationQueue = activationQueue;
    }

    @Override
    public void run() {
        while (running) {
            try {
                MethodRequest request = activationQueue.dequeue();
                if (request.guard()) {
                    request.call();
                } else {
                    synchronized (activationQueue) {
                        activationQueue.enqueue(request);
                    }
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

```

        public void stopScheduler() {
            running = false;
        }
    }
}

```

## 6 ActiveBufferProxy

```

class ActiveBufferProxy {
    private final ActivationQueue activationQueue;
    private final Buffer buffer;

    public ActiveBufferProxy(ActivationQueue activationQueue, Buffer buffer) {
        this.activationQueue = activationQueue;
        this.buffer = buffer;
    }

    public void put(String item) {
        MethodRequest request = new PutRequest(buffer, item);
        activationQueue.enqueue(request);
    }

    public void get() {
        MethodRequest request = new GetRequest(buffer);
        activationQueue.enqueue(request);
    }
}

```

## 7 ShutdownRequest

```

class ShutdownRequest extends MethodRequest {
    private final Scheduler scheduler;

    public ShutdownRequest(Buffer buffer, Scheduler scheduler) {
        super(buffer);
        this.scheduler = scheduler;
    }

    @Override
    public boolean guard() {
        return true; // Zawsze gotowe do przetworzenia
    }

    @Override

```

```

        public void call() {
            scheduler.stopScheduler();
        }
    }
}

```

## 8 ActiveObjectExample

```

public class ActiveObjectExample {
    public static void main(String[] args) throws InterruptedException {
        Buffer buffer = new Buffer(5); // Bufor o pojemności 5
        ActivationQueue activationQueue = new ActivationQueue();
        Scheduler scheduler = new Scheduler(activationQueue);
        ActiveBufferProxy proxy = new ActiveBufferProxy(activationQueue, buffer);

        scheduler.start();

        // Producenci
        Thread producerThread = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                String item = "Item " + i;
                proxy.put(item);
                System.out.println("Request to produce: " + item);
                try {
                    Thread.sleep(100); // Symulacja czasu produkcji
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });

        // Konsumenci
        Thread consumerThread = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                proxy.get();
                System.out.println("Request to consume.");
                try {
                    Thread.sleep(150); // Symulacja czasu konsumpcji
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });

        producerThread.start();
        consumerThread.start();
    }
}

```

```
        producerThread.join();
        consumerThread.join();

        // Dodanie żądania zakończenia pracy Scheduler
        MethodRequest shutdownRequest = new ShutdownRequest(buffer, scheduler);
        activationQueue.enqueue(shutdownRequest);

        scheduler.join();
        System.out.println("Scheduler stopped. Program completed.");
    }
}
```

## 9 Wyniki programu

The screenshot shows the Run console output for the program. The output consists of alternating messages from the producer and consumer threads, followed by a completion message.

```

Produced: Item 18
Request to produce: Item 19
Request to consume.
Consumed: Item 14
Produced: Item 19
Request to consume.
Consumed: Item 15
Request to consume.
Consumed: Item 16
Request to consume.
Consumed: Item 17
Request to consume.
Consumed: Item 18
Request to consume.
Consumed: Item 19
Request to consume.
Request to consume.
Request to consume.
Request to consume.
Request to consume.
Request to consume.
Request to consume.
Request to consume.
Request to consume.
Request to consume.
Scheduler stopped. Program completed.

Process finished with exit code 0

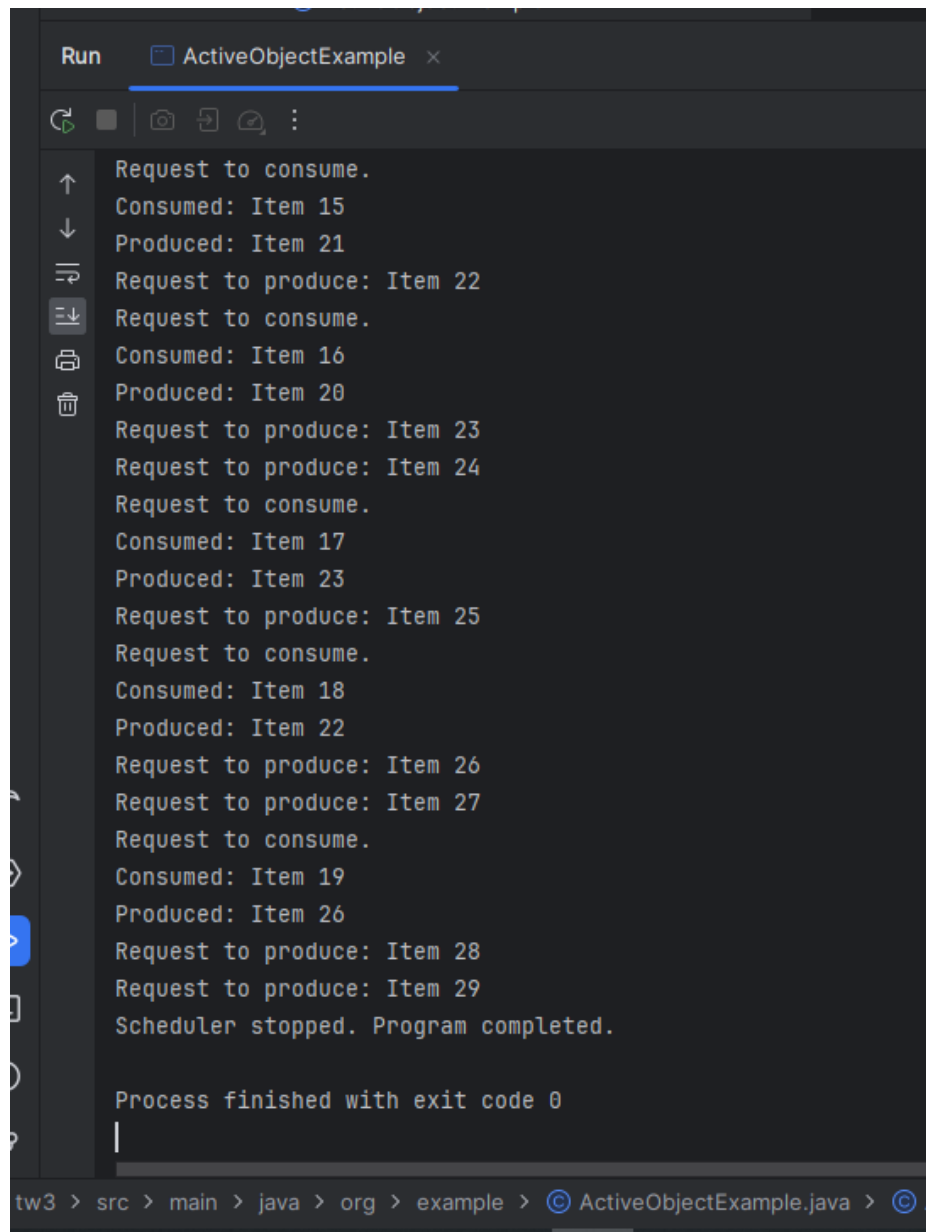
```

