# Programowanie Współbieżne Projekt

Krzysztof Nowak

Grupa: WCY23IY2S1

Nr. Albumu: 86038

Instrukcja uruchomienia (wywołujemy z roota projektu):

1. Pobrać release tag
2. cd Java_Hairdresser_Concurrency-1.0.2
3. Uruchomienie (Linux): java -jar target/Test-1.0-SNAPSHOT.jar

PW-6

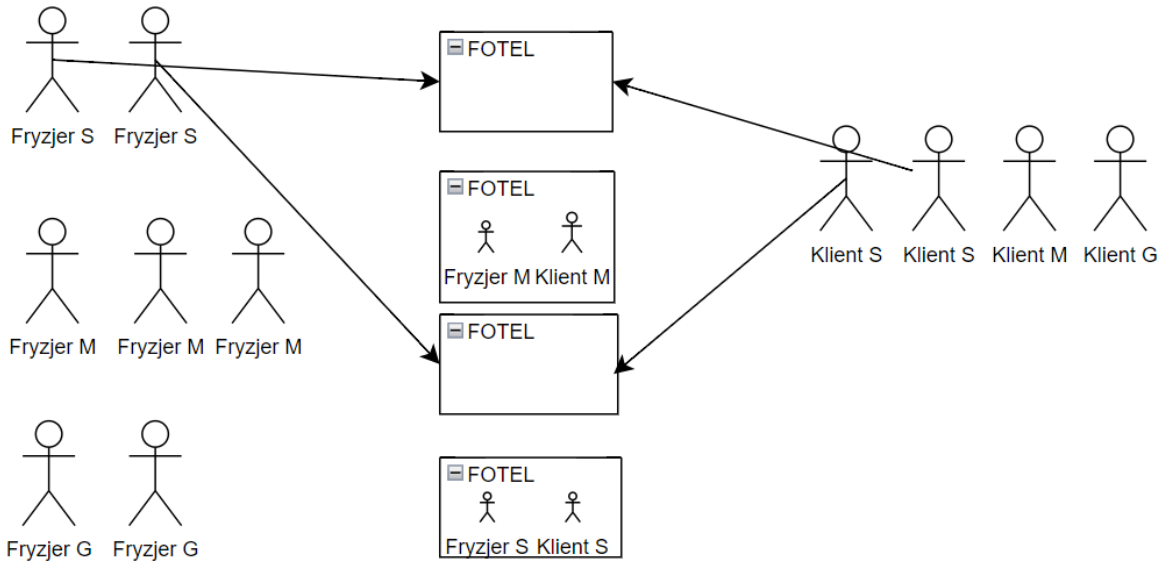
Problem do rozwiązania:
Zakład fryzjerski.
Założenia:
Zakład fryzjerski realizuje N rodzajów usług (np.: strzyżenie, modelowanie, golenie, …) i obsługuje klientów przybywających do niego w losowych odstępach czasu.
Klienci trafiają najpierw do poczekalni o ustalonej pojemności i czekają na wykonanie usługi wybranego rodzaju.
W zakładzie pracuje P fryzjerów, którzy są wyspecjalizowani w określonych usługach:

a. - $P_1$ – liczba fryzjerów wyspecjalizowanych w strzyżeniu,
b. - $P_2$ – liczba fryzjerów wyspecjalizowanych w modelowaniu,
c. - $P_3$ – liczba fryzjerów wyspecjalizowanych w goleniu,

($P_1$, $P_2$, $P_3$, … <= P).
Do dyspozycji fryzjerów jest L foteli (L < P).

