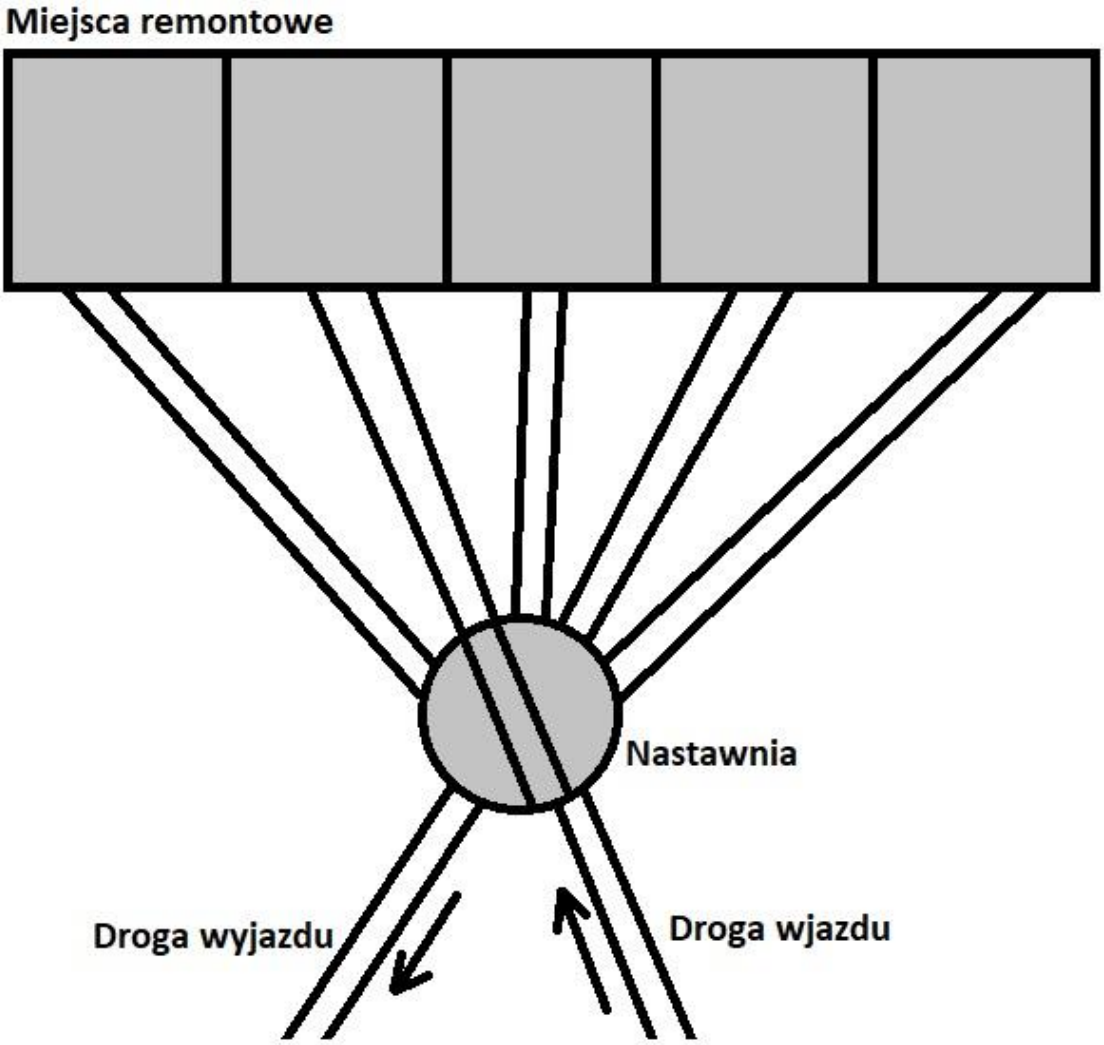
Sprawozdanie z projektu
Zadanie nr: PW-13/2021
„Parowozownia”

Opracowanie: Patrycja Baczewska

Problem do rozwiązania:
Parowozownia.

Założenia:
W pewnej parowozowni istnieje M miejsc remontowych (równocześnie może być naprawianych M lokomotyw). Naprawa taka trwa pewien nieznany, ale skończony okres czasu, po czym lokomotywa opuszcza parowozownię i rozpoczyna się jej normalna eksploatacja. Cykl ten powtarza się. Aby wjechać do parowozowni, należy przejechać przez obrotową nastawnię. Do nastawni prowadzą jedynie dwa tory, którymi można wjechać na teren parowozowni. Należy zagwarantować, że nie dojdzie do sytuacji, w której na jednym odcinku toru znajdą się 2 lokomotywy, gdyż grozi to zderzeniem. Należy założyć że pracownik obsługujący nastawnię posiada wiedzę, która pozwala mu operować nastawnią w ten sposób, iż wybierane będą, o ile są dostępne, wolne tory prowadzące do parowozowni lub do bram. Co pewien czas w parowozowni musi zostać przeprowadzony remont. Decyzję o tym podejmuje szef robót remontowych. On także decyduje o jego zakończeniu. Dla bezpieczeństwa robotników, w czasie remontu żadna lokomotywa nie może znajdować się na terenie parowozowni.

