Sprawozdanie z projektu

Przedmiot: Programowanie współbieżne

Temat: Symulacja obsługi kas w supermarkecie.

Wykonał: Hubert Witkowski WCY23IY2S1

Nr. Albumu: 86071

Data: 06.06.2025

Spis treści

1. Treść zadania:

Zadanie nr: PW-2 Język implementacji: Java: {Java, Kotlin, Scala}, .net {C#, F#}, Go, propozycja studenta Środowisko implementacyjne: JetBrains Termin wykonania: ostatnie zajęcia

Zasadnicze wymagania:
a. liczba procesów sekwencyjnych powinna być dobrana z wyczuciem tak, aby zachować czytelność interfejsu i jednocześnie umożliwić zobrazowanie reprezentatywnych przykładów,
b. kod źródłowy programu musi być tak skonstruowany, aby można było „swobodnie" modyfikować liczbę procesów sekwencyjnych (z wyjątkiem zadań o ściśle określonej liczbie procesów),
c. graficzne zobrazowanie działania procesów współbieżnych,
d. odczyt domyślnych danych wejściowych ze sformatowanego, tekstowego pliku danych (yaml, json, inne),
e. [opcjonalnie] możliwość modyfikacji danych wejściowych poprzez GUI.

Sprawozdanie (w formie elektronicznej) powinno zawierać następujące elementy:
1) stronę tytułową,
2) numer i niniejszą treść zadania,
3) syntetyczny opis problemu – w tym wszystkie przyjęte założenia,
4) wykaz współdzielonych zasobów,
5) wykaz wyróżnionych punktów synchronizacji,
6) wykaz obiektów synchronizacji,
7) wykaz procesów sekwencyjnych,
Powyższe ilustrować adekwatnymi sekcjami kodu źródłowego programu.

Problem do rozwiązania: Symulacja obsługi kas w supermarkecie.
Założenia:
- M – liczba kas;
- N – maksymalna liczba klientów w sklepie;
- Osobne kolejki do każdej kasy;
- Przerwy kasjerów oraz ich zmiana po upływie określonego czasu (kasa jest w tym czasie zamknięta);
- Dokończenie obsługi wszystkich klientów z kolejki przed zamknięciem kasy.

2. Opis problemu:

Symulacja modeluje działanie supermarketu z wieloma kasami obsługującymi klientów. System zarządza przepływem klientów przez sklep, ich przydzielaniem do kolejek oraz obsługą przez kasjerów, którzy mogą udawać się na przerwy.

Przyjęte założenia:

- **Ograniczona pojemność sklepu**: Maksymalnie N klientów może przebywać w sklepie jednocześnie
- **Kasy**: M kas działa równolegle, każda z własną kolejką
- **Inteligentny wybór kolejki**: Klienci wybierają najkrótszą dostępną kolejkę
- **Przerwy kasjerów**: Kasjerzy losowo decydują o przerwach z konfigurowalnymi parametrami (z każdym klientem jest szansa że kasjer pójdzie na przerwę)
- **Dokończenie obsługi**: Przed zamknięciem kasy kasjer obsługuje wszystkich klientów z kolejki
- **Proces zakupów**: Klienci po wejściu do sklepu spędzają losowy czas na zakupach przed udaniem się do kasy (nie wchodzą od razu do kolejki do kasy)

2.1 Konfiguracja programu

System odczytuje konfigurację z pliku config.json:

```
class Config {  14 usages  new *
    @JsonProperty("liczba_kas")  9 usages
    public int liczbKas;
    @JsonProperty("max_klientow")  14 usages
    public int maxKlientow;
    @JsonProperty("czas_obslugi_min")  8 usages
    public int czasObslugiMin;
    @JsonProperty("czas_obslugi_max")  7 usages
    public int czasObslugiMax;
    @JsonProperty("czas_przerwy_kasjera")  7 usages
    public int czasPrzerwyKasjera;
    @JsonProperty("czas_przychodu_klienta")  7 usages
    public int czasPrzychoduKlienta;
    @JsonProperty("szansa_na_przerwe")  7 usages
    public double szansaNaPrzerwe;
```

```
{
  "liczba_kas": 4,
  "max_klientow": 40,
  "czas_obslugi_min": 2000,
  "czas_obslugi_max": 4000,
  "czas_przerwy_kasjera": 7000,
  "czas_przychodu_klienta": 600,
  "szansa_na_przerwe": 0.15
}
```

