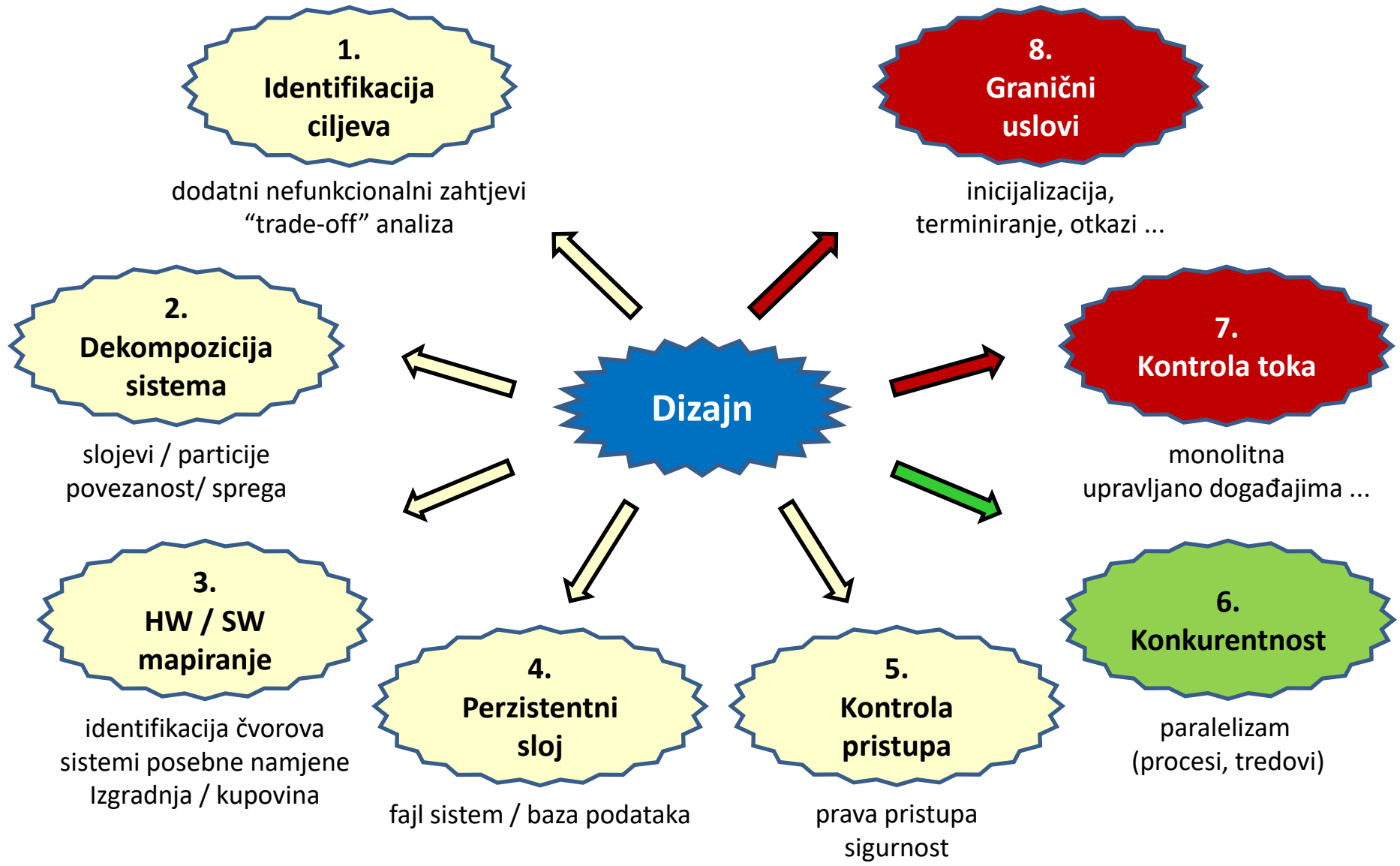
UNIVERZITET U BANJOJ LUCI
ELEKTROTEHNIČKI FAKULTET

Prof. dr Dražen Brđanin

PROJEKTOVANJE SOFTVERA
/konkurentnost/

Banja Luka
2024.

8 bitnih aktivnosti u projektovanju



6. Konkurentnost

Sadržaj prezentacije

- pasivni i aktivni objekti
- sekvencijalne i konkurentne aplikacije
- problemi u komunikaciji konkurentnih procesa
- sinhronizacija konkurentnih procesa
- razmjena podataka između konkurentnih procesa
- identifikacija konkurentnosti u dinamičkom modelu sistema
- mapiranje dinamičkog modela
- implementacija konkurentnosti

6. Konkurentnost

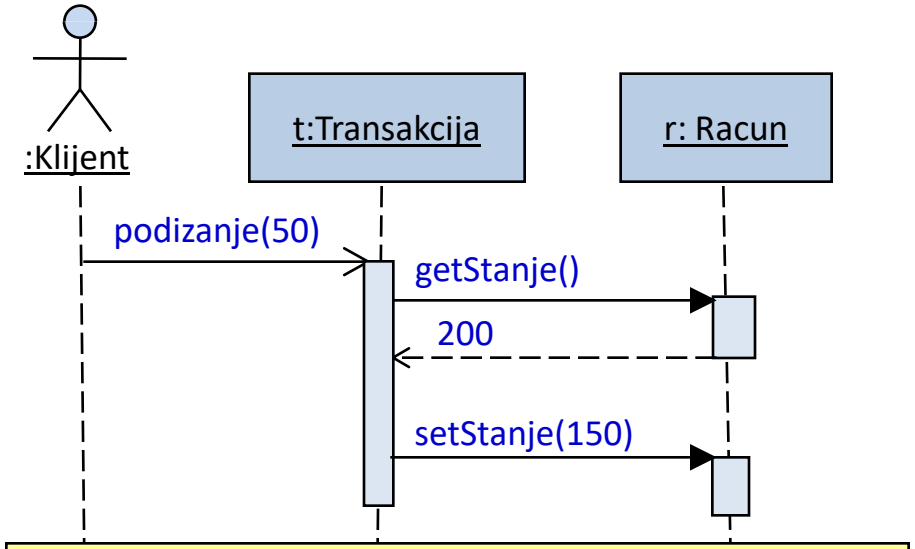
Pasivni i aktivni objekti