# Raspoređivač i izvršitelj asinkronih zadataka

**Bošnjak, Mateo**

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

**Mateo Bošnjak**

# Raspoređivač i izvršitelj asinkronih zadataka

Diplomski rad

Pula, rujan, 2019. godine

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

**Mateo Bošnjak**

# Raspoređivač i izvršitelj asinkronih zadataka

Diplomski rad

**JMBAG:** 0248039345, redoviti student
**Studijski smjer:** Informatika

**Predmet:** Suvremene tehnike programiranja
**Znanstveno područje:** Društvene znanosti
**Znanstveno polje:** Informacijske i komunikacijske znanosti
**Znanstvena grana:** Informacijski sustavi i informatologija
**Mentor:** doc. dr. sc. Siniša Sovilj
**Komentor:** dr. sc. Nikola Tanković

Pula, rujan, 2019. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Mateo Bošnjak, kandidat za magistra informatike ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomksog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, rujan, 2019. godine

IZJAVA
o korištenju autorskog djela

Ja, Mateo Bošnjak dajem odobrenje Sveučilištu Jurja Dobrile
u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom „Raspoređivač i izvršitelj asinkronih zadataka“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu diplomskih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.
Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, rujan, 2019. godine

Potpis

_____

# Sadržaj

# 1. Uvod

Od prvog računala namijenjenog za opću upotrebu, do uređaja veličine dlana koji čovjeku pruža sve znanje ovog svijeta u istom trenutku, prošlo je samo 70-ak godina. Kako napreduje tehnologija tako i napreduje sve oko nje. Danas svaki korisnik želi imati sve dostupno u što kraćem roku, inače gubi interes i pronalazi zamjenski proizvod.

Jedan od načina postizanje velike brzine dostupnosti sadržaja je asinkroni model programiranja. Asinkroni model programiranja dozvoljava izvršavanje većeg broja radnji u isto vrijeme. Što znači da glavni program može nastaviti sa svojim radom dok se ostale radnje izvršavaju uz njega. Kada se neka radnja završi, glavni program dobije informaciju o tome. Kod sinkronog modela programiranja radnje se izvršavaju jedna za drugom. Što bi značilo da glavni program mora čekati dok se neka radnja ne izvrši da bi nastavio dalje sa svojim radom. (Oaks & Wong, 2004)

U ovom radu prikazana je biblioteka pod nazivom Beeq, koja služi kao asinkroni izvršitelj i raspoređivač zadataka. Opisani su koncepti sinkronog i asinkronog modela programiranja, dretve te načini na koji se kod može sigurno izvoditi sa većim brojem dretvi. Također je prikazana implementacija spomenute biblioteke.

Drugo poglavlje opisuje konkurentnost. Govori o sinkronom i asinkronom izvođenju algoritma. Daje uvod u temu dretvi te njihovo korištenje u današnjem programiranju. Ilustrira koncepte sinkronog programiranja, asinkronog programiranja te dretvi kroz primjere koda.

Treće poglavlje dublje ulazi u dretve. Prikazuje prednosti i mane dretvi kroz koncepte kao što su stanje nadmetanja (*race condition*), vidljivost memorije (*memory visibility*), zastoj (*dead lock*) te *reentrant* ključevi. Opisuje kako napisati kod koji je siguran za interakciju sa većim brojem dretvi.

Četvrto poglavlje prikazuje kolekcije i klase u Javi koje su napravljene u svrhu sigurne interakcije sa više dretvi. Klase kao što su *Atomic Integer*, *Executor Service* te *Copy on Write* kolekcije.

Peto poglavlje sadrži klasne dijagrame koji prikazuju veze između klasa, te njihove atribute i funkcije.

U šestom poglavlju su prikazani slijedni dijagrami. Slijedni dijagrami prikazuje interakciju između objekata prikazanih u vremenskom slijedu. Opisuje različite scenarije među objektima i poruke koje se šalju između njih.

Sedmo poglavlje prikazuje jednostavnu programsku implementaciju biblioteke.

Zadnje poglavlje opisuje tehnologije koje su korištene prilikom razvoja biblioteke.

# 2. Konkurentnost

U računalnoj znanosti, konkurentnost predstavlja sposobnost različitih dijelova programa, algoritma ili problema, redoslijedno ili djelomično redoslijedno izvršavanje bez utjecaja na konačni ishod. Konkurentnost omogućuje paralelno izvršavanje istodobnih jedinica, što može značajno poboljšati ukupnu brzinu izvršavanja u višeprocesorskim sustavima. (Goetz, i dr., 2006)

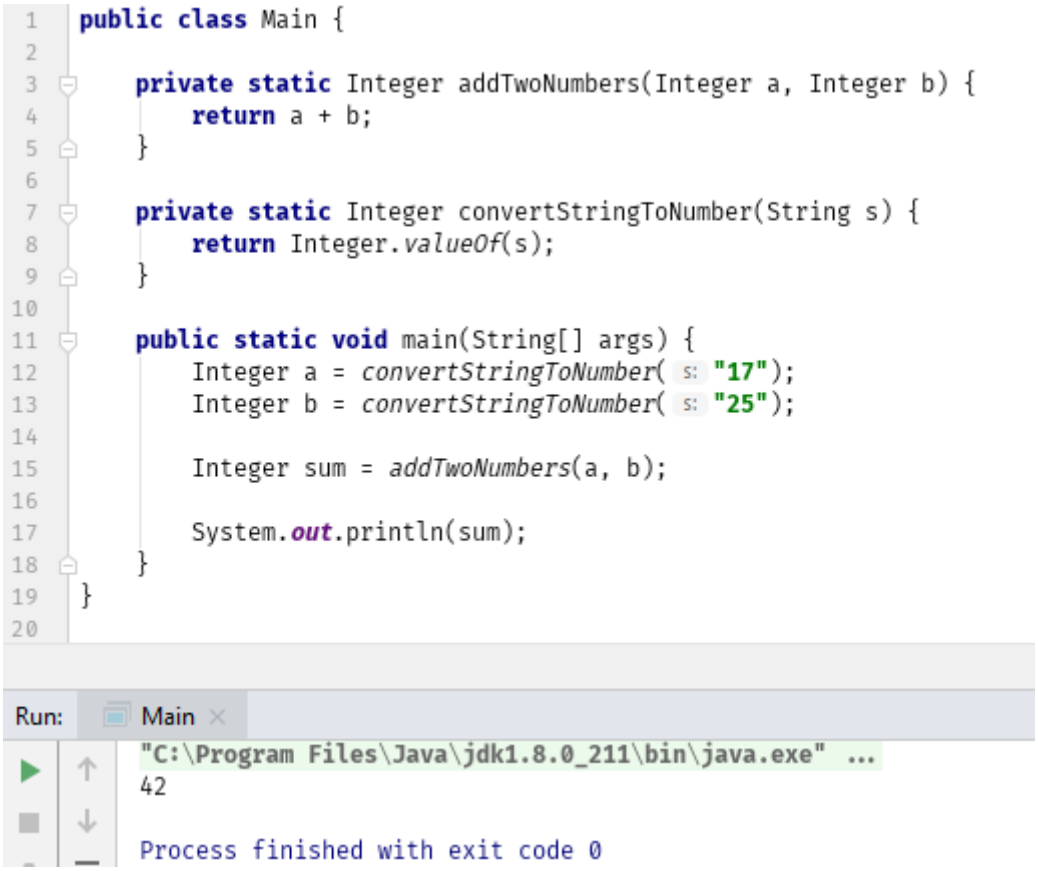
Postoji znatna razlika između konkurentnog i paralelnog izvršavanja programa. Konkurentnost je mogućnost rješavanja velikog broja radnji odjednom, dok je paralelnost izvršavanje velikog broja radnji u isto vrijeme. Bolje objašnjene prikazuje slika broj jedan.


Slika 1. Razlika paralelnog i konkurentnog izvršavanja

## 2.1 Sinkrono izvođenje algoritma

Prilikom sinkronog izvršavanje programa, program čeka na završetak jedne radnje prije nego što krene na sljedeću. Što znači da se radnje izvršavaju jedna za drugom. Ako je radnja kompleksna, program će dugo vremena provesti čekajući da se radnja završi. Sinkroni način izvršavanja programa se koristi onda kada je bitan redoslijed izvršavanja radnji. (Bloch, 2018)

Primjer takvog programa je prikazan na slici dva.

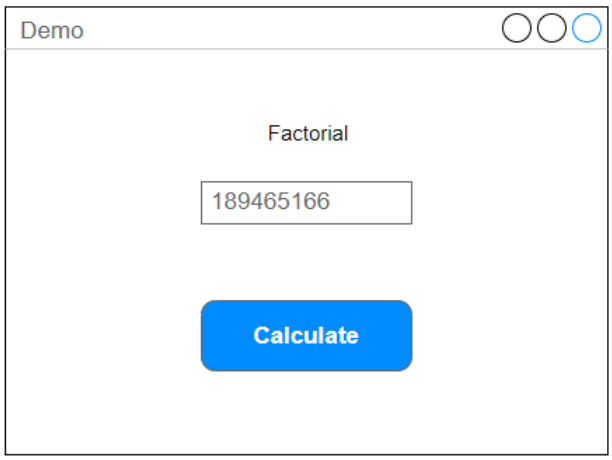



Slika 2. Primjer sinkronog programa

Program pretvara dvije vrijednosti tipa *String* u tip *Integer*. Nakon pretvorbe ih zbroji te ispiše rezultat. Slijed radnji se odvija sinkrono. Što znači da će se prvo izvršiti pretvorba vrijednosti iz tipa *String* u tip *Integer* te nakon toga zbrajanje. Kod ovog programa je bitan slijed naredbi jer program želi zbrojiti dvije vrijednosti tipa *Integer* a ne tipa *String*.

Sljedeći primjer na slici tri, pokazuje poteškoće s kojima se može susresti kada je program napisan sinkrono.

```
private static Integer calculateFactorial(Integer n) {
    if (n == 0) {
        return 1;
    }
    return n * calculateFactorial(--n);
}
```




Slika 3. Primjer sinkronog korisničkog sučelja

Interakcija s programom se vrši putem grafičkog sučelja. Na grafičkom sučelju se nalazi polje za unos broja, te gumb koji poziva metodu za računanje faktorijela unesenog broja. Vrijeme potrebno za računanje se povećava eksponencijalno. Problem koji će se javiti u ovom primjeru je sljedeći. Ako korisnik unese broj kod kojeg će računanje faktorijela broja potrajati par sekundi, cijelo grafičko sučelje će zablokirati i korisnik ga neće moći koristit sve dok računanje ne završi. U prikazanom primjeru korisnik zapravo ni nema drugih radnji koje može obavljati dok se računanje odvija. Kada bi prikazano grafičko sučelje imalo opciju pregledavanja povijesti svih izračuna, korisnik ne bi mogao koristiti tu opciju dok se računanje ne završi. U kasnijim poglavljima dotaknuti će se rješenje ovog problema.

Kada koristiti sinkroni model programiranja

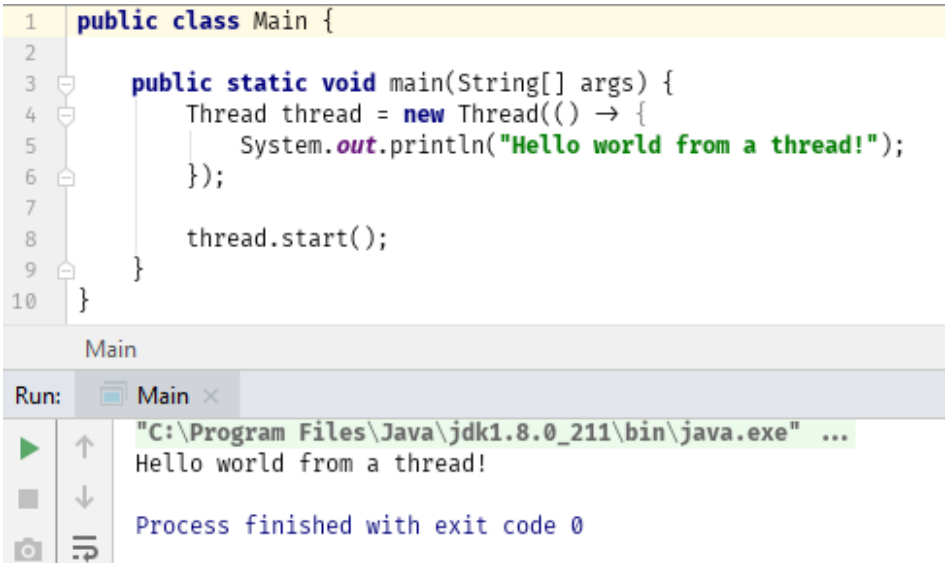
- radnje ovise jedna o drugoj tj. moraju se izvršiti slijedno
- problem nije kompleksan
- brzina nije pretjerano bitna
- potrebno je jednostavno rješenje

## 2.2 Uvod u dretve

Izvršni program je niz instrukcija mikroprocesoru smještenih u datoteku. Pored instrukcija izvršni program obično sadrži i podatke. Proces je instanca izvršnog programa koja se izvršava u memoriji računala. Različiti procesi izvršavaju se prividno paralelno tako što operacijski sustav alocira hardverske resurse svakom pojedinom procesu tek jedan kratak vremenski interval. Na višeprocesorskim računalima paralelizam može biti realan. (Herlihy, 2012)

Dretve predstavljaju oblik paralelizacije na razini procesa. Svaki se proces može podijeliti u jedan broj dretvi koje se izvršavaju prividno paralelno, na isti način kao i procesi. Za razliku od procesa, sve dretve unutar jednog procesa dijele zajedničku memoriju što omogućava efikasnu komunikaciju između različitih dretvi. (Kleiman, 1996)

Na slici broj četiri je prikazan primjer korištenja dretvi u programskom jeziku Java.