# 1. Opis problemu

Głównym problemem powyższego zadania jest synchronizacja zasobów takich jak kolejka, ilość fryzjerów oraz ilość krzeseł tak aby system odzwierciedlał realny lub też bardzo zbliżony zakład fryzjerski (BARBERSHOP). Przedstawmy powyższe zadanie na przykładzie. Zakład fryzjerski posiada wyspecjalizowanych fryzjerów w **N** dziedzinach, w naszym przykładzie będę przyjmował że fryzjer jest wyspecjalizowany tylko w jednej usłudze (Strzyżenie, Golenie, Modelowanie) i tylko tą usługę może wykonywać. Fryzjerów wyspecjalizowanych w danej usłudze jest **K**. Klienci przychodzą do salonu fryzjerskiego na jedną wybraną usługę. Klienci wchodzą do zakładu fryzjerskiego i jeśli nie ma kolejki zostają od razu przydzieleni na jedno z **L** krzeseł. Jeśli krzesła są już okupowane klienci muszą czekać w kolejce **FIFO** (pierwszy klient który przychodzi zostaje pierwszy obsłużony). Jedna modyfikacja tej kolejki polega na tym, że w przypadku gdy pierwszy klient w kolejce nie może zostać obsłużony, tzn. jest wolne krzesło ale nie ma dostępnego fryzjera który może wykonać usługę na jaką dany klient przyszedł, wtedy pierwszy klient z kolejki dla którego jest dostępny fryzjer o danej specjalności jest obsłużony. W realnym świecie gdy kolejka jest zbyt długa do fryzjera, rezygnujemy z usługi i odchodzimy (chyba że jesteśmy bardzo zdeterminowani), kolejnym założeniem jest określona wielkość kolejki FIFO, nowi klienci nie mogą wejść do zakładu fryzjerskiego jeśli kolejka jest pełna. Fryzjerzy nie są rozróżniani (reprezentowani jedynie jako liczba zasobów danego typu) Przykładowa wizualizacja opisanego procesu:




*Zdj. Salon Fryzjerski*

Przebieg programu krok po kroku:

a. Wczytanie defaultowych danych konfiguracyjnych z pliku ()

```
{
  "F" : {  - Fryzjerzy
    "S" : 2, - Liczba fryzjerów typu S
    "M" : 2, - Liczba fryzjerów typu M
    "G" : 2  - Liczba fryzjerów typu G
  },
  "LF" : 4, - Liczba foteli
  "WK" : 10 - Wielkość kolejki
}
```

b. Zainicjalizowanie zasobów współdzielonych (kolejka, krzesła, fryzjerzy)
c. Zainicjalizowanie GUI
d. Konfiguracja parametrów w GUI po czym zapisanie ich do pliku konfiguracyjnego
e. Uruchomienie 2 typów wątków:
   - pierwszy typ obsługujący wprowadzanie danych do kolejki
   - drugi typ obsługujący przydzielanie klientów z kolejki na fotele wraz z przydzielaniem odpowiednich fryzjerów
f. Wątek obsługujący wprowadzanie danych do kolejki robi to w losowych odstępach czasu dobranych do poprawnej wizualizacji w gui. Wprowadzanie do kolejki odbywa się tylko i wyłącznie jeśli kolejka nie jest pełna (w innym przypadku klienci „odbijają” się od kolejki i nie zostają do niej dołączeni). Jej wielkość jest określona w pliku konfiguracyjnym.

```
if (queue.isFull()) {
//                    System.out.println("Queue full");
                } else{
                    try {
                        String customer = counters[option].getCounter();
                        queue.enqueue(options[option] + customer);
                        counters[option].increment();

                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
```

g. W zależności od podanej w konfiguracji liczby foteli tyle jest wątków pracowniczych. Podstawowa ich praca to działanie na zsynchronizowany obiekcie kolejki, sprawdzane jest czy są jacyś klienci czekający oraz czy są dostępne krzesła. Jeśli tak to sprawdzamy dostępność fryzjera typu dpowiadającemu usłudze na jaki przyszedł dany klient (modelowanie, strzyżenie, golenie). Jeśli nie ma odpowiadającego fryzjera dla danego klienta, skanujemy kolejkę w reżymie FIFO poszukiwaniu klienta który może zostać obsłużony. Kleint który był pierwszy i nie został obsłużony dalej pozostaje pierwszy w kolejce FIFO i czeka aż fryzjer typu jego usługi będzie dostępny oraz będzie wolny fotel po czym zostanie on obsłużony.

```
synchronized (queue) {
    if (queue.size() > 0 && chairs.hasAvailableChairs()) {
        HashMap<String, Boolean> hairdressersAvailable = new HashMap<>();
        for (String svc : services) {
            hairdressersAvailable.put(svc,
hairDressers.hasAvailableHairdressers(svc));
        }

        customer = queue.returnFirstOccurence(hairdressersAvailable);
        if (!customer.isEmpty()) {
            queue.removeFromFifo(customer);
            service = String.valueOf(customer.charAt(0));
            serviceTime = calculateServiceTime(service);
            hairDressers.decrementHairdressers(service);
            chairs.acquireChair(customer, serviceTime);
        }
    }
}
```