The bottom status bar indicates the file path: `tw3 > src > main > java > org > example > © ActiveObjectExample.java > © ActiveObjectExample.java`.

Rysunek 1: Producenci: 10, Konsumenci: 10







```
Run ActiveObjectExample x
Request to consume.
Consumed: Item 15
Produced: Item 21
Request to produce: Item 22
Request to consume.
Consumed: Item 16
Produced: Item 20
Request to produce: Item 23
Request to produce: Item 24
Request to consume.
Consumed: Item 17
Produced: Item 23
Request to produce: Item 25
Request to consume.
Consumed: Item 18
Produced: Item 22
Request to produce: Item 26
Request to produce: Item 27
Request to consume.
Consumed: Item 19
Produced: Item 26
Request to produce: Item 28
Request to produce: Item 29
Scheduler stopped. Program completed.

Process finished with exit code 0
```

tw3 > src > main > java > org > example > © ActiveObjectExample.java > ©

Rysunek 3: Producenci: 30, Konsumenci: 20

## 10 Wnioski

- **Modularność i Dekompzycja:**
  1. Wzorzec aktywnego obiektu dobrze dzieli odpowiedzialności między komponentami: Proxy, Servant, Scheduler, MethodRequest itp.
  2. Każdy komponent realizuje jasno określone zadanie, co ułatwia rozwój i utrzymanie kodu.
- **Synchronizacja i Konkurencyjność:**
  1. Implementacja skutecznie radzi sobie z synchronizacją między producentami i konsumentami. Scheduler decyduje, które żądanie jest gotowe do przetworzenia, bazując na metodach strażników (guard).
  2. Wyeliminowano ryzyko wyścigu dzięki odpowiedniej kolejce aktywacji (ActivationQueue) i synchronizacji wątku Scheduler.
- **Skalowalność:**
  1. Rozwiązanie jest łatwe do skalowania, można w prosty sposób dodać więcej producentów i konsumentów bez istotnych zmian w logice.
  2. Rozdzielenie pracy między wątki pozwala efektywnie wykorzystać systemy wielordzeniowe.
- **Poprawność i Stabilność:**
  1. Problemy z zawieszaniem się programu zostały wyeliminowane dzięki wprowadzeniu specjalnego ShutdownRequest oraz poprawionej logiki zarządzania cyklem życia wątków.
- **Realizacja Wzorca Aktywnego Obiektu:**
  1. Proxy obsługuje żądania klientów.
  2. Servant (bufor) realizuje rzeczywiste operacje.
  3. Scheduler zarządza kolejką żądań (ActivationQueue) i decyduje o ich realizacji.
  4. MethodRequest umożliwia implementację strażników i enkapsuluje logikę wywołań metod.
  5. ActivationQueue przechowuje żądania metod w kolejności zgłaszania, synchronizując dostęp klientów (Proxy) i Scheduler.

## 11 Bibliografia

Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Operating System Concepts*. Wiley, 2020. ISBN: 978-1119800361.

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006. ISBN: 978-0321349606.

Andrew S. Tanenbaum, Herbert Bos. *Modern Operating Systems*. Pearson, 2015. ISBN: 978-0133591620.

Maurice Herlihy, Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2020. ISBN: 978-0124159501.

A. S. Tanenbaum, M. Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, 2007. ISBN: 978-0132392273.

Douglas Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2000. ISBN: 978-0201310092.