3. Wykaz współdzielonych zasobów:

- Semafor zliczający maksymalną liczbę osób w sklepie definiowaną w config lub braną domyślnie.

```
private final Semaphore miejscaWSklepie; // semafor ograniczajacy liczbe klientów w sklepie  7 usages
```

- Kolejka przechowująca klientów w kolejce do danej kasy
- Zmienne atomowe typu boolean odpowiedzialne za status danej kasy
- Mutex do synchronizacji operacji na kasie
- GUI – aktualizowane przez poszczególne metody

```
class Kasa {  11 usages  new *
    private final int id;  6 usages
    private final ConcurrentLinkedQueue<Klient> kolejka; //  kolejka klientów  6 usages
    private final AtomicBoolean otwarta;  2 usages
    private final AtomicBoolean kasjerDostepny; // czy kasjer jest nie na przerwie  5 usages
    private final AtomicBoolean kasjerChcePrzerwe;  5 usages
    private final ReentrantLock kasaMutex; // Mutex do synchronizacji operacji na kasie  6 usages
    private final GUI gui;  3 usages
```

- Zmienna atomowa do startowania i resetowania symulacji.

```
private final AtomicBoolean running;  3 usages
```

- Status obsługiwania i przerwy kasjera

```
private final AtomicBoolean currentlyServing = new AtomicBoolean( initialValue: false);
private final AtomicBoolean onBreak = new AtomicBoolean( initialValue: false);  4 usages
```

4. Punkty synchronizacji

- Wejście klienta do sklepu

```
if (miejscaWSklepie.tryAcquire()) { //nowy klient wchodzi do sklepu
    boolean klientDodany = false;

    try {         // losuje czas obsługi dla nowego klienta

        int czasObslugi = config.czasObslugiMin +
                random.nextInt( bound: config.czasObslugiMax - config.czasObslugiMin + 1);
        Klient klient = new Klient(czasObslugi);
        Kasa wybranaKasa = wybierzNajkrotszaKolejke(); // szukanie najkrotszej kolejki
        if (wybranaKasa != null) {
            if (wybranaKasa.dodajKlienta(klient)) {
                klientDodany = true;
                // uruchamia proces klienta (zakupy + oczekiwanie na obsługę)
                new ProcesKlienta(klient, miejscaWSklepie, gui, config.maxKlientow, scheduler).start();
```

- Opuszczanie sklepu

```
private void opuscSklep() { //zwalnianie miejsca i aktualizacja gui  1 usage  new *
    miejscaWSklepie.release();
    SwingUtilities.invokeLater(() -> {
        gui.updateCustomerCount( currentCustomers: maxKlientow - miejscaWSklepie.availablePermits(), maxKlientow);
    });
    System.out.println(klient + " opuścił sklep (wolne miejsca: " +
            miejscaWSklepie.availablePermits() + ")");
}
```

- Dodanie klienta do kolejki

```
public boolean dodajKlienta(Klient klient) {  1 usage  new *
    kasaMutex.lock(); // blokada
    try {
        if (czyPrzyjmujeKlientow()) {
            kolejka.offer(klient);
            gui.updateKasaDisplay(id, kolejka.size(), czyPrzyjmujeKlientow(), kasjerDostepny.get());
            System.out.println(klient + " dołączył do kolejki kasy " + id);
            return true;
        }
        return false;
    } finally {
        kasaMutex.unlock();
    }
}
```