Po zablokowaniu danego krzesła, usunięcia obsługiwanego klienta z kolejki oczekujących oraz zdekrementowaniu ilości fryzjerów danego typu wątek przechodzi do wykonania akcji Thread.sleep(serviceTime) symulującego wykonywanie danej usługi.

```
private void processService(String customer, String service, int
serviceTime) throws InterruptedException, IOException {

//        System.out.println(customer + " Available chairs: " +
chairs.availableChairs());
        Thread.sleep(serviceTime);
        chairs.releaseChair(customer);
```

```
        hairDressers.incrementHairdressers(service);
//        System.out.println(customer + " Available chairs: " +
chairs.availableChairs());
//        System.out.println(customer + " Available Hairdressers of type "
+ service + " : " + hairDressers.availableHairdressers(service));
    }
```

Każda usługa ma przypisaną dolną i górną granicę czasu ile powinna się wykonywać i jest losowo obliczana i później przekazywana do funkcji Thread.sleep()

```
private int calculateServiceTime(String service) {
    Random random = new Random();
    return switch (service) {
        case "S" -> // Slow service
                random.nextInt(8000, 12000);
        case "M" -> // Medium service
                random.nextInt(5000, 8000);
        case "G" -> // Quick service
                random.nextInt(2000, 5000);
        default -> 3000; // Default service time
    };
}
```

Po wykonaniu usługi zablokowane krzesło zostaje zwolnione a ilość fryzjerów danego typu zostaje zinkrementowana.

h. Aktualizowanie GUI odbywa się poprzez słuchaczy na obiektach zasobów współdzielonych. Nasłuchują oni zmiany i w przypadku aktualizacji zasobów informacja zostaje przekazana do GUI, po czym zostaje ona odpowiednio zasygnalizowana na ekranie.

## 2. Założenia

1. Jeden fryzjer ma w ofercie tylko jedną usługę
2. Jeden klient przychodzi na wybraną 1 usługę (Strzyżenie, Modelowanie, Golenie)
3. Brak możliwości wejścia do salonu fryzjerskiego przez klientów gdy pełna kolejka

## 3. Wykaz współdzielonych zasobów

1. Fryzjerzy typu S, M, G
2. Fotele do strzyżenia
3. Kolejka oczekiwania FIFO

Fryzjerzy typu S, M, G są przydzielani do klienta (przychodzącego na jedną z usług S, M, G) i stawiani do konkretnego fotela. Liczba foteli, fryzjerów danego typu oraz wielkość poczekalni jest ograniczona.

# 4. Wykaz punktów synchronizacji

### 1. Klasa krzesła

```
public synchronized void addChairsListener(final ChairsEventListener
listener) {
    if (!listeners.contains(listener)) {
        listeners.add(listener);
    }
}
public synchronized void removeChairsListener(final ChairsEventListener
listener) {
    listeners.remove(listener);
}
```

Listenery zmian statusu krzesła (używane w celu nasłuchiwania zmian i w rezultacie wyświetlenia odpowiedniej zmiany w gui).

```
public synchronized int availableChairs(){
        return MAX_CHAIRS - chairCustomer.size();
    }

    public synchronized void releaseChair(String customer){
//        System.out.println("Customer " + customer + " has been served");

        Integer chairToRelease = null;
        for(Integer chairId : chairCustomer.keySet()) {
            if(chairCustomer.get(chairId).equals(customer)) {
                chairToRelease = chairId;
                break;
            }
        }
        if (chairToRelease != null) {
            chairCustomer.remove(chairToRelease);
            chairEndTimes.remove(chairToRelease);
//            System.out.println("Customer " + customer + " has left chair
" + chairToRelease);
            notifyListeners();
        }
    }

    public synchronized void acquireChair(String customer, int serviceTime)
{
//        System.out.println("Serving customer: " + customer);
        int chairId = 0;
        while(chairCustomer.containsKey(chairId)) {
```

```
            chairId++;
        }
        chairCustomer.put(chairId, customer);
        long endTime = System.currentTimeMillis() + serviceTime;
        chairEndTimes.put(chairId, endTime);
//          chairCapacity--;
//          System.out.println("Serving customer: " + customer + " in chair "
+ chairId);
        notifyListeners();
    }

    public synchronized boolean hasAvailableChairs() {
        return chairCustomer.size() < MAX_CHAIRS;
    }
```

Funkcje zajmowania oraz zwalniania krzesła przez klienta oraz funkcje używane do sprawdzania statusu zasobu krzeseł.