Slika 4. Primjer programa koji koristi dretve

Program je vrlo jednostavan. U *main* metodi se kreira novi objekt tipa *Thread*, te se kao argument predaje *lambda* u kojoj je naredba za ispisivanje poruke u konzolu. Nad kreiranim objektom se poziva metoda start, koja kreira novu dretvu u te izvršava njene naredbe.

## 2.3 Asinkrono izvođenje algoritma

Asinkroni model omogućuje istovremeno izvršavanje većeg broja radnji. (Bloch, 2018) Kada se pokrene jedna radnja, program neće čekati na njen kraj nego će nastaviti dalje. Kada se radnja završi program će dobiti „obavijest“ o njenom završetku i eventualnom rezultatu.

Na slici broj pet je prikazan primjer asinkronog programa.

```
1  public class Main {
2
3      public static void main(String[] args) {
4          final Integer a = 10;
5          final Integer b = 7;
6
7          Thread additionThread = new Thread(() → {
8              System.out.printf("%d + %d = %d\n", a, b, a + b);
9          });
10
11         Thread subtractionThread = new Thread(() → {
12             System.out.printf("%d - %d = %d\n", a, b, a - b);
13         });
14
15         additionThread.start();
16         subtractionThread.start();
17     }
18 }
```

```
Run:  Main
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
10 - 7 = 3
10 + 7 = 17

Process finished with exit code 0
```

Slika 5. Primjer asinkronog programa

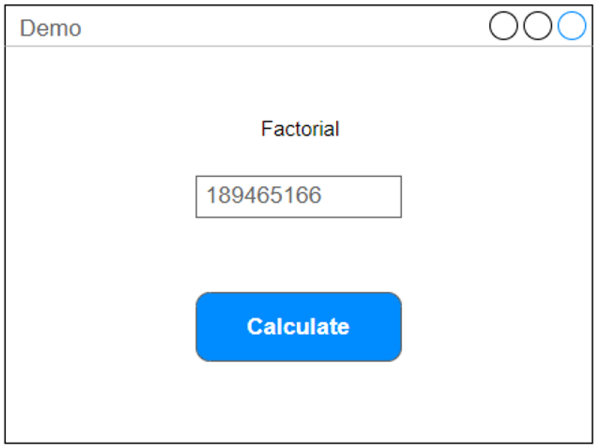
Program kreira dva objekta tipa *Thread*. U jednom objektu se nalazi naredba koja će ispisati zbroj dva broja, a u drugom objektu se nalazi naredba koja će ispisati razliku dva broja. Na posljetku se pokreću obje dretve. U konzoli se nalazi ispis dretvi. Može se primijetiti da je dretva za zbroj pokrenuta prije dretve za razliku, ali se ipak rezultat dretve za razliku ispisao prije dretve za zbroj. Uzrok tome je asinkrono izvođenje dretvi. Obe dretve se izvode u isto vrijeme tako da svaka dretva ima jednaku šansu da prva završi sa svojom radnjom.

Asinkroni model programiranja predstavlja rješenje za problem koji je prethodno prikazan na slici broj tri. Pomoću asinkronog modela programiranja metodu koja je izračunava faktorijele je moguće pozvati u zasebnoj dretvi. Što znači da glavana dretva, koja je zaslužna za prikaz grafičkog sučelja, neće biti blokirana dok se čeka izračun metode za faktorijele. Na taj način bi korisnik nesmetano mogao koristit ostale radnje na grafičkom sučelju.

```
public class Main {

    private static Integer calculateFactorial(Integer n) {
        if (n == 0) {
            return 1;
        }
        return n * calculateFactorial(--n);
    }

    public static void main(String[] args) {
        Integer n = 189465166;
        Thread thread = new Thread(() -> calculateFactorial(n));
        thread.start();
    }
}
```




Slika 6. Primjer asinkronog korisničkog sučelja

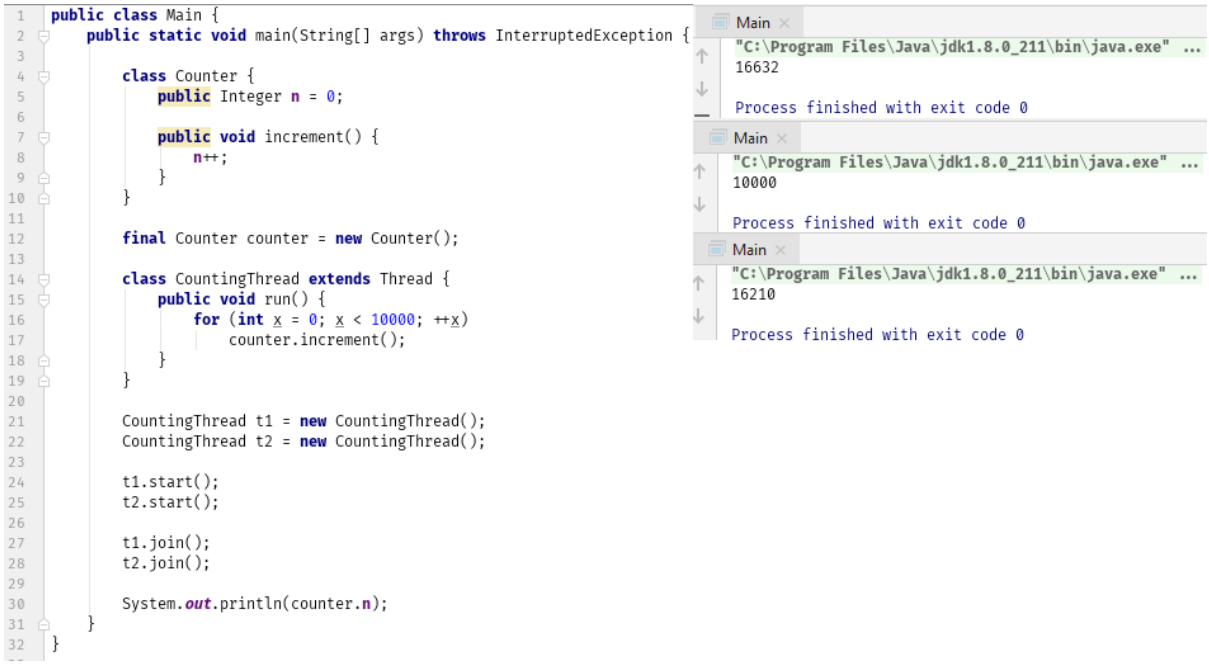
Kada koristit asinkroni model programiranja

- radnje su neovisne tj. ne moraju se izvršiti slijedno
- performanse su bitne (paralelizam znatno ubrzava program)
- kada program mora odgovarati na razne događaje

# 3. Dretve

## 3.1 Stanje nadmetanja

Kada veći broj dretvi pristupa zajedničkoj memoriji, vrlo lako se desi da obje dretve to učine u isto vrijeme. Takva situacija se izbjegava uzajamnim isključivanjem putem ključeva, gdje ključ može držati samo jedna dretva.




Slika 7. Primjer interakcije dretvi

Slika broj sedam prikazuje primjer programa gdje dvije dretve imaju međusobnu interakciju. Klasa *Counter* kao svojstvo ima varijablu n, te metodu koja povećava vrijednost varijable n za jedan. U nastavku svaka dretva poziva metodu *increment* deset tisuća puta. Svaki put kada se pokrene ovaj program, rezultat će biti drugačiji od zadnjeg pokretanja. Na desnoj strani slike su prikazana tri uzastopna pokretanja programa. Svaki od tri rezultata je drugačiji. Naziv za ovakvo ponašanje je stanje nadmetanja (Butcher, 2014).

Ponašanje ovog programa je sljedeće. Obje dretve pozovu metodu *increment*, prva dretva pročita vrijednost n, npr. 42, te ju krene povećavati. Prije nego je dretva broj jedan povećala vrijednost n, druga dretva ju je već pročitala. Stvorila se situacija gdje

obje dretve imaju istu vrijednost za varijablu n, te će obje tu vrijednost povećati na isti broj. Posljedica toga je povećanje vrijednosti n jedanput umjesto dvaput.

Rješenje za ovaj problem predstavljaju ključevi. Jedan takav način rješenja je ključna riječ *synchronized*. Stavljanjem ključne riječi *synchronized* ispred tipa metode, metoda postaje sinkronizirana. Što znači da se koristi unutrašnji ključ (*intrinsic lock*) kojeg sadrži svaki objekt u programskom jeziku Java. (Goetz, i dr., 2006)

```
public class Main {
    public static void main(String[] args) throws InterruptedException {

        class Counter {
            public Integer n = 0;

            public synchronized void increment() {
                n++;
            }
        }

        final Counter counter = new Counter();

        class CountingThread extends Thread {
            public void run() {
                for (int x = 0; x < 10000; ++x)
                    counter.increment();
            }
        }

        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println(counter.n);
    }
}
```

```
Main
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
20000

Process finished with exit code 0
```

```
Main
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
20000

Process finished with exit code 0
```

```
Main
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
20000

Process finished with exit code 0
```

Slika 8. Primjer korištenja ključne riječi *synchronized*

Slika broj osam prikazuje primjer koji koristi ključnu riječ *synchronized*. Sada je rezultat svaki put dvadeset tisuća. Sada dretva uzima ključ kada pozove metodu *increment*. Što znači da će metoda *increment* u isto vrijeme moći biti izvršena samo od strane jedne dretve. Svaka druga dretva koja pokuša pozvati metodu *increment* će preći u blokirano (*blocking*) stanje, te će se nastavit izvršavat samo kada dretva ispusti ključ koji je prethodno uzela.

## 3.2 Vidljivost memorije

Primjer na slici broj devet savršeno objašnjava problem vidljivosti memorije (Butcher, 2014).

```
1  public class Main {
2      static boolean isReady = false;
3      static int answer = 0;
4
5      static Thread t1 = new Thread(() → {
6          answer = 42;
7          isReady = true;
8      });
9
10     static Thread t2 = new Thread(() → {
11         if(isReady) {
12             System.out.println("The meaning of life is " + answer);
13         } else {
14             System.out.println("I don't know the answer");
15         }
16     });
17
18     public static void main(String[] args) throws InterruptedException {
19         t1.start();
20         t2.start();
21         t1.join();
22         t2.join();
23     }
24 }
```

Slika 9. Primjer problema vidljivosti memorije

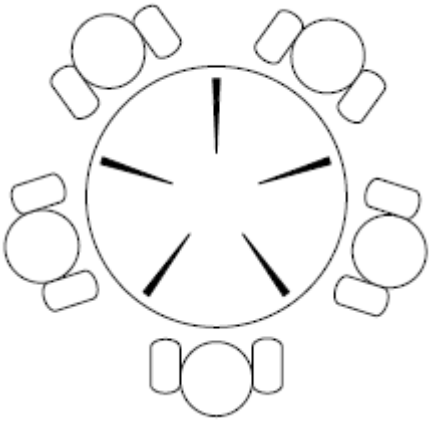
Rezultat ovog programa može biti jedan od sljedećih:

1. „*The meaning of life is 42*“,
2. „*I don't know the answer*“
3. „*The meaning of life is 0*“

Na prvi pogled je zadnji rezultat naprosto nemoguć. Ali se ipak može dogoditi. Razlog tome je optimizacija koda. Sam kompajler, JVM i hardware mogu premještati linije koda radi bolje optimizacije. Ovakve stvari nažalost nije moguće izbjeći jer su one zaslužne za ogromni napredak u informatici u proteklih nekoliko godina. (Subramaniam, 2011)

## 3.3 Zastoj

Sljedeći problem koji se može javiti u programiranju dretvama je zastoj (Butcher, 2014). Zastoj može biti demonstriran često korištenim primjerom gladni filozofi (*dining philosopohers)*.



Slika 10. Ilustracija problema gladni filozofi

Na slici broj 10 prikazana ilustracija problema gladni filozofi. Za stolom sjedi pet filozofa i na stolu se nalazi pet štapića za jelo. Filozof može razmišljati ili biti gladan. Ako je filozof gladan uzeti će štapiće za jelo, te će početi jesti. Kada je gotov vratiti će štapiće nazad na stol.

```java
public class Philosopher extends Thread {
    private Chopstick left, right;
    private Random random;

    public Philosopher(Chopstick left, Chopstick right) {
        this.left = left;
        this.right = right;
        random = new Random();
    }

    public void run() {
        try{
            while(true) {
                Thread.sleep(random.nextInt( bound: 1000));
                synchronized (left) {
                    synchronized (right) {
                        Thread.sleep(random.nextInt( bound: 1000));
                    }
                }
            }
        } catch (InterruptedException e) {}

    }
}
```

Slika 11. Implementacija problema gladni filozofi

Slika broj 11 prikazuje implementaciju problema filozofa. U glavnom programu se nalazi pet ovakvih filozofa, *Philosopher*, i ukupno pet različitih štapića za jelo, *Chopstick*. Program može biti pokrenut danima bez ikakvog prekida, ali kad-tad će se desiti sljedeća situacija. Svih pet filozofa će postati gladni i odlučiti će da žele jesti. Svaki od njih će brzo uzeti po jedan štapić za jelo, te će nastati situacija u kojoj svaki filozof ima jedan štapić za jelo. Takva situacija se naziva zastoj. Svaki filozof ima jedan štapić i svaki od njih čeka da netko drugi spusti štapić. Problem je što to nitko neće napraviti i program će ostati blokiran zauvijek.