- Obsługa przez kasjera

```
private void obsluzKlienta(Klient klient) {  1 usage  new *
    if (currentlyServing.compareAndSet( expectedValue: false,  newValue: true)) {
        System.out.println("Kasjer kasy " + kasa.getId() + " obsługuje " + klient +
                " (pozostało w kolejce: " + kasa.rozmiarKolejki() +
                (kasa.isKasjerChcePrzerwe().get() ? ", nie przyjmuje nowych" : "") + ")");

        scheduler.schedule(() -> {
            try {
                klient.oznaczJakoObsluzony();
                System.out.println("Kasjer kasy " + kasa.getId() + " zakończył obsługę " + klient);

                SwingUtilities.invokeLater(() -> {
                    gui.updateKasaDisplay(kasa.getId(), kasa.rozmiarKolejki(),
                            kasa.czyPrzyjmujeKlientow(), kasa.isKasjerDostepny().get());
                });
            } finally {
                currentlyServing.set(false);
            }
        }, klient.getCzasObslugi(), TimeUnit.MILLISECONDS);
    }
}
```

- Synchronizacja przerwy kasjera

```
private void wykonajPrzerwe() { //wykonywanie przerwy  1 usage  new *
    if (onBreak.compareAndSet( expectedValue: false,  newValue: true)) {
        System.out.println("Kasjer kasy " + kasa.getId() + " idzie na przerwę (kolejka opróżniona)");

        kasa.isKasjerDostepny().set(false); // kasjer niedostepny
        SwingUtilities.invokeLater(() -> {
            gui.updateKasaDisplay(kasa.getId(), kasa.rozmiarKolejki(),
                    kasa.czyPrzyjmujeKlientow(),  kasjerDostepny: false);
        });

        scheduler.schedule(() -> {
            try {
                kasa.isKasjerDostepny().set(true);
                System.out.println("Kasjer kasy " + kasa.getId() + " wrócił z przerwy");
                SwingUtilities.invokeLater(() -> {
                    gui.updateKasaDisplay(kasa.getId(), kasa.rozmiarKolejki(),
                            kasa.czyPrzyjmujeKlientow(),  kasjerDostepny: true);
                });
            } finally {
                onBreak.set(false);
            }
        }, config.czasPrzerwyKasjera, TimeUnit.MILLISECONDS);
    }
}
```

- Sprawdzanie rozmiaru kolejki

```
public int rozmiarKolejki() {  6 usages  new *
    kasaMutex.lock();
    try {
        return kolejka.size();
    } finally {
        kasaMutex.unlock();
    }
}
```

- Sygnalizacja statusu kasjera

```
public boolean czyPrzyjmujeKlientow() { //sprawdza czy kasa przyjmuje aktualnie klientow  7 usages  new *
    return otwarta.get() && kasjerDostepny.get() && !kasjerChcePrzerwe.get();
}
public void sygnalizujChęćPrzerwy() { // kasjer chce isc na przerwe - obsluguje reszte kolejki i idzie  1 usag
    kasjerChcePrzerwe.set(true);
    gui.updateKasaDisplay(id, kolejka.size(), czyPrzyjmujeKlientow(), kasjerDostepny.get());
    System.out.println("Kasa " + id + " nie przyjmuje nowych klientów - kasjer chce przerwę");
}

public void anulujChęćPrzerwyBezAktualizacjiGUI() {  1 usage  new *
    kasjerChcePrzerwe.set(false);
}
```

- Aktualizacja GUI

```
// aktualizacja wyświetlania
public void updateKasaDisplay(int kasaId, int rozmiarKolejki, boolean przyjmujeKlientow, boolean kasjerDostepny) {  6 usages  new *
    long currentTime = System.currentTimeMillis();
    Long lastUpdate = lastUpdateTime.get(kasaId);
    if (lastUpdate == null || currentTime - lastUpdate > MIN_UPDATE_INTERVAL) {
        SwingUtilities.invokeLater(() -> {
            if (kasaId < kasaPanels.size()) {
                JPanel panel = kasaPanels.get(kasaId);
                JLabel statusLabel = findLabelByName(panel,  name: "status");
                JLabel kolejkaLabel = findLabelByName(panel,  name: "kolejka");
                JLabel kasjerLabel = findLabelByName(panel,  name: "kasjer");
                if (statusLabel != null) {
                    String status;
                    Color backgroundColor;
                    if (!kasjerDostepny) { // zamiany kolorów w zależnosci od statusu kasy
                        status = "Kasjer na przerwie";
                        backgroundColor = Color.RED;
                    } else if (przyjmujeKlientow) {
                        status = "Otwarta";
                        backgroundColor = Color.GREEN;
                    } else {
                        status = "Nie przyjmuje nowych";
                        backgroundColor = Color.YELLOW;
                    }
                    statusLabel.setText(status);
                    panel.setBackground(backgroundColor);
                }
                    }
                    statusLabel.setText(status);
                    panel.setBackground(backgroundColor);
                }

                if (kolejkaLabel != null) {
                    kolejkaLabel.setText("Kolejka: " + rozmiarKolejki);
                }

                if (kasjerLabel != null) {
                    kasjerLabel.setText("Kasjer: " + (kasjerDostepny ? "Dostępny" : "Na przerwie"));
                }
            }
        });
        lastUpdateTime.put(kasaId, currentTime);
    }
}
```