2. **Klasa fryzjerów**

```
public synchronized void addHairdressersListener(final
HairdressersEventListener listener) {
    if (!listeners.contains(listener)) {
        listeners.add(listener);
    }
}
public synchronized void removeHairdressersListener(final
HairdressersEventListener listener) {
    listeners.remove(listener);
}
```

Listenery zmian statusu zasobu fryzjerów (używane w celu nasłuchiwania zmian i w rezultacie wyświetlenia odpowiedniej zmiany w gui).

```
public synchronized void decrementHairdressers(String hairdresser) {
    switch (hairdresser) {
        case "S" -> hairdressersS--;
        case "M" -> hairdressersM--;
        case "G" -> hairdressersG--;
    }
    notifyListeners();
}

public synchronized void incrementHairdressers(String hairdresser) {
    switch (hairdresser) {
        case "S" -> hairdressersS++;
        case "M" -> hairdressersM++;
        case "G" -> hairdressersG++;
    }
    notifyListeners();
}
```

Funkcje inkrementacji (w przypadku zwolnienia fryzjera danego typu) oraz dekrementacji (w przypadku zajęcia fryzjera danego typu).

```
public synchronized boolean hasAvailableHairdressers(String hairdresser) {
    switch (hairdresser) {
        case "S" -> {
            return hairdressersS > 0;
        }
        case "M" -> {
            return hairdressersM > 0;
        }
        case "G" -> {
            return hairdressersG > 0;
        }
    }
    return false;
}

public synchronized int availableHairdressers(String hairdresser) {
    switch (hairdresser) {
        case "S" -> {
            return hairdressersS;
        }
        case "M" -> {
            return hairdressersM;
        }
        case "G" -> {
            return hairdressersG;
        }
    }
    return 0;
}
```

Funkcje sprawdzania statusu zasobu fryzjerów.

### 3. Klasa Poczekalni

```
public synchronized void addQueueListener(final QueueEventListener
listener) {
    if (!listeners.contains(listener)) {
        listeners.add(listener);
    }
}
public synchronized void removeQueueListener(final QueueEventListener
listener) {
    listeners.remove(listener);
}

private void notifyListeners() {
    for(QueueEventListener listener : listeners) {
        listener.onQueueChanged(this);
```

```
    }
}
```

Listenery zmian statusu kolejki. (używane w celu nasłuchiwania zmian w nadchodzących i opuszających kolejkę klientach i w rezultacie wyświetlenia odpowiedniej zmiany w gui).

```
public synchronized void enqueue(String item) throws InterruptedException {
        while(isFull()){
//              System.out.println("Queue full");
            wait();
        }
        queue.add(item);

//        System.out.println("Enqueued " + item + " People in queue: " +
queue.size());
        notifyListeners();
        notifyAll();
    }
public synchronized String dequeue() throws InterruptedException {
        while(queue.isEmpty()) {
            wait();
        }

        String item =  queue.poll();
//        System.out.println("People in queue: " + queue.size() + "
Dequeued " + item);
        notifyListeners();
        notifyAll();
        return item;

    }
```

Funkcje dodawania nowych klientów do kolejki FIFO oraz usuwania z niej.

```
public synchronized boolean isFull(){
        if(queue.size() < MAX_CAPACITY) {
            return false;
        } return true;
    }
    public synchronized int size() {
        return queue.size();
    }

    public synchronized String returnFirstOccurence(HashMap<String,
Boolean> hairdressersAvailable) {
        for(String customer : queue){
            if(hairdressersAvailable.get(customer.substring(0, 1))) return
customer;
        }
        return "";
    }

    public synchronized void removeFromFifo(String customer) throws
InterruptedException {
```

```
        queue.remove(customer);
        notifyListeners();
    }

    public synchronized Queue<String> getQueueCopy(){
        return new LinkedList<>(queue);
    }
```

Funkcje sprawdzania zapełnienia kolejki, jej aktualnej wielkości, funkcja usuwania konkretnego użytkownika z kolejki, funkcja zwracająca kopię kolejki.