Syntetyczny opis problemu – w tym wszystkie przyjęte założenia:



Do parowozowni prowadzi jedna droga przechodząca przez nastawnię. Tam jest jednocześnie pierwsza i ostatnia synchronizacja obiektu Pociąg. Pociągi zgłaszają swoją chęć wejścia do hangaru i ustawiają się w kolejce przed nastawnią. Po przejściu przez nastawnię pociąg kierowany jest do wolnego miejsca remontowego. W hangarze może być maksymalnie tyle pociągów ile miejsc remontowych. Po remoncie pociąg kieruje się do nastawni, bez kolizji wyjeżdża z terenu parowozowni drogą wyjazdu.
Co pewien czas szef parowozowni zarządza remont. Jeśli zgłosi taką chęć to od tego momentu żaden z oczekujących pociągów nie wjeżdża do parowozowni, a obecnie znajdujące się w hangarze wyjeżdżają po ukończeniu naprawy. Remont parowozowni trwa skończoną liczbę

czasu i po jego upłynięciu parowozownia jest znowu otwierana. W moim programie przyjęłam liczbę miejsc w hangarze równą 5. Parowozownia ta obsługuje 12 pociągów które podczas całej swojej eksploatacji 5 razy przyjeżdżają do naprawy. Szef również zgłasza chęć remontu hangaru 5 razy. Czas trwania remontów jest losowany z określonego przedziału.

Wykaz współdzielonych zasobów oraz punktów synchronizacji:

Pierwszym współdzielonym zasobem jest dostęp pociągów do nastawni. Tam też jest pierwszy punkt synchronizacji.
Kolejnym współdzielonym zasobem są miejsca remontowe w hangarze. W nich również zachodzi proces synchronizacji.
Ostatnim punktem synchronizacji jest kolejka pociągów oraz szefa, który może ją przerwać zgłaszając żądanie naprawy hangaru.

Wykaz obiektów synchronizacji:

W programie synchronicznie działają ze sobą pociągi oraz szef.

Wykaz procesów sekwencyjnych:

Dla pociągu: zgłoszenie chęci remontu, oczekiwanie w kolejce na wjazd do hangaru, przejazd przez nastawnię, remont w parowozowni, wyjazd z hangaru, przejazd przez nastawnię

Dla szefa: zgłoszenie chęci remontu hangaru, poczekanie aż wszystkie pociągi opuszczą parowozownię, zajęcie hangaru na pewien czas, zwolnienie hangaru