Rješenje za ovakav problem je sljedeći. Ključevi, u ovom slučaju štapići za jelo, se uvijek uzimaju u fiksnom globalnom poretku. Slika broj 12 prikazuje jedan način da se postigne takvo rješenje.

```java
public class Philosopher extends Thread {
    private Chopstick first, second;
    private Random random;

    public Philosopher(Chopstick left, Chopstick right) {
        if (left.getId() < right.getId()) {
            this.first = left;
            this.second = right;
        } else {
            this.first = right;
            this.second = left;
        }
        random = new Random();
    }

    public void run() {
        try{
            while(true) {
                Thread.sleep(random.nextInt( bound: 1000));
                synchronized (first) {
                    synchronized (second) {
                        Thread.sleep(random.nextInt( bound: 1000));
                    }
                }
            }
        } catch (InterruptedException e) {}

    }
}
```

Slika 12. Primjer upotrebe ključeva u problemu gladni filozofi

Umjesto da filozof uzima lijevi i desni štapić za jelo, uzeti će štapiće u redoslijedu. Pomoću ovog rješenja program se može vrtjeti zauvijek.

### 3.4 Reentrant ključ

Problem kod prethodnog načina dohvaćanja ključeva je nemogućnost prekidanja dretvi. Točno to je i razlog zbog kojeg je nemoguće izaći iz zastoja. Taj problem je moguće riješiti putem *reentrant* ključa (Butcher, 2014). *Reentrant* ključ omogućuje prekid dretve koja se nalazi u zastoju. Slika 13 prikazuje primjer korištenja reentrant ključa.

```
1  import ...
6
7  public class Philosopher extends Thread {
8      private ReentrantLock left, right;
9      private Random random;
10
11     public Philosopher(ReentrantLock left, ReentrantLock right) {
12         this.left = left;
13         this.right = right;
14         random = new Random();
15     }
16
17     public void run() {
18         try{
19             while(true) {
20                 Thread.sleep(random.nextInt( bound: 1000));
21                 left.lock();
22                 try{
23                     if (right.tryLock( timeout: 1000, TimeUnit.MILLISECONDS)) {
24                         try {
25                             Thread.sleep(random.nextInt( bound: 1000));
26                         } finally {
27                             right.unlock();
28                         }
29                     } else {}
30                 } finally {
31                     left.unlock();
32                 }
33             }
34         } catch (InterruptedException e) {}
35
36     }
37 }
```

Slika 13. Primjer korištenja reentrant ključa

Sada će filozof, nakon što uzme lijevi štapić, pokušati uzeti desni štapić, i ako to nakon tisuću milisekundi ne uspije, otpustit će lijevi štapić. Ovakva verzija programa još uvijek može proizvesti zastoj, ali on neće trajati zauvijek.

# 4. Java kolekcije za rukovanje dretvama

Java posjeduje veliku kolekciju temeljito istestiranih i visoko performantnih struktura podataka te općenitih korisnih metoda. U većini slučajeva su spomenute kolekcije bolji odabir od vlastitih implementacija.

## 4.1 Atomske varijable

Jedno od takvih rješenja je klasa *AtomicInteger* (Naftalin & Wadler, 2006).

U prethodnom problemu u kojemu su dvije dretve pokušale povećavati jednu vrijednost je uspješno riješen sinkronizacijom metode. Java već nudi slično rješenje u obliku *AtomicInteger* klase. Na slici 14 je prikazan primjer uporabe *AtomicInteger* klase.

```
1  import ...
9
10 public class Main {
11     public static void main(String[] args) throws InterruptedException {
12         final AtomicInteger counter = new AtomicInteger();
13         class CountingThread extends Thread {
14             public void run() {
15                 for (int x = 0; x < 10000; ++x)
16
17                     counter.incrementAndGet();
18             }
19         }
20         CountingThread t1 = new CountingThread();
21         CountingThread t2 = new CountingThread();
22         t1.start();
23         t2.start();
24         t1.join();
25         t2.join();
26
27         System.out.println(counter.get());
28     }
29 }
```

```
Run:  Main
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
20000

Process finished with exit code 0
```

Slika 14. Primjer upotrebe *AtomicInteger* klase

Sama *AtomicInteger* klasa funkcionira kao i sinkronizacija metode tj. korištenje mehanizma ključa. Samo jedna dretva je u stanju pozvati metodu *incrementAndGet*. Čime se postiže sinkronizacija dretvi.

## 4.2 Kreiranje dretvi

```
1  import ...
6
7  public class Main {
8      public static void main(String[] args) throws IOException {
9          class ConnectionHandler implements Runnable {
10             InputStream in;
11             OutputStream out;
12
13             ConnectionHandler(Socket socket) throws IOException {
14                 in = socket.getInputStream();
15                 out = socket.getOutputStream();
16             }
17
18             public void run() {
19                 try {
20                     int n;
21                     byte[] buffer = new byte[1024];
22                     while ((n = in.read(buffer)) != -1) {
23                         out.write(buffer,  off: 0, n);
24                         out.flush();
25                     }
26                 } catch (IOException e) {
27                 }
28             }
29         }
30
31         ServerSocket server = new ServerSocket( port: 4567);
32         while (true) {
33             Socket socket = server.accept();
34             Thread handler = new Thread(new ConnectionHandler(socket));
35             handler.start();
36         }
37     }
38 }
```

Slika 15.Primjer lošeg načina kreiranje dretvi

Na slici 15 se nalazi primjer programa koji ispisuje poruku primljenog zahtjeva. Linije 33, 34 i 35 prikazuju česti način zaprimanja i obrađivanja novog zahtjeva. Kreiranje dretvi nije jeftina radnja. Ako program krene zaprimati ogromnu količinu zahtjeva, broj

dretvi će znatno narasti. Za taj problem java nudi rješenje u obliku *Executor Service*-a (Oaks & Wong, 2004).

```java
import ...

public class Main {
    public static void main(String[] args) throws IOException {
        class ConnectionHandler implements Runnable {
            InputStream in;
            OutputStream out;

            ConnectionHandler(Socket socket) throws IOException {
                in = socket.getInputStream();
                out = socket.getOutputStream();
            }

            public void run() {
                try {
                    int n;
                    byte[] buffer = new byte[1024];
                    while ((n = in.read(buffer)) != -1) {
                        out.write(buffer, off: 0, n);
                        out.flush();
                    }
                } catch (IOException e) {
                }
            }
        }

        ServerSocket server = new ServerSocket( port: 4567);

        int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2;
        ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize);
        while (true) {
            Socket socket = server.accept();
            executor.execute(new ConnectionHandler(socket));
        }
    }
}
```

Slika 16.Primjer dobrog načina kreiranje dretvi

Na slici 16 je prikazan program koji umjesto ručnog kreiranja dretve koristi *Executor Service*. U ovom primjeru se koristi *FixedThreadPool*. Njegova odlika je ta što on ima fiksan broj dretvi u svom životnom ciklusu. Što znači da ako sve dretve obavljaju neku radnju, svi ostali zahtjevi će biti stavljeni u red čekanja. Ako program zaprimi ogroman broj zahtjeva, broj dretvi se neće povećati nego će svi zahtjevi koji ne mogu biti obrađeni ići u red čekanja.

### 4.3 Preslika pri pisanju

Još jedan u nizu problema, kada su u pitanju dretve, je problem rada na listama. Dok jedna dretva čita listu i vrši neku radnju nad njome, druga dretva može brisati određene elemente te liste. Na kraju obavljanja spomenutih radnji, lako je moguće da prva dretva nije uspjela pročitati sve elemente liste jer je druga dretva prije nje izbrisala spomenute elemente.

Rješenje za takav problem nudi Java u obliku *CopyOnWriteArrayList* klase (Oaks & Wong, 2004). *CopyOnWriteArrayList* klasa radi kopiju liste kada se nešto u njoj mijenja. Što znači da će postojeće iteracije po listi ostati netaknute.

```
import ...

public class Main {
    public static void main(String[] args) throws InterruptedException {
        CopyOnWriteArrayList<Integer> numbers = new CopyOnWriteArrayList<>(new Integer[]{1, 2, 3, 4, 5});

        Thread t1 = new Thread( () ->{
            Iterator<Integer> iterator = numbers.iterator();
            iterator.forEachRemaining(System.out::println);
        });

        Thread t2 = new Thread(() -> {
            Iterator<Integer> iterator = numbers.iterator();
            iterator.forEachRemaining(numbers::add);
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
1
2
3
4
5

Process finished with exit code 0
```

Slika 17.Primjer uporabe klase *CopyOnWriteArrayList*

Primjer programa na slici 17 prikazuje uporabu klase *CopyOnWriteArrayList*. Dretva broj dva prolazi kroz listu i svaki broj iz liste ponovno doda u listu. Dretva broj jedan je

pokrenuta odmah nakon dretve broj dva. Zbog svojstva klase *CopyOnWriteArrayList*, dretva broj jedan će ispisati originalnu listu jer dretva broj dva radi kopiju liste kako bi originalni podaci ostali sačuvani.

```
1  import ...
14
15 public class Main {
16     public static void main(String[] args) throws InterruptedException {
17         List<Integer> numbers = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
18
19         Thread t1 = new Thread( () ->{
20             numbers.forEach(System.out::println);
21         });
22
23         Thread t2 = new Thread(() -> {
24             numbers.addAll(numbers);
25         });
26
27         t1.start();
28         t2.start();
29
30         t1.join();
31         t2.join();
32     }
33 }
```

```
Run:  Main
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe" ...
1
2
3
4
5
1
2
3
4
5

Process finished with exit code 0
```

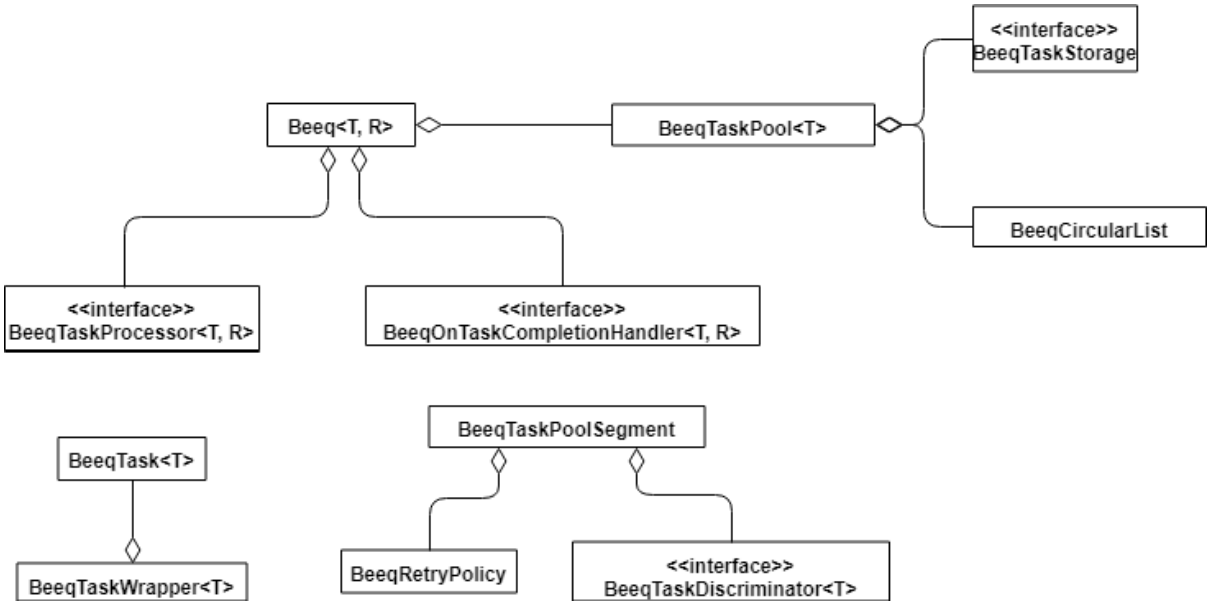
Slika 18.Primjer bez uporabe klase *CopyOnWriteArrayList*

Na slici broj 18 je prikazan primjer koji ne koristi klasu *CopyOnWriteArrayList*. Sada će brojevi koje dretva broj dva umeće u listu biti vidljivi dretvi broj jedan koja iterira kroz nju.

# 5. Klasni dijagrami

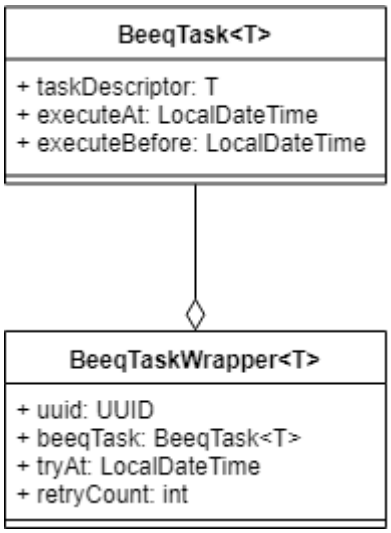
U softverskom inženjerstvu, klasni dijagram je dijagram statične strukture pomoću kojeg se prikazuju klase, atributi, funkcije i veze između sistemskih objekata.