## 5. Wykaz obiektów synchronizacji

### 1. Wątek obsługujący kolejkę i zarządzający przydzielaniem miejsca w fotelach

```
synchronized (queue) {
    if (queue.size() > 0 && chairs.hasAvailableChairs()) {
        HashMap<String, Boolean> hairdressersAvailable = new HashMap<>();
        for (String svc : services) {
            hairdressersAvailable.put(svc,
hairDressers.hasAvailableHairdressers(svc));
        }

        customer = queue.returnFirstOccurence(hairdressersAvailable);
        if (!customer.isEmpty()) {
            queue.removeFromFifo(customer);
            service = String.valueOf(customer.charAt(0));
            serviceTime = calculateServiceTime(service);
            hairDressers.decrementHairdressers(service);
            chairs.acquireChair(customer, serviceTime);
        }
    }
}
```

Działanie wyłączne jednego wątku na kolejce, sprawdzanie czy są klienci w kolejce oraz czy są dostępne fotele w salonie fryzjerskim. W przypadku jeśli oba warunki są spełnione przechodzimy przez kolejkę w poszukiwaniu pierwszego klienta dla którego również mamy dostępnego fryzjera po czym przechodzimy do jego obsługi. Usuwamy go z kolejki, zmniejszamy liczbę fryzjerów typu odpowiadającego usłudze na jaką klient przyszedł. Aktualizowanie statusu krzesła jako zajęte – acquireChair.

## 2. GUI

```
private void updateQueueDisplay() {
    synchronized (screen) {
        try {
            TerminalSize size = screen.getTerminalSize();
            int startX = size.getColumns() * 6/10;
            int startY = size.getRows() / 10;

            String header = "QUEUE STATUS";
            for(int i=0; i < header.length(); i++) {
                screen.setCharacter(startX + i, startY - 2, new
TextCharacter(header.charAt(i)));
            }

            screen.setCharacter(startX, startY, new TextCharacter('['));
            screen.setCharacter(startX + maxQueueSize * 3 + 1, startY, new
TextCharacter(']'));

            for(int i=1; i < maxQueueSize * 3 + 1; i++) {
                screen.setCharacter(startX + i, startY, new
TextCharacter('·'));
            }


            Queue<String> queueCopy = queueResource.getQueueCopy();
            int i = 0;
            for (String item : queueCopy) {
                char displayChar = item.charAt(0);
                TextColor itemColor;

                switch (displayChar) {
                    case 'S':
                        itemColor = TextColor.ANSI.RED;
                        break;
                    case 'M':
                        itemColor = TextColor.ANSI.GREEN;
                        break;
                    case 'G':
                        itemColor = TextColor.ANSI.BLUE;
                        break;
                    default:
                        itemColor = TextColor.ANSI.WHITE;
                }
                for(int j = 0; j < item.length(); j++) {
                    screen.setCharacter(startX + i + 1 + j, startY, new
TextCharacter(item.charAt(j), itemColor, TextColor.ANSI.BLACK));
                }
                i += item.length();
            }

            screen.refresh();

        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Aktualizowanie kolejki (tylko jeden wątek na raz może aktualizować gui).

```
private void updateChairDisplay() {
        HashMap<Integer, String> chairsCopy;
        chairsCopy = chairsResource.getChairsCopy();
        HashMap<Integer, Integer> remainingTimes =
chairsResource.getRemainingTimes();

        synchronized (screen) {
            try {
                TerminalSize size = screen.getTerminalSize();

                final int CHAIR_WIDTH = 5;
                final int CHAIR_HEIGHT = 3;
                final int CHAIR_SPACING = 4;

                int startX = size.getColumns() * 3/10;
                int initialY = size.getRows() / 10;

                String header = "CHAIRS STATUS";
                for (int i = 0; i < header.length(); i++) {
                    screen.setCharacter(startX + i, initialY - 2, new
TextCharacter(header.charAt(i)));
                }

                for (int chairId = 0; chairId < maxChairs; chairId++) {
                    int chairY = initialY + (chairId * CHAIR_SPACING);

                    drawChair(screen, startX, chairY, CHAIR_WIDTH,
CHAIR_HEIGHT);

                    screen.setCharacter(startX - 2, chairY + 1,
                            new TextCharacter(Character.forDigit(chairId +
1, 10)));

                    if (chairsCopy.containsKey(chairId)) {
                        String customer = chairsCopy.get(chairId);
                        char displayChar = customer.charAt(0);

                        TextColor itemColor;
                        switch (displayChar) {
                            case 'S':
                                itemColor = TextColor.ANSI.RED;
                                break;
                            case 'M':
                                itemColor = TextColor.ANSI.GREEN;
                                break;
                            case 'G':
                                itemColor = TextColor.ANSI.BLUE;
                                break;
                            default:
                                itemColor = TextColor.ANSI.WHITE;
                        }
                        int customerX = startX + CHAIR_WIDTH/2 -
customer.length()/2;