Listing programu (wraz z komentarzami pomocniczymi w kodzie):

```
public class Main {

    public static void main(String[] args) {
        Hangar hangar = new Hangar();

        // tablica na obiekty ktore moga przebywac w hangarze
        ProblemObject []objects = new ProblemObject[15];

        // tworzenie pociagow
        objects[0] = new Train(hangar, "Tomek", 10,30, 5);
        objects[1] = new Train(hangar, "Norman", 10,30, 5);
        objects[2] = new Train(hangar, "Edek", 10,30, 5);
        objects[3] = new Train(hangar, "Basia", 10,30, 5);
        objects[4] = new Train(hangar, "Rysio", 10,30, 5);
        objects[5] = new Train(hangar, "Gabrys", 10,30, 5);
        objects[6] = new Train(hangar, "Emilka", 10,30, 5);
        objects[7] = new Train(hangar, "Diesel 10", 10,30, 5);
        objects[8] = new Train(hangar, "Kuba", 10,30, 5);
        objects[9] = new Train(hangar, "Artur", 10,30, 5);
        objects[10] = new Train(hangar, "Ksiezna", 10,30, 5);
        objects[11] = new Train(hangar, "Marcin", 10,30, 5);

        // tworzenie szefow
        objects[12] = new Boss(hangar, "Krzysztof Jarzyna",
30,50, 5);
        //objects[13] = new Boss(hangar, "Pan Zabka", 30,50,
5);

        // petla uruchamia wszystkie watki
        for(int i=0; i<13; ++i){
            objects[i].start();
        }
    }
}
```

```
public abstract class ProblemObject extends Thread{

    // imie obiektu
    protected final String name;

    // opoznienie pomiedzy wykonywaniem operacji w ms
    protected final int delayInMS;

    // hangar z ktorego bedzie korzystal obiekt
```

```
    protected final Hangar hangar;

    // ilosc prob wejsc do hangaru. kazdy obiekt zanim
zakonczy dzialanie wejdzie tryCount razy
    protected final int tryCount;

    public ProblemObject(Hangar hangar, String name, int
lowerMSBound, int upperMSBound, int tryCount){
        this.name = name;
        this.tryCount = tryCount;
        this.delayInMS = Math.max(0, new
Random().nextInt(lowerMSBound, upperMSBound));
        this.hangar = hangar;
    }

    public ProblemObject(Hangar hangar, String name){
        this(hangar, name, 20, 50, 5);
    }

    protected void enter() throws Exception{
        hangar.add(this); // dodanie obiektu do hangaru ->
logika dodawnia odbywa sie wewnatrz funkcji add hangaru
        Thread.sleep(delayInMS); // opoznienie w dodaniu
    }

    protected void fix() throws InterruptedException{
        Thread.sleep(delayInMS); // opoznienie w
przeprowadzaniu naprawy/kontroli
    }

    protected void leave() throws Exception{
        hangar.remove(this); // usuniecie obiektu z hangaru ->
logika usuwania odbywa sie wewnatrz funkcji remove hangaru
        Thread.sleep(delayInMS); // opoznienie w usunieciu
    }

    @Override
    public void run() {
        try{
            int counter = 0;
            // glona petla watku
            do {
                enter(); // watek wchodzi do hangaru. metoda
konczy dzialanie gdy uda sie wejsc inaczej watek ktoremu nie
udalo sie wejsc czeka
                fix(); // odczekanie na naprawe/kontrole
                leave(); // wyjdzie z hangaru
                Thread.sleep(2L * delayInMS); // opoznienie w
kolejnej probie dolaczenia do hangaru
            } while (++counter < tryCount);
        }catch(InterruptedException e){
```

```
            System.out.println("Thread was interrupted.
Message: " + e.getMessage());
            Thread.currentThread().interrupt(); // ponowne
przerwanie watku ktory zostal juz przerwany
        }catch(Exception e){
            System.out.println("Unexpected exception occurred.
The exception message: " + e.getMessage());
        }
    }

    @Override
    public String toString() {
        return "Object{" + "name: '" + name + "'}";
    }
}
```

```
public class Train extends ProblemObject{

    public Train(Hangar hangar, String name, int lowerMSBound,
int upperMSBound, int tryCount){
        super(hangar, name, lowerMSBound, upperMSBound,
tryCount);
    }

    public Train(Hangar hangar, String name){
        super(hangar, name);
    }

    @Override
    public String toString() {
        return "Train{" + "name: '" + name + "'}";
    }
}
```

```
public class Boss extends ProblemObject {

    public Boss(Hangar hangar, String name, int lowerMSBound,
int upperMSBound, int tryCount){
        super(hangar, name, lowerMSBound, upperMSBound,
tryCount);
    }

    public Boss(Hangar hangar, String name){
        super(hangar, name);
    }

    @Override
    public String toString() {
        return "Boss{" + "name: '" + name + "'}";
```

```
    }
}
```

```
public class Gate {

    // opoznienie w przechodzeniu przez brame w ms
    private final int delayInMS;

    // kolejka osob czekajacych na przejscie przez brame
    private final Queue<ProblemObject> queue;

    // obiekt aktualnie przechodzacy przez brame
    private ProblemObject currentlyInTheGate;

    public Gate(int delayInMS){
        this.delayInMS = delayInMS;
        this.queue = new LinkedList<>();
        this.currentlyInTheGate = null;
    }

    // metoda synchronized -> tylko 1 watek moze w tym samym
czasie korzystac z tej metody
    public synchronized void goThrough(ProblemObject object,
String prefix) throws InterruptedException{
        if(object == null) return;