U nastavku ovog poglavlja slijedi opisi programskog rješenja korištenjem klasnih dijagrama. Na slici 19 je prikazan klasni dijagrama za sveukupno programsko rješenje.




Slika 19.Klasni dijagram biblioteke

## 5.1 Klasni dijagram zadataka




Slika 20.Klasni dijagram zadataka

Na slici 20 se nalazi klasni dijagram za klase koje predstavljaju zadatke koji se izvršavaju.

***BeeqTask*** predstavlja klasu koja u sebi ima specifičan zadatak te atribute koji opisuju rokove izvršavanja zadatka.

Atributi:

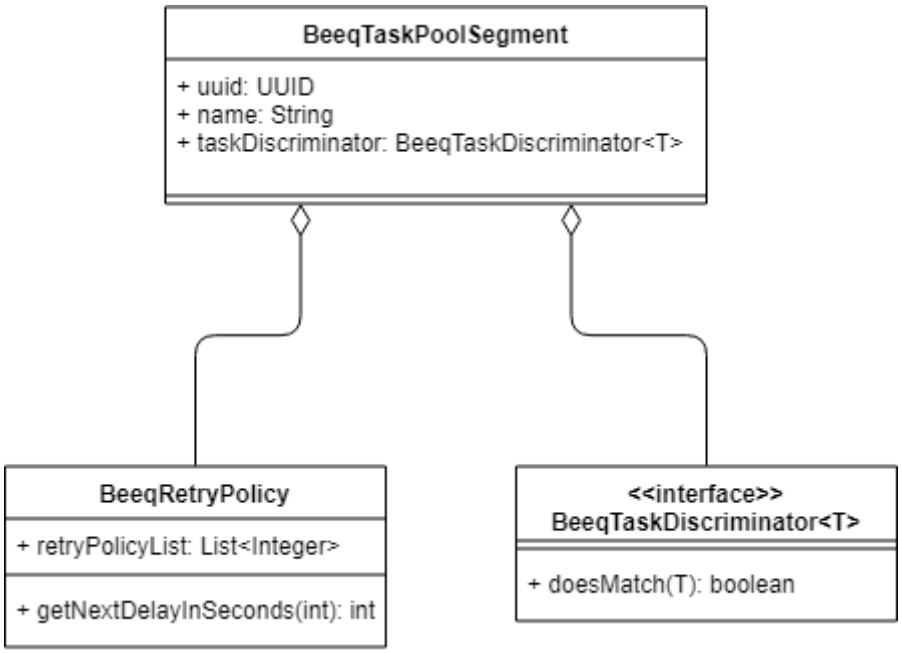
- *taskDescriptor*: T – specifičan zadatak koji se treba izvršiti
- *executeAt*: *LocalDateTime* – vrijeme kada zadatak mora biti izvršen
- *executeBefore*: *LocalDateTime* – vrijeme do kojeg zadatak mora biti izvršen

***BeeqTaskWrapper*** predstavlja klasu koja uz prethodno opisanu klasu čuva još neke dodatne atribute.

Atributi:

- *beeqTask*: *BeeqTask* – prethodno opisana klasa
- *uuid*: UUID – unikatno univerzalni identifikator
- *tryAt*: *LocalDateTime* – vrijeme kada bi se zadatak ponovno trebao izvršiti
- *retryCount*: *Int* – broj trenutno pokušaja izvršavanja

## 5.2 Klasni dijagram segmenta




Slika 21.Klasni dijagram segmenta

Na slici 21 je prikazan klasni dijagram *BeeqTaskPoolSegment* klase, te svih klasa koje sadrži *BeeqTaskPoolSegment*.

***BeeqTaskPoolSegment*** predstavlja klasu pomoću koje se zadaci razvrstavaju.

Atributi:

- *uuid*: *UUID* – unikatno univerzalni identifikator
- *name*: *String* – atribut koji predstavlja ime segmenta
- *taskDiscriminator*: *BeeqTaskDiscriminator*<T> - atribut pomoću kojeg se zadaci razvrstavaju. Detaljnije će biti pojašnjeno u nastavku
- *batchSize*: *int* – atribut koji predstavlja najveći broj uzastopno izvršenih zadataka za segment

***BeeqRetryPolicy*** je klasa koja opisuje politiku izvršavanja zadataka koji pripadaju segmentu. Kao parametar prima listu koja sadrži razmake između pojedinih ponavljanja. Ponavljanja su izražena u sekundama.

Atributi:

- *retyrPolicyList*: *List<Integer>* - lista koja sadrži vrijednosti razmaka pojedinih ponavljanja izražene u sekundama

Funkcije:

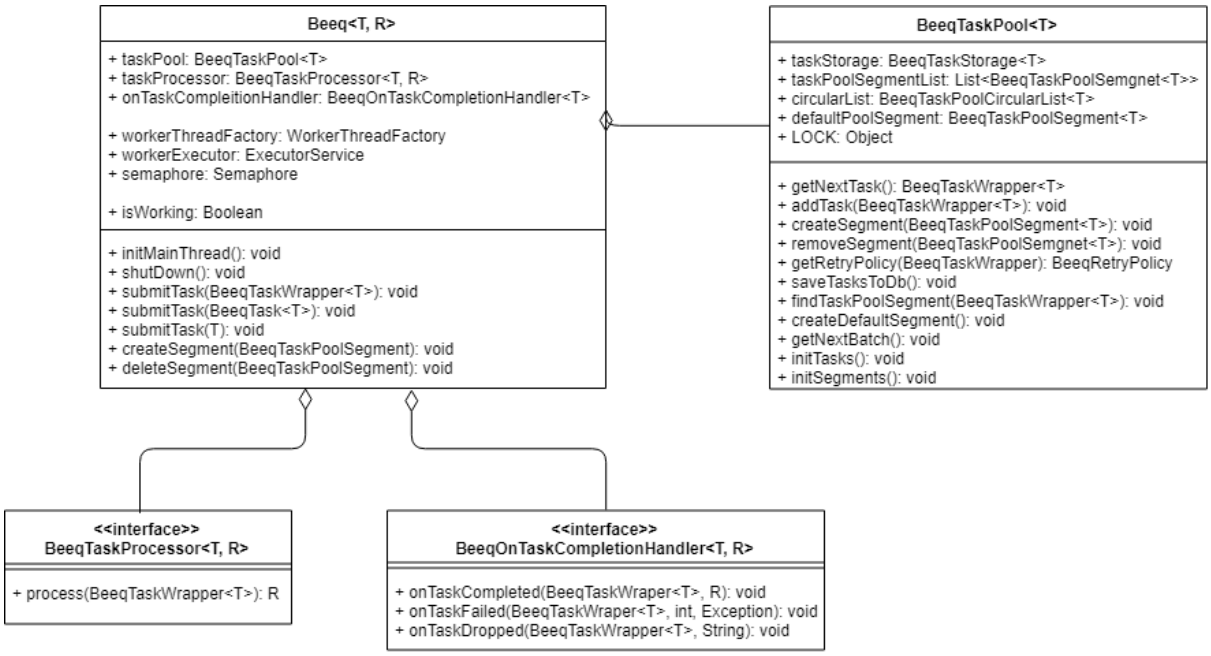
- *getNextDelayInSeconds*(*int*): *int* – funkcija koja na osnovu broja trenutnog pokušaja ponavljanja vraća vrijednost, izraženu u sekundama, do sljedećeg ponavljanja

***BeeqTaskDiscriminator***<T> je sučelje pomoću kojeg se zadaci razvrstavaju po segmentima

Funkcije:

- *doesMatch* (T): *boolean* – funkcija pomoću koje se razvrstavaju zadaci po segmentima

## 5.3 Klasni dijagram ulazne točke




Slika 22.Klasni dijagram ulazne točke

Na slici 22 je prikazana ulazna točka programa, a samim time i dio gdje se dešava obrada zadataka. Također su prikazani najbliži elementi *Beeq* klase

***Beeq*** klasa predstavlja jedini način komunikacije korisnika sa ostatkom biblioteke. Klasa putem konstruktora prima parametar koji određuje broj dretvi. Klasa sadrži sljedeće atribute:

- *taskPool*: *BeeqTaskPool*<T> – atribut koji je zaslužan za daljnje propagiranje zadataka. Detaljnije će biti pojašnjen u nastavku poglavlja
- *taskProccesor*: *BeeqTaskProcesor*<T, R> – atribut koji obrađuje zadatke. Detaljnije će biti pojašnjen u nastavku poglavlja
- *onTaskCompletitionHandler*: *BeeqOnTaskCompletionHandler*<T> – atribut koji odlučuje što će biti sa zadatkom nakon njegove obrade. Detaljnije će biti pojašnjen u nastavku poglavalja
- *workerThreadFactory*: *ThreadFactory* – atribut pomoću kojeg se dodjeljuje ime kreiranim dretvama

- *workerExecutor*: *ExecutorService* – atribut kojemu se predaju zadaci na obradu
- *semaphore*: *Semaphore* – atribut pomoću kojeg se na sinkroniziran način kontrolira broj trenutnih zadataka u obradi
- *isWorking*: *Boolean* – atribut koji otkriva obrađuje li izvršitelj zadataka neki zadatak

Klasa također sadrži niz funkcija:

- *initMainThread*(): *void* – funkcija pokreće glavnu dretvu koja dohvaća i šalje zadatke na obradu
- *shutDown*(): *void* – funkcija pomoću koje se na siguran način isključuje program tako što se poziva funkcija koja zadatke u memoriji sprema u bazu podataka (ovisi o implementaciji korisnika)
- *submitTask* (*BeeqTaskWrapper*<T>/*BeeqTask*<T>/T): *void* – niz preopterećenih funkcija pomoću kojih se dodaju novi zadaci
- *createSemgent* (*BeeqTaskPoolSegment*<T>): *void* – metoda za kreiranje novog segmenta. Segment će biti detaljnije objašnjen u nastavku poglavlja
- *deleteSegment* (*BeeqTaskPoolSegment*<T>): *void* – metoda za brisanje postojećeg segmenta

**BeeqTaskPool** klasa je zaslužna za raspodjeljivanje zadataka.

Atributi:

- *taskStorage*: *BeeqTaskStorage*<T> – zaslužan za spremanje zadataka. Pobliže će biti objašnjen u nastavku
- *taskPoolSegmentList*: List<*BeeqTaskPoolSegment*<T>> – lista koja sadrži sve dodane segmente
- *circularList*: *BeeqTaskPoolCircularList*<T> – služi kao pomoć prilikom odabira segmenta iz kojeg će se uzeti zadatak za obradu
- *defaultPoolSegment*: *BeeqTaskPoolSegment*<T> – zadani segment u slučaju da korisnik ne doda ni jedan vlastiti
- *LOCK*: *Object* – služi kao ključ za sinkronizirano dodavanje i čitanje zadataka

Funkcije:

- *getNextTask*(): *BeeqTaskWrapper*<T> – funkcija za dohvaćanje sljedećeg zadatka, funkcija je sinkronizirana korištenjem mehanizma ključa
- *addTask* (*BeeqTaskWrapper*<T>) : *void* – funkcija za dodavanje novog zadatka, također je sinkronizirana korištenjem mehanizma ključa
- *createSegment* (*BeeqTaskPoolSegment*<T>): *void* – funkcija koja kreira novi segment
- *removeSegment* (*BeeqTaskPoolSegment*<T>: *void* – funkcija koja uklanja segment
- *getRetryPolicy* (*BeeqTaskWrapper*<T>): *BeeqRetryPolicy* – funkcija koja dohvaća objekt koji opisuje politiku ponavljanja zadatka. Pobliže će biti opisan u nastavku poglavlja
- *saveTasksToDb*(): *void* – funkcija koja sprema podatke u bazu podataka (ukoliko se korisnik odluči na implementaciju baze podataka)
- *findTaskPoolSegment* (*BeeqTaskWrapper*<T>): *BeeqTaskPoolSegment*<T> – funkcija koja služi za pronalazak segmenta u koji će zadatak biti spremljen
- *getNextBatch* (): *void* – funkcija koja dohvaća određeni broj zadataka iz baze podataka (ovisi o implementaciji)
- *initTasks* (): *void* – funkcija dohvaća zadatke iz baze podataka (ovisi o implementaciji)
- *initSegments* (): *void* – funkcija dohvaća segmente iz baze podataka (ovisi o implementaciji)

***BeeqTaskProcessor*** je prvo sučelje do sad. Ono predstavlja način na koji će se zadaci obrađivati

Funkcije:

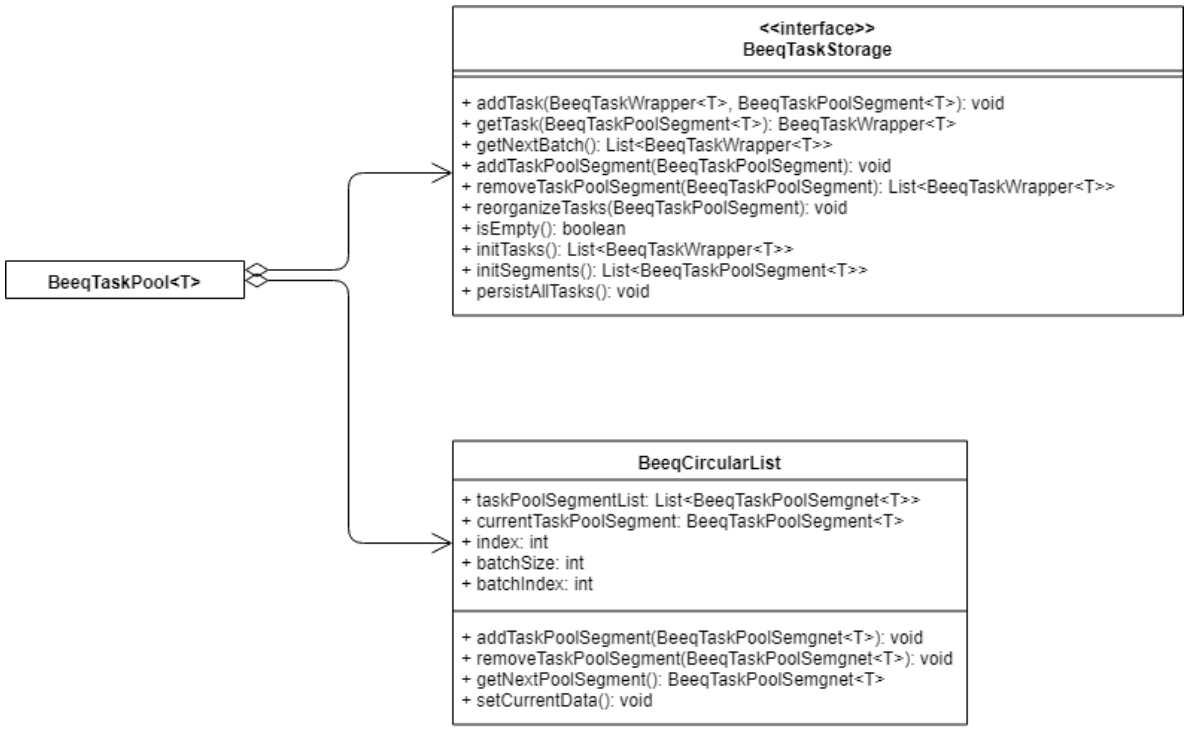
- *process*(*BeeqTaskWrapper*<T>): R – funkcija koja obraduje zadatak i vraća rezultat

***BeeqOnTaskCompletionHandler*** je sučelje koje opisuje radnje koje se izvršavaju na osnovu rezultata obrađenog zadatka

Funkcija:

- *onTaskCompleted* (*BeeqTaskWrapper*<T>, R): *void* – funkcija koja se poziva prilikom uspješnog izvršavanja zadatka
- *onTaskFailed* (*BeeqTaskWrapper*<T>, *int*, *Exception*): *void* – funkcija koja se poziva prilikom neuspješnog izvršavanja zadatka. Funkcija kao argument prima broj neuspješnog izvršavanja zadatka te parametar tipa *Exception*.
- *onTaskDropped* (*BeeqTaskWrapper*<T>, *String*): *void* – funkcija koja se poziva kada se zadatak odbaci od daljnje obrade. Atribut tipa *String* predstavlja poruku zbog kojeg se desilo odbacivanje zadatka

## 5.4 Klasni dijagram pohrane




Slika 23.Klasni dijagram pohrane

Na slici 23 je prikazan klasni dijagram za *BeeqTaskStorage* i *BeeqCircularList*. *BeeqTaskPool* je već ranije pojašnjen tako da u nastavku slijedi opis samo za *BeeqTaskStorage* i *BeeqCircularList*.

***BeeqTaskStorage*** predstavlja sučelje koje služi kao pohrana za zadatke.

Funkcije:

- *addTask* (*BeeqTaskWrapper*<T>, *BeeqTaskPoolSegment*<T>): *void* – funkcija koja sprema zadatak u neki određeni segment
- *getTask* (*BeeqTaskPoolSegment*<T>): *BeeqTaskWrapper*<T> – funkcija koja dohvaća zadatak za određeni segment
- *getNextBatch*(): List<*BeeqTaskWrapper*<T> – funkcija koja dohvaća određeni broj zadataka iz baze podataka (ovisi o implementaciji)
- *addTaskPoolSegment* (*BeeqTaskPoolSegment*<T>): *void* – funkcija koja dodaje novi segment

- *removeTaskPoolSegment* (*BeeqTaskPoolSegment*<T>): List<*BeeqTaskWrapper*<T>> – funkcija koja uklanja postojeći segment i vraća listu zadataka koji su bili u izbrisanom segmentu
- *reorganizeTasks* (*BeeqTaskPoolSegment*): *void* – funkcija koja preraspodjeljuje zadatke iz postojećih segmenata u novi segment
- *isEmpty* (): *boolean* – funkcija koja kao rezultat vraća istinu ako ne postoji niti jedan zadatak koji čeka na obradu, inače vraća laž
- *initTasks* (): *void* – funkcija dohvaća zadatke iz baze podataka (ovisi o implementaciji)
- *initSegments* (): *void* – funkcija dohvaća segmente iz baze podataka (ovisi o implementaciji)
- *persistAllTasks* (): *void* – funkcija koja sprema sve zadatke u bazu podataka (ovisi o implementaciji)

***BeeqCircularList*** je pomoćna klasa koja se koristi prilikom segmentiranja zadataka.

Atributi:

- *taskPoolSegmentList*: List<*BeeqTaskPoolSegment*<T>> – lista koja sadrži sve trenutne segmente
- *currentTaskPoolSegment*: *BeeqTaskPoolSegment*<T> – trenutni segment iz kojeg se izvršavaju zadaci
- *index*: *int* – pomoćni atribut koji pamti trenutni položaj u *taskPoolSegmentList*-i
- *batchSize*: *int* – pomoćni atribut koji pamti *batchSize* trenutnog segmenta
- *batchIndex*: *int* – pomoćni atribut koji pamti broj dosadašnje izvršenih *taskova*

Funkcije:

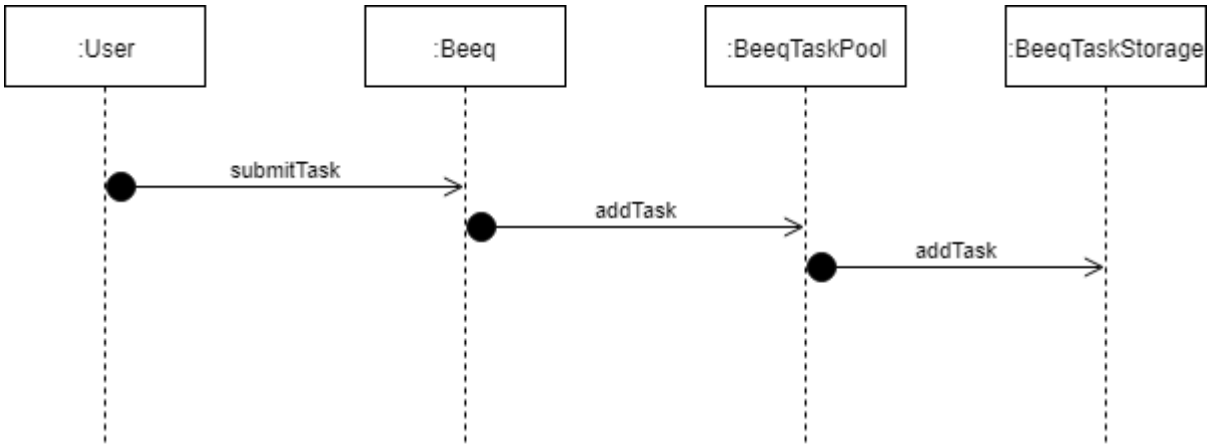
- *addTaskPoolSegment* (*BeeqTaskPoolSegment*<T>): *void* – funkcija koja dodaje novi segment u *taskPoolSegmentList*-u
- *removeTaskPoolSegment* (*BeeqTaskPoolSemgnet*<T>): *void* – funkcija koja uklanja segment iz *taskPoolSegmentList*-e
- *getNextPoolSegment*(): *BeeqTaskPoolSegment*<T> – funkcija koja vraća trenutni segment
- *setCurrentData*(): *void* – funkcija koja ažurira podatke prilikom dodavanja ili uklanjanja segment

# 6. Slijedni dijagrami

Slijedni dijagrami prikazuje interakciju između objekata prikazanih u vremenskom slijedu. Opisuje različite scenarije među objektima i poruke koje se šalju između njih.

U nastavku poglavalja slijede slijedni dijagrami koji će pobliže objasniti neke od bitnijih radnji biblioteke.

## 6.1 Slijedni dijagram dodavanja zadatka




Slika 24.Slijedni dijagram dodavanja zadataka

Slika broj 24 prikazuje slijedni dijagram za akciju dodavanja novog zadatka. Korisnik poziva jednu od funkcija pod nazivom *submitTask* iz klase *Beeq*.

```
public void submitTask(BeeqTaskWrapper<T> task) {
    taskPool.addTask(task);
}

public void submitTask(BeeqTask<T> task) {
    taskPool.addTask(new BeeqTaskWrapper<>(task));
}

public void submitTask(T task) {
    taskPool.addTask(new BeeqTaskWrapper<>(new BeeqTask<>(task)));
}
```

Slika 25. Prikaz *submitTask* funkcija

*Beeq* tada zove funkciju *addTask* iz klase *BeeqTaskPool*.

```
void addTask(BeeqTaskWrapper<T> task) {
    BeeqTaskPoolSegment<T> segmentPool = findTaskPoolSegment(task);

    synchronized (LOCK) {
        taskStorage.addTask(task, segmentPool);
        LOCK.notifyAll();
    }
}
```

Slika 26.Prikaz *addTask* funkcije

Funkcija *addTask* iz klase *BeeqTaskPool* pronalazi segment kojem zadatak pripada. Nakon pronalaska *segmeneta*, u sinkroniziranom bloku se poziva funkcija *addTask* iz klase *BeeqTaskStorage*. Sinkronizirani blok koda sluzi kao bloker dretvama prilikom dodavanja i čitanja zadataka. Svrha blokera će biti pobliže objašnjena u nastavku poglavlja. Implementacija funkcije *addTask*, koja se nalazi u sučelju *BeeqTaskStorage*, ovisi o samom korisniku.

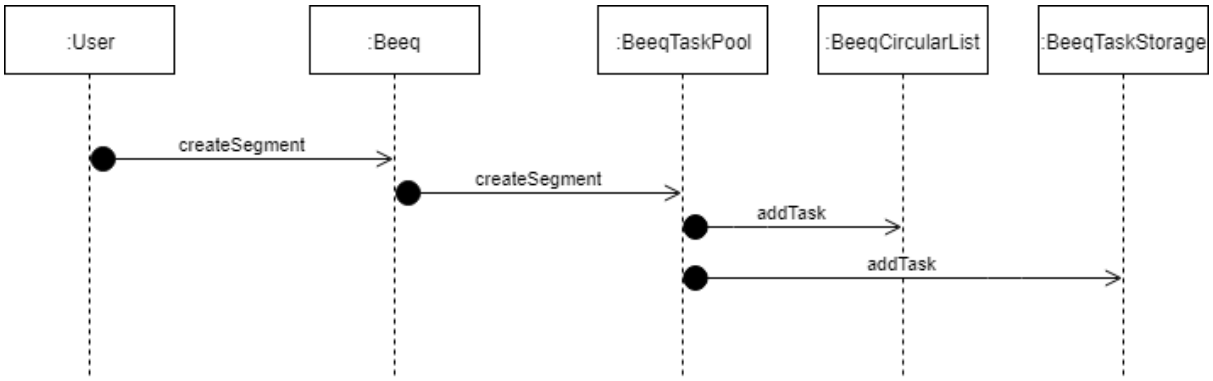
```
private BeeqTaskPoolSegment<T> findTaskPoolSegment(BeeqTaskWrapper<T> task) {
    Optional<BeeqTaskPoolSegment<T>> segmentPool = taskPoolSegmentList.stream()
            .filter(segment → segment.getTaskDiscriminator().doesMatch(task.getTaskDescriptor()))
            .findFirst();

    return segmentPool.orElseGet(() → defaultTaskPoolSegment);
}
```

Slika 27.Prikaz *findTaskPoolSegment* funkcije

Funkcija *findTaskPoolSegment* prima zadatak kao parametar te pokušava naći segment za njega. Na samom početku funkcija filtrira sve segmente tako da njihov *taskDiscriminator* odgovara *taskDescriptor*-u zadatka, te dohvati prvi od njih. Na posljetku funkcija vraća pronađeni segment, ili ako ga nije pronašla vraća zadani *segement*.