//                        Czyszczenie
                        for(int i = 0; i < 3; i++) {
                            screen.setCharacter(customerX + i, chairY +
```

```
CHAIR_HEIGHT/2,
                                    new TextCharacter(' ', itemColor,
TextColor.ANSI.BLACK));
                        }

//                        Wyswietlanie
                        for(int i = 0; i < customer.length(); i++) {
                            screen.setCharacter(customerX + i, chairY +
CHAIR_HEIGHT/2,
                                    new TextCharacter(customer.charAt(i),
itemColor, TextColor.ANSI.BLACK));
                        }

                        int remainingTime =
remainingTimes.getOrDefault(chairId, 0);
                        String timeStr = String.format("%02d:%02d",
remainingTime / 60, remainingTime % 60);

                        TextGraphics tg = screen.newTextGraphics();
                        tg.setForegroundColor(TextColor.ANSI.YELLOW);
                        tg.putString(startX + 8, chairY + 1, timeStr);


                    }
                }
                screen.refresh();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
```

Aktualizowanie okupowania krzeseł (tylko jeden wątek na raz może aktualizować gui).

```
private void updateHairdresserDisplay(){
    synchronized (screen) {
        try {
            TerminalSize size = screen.getTerminalSize();
            int startX = size.getColumns() * 1/10;
            int startY = size.getRows() / 10;

            String header = "HAIRDRESSERS STATUS";
            for(int i=0; i < header.length(); i++) {
                screen.setCharacter(startX + i, startY - 2, new
TextCharacter(header.charAt(i)));
            }

            // Czyszczenie wczesniejszych wartosci
            for(int i = 0; i < 10; i ++) {
                for(int j = 0; j< header.length(); j++){
                    screen.setCharacter(startX + j, startY + i, new
TextCharacter(' '));
                }
            }

            int hairdressersS =
hairdressersResource.availableHairdressers("S");
            int hairdressersM =
```

```
hairdressersResource.availableHairdressers("M");
            int hairdressersG =
hairdressersResource.availableHairdressers("G");

            TextGraphics tg = screen.newTextGraphics();

            tg.putString(startX, startY, "Usługa S: ");
            for(int i=0; i < hairdressersS; i++) {
                screen.setCharacter(startX + 11 + i, startY, new
TextCharacter('■', TextColor.ANSI.RED, TextColor.ANSI.BLACK));
            }

            tg.putString(startX, startY + 2, "Usługa M: ");
            for(int i=0; i < hairdressersM; i++) {
                screen.setCharacter(startX + 11 + i, startY + 2, new
TextCharacter('■', TextColor.ANSI.GREEN, TextColor.ANSI.BLACK));
            }

            tg.putString(startX, startY + 4, "Usługa G: ");
            for(int i=0; i < hairdressersG; i++) {
                screen.setCharacter(startX + 11 + i, startY + 4, new
TextCharacter('■', TextColor.ANSI.BLUE, TextColor.ANSI.BLACK));
            }

            screen.refresh();

        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Aktualizowanie zajętości fryzjerów (tylko jeden wątek na raz może aktualizować gui).

## 6. Wykaz procesów sekwencyjnych

**Proces klienta**

1. **Wejście do salonu**
    - Sprawdzenie, czy jest miejsce w kolejce (queue.size() < MAX_QUEUE).
    - Jeśli nie ma miejsca → klient odchodzi (**utrata klienta**).
    - Jeśli jest miejsce → klient wchodzi do kolejki FIFO.
2. **Oczekiwanie na obsługę**
    - Jeśli jest wolne krzesło i dostępny fryzjer odpowiedniej specjalizacji → klient jest obsługiwany.
    - Jeśli nie ma fryzjera, ale jest wolne krzesło → klient **czeka w kolejce**, a system sprawdza następnego klienta (moduł **"przeskakiwania kolejki"**).
3. **Obsługa klienta**
    - Fryzjer wykonuje usługę (symulowane przez sleep()) .
    - Po zakończeniu:

    - Klient opuszcza salon.
    - Krzesło staje się wolne.
    - Fryzjer staje się dostępny.

---

**Proces fryzjera**

1. **Sprawdzenie dostępności klienta**
    - Fryzjer cyklicznie sprawdza, czy w kolejce jest klient wymagający jego specjalizacji.
    - Jeśli tak → przydziela go do wolnego krzesła.
2. **Wykonanie usługi**
    - Blokuje krzesło i swoją dostępność na czas usługi.
    - Po zakończeniu zwalnia zasoby.