5. Obiekty synchronizacji

- Semafory:

```
private Semaphore miejscaWSklepie;  2 usages
```

```
private final Semaphore obsluzony; // semafor do sygnalizacji zakończenia obsługi


public Klient(int czasObslugi) {  1 usage  new *
    this.id = nextId.getAndIncrement();
    this.czasObslugi = czasObslugi;
    this.obsluzony = new Semaphore( permits: 0); // poczatkowo zablokowany
}
```

- Mutexy

```
private final ReentrantLock kasaMutex; // Mutex do synchronizacji operacji na kasie  6 usages
private final GUI gui;  3 usages

public Kasa(int id, GUI gui) {  1 usage  new *
    this.id = id;
    this.kolejka = new ConcurrentLinkedQueue<>();
    this.otwarta = new AtomicBoolean( initialValue: true);
    this.kasjerDostepny = new AtomicBoolean( initialValue: true);
    this.kasjerChcePrzerwe = new AtomicBoolean( initialValue: false);
    this.kasaMutex = new ReentrantLock();
    this.gui = gui;
}
```

- Zmienne atomowe

```
private final AtomicBoolean otwarta;  2 usages
private final AtomicBoolean kasjerDostepny; // czy kasjer jest nie na przerwie  5 usages
private final AtomicBoolean kasjerChcePrzerwe;  5 usages
```

```
private final AtomicBoolean currentlyServing = new AtomicBoolean( initialValue: false);  4 us
private final AtomicBoolean onBreak = new AtomicBoolean( initialValue: false);  4 usages
```

```
 private final AtomicBoolean running;
```

```
private static final AtomicInteger nextId = new AtomicInteger( initialValue: 1);  1 usage
```

- Kolejka

```
private final ConcurrentLinkedQueue<Klient> kolejka; //  kolejka klientów
```

```
private final Map<Integer, Long> lastUpdateTime = new ConcurrentHashMap<>();  2 usages
private static final long MIN_UPDATE_INTERVAL = 100;  1 usage
```

- Scheduler do zarządzania wątkami

```
private final ScheduledExecutorService scheduler;
```

6. Procesy sekwencyjne

- Generowanie klientów:

```
public void start() { // generowanie klienta co czas okreslony w ustawieniach  1 usage  new *
    generatorTask = scheduler.scheduleAtFixedRate(this::sprobujDodacKlienta,
            initialDelay: 0, config.czasPrzychoduKlienta, TimeUnit.MILLISECONDS);
}
```

```
private void sprobujDodacKlienta() {  1 usage  new *
    if (!running.get()) {
        return;
    }

    if (miejscaWSklepie.tryAcquire()) { //nowy klient wchodzi do sklepu
        boolean klientDodany = false;

        try {        // losuje czas obsługi dla nowego klienta

            int czasObslugi = config.czasObslugiMin +
                    random.nextInt( bound: config.czasObslugiMax - config.czasObslugiMin + 1);
            Klient klient = new Klient(czasObslugi);
            Kasa wybranaKasa = wybierzNajkrotszaKolejke(); // szukanie najkrotszej kolejki
            if (wybranaKasa != null) {
                if (wybranaKasa.dodajKlienta(klient)) {
                    klientDodany = true;
                    // uruchamia proces klienta (zakupy + oczekiwanie na obsługę)
                    new ProcesKlienta(klient, miejscaWSklepie, gui, config.maxKlientow, scheduler).start();
                    gui.updateCustomerCount( currentCustomers: config.maxKlientow - miejscaWSklepie.availablePermits(), config.maxKlientow);
                    System.out.println(klient + " wszedł do sklepu (miejsca: " +
                            miejscaWSklepie.availablePermits() + "/" + config.maxKlientow + ")");
                }
            }
            if (!klientDodany) {
                miejscaWSklepie.release();
            }
        } catch (Exception e) {
            if (!klientDodany) {
                miejscaWSklepie.release();
            }
            System.err.println("Błąd przy dodawaniu klienta: " + e.getMessage());
        }
    }
}
```