## 6.2 Slijedni dijagram dodavanja segmenta




Slika 28. Slijedni dijagram dodavanja segmenta

Na slici 28 je prikazan slijedni dijagram za dodavanje novog segmenta. Korisnik poziva funkciju *createSegment* koja se nalazi u klasi *Beeq*.

```
public void createSegment(BeeqTaskPoolSegment<T> taskPoolSegment) {
    taskPool.createSegment(taskPoolSegment);
}
```

Slika 29. Prikaz *createSegment* funkcije klase *Beeq*

Funkcija kao argument prima segment te ga prosljeđuje u funkciju *createSegment* iz klase *BeeqTaskPool*

```
void createSegment(BeeqTaskPoolSegment<T> taskPoolSegment) {
    if (defaultTaskPoolSegment.getBatchSize() > taskPoolSegment.getBatchSize()) {
        defaultTaskPoolSegment.setBatchSize(taskPoolSegment.getBatchSize());
    }

    // We want the default task pool segment to be last
    taskPoolSegmentList.add( index: 0, taskPoolSegment);
    circularList.addTaskPoolSegment(taskPoolSegment);
    taskStorage.addTaskPoolSegment(taskPoolSegment);
}
```

Slika 30. Prikaz *createSegment* funkcije klase *BeeqTaskPool*

Funkcija *createSegment* na samom početku postavlja nove vrijednosti za zadani segment. Svrha zadanog segmenta je da poprimi najmanji *batchSize* od svih segmenata koji su kreirani od strane korisnika. Nakon toga segment se dodaje na prvo mjesto u *taskPoolSegmentList* atribut, kako bi zadani segment bio zadnji segment u listi. Na samom kraju funkcije se pozivaju *addTaskPoolSegment* funkcije iz klase *BeeqCircularList* i sučelja *BeeqTaskStorage*.

```
public void addTaskPoolSegment(BeeqTaskPoolSegment<T> taskPoolSegment){
    // We want the default task pool segment to be last
    taskPoolSegmentList.add( index: 0, taskPoolSegment);
    this.setCurrentData();
}
```

Slika 31. Prikaz *addTaskPoolSegment* funkcije

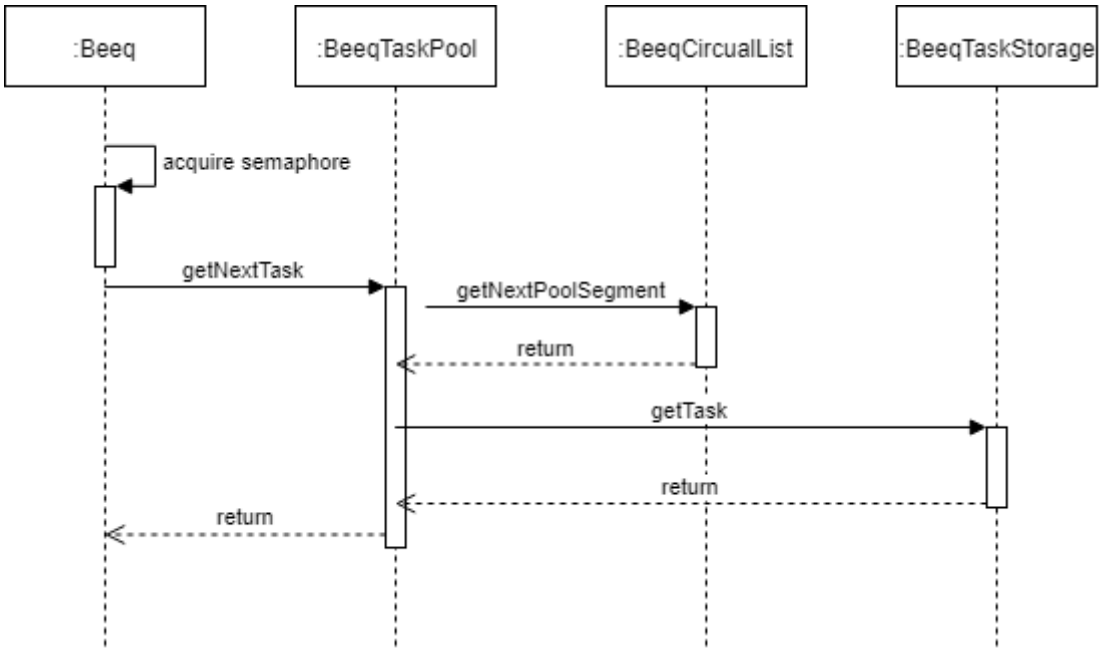
Funkcija *addTaskPoolSegment* iz klase *BeeqCircularList* dodaje segment na prvo mjesto svoje liste, te poziva funkcija *setCurrentData* koja postavlja trenutni segment i trenutni *batchSize*.

```
private void setCurrentData(){
    currentTaskPoolSegment = taskPoolSegmentList.get(index);
    batchSize = currentTaskPoolSegment.getBatchSize();
}
```

Slika 32. Prikaz *setCurrentData* funkcije

Funkcija *addTaskPoolSegment* iz klase *BeeqTaskStorage* ovisi o implementaciji korisnika.

## 6.3 Slijedni dijagram dohvaćanja zadatka




Slika 33. Slijedni dijagram dohvaćanja zadatka

Na slici 33 je prikazan slijedni dijagram za dohvaćanje zadatka.

```
try {
    semaphore.acquire();
} catch (InterruptedException e) {
    e.printStackTrace();
}

BeeqTaskWrapper<T> task = taskPool.getNextTask();
```

Slika 34. Prikaz početka bloka izvršne dretve

Glavna dretva na samom početku pokušava uzeti jednu „kartu“ semafora. Proces blokira dretvu sve dok ne uspije uzeti „kartu“. Nakon toga dretva poziva funkciju *getNextTask* iz klase *BeeqTaskPool*.

```
BeeqTaskWrapper<T> getNextTask() {
    synchronized (LOCK) {
        if (taskStorage.isEmpty()) {
            try {
                LOCK.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        BeeqTaskWrapper<T> taskWrapper = taskStorage.getTask(circularList.getNextPoolSegment());

        if (taskWrapper == null) {
            taskWrapper = getNextTask();
        }

        if (taskStorage.isReadyForNextBatch()) {
            getNextBatch();
        }

        return taskWrapper;
    }
}
```

Slika 35. Prikaz bloka izvršne dretve

Cjelokupni kod funkcije se nalazi u sinkroniziranom bloku. Razlog tome je blokiranje dretve. Glavna dretva koja poziva ovu funkciju se nalazi u beskonačnoj petlji te se iz tog razloka koristi mehanizam blokiranja. Kada mehanizam blokiranja ne bi bio korišten dretva bi neprestano pozivala ovu funkciju, iako ne bi bilo nijednog dostupnog zadatka. Blokirana dretva kreće u ponovno izvršavanje pozivom metode *addTask*. Nakon što dretva prestane biti u blokiranom stanju, dohvaća se sljedeći zadatak za obradu. Poziva se funkcija *getTask*, iz klase *BeeqTaskStorage*, kojoj se kao parametar prosljeđuje povratna informacija, točnije segment, funkcije *getNextPoolSegment* iz klase *BeeqCircularList*. Ako funkcija ne uspije vratiti zadatak za ponuđeni segment, slijedi rekurzivni poziv. Prije samog vraćanja zadatka slijedi provjera koja govori da li je *BeeqTaskStorage* spreman za sljedeći broj zadataka, ako da slijedi poziv funkcije *getNextBatch* koja dohvaća određeni broj zadataka. Implementacije funkcije *getTask*, *getNextBatch* te *isReadyForNextBatch* ovise o korisniku.

```
public BeeqTaskPoolSegment<T> getNextPoolSegment(){
    if(batchIndex == batchSize){
        index = (index + 1) % taskPoolSegmentList.size();
        setCurrentData();
        batchIndex = 1;
        return currentTaskPoolSegment;
    }

    batchIndex++;

    return currentTaskPoolSegment;
}
```

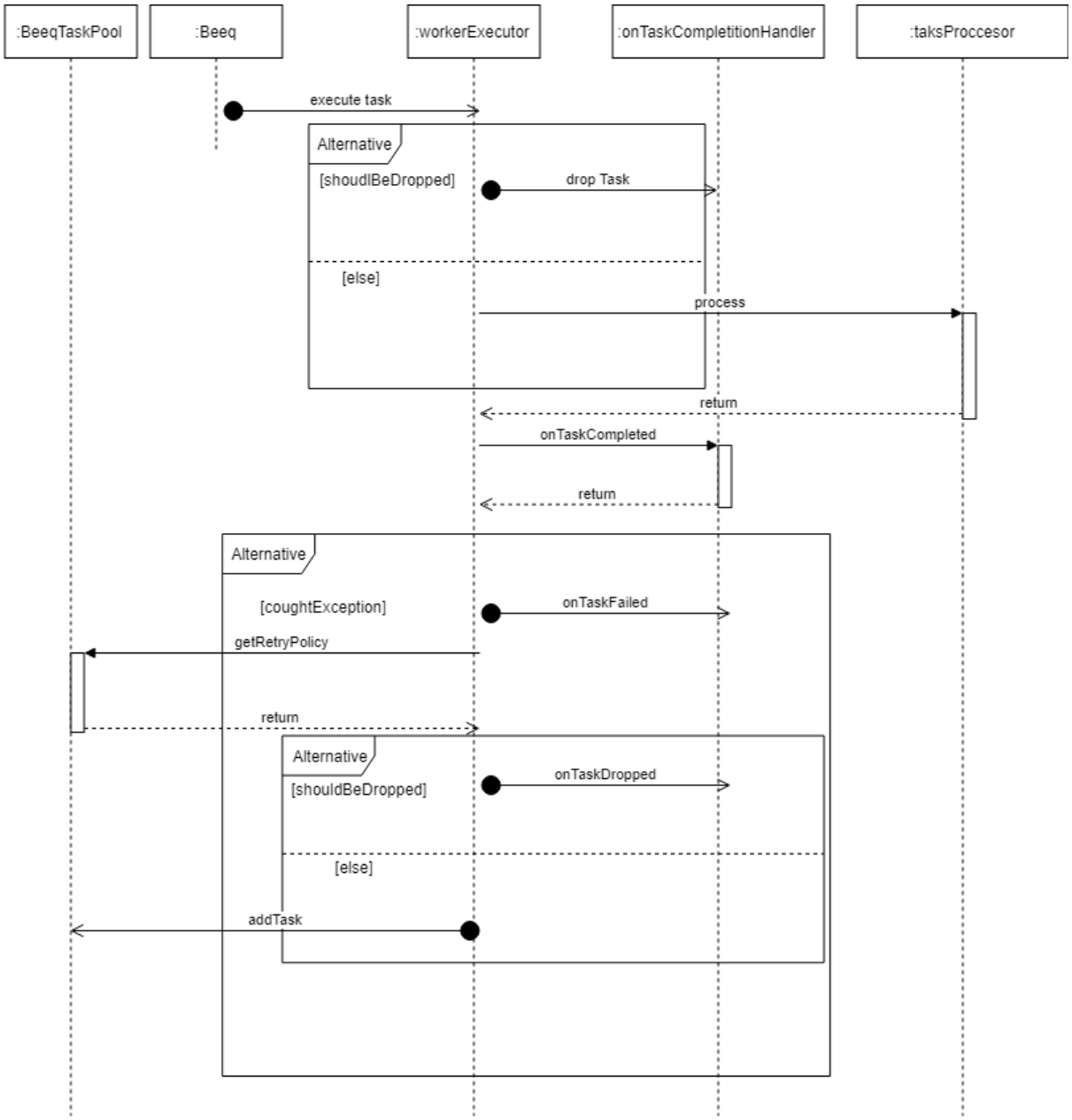
Slika 36. Prikaz *getNextPoolSegment* funkcije

Funkcija *getNextPoolSegment* iz klase *BeeqCircularList* vraća trenutni segment ako trenutni broj obrađenog zadatka nije jednak vrijednosti *batchSize* trenutnog segmenta. Inače funkcija povećava vrijednost *index* atributa, pazeći da vrijednost ne nadmaši ukupnu dužinu liste svih segmenata. Nakon toga slijedi postavljanje trenutnih podataka te vraćanje trenutnog segmenta.

```
private void getNextBatch() {
    List<BeeqTaskWrapper<T>> taskWrapperList = taskStorage.getNextBatch();
    taskWrapperList.forEach(this::addTask);
}
```

Slika 37. Prikaz *getNextBatch* funkcije

## 6.4 Slijedni dijagram izvršavanja zadatka




Slika 38. Slijedni dijagram izvršavanja zadatka

Na slici broj 38 je prikazan slijedni dijagram za glavni tok biblioteke.

```java
private void initMainThread() {
    Thread beeqMainThread = new Thread(() -> {
        while (isWorking) {
            try {
                semaphore.acquire();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            if (workerExecutor.isShutdown() || workerExecutor.isTerminated() || !isWorking) {
                break;
            }

            BeeqTaskWrapper<T> task = taskPool.getNextTask();

            workerExecutor.execute(() -> {
                try {
                    if (task.shouldBeDropped()) {
                        onTaskCompletionHandler.onTaskDropped(task,  message: "Time of execution has passed");
                        semaphore.release();
                        return;
                    }

                    R r = taskProcessor.process(task);
                    onTaskCompletionHandler.onTaskCompleted(task, r);

                    semaphore.release();

                } catch (Exception e) {
                    task.incrementRetryCount();

                    onTaskCompletionHandler.onTaskFailed(task, task.getRetryCount(), e);

                    BeeqRetryPolicy retryPolicy = taskPool.getRetryPolicy(task);

                    int retryDelay = retryPolicy.getNextDelayInSeconds(task.getRetryCount());

                    if (retryDelay == -1) {
                        onTaskCompletionHandler.onTaskDropped(task,  message: "Maximum number of retries reached");
                    } else {
                        task.setTryAt(LocalDateTime.now().plusSeconds(retryDelay));
                        taskPool.addTask(task);
                    }

                    semaphore.release();
                }
            });
        }
    });

    beeqMainThread.setName("Beeq-main-thread");
    beeqMainThread.start();
}
```

Slika 39. Prikaz bloka glavne dretve

Sve funkcije koje ne budu pobliže objašnjene ovise o implementaciji samog korisnika.

Na samom početku slijedi proces dohvaćanja zadatka koji je prethodno objašnjen. Iz prethodnog objašnjenja se izostavio uvjet koji prije same predaje zadatka na obradu, provjerava da li je program u procesu gašenja. Kada se zadatak preda izvršitelju, prva provjera koja se desi je ta da li je zadatak za odbacivanje. Ako je zadatak za odbacivanje, on se odbaci pozivom funkcije *onTaskDropped*, koja pripada sučelju *BeeqTaskOnCompletinoHandler*, s opisnom porukom, te se vraća „karta“ semafora.