        // dodawanie obiektu do kolejki
        queue.add(object);
        while(true){ // nieskonczona petla do momentu gdy uda
sie przejsc przez brame
            if(queue.peek() == object && currentlyInTheGate ==
null){ // jesli brama jest pusta i object to osoba ktora
aktualnie ma przechodzic przez brame to:
                queue.remove(); // usun pierwsza osobe z
kolejnki
                currentlyInTheGate = object; // ustaw nowa
osobe ktora przechodzi przez brame
                Thread.sleep(delayInMS); // poczekaj az osoba
przejdzie przez brame

                System.out.println(prefix + this); //
wyswietlw informacje o przejsciu

                currentlyInTheGate = null; // zwolnij brame
                notifyAll(); // powiadom inne czekajace watki
ze brama jest juz wolna
                break;
            }else{
                wait(); // zatrzymaj watek do momentu
wywolania notifyAll
```

```
            }
        }
    }

    @Override
    public String toString() {
        return "Gate: " + currentlyInTheGate;
    }
}
```

```
public class Hangar {

    // okresla ilosc pociagow ktore moga naraz byc w hangarze
    public static final int N = 5;

    // enum ktory przechowuje informacje na temat rodzaju
kolejki
    // INSIDE -> obiektu wewnatrz hangaru
    // QUEUE -> obiekty ktore oczekuja na wejscie
    public enum Where{
        INSIDE, QUEUE
    }

    // enum ktory okresla typ obiektu ktory moze sie pojawic w
hangarze
    public enum What{
        TRAIN, BOSS
    }

    // brama przez ktora trzeba przejsc wchodzac i wychodzac z
hangaru
    private final Gate gate;

    // kolejka oczekujacych na wejscie do hangaru
    private final List<ProblemObject> queue;

    // lista ktora przechowuje informacje na temat obiektow
znajdujacych sie wewnatrz hangaru
    private final List<ProblemObject> inside;

    public Hangar(){
        gate = new Gate(10);
        queue = new ArrayList<>();
        inside = new ArrayList<>();
    }

    // metoda ma za zadanie policzyc ilosc wystapien podanego
typu obiektow w okreslonym miejscu
    public int count(What what, Where where){
        // wybranie listy w ktorej bedzie wykonywane
```

```
obliczanie. kolejka oczekujacych lub lista z obiektami w
hangarze
        List<ProblemObject> source = switch(where){
            case QUEUE -> queue;
            case INSIDE -> inside;
        };

        // obliczenie wystapien podanego typu obiektow
        int counter = 0;
        switch(what){
            case BOSS -> {
                for(ProblemObject o : source){
                    if(o instanceof Boss) ++counter; // jesli
o jest typu Boss to zwieksz counter
                }
            }
            case TRAIN -> {
                for(ProblemObject o : source){
                    if(o instanceof Train) ++counter;
                }
            }
            default -> {}
        }
        return counter; // zwrocenie obliczonej wartosci
    }

    public synchronized void add(ProblemObject object) throws
Exception{
        if(object == null) return;

        queue.add(object); // dodanie obiektu do kolejki
oczekujacych
        while(true){ // petla do momentu dodania do hangaru
            if(isAvailableFor(object)){ // funckja sprawdza
czy mozna dany obiekt dodac do hangaru -> opisana nizej
                queue.remove(object); // usuniecie elementu z
kolejki oczekujacych
                gate.goThrough(object, "Entering through the
gate. "); // przejscie przez brame -> cale przejscie przez
brame jest w metodzie synchronized zatem przed brama nie beda
sie tworzyc kolejki
                inside.add(object); // dodanie obiektu do list
obiektow znajdujacych sie wewnatrz hangaru
                System.out.println(this); // wypisanie danych
na temat hangaru
                break;
            }else{
                wait(); // zatrzymanie watku gdy nie mozna
dolaczyc
            }
        }
```

```
    }

    public synchronized void remove(ProblemObject object)
throws Exception{
        inside.remove(object); // usuniecie obiektu z list
obiektow znajdujacych sie wewnatrz hangaru
        gate.goThrough(object, "Exiting through the gate. " );
// wyjscie przez brame
        notifyAll(); // powiadomienie innych oczekujacych
watkow
    }

    @Override
    public String toString() {
        return "Hangar status: Inside (bosses: " +
count(What.BOSS, Where.INSIDE) + " trains: " +
count(What.TRAIN, Where.INSIDE) + " ) " +
                "Queue (bosses: " + count(What.BOSS,
Where.QUEUE) + " trains: " + count(What.TRAIN, Where.QUEUE) +
" )";
    }

    private boolean isAvailableFor(ProblemObject object)
throws InvalidAttributeValueException{
        if(object instanceof Boss){ // jesli obiekt to szef
to:
            return inside.isEmpty(); // sprawdz czy hangar
jest pusty
        }else if(object instanceof Train){ // jesli obiekt to
pociag to:
            return count(What.BOSS, Where.INSIDE)==0 &&
count(What.BOSS, Where.QUEUE)==0 && inside.size() < N; //
sprawdz w hangarze nie ma szefa && sprawdz czy w kolejce nie
ma szefa && sprawdz czy jest miejsce w hangarze
        }else{
            throw new InvalidAttributeValueException("The
class object is neither Boss nor Train"); // obiekt inny niz
pociag i szef -> wyrzuc wyjatek
        }
    }
}
```