- Proces kasjera

```
public void start() { // uruchomienie kasjera i sprawdzanie kolejki co 100ms  1 usage  new *
    mainTask = scheduler.scheduleAtFixedRate(this::sprawdzKolejke,  initialDelay: 0,  period: 100, TimeUnit.MILLISECONDS);
}
```

```
private void sprawdzKolejke() {  1 usage  new *
    try {
        if (!running.get() || !kasa.isKasjerDostepny().get() ||
                currentlyServing.get() || onBreak.get()) {
            return;
        }

        obsluzKolejkeZPrzerwami();
    } catch (Exception e) {
        System.err.println("Błąd w kasjerze " + kasa.getId() + ": " + e.getMessage());
    }
}
```

```java
private void obsluzKolejkeZPrzerwami() { //obsługa kolejki  1 usage  new *

    if (kasa.getKolejka().isEmpty() && kasa.isKasjerChcePrzerwe().get()) {  // jak kolejka pusta i kasjer chce przerwę
        kasa.anulujChęćPrzerwyBezAktualizacjiGUI();
        wykonajPrzerwe();
        return;
    }
    if (kasa.getKolejka().isEmpty() || !running.get() || !kasa.isKasjerDostepny().get()) { // jeśli brak klientów lub
        return;
    }
    // losowo decyduje czy chce przerwę z szansą co klienta
    if (!kasa.isKasjerChcePrzerwe().get() &&
            !kasa.getKolejka().isEmpty() &&
            random.nextDouble() < config.szansaNaPrzerwe) {

        kasa.sygnalizujChęćPrzerwy();
        System.out.println("Kasjer kasy " + kasa.getId() +
                " chce przerwę po obsłudze pozostałych " + kasa.rozmiarKolejki() + " klientów");
    }
    Klient klient = kasa.getKolejka().poll(); // obsluga klienta
    if (klient != null) {
        obsluzKlienta(klient);
    }
}
```

```java
private void obsluzKlienta(Klient klient) {  1 usage  new *
    if (currentlyServing.compareAndSet( expectedValue: false,  newValue: true)) {
        System.out.println("Kasjer kasy " + kasa.getId() + " obsługuje " + klient +
                " (pozostało w kolejce: " + kasa.rozmiarKolejki() +
                (kasa.isKasjerChcePrzerwe().get() ? ", nie przyjmuje nowych" : "") + ")");

        scheduler.schedule(() -> {
            try {
                klient.oznaczJakoObsluzony();
                System.out.println("Kasjer kasy " + kasa.getId() + " zakończył obsługę " + klient);

                SwingUtilities.invokeLater(() -> {
                    gui.updateKasaDisplay(kasa.getId(), kasa.rozmiarKolejki(),
                            kasa.czyPrzyjmujeKlientow(), kasa.isKasjerDostepny().get());
                });
            } finally {
                currentlyServing.set(false);
            }
        }, klient.getCzasObslugi(), TimeUnit.MILLISECONDS);
    }
}
```