Ako je zadatak spreman za obradu, poziva se funkcija *process* sučelja *BeeqTaskProcessor* te se njen rezultat sprema u varijablu r. Nakon toga se poziva funkcija *onTaskCompleted*, te joj se kao parametar u sam zadatak predaje i njegov rezultat. Na posljetku se vraća „karta“ semafora.

U slučaju da se uhvati iznimka prvo što funkcija napravi je povećava broj pokušaja zadatka. Nakon toga se poziva metoda *onTaskFailed*. Funkcija uz sam zadatak prima dva dodatna parametra koji su broj pokušaja izvršavanja zadatka te uhvaćena iznimka. Tada se zove funkcija *getRetryPolicy* klase *BeeqTaskPool* koja vraća politiku ponavljanja zadataka. Očitava se sljedeća vrijednost ponavljanja. Ako je vrijednost -1, oznaka da je broj ponavljanja dosegnuo granicu, poziva se funkcija onTakDropped te se zadatak odbacuje. U protivnom se postavlja novo zakazano vrijeme izvršavanja zadatka te se zadatak vraća nazad u pohranu. Na samom posljetku se vraća „karta“ semafora.

```
BeeqRetryPolicy getRetryPolicy(BeeqTaskWrapper<T> task) {
    Optional<BeeqTaskPoolSegment<T>> taskPoolSegment = taskPoolSegmentList.stream()
            .filter(segment → segment.getTaskDiscriminator().doesMatch(task.getTaskDescriptor()))
            .findFirst();

    if (taskPoolSegment.isPresent()) {
        return taskPoolSegment.get().getRetryPolicy();
    } else {
        return defaultTaskPoolSegment.getRetryPolicy();
    }
}
```

Slika 40. Prikaz *getRetryPolicy* funkcije

Funkcija *getRetryPolicy* klase *BeeqTaskPool* vraća *retryPolicy* odgovarajućeg segmenta za zadatak. Ako ne uspije pronaći segment onda vraća *retryPolicy* zadanog segmenta.

```
public int getNextDelayInSeconds(int retryNumber){
    if(retryNumber > retryPolicyList.size() - 1) return -1;
    return retryPolicyList.get(retryNumber-1);
}
```

Slika 41. Prikaz *getNextDelayInSeconds* funkcije

Funkcija *getNextDelayInSeconds* klase *BeeqRetryPolicy* vraća sljedeću vrijednost izraženu u sekundama. Ako je dosegnuta granica ponavljanja zadataka, tada je vraćeni rezultat -1, tj. znak da zadatak treba biti odbačen.

# 7. Implementacija

Poglavlje obuhvaća prikaz jednostavne implementacije do sad opisane biblioteke. Implementacija će kao zadatak izvršavati nasumično generiranje brojeva. Postojat će više vrsti generatora, tj. generatori će generirati različite veličine brojeva. Zadatak će biti neuspješno izvršen ako generator generira neparni broj. Svi zadaci će se pohranjivati u memoriji programa.

```
public class NumberGenerator {

    private int minNumber;
    private int maxNumber;

    public NumberGenerator(int minNumber, int maxNumber) {
        this.minNumber = minNumber;
        this.maxNumber = maxNumber;
    }

    public int getMax() { return maxNumber; }

    public int getMin() { return minNumber; }

    @Override
    public String toString() {
        return "NumberGenerator{" +
                "minNumber=" + minNumber +
                ", maxNumber=" + maxNumber +
                '}';
    }
}
```

Slika 42. Prikaz *NumberGenerator* klase

Na slici broj 42 je prikazana klasa zadatka. Klasa ima dva atributa tipa *int* koji govore o rasponu broja koji se može generirati. Klasa ima jedan konstruktor koja prima dva parametra tipa *int*. Klasa također sadrži *get* funkcije te funkciju koja ispisuje objekt u obliku *String*-a.

```
private static BeeqTaskPoolSegment<NumberGenerator> getSmallPoolSegment() {
    return new BeeqTaskPoolSegment<>(
            name: "Small numbers segment",
            (task) -> task.getMaxNumber() <= 100,
            batchSize: 1,
            new BeeqRetryPolicy(Collections.singletonList(1)));
}

private static BeeqTaskPoolSegment<NumberGenerator> getLargePoolSegment() {
    return new BeeqTaskPoolSegment<>(
            name: "Large numbers segment",
            (task) -> task.getMaxNumber() > 100 && task.getMaxNumber() < 1000,
            batchSize: 1,
            new BeeqRetryPolicy(Collections.singletonList(1)));
}
```

Slika 43. Prikaz kreiranja segmenata

Na slici broj 43 su prikazane dvije funkcije koje kreiraju dva nova segmenta. Segmenti kao prvi parametar primaju svoj naziv. U ovom slučaju su to „*Small numbers segment*“ i „*Large numbers segment*“. Drugi parametar je *lambda*. Ona opisuje zadatke koji mogu pripasti odgovarajućem segmentu. U prvom slučaju su to zadaci koji generiraju brojeve do 100, a u drugom slučaju su to zadaci koji generiraju brojeve od 100 do 1000. Treći argument predstavlja broj uzastopno obrađenih zadataka za segment. Oba segmenta imaju *batchSize* jednak jedan. Četvrti argument predstavlja politiku ponavljanja izvršavanja zadatka. U ovom slučaju su obje vrijednosti liste koje sadrže broj jedan.

```
public class NumberGeneratorProcessor<T, R> implements BeeqTaskProcessor<T, R> {

    @Override
    public R process(BeeqTaskWrapper<T> task) throws Exception {
        NumberGenerator ng = (NumberGenerator) task.getTaskDescriptor();
        Random random = new Random();

        Integer generatedNumber = random.nextInt( bound: ng.getMax() - ng.getMin() + 1) + ng.getMin();

        if (generatedNumber % 2 != 0) {
            throw new Exception("Generated number can't be odd. Number: " + generatedNumber);
        }

        return (R) generatedNumber;
    }
}
```

Slika 44. Prikaz implementacije sučelja *BeeqTaskProcessor*

Na slici broj 44 je prikazana implementacija *BeeqTaskProcessor* sučelja. Procesor generira jedan nasumičan broj u zadanom rasponu koji se očitava iz parametra *task*.

Ako je broj neparan, baca se iznimka sa porukom, inače se vraća generirani broj.

```
public class OnNumberGeneratorCompletionHandler<T, R> implements BeeqOnTaskCompletionHandler<T, R> {
    @Override
    public void onTaskCompleted(BeeqTaskWrapper<T> completedTask, R completionResult) {
        System.out.println(Thread.currentThread().getName() + " Generated number " + completionResult);
    }

    @Override
    public void onTaskFailed(BeeqTaskWrapper<T> failedTask, int retryNumber, Exception error) {
        System.out.println(Thread.currentThread().getName() + " Task " + failedTask.getTaskDescriptor() +
                " failed " + retryNumber + " time(s), with error " + error.getMessage());
    }

    @Override
    public void onTaskDropped(BeeqTaskWrapper<T> failedTask, String message) {
        System.out.println(Thread.currentThread().getName() + " Task" + failedTask.getTaskDescriptor() +
                " was dropped. Reason: " + message);
    }
}
```

Slika 45. Prikaz implementacije sučelja *BeeqOnTaskCompletionHandler*

Na slici broj 45 je prikazana implementacija sučelja *BeeqOnTaskCompletionHandler*. *onTaskCompleted* ispisuje generirani broj. Funkcija *onTaskFailed* ispisuje broj pokušaja izvršavanja zadatka, te poruku zbog koje je izvršavanje bilo neuspješno. *onTaskDropped* ispisuje da je zadatak odbačen, te razlog odbacivanja zadataka. Svaka funkcija dodatno ispisuje ime dretve koja je pozvala tu funkciju.

```java
public class InMemoryBeeqTaskStorage<T> implements BeeqTaskStorage<T> {

    private Map<BeeqTaskPoolSegment<T>, List<BeeqTaskWrapper<T>>> tasks;
    private AtomicInteger taskCount = new AtomicInteger( initialValue: 0);

    public InMemoryBeeqTaskStorage() { this.tasks = new HashMap<>(); }

    @Override
    public void addTask(BeeqTaskWrapper<T> task, BeeqTaskPoolSegment<T> taskPoolSegment) {
        tasks.computeIfAbsent(taskPoolSegment, v -> new LinkedList<>()).add(task);
        taskCount.incrementAndGet();
    }

    @Override
    public BeeqTaskWrapper<T> getTask(BeeqTaskPoolSegment<T> taskPoolSegment) {
        if (tasks.get(taskPoolSegment) == null || tasks.get(taskPoolSegment).isEmpty())
            return null;

        BeeqTaskWrapper<T> task = tasks.get(taskPoolSegment).get(0);

        if (task.getTryAt().isAfter(LocalDateTime.now()))
            return null;

        taskCount.decrementAndGet();
        tasks.get(taskPoolSegment).remove(task);

        return task;
    }

    @Override
    public void addTaskPoolSegment(BeeqTaskPoolSegment<T> taskPoolSegment) { reorganizeTasks(taskPoolSegment); }

    @Override
    public List<BeeqTaskWrapper<T>> removeTaskPoolSegment(BeeqTaskPoolSegment<T> taskPoolSegment) {
        return tasks.remove(taskPoolSegment);
    }

    @Override
    public void reorganizeTasks(BeeqTaskPoolSegment<T> taskPoolSegment) {
        tasks.forEach((key, value) -> value
                .forEach(task -> {
                    if (taskPoolSegment.getTaskDiscriminator().doesMatch(task.getTaskDescriptor())) {
                        value.remove(task);
                        this.addTask(task, taskPoolSegment);
                    }
                }));
    }

    @Override
    public boolean isEmpty() { return taskCount.intValue() <= 0; }
```

Slika 46. Prikaz implementacije sučelja *BeeqTaskStorage*

Na slici 46 je prikazana implementacija sučelja *BeeqTaskStorage*. Klasa sadrži dva atribut. Atribut *tasks* je *hash* mapa u kojoj ključ predstavlja segment dok je vrijednost svakog zapisa lista zadataka. *taskCount* atribut otkriva ukupni broj zadataka u pohrani. Funkcija *addTask* dodaje novi zadatak u mapu zadatka, te povećava atribut *taskCount*. Funkcija *getTask* pronalazi zadatak na osnovu predanog segmenta. U slučaju da za predani segment ne postoji ni jedan zadatak funkcija vraća *null* vrijednost. Ako je rok izvršavanja zadatka prošao, također se vraća *null* vrijednost. Inače se vraća pronađeni zadatak i smanjuje se vrijednost *taskCount* atributa. Funkcija

*addTaskPoolSegment* poziva funkciju *reorganizeTasks* te joj predaje primljeni segment. Funkcija *reorganizeTasks* provjerava pripada li neki od postojećih zadataka u novi segment. *RemoveTaskPoolSegment* funkcija briše segment iz mape, te sve njegove zadatke vraća kao rezultat. Funkcija *isEmpty* vraća istinu ako je broj *task*-ova u pohrani jednak nula, inače vraća laž.

```
public class UsageExample {
    public static void main(String[] args) {
        NumberGeneratorProcessor<NumberGenerator, Integer> processor = new NumberGeneratorProcessor<>();
        InMemoryBeeqTaskStorage<NumberGenerator> storage = new InMemoryBeeqTaskStorage<>();
        OnNumberGeneratorCompletionHandler<NumberGenerator, Integer> completionHandler =
                new OnNumberGeneratorCompletionHandler<>();

        Beeq<NumberGenerator, Integer> beeq = new Beeq<>(storage, processor, completionHandler,  workerCount: 2);

        BeeqTaskPoolSegment<NumberGenerator> smallNumber = getSmallPoolSegment();
        BeeqTaskPoolSegment<NumberGenerator> largeNumbers = getLargePoolSegment();

        beeq.createSegment(smallNumber);
        beeq.createSegment(largeNumbers);

        NumberGenerator numberGeneratorSmallOne = new NumberGenerator(1, 100);
        NumberGenerator numberGeneratorSmallTwo = new NumberGenerator(1, 100);
        NumberGenerator numberGeneratorLarge = new NumberGenerator(101, 1000);
        NumberGenerator numberGeneratorExtraLarge = new NumberGenerator(1001, 10000);

        beeq.submitTask(numberGeneratorSmallOne);
        beeq.submitTask(numberGeneratorSmallTwo);
        beeq.submitTask(numberGeneratorLarge);
        beeq.submitTask(numberGeneratorExtraLarge);
    }

    private static BeeqTaskPoolSegment<NumberGenerator> getSmallPoolSegment() {
        return new BeeqTaskPoolSegment<>(
                 name: "Small numbers segment",
                (task) -> task.getMax() <= 100,
                 batchSize: 1,
                new BeeqRetryPolicy(Collections.singletonList(1)));
    }

    private static BeeqTaskPoolSegment<NumberGenerator> getLargePoolSegment() {
        return new BeeqTaskPoolSegment<>(
                 name: "Large numbers segment",
                (task) -> task.getMax() > 100 && task.getMax() < 1000,
                 batchSize: 1,
                new BeeqRetryPolicy(Collections.singletonList(1)));
    }
}
```

Slika 47. Prikaz glavne funkcije programa

Na slici 47 je prikaz glavne funkcije koja kreira sve potrebne objekte te poziva funkcije za dodavanje zadatka iz *Beeq* klase. Prilikom kreiranja klase *Beeq*, parametar koji označava broj dretvi ima vrijednost dva.

U nastavku slijede dva primjera ispisa iz konzole.

```
Beeq-Worker-1 Generated number 96
Beeq-Worker-2 Generated number 602
Beeq-Worker-1 Task NumberGenerator{minNumber=1, maxNumber=100} failed 1 time(s), with error Generated number can't be odd. Number: 91
Beeq-Worker-2 Task NumberGenerator{minNumber=1001, maxNumber=10000} failed 1 time(s), with error Generated number can't be odd. Number: 2481
Beeq-Worker-2 TaskNumberGenerator{minNumber=1001, maxNumber=10000} was dropped. Reason: Maximum number of retries reached
Beeq-Worker-1 TaskNumberGenerator{minNumber=1, maxNumber=100} was dropped. Reason: Maximum number of retries reached
```

Slika 48. Prikaz ispisa iz konzole primjer jedan

1. Dretva broj jedan uzima prvi zadatak. Zadatak je uspješan.
2. Dretva broj dva uzima drugi zadatak. Zadatak je uspješan.
3. Dretva broj jedan uzima treći zadatak. Zadatak je neuspješan jer generira neparni broj.
4. Dretva broj dva uzima četvrti zadatak. Zadatak je neuspješan jer generira neparni broj.
5. Dretva broj dva ispisuje poruku kako je četvrti zadatak odbačen zbog prekoračenja maksimalnog broja pokušaja.
6. Dretva broj jedan ispisuje poruku kako je treći zadatak odbačen radi prekoračenja maksimalnog broja pokušaja.

```
Beeq-Worker-2 Generated number 922
Beeq-Worker-1 Task NumberGenerator{minNumber=1, maxNumber=100} failed 1 time(s), with error Generated number can't be odd. Number: 95
Beeq-Worker-2 Generated number 16
Beeq-Worker-2 Generated number 7264
Beeq-Worker-1 TaskNumberGenerator{minNumber=1, maxNumber=100} was dropped. Reason: Maximum number of retries reached
```

Slika 49. Prikaz ispisa iz konzole primjer dva

1. Dretva broj dva uzima drugi zadatak. Zadatak je uspješan.
2. Dretva broj jedan uzima prvi zadatak. Zadatak je neuspješan jer generira neparni broj.
3. Dretva broj dva uzima treći zadatak. Zadatak je uspješan.
4. Dretva broj dva uzima četvrti zadatak. Zadatak je uspješan.
5. Dretva broj jedan ispisuje poruku kako je prvi zadatak odbačen zbog prekoračenja maksimalnog broja pokušaja.

Može se primijetiti kako zadaci manjeg tipa nikada nisu u obradi u isto vrijeme. Razlog tome je postavljanje atributa *batchSize* za svaki segment na vrijednost jedan.

# 8. Korištene tehnologije

Tehnologije korištene za izradu programskog rješenja su programski jezik Java, upravljač paketa Maven te integrirano razvojno okruženje IntelliJ IDEA.

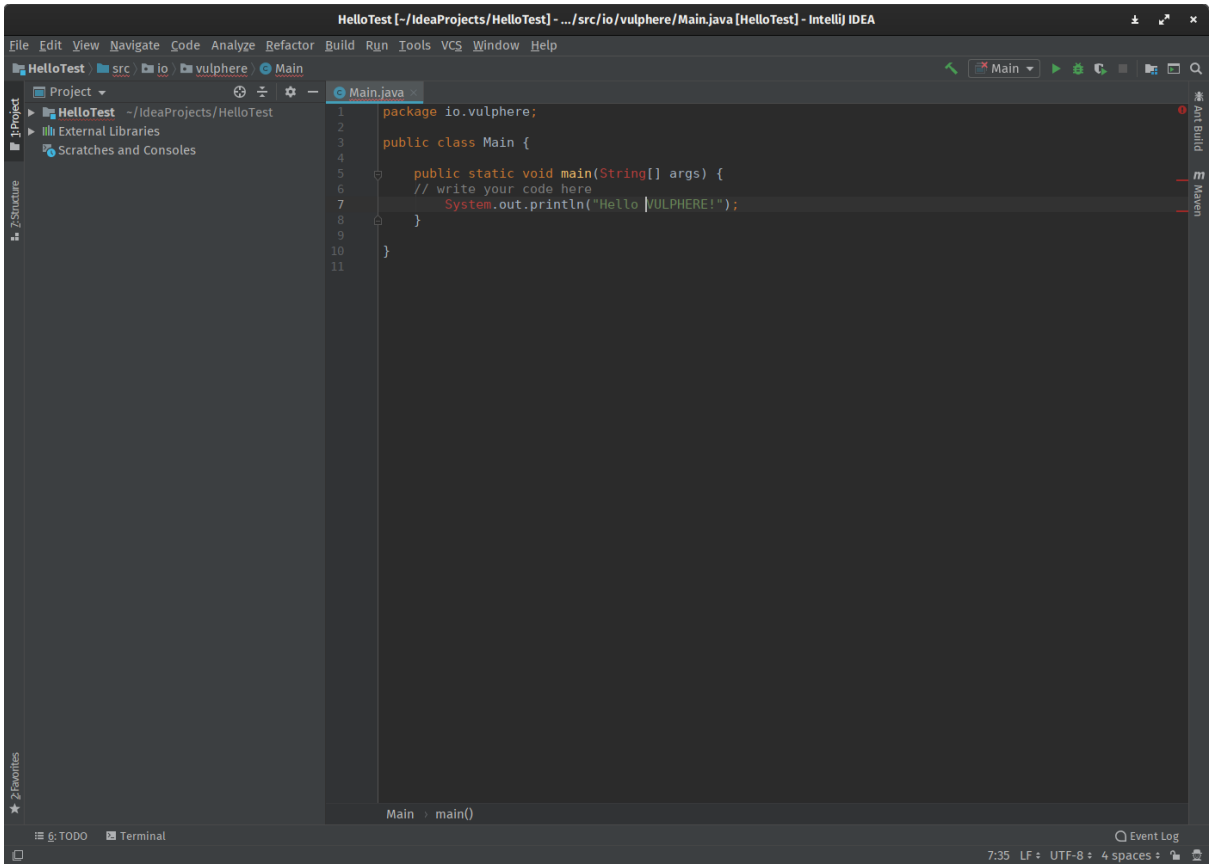
Java je objektno-orijentirani programski jezik opće namjene koji je osmišljen tako da ima što je manje moguće ovisnosti o implementaciji. Javu je originalno razvijao James Gosling u Sun Microsystems, te je izdana 1995. godine kao glavna komponenta Sun Microsystems Java platforme. (Sierra & Bates, 2005)

Ideja iza Jave je omogućiti programerima da pišu jednom i pokreću svuda (*write once, run anywhere*). Što znači da se kompilirani Java kod može pokrenuti na bilo kojoj platformi koja podržava Javu, bez da se kod ponovno kompilira. Java kod se kompilira u bajt kod koji može biti pokrenut na bilo kojem Java virtualnom stroju, bez obzira na samu arhitekturu računalnog sustava. Sintaksa je vrlo slična programskim jezicima C i C++, s time da ima manje doticaja sa nižim nivoom programiranja.

Maven je alat za automatizaciju gradnje i održavanja programskog rješenja te je održavan i razvijen od strane Apache Software Foundation-a. Primarno je korišten za Java projekte. Maven dinamički preuzima Java biblioteke i Maven dodatke iz jednog ili više repozitorija, kao što je Maven 2 Central Repository, i pohranjuje ih u lokalnu predmemoriju. (Sonatype, 2008)

Maven se također može koristiti za projekte koji su napisani u C#, Ruby, Scala, i mnogim drugima programskim jezicima.

IntelliJ IDEA je Java integrirano razvojno okruženje za razvoj računalnih programa. Alat je razvijen od tvrtke JetBrains, prethodno poznati kao IntelliJ. Prva verzija alata je izašla u siječnju 2001. godine, te je bio jedan od prvih naprednijih integriranih razvojnih okruženja za Java programere. Alat pruža veliki broj mogućnosti kao što su dovršavanje koda, analiziranje konteksta, refaktoriranje, *linting*, verzioniranje, te mnoge druge. (Krochmalski, 2014)




Slika 50. Prikaz alata IntelliJ IDEA

# 9. Zaključak

U ovom radu je detaljno objašnjeno razvijanje biblioteke koja pruža korisniku način da asinkrono izvršava svoje zadatke uz dodatne mogućnosti raspoređivanja istih. Obrađene su teme sinkronog i asinkronog modela programiranja te razlike između njih. Opisane su dretve te načini na koji se kod može napisati tako da je siguran u radu sa više dretvi. Biblioteka je napisana u Java programskom jeziku. Prilikom razvoja korišten je Maven kao alat za automatsku gradnju programa i održavanje samog programskog rješenja. Također je korišteno integrirano razvojno okruženje IntelliJ IDEA.

Biblioteka može biti korištena u svrhu automatizacije i brzog izvršavanja raznih asinkronih zadataka. Projekt je otvorenog koda tako da može biti korišten od bilo koga bez ikakve naknade.

# Literatura


Bloch, J. (2018). *Effective Java.* Addison-Wesley Professional.

Butcher, P. (2014). *Seven Concurrency Models in Seven Weeks: When Threads Unravel.* Pragmatic Bookshelf.

Downey, A. B. (2016). *The Little Book of Semaphores.*

Evans, B., Gough, J., & Newland, C. (2018). *Optimizing Java: Practical Techniques for Improving JVM Application Performance.* O'Reilly Media.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). *Java Concurrency in Practice.* Addison-Wesley Professional.

Gonzalez, J. F. (2016). *Mastering Concurrency Programming with Java 8.* Packt Publishing.

Herlihy, M. (2012). *The Art of Multiprocessor Programming.* Morgan Kaufmann.

Kleiman, S. (1996). *Programming With Threads.* Prentice Hall.

Krochmalski, J. (2014). *IntelliJ IDEA Essentials.* Packt Publishing.

Naftalin, M., & Wadler, P. (2006). *Java Generics and Collections: Speed Up the Java Development Process.* O'Reilly Media.

Oaks, S. (2014). *Java Performance: The Definitive Guide: Getting the Most Out of Your Code.* O'Reilly Media.

Oaks, S., & Wong, H. (2004). *Java Threads: Understanding and Mastering Concurrent Programming.* O'Reilly Media.

Sierra, K., & Bates, B. (2005). *Head First Java.* O'Reilly Media.

Sonatype. (2008). *Maven: The Definitive Guide.* O'Reilly Media.

Subramaniam, V. (2011). *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors.* Pragmatic Bookshelf.

Bloch, J. (2018). *Effective Java.* Addison-Wesley Professional.

Butcher, P. (2014). *Seven Concurrency Models in Seven Weeks: When Threads Unravel.* Pragmatic Bookshelf.

Downey, A. B. (2016). *The Little Book of Semaphores.*

Evans, B., Gough, J., & Newland, C. (2018). *Optimizing Java: Practical Techniques for Improving JVM Application Performance.* O'Reilly Media.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). *Java Concurrency in Practice.* Addison-Wesley Professional.

Gonzalez, J. F. (2016). *Mastering Concurrency Programming with Java 8.* Packt Publishing.

Herlihy, M. (2012). *The Art of Multiprocessor Programming.* Morgan Kaufmann.

Kleiman, S. (1996). *Programming With Threads.* Prentice Hall.

Krochmalski, J. (2014). *IntelliJ IDEA Essentials.* Packt Publishing.

Naftalin, M., & Wadler, P. (2006). *Java Generics and Collections: Speed Up the Java Development Process.* O'Reilly Media.

Oaks, S. (2014). *Java Performance: The Definitive Guide: Getting the Most Out of Your Code.* O'Reilly Media.

Oaks, S., & Wong, H. (2004). *Java Threads: Understanding and Mastering Concurrent Programming.* O'Reilly Media.

Sierra, K., & Bates, B. (2005). *Head First Java.* O'Reilly Media.

Sonatype. (2008). *Maven: The Definitive Guide.* O'Reilly Media.

Subramaniam, V. (2011). *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors.* Pragmatic Bookshelf.

# Popis slika

## Sažetak

Rad detaljno prikazuje razvijane biblioteke koja korisniku pruža način da asinkrono izvršava svoje zadatke uz dodatne mogućnosti raspoređivanja istih. Također je prikazan jednostavan primjer korištenja biblioteke. U teoretskom dijelu objašnjeno je sinkroni i asinkroni model programiranja. Dotaknute se dretve te način na koji se kod osigurava za rad sa većim brojem dretvi. Opisane su kolekcije i klase u Javi koje su napravljene u svrhu sigurnog rada s dretvama. U praktičnom dijelu rada je prikazana implementacija biblioteke

Ključne riječi: asinkronost, sinkronost, dretve, konkurentnost, višedretvenost, dead lock, locking, sinkroniziranost, atomske varijable

## Abstract

This thesis shows the process of making the library that allows the user to schedule and prioritize various asynchronous tasks. It also shows a simple usage example. The theoretical part describes synchronous and asynchronous programming. The subject of threads is touched upon, and the ways to make code more thread-safe. Java classes and collections for thread-safe interactions are described aswell.

Key words: async, sync, threads, concurrency, multithreading, deadlock, locks, synchronous, atomic variables