- Proces klienta

```
    public void start() { // proces klienta  1 usage  new *
        //losuje czas zakupów (1-3 sekundy) - klient po wejsciu do sklepu nie idzie odrazu do kasy
        int czasZakupow = 1000 + random.nextInt( bound: 2000);

        shoppingTask = scheduler.schedule(() -> {
            System.out.println(klient + " zakończył zakupy i czeka w kolejce na obsługę");
            //czeka na obsluzenie
            CompletableFuture.runAsync(() -> {
                try {
                    klient.czekajNaObsluge();
                    System.out.println(klient + " został obsłużony i opuszcza sklep");
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    opuscSklep();
                }
            });
        }, czasZakupow, TimeUnit.MILLISECONDS);
    }

    private void opuscSklep() { //zwalnianie miejsca i aktualizacja gui  1 usage  new *
        miejscaWSklepie.release();
        SwingUtilities.invokeLater(() -> {
            gui.updateCustomerCount( currentCustomers: maxKlientow - miejscaWSklepie.availablePermits(), maxKlientow);
        });
        System.out.println(klient + " opuścił sklep (wolne miejsca: " +
                miejscaWSklepie.availablePermits() + ")");
    }
}
```

- Obsługa GUI

```java
public void updateKasaDisplay(int kasaId, int rozmiarKolejki, boolean przyjmujeKlientow, boolean kasjerDostepny) {  6 usages
    long currentTime = System.currentTimeMillis();
    Long lastUpdate = lastUpdateTime.get(kasaId);
    if (lastUpdate == null || currentTime - lastUpdate > MIN_UPDATE_INTERVAL) {
        SwingUtilities.invokeLater(() -> {
            if (kasaId < kasaPanels.size()) {
                JPanel panel = kasaPanels.get(kasaId);
                JLabel statusLabel = findLabelByName(panel, name: "status");
                JLabel kolejkaLabel = findLabelByName(panel, name: "kolejka");
                JLabel kasjerLabel = findLabelByName(panel, name: "kasjer");
                if (statusLabel != null) {
                    String status;
                    Color backgroundColor;
                    if (!kasjerDostepny) { // zamiany kolorów w zależnosci od statusu kasy
                        status = "Kasjer na przerwie";
                        backgroundColor = Color.RED;
                    } else if (przyjmujeKlientow) {
                        status = "Otwarta";
                        backgroundColor = Color.GREEN;
                    } else {
                        status = "Nie przyjmuje nowych";
                        backgroundColor = Color.YELLOW;
                    }
                    statusLabel.setText(status);
                    panel.setBackground(backgroundColor);
                }

                if (kolejkaLabel != null) {
                    kolejkaLabel.setText("Kolejka: " + rozmiarKolejki);
                }

                if (kasjerLabel != null) {
                    kasjerLabel.setText("Kasjer: " + (kasjerDostepny ? "Dostępny" : "Na przerwie"));
                }
            }
        });
        lastUpdateTime.put(kasaId, currentTime);
    }
}
```

- Główny proces symulacji

```java
public void startSimulation() { // start  1 usage  new *
    if (running.get()) return;

    running.set(true);
    gui.setStatus("Symulacja uruchomiona");
    refreshGUI();
    for (Kasjer kasjer : kasjerzy) {
        kasjer.start();
    }

    generatorKlientow.start();

    System.out.println("Symulacja uruchomiona z " + config.liczbKas +
            " kasami i max " + config.maxKlientow + " klientami");
}
```

```java
public void stopSimulation() { //stop  2 usages  new *
    if (!running.get()) return;

    running.set(false);
    gui.setStatus("Zatrzymywanie symulacji...");

    for (Kasjer kasjer : kasjerzy) {
        kasjer.stop();
    }

    if (generatorKlientow != null) {
        generatorKlientow.stop();
    }

    gui.setStatus("Symulacja zatrzymana");
    System.out.println("Symulacja zatrzymana");
    gui.updateCustomerCount( currentCustomers: 0, config.maxKlientow);
}

public void shutdown() { // zamkniecie i zwolnienie zasobów  1 usage  new *
    stopSimulation();
    mainScheduler.shutdown();
    try {

        if (!mainScheduler.awaitTermination( timeout: 5, TimeUnit.SECONDS)) {
            mainScheduler.shutdownNow();
        }
    } catch (InterruptedException e) {
        mainScheduler.shutdownNow();
        Thread.currentThread().interrupt();
    }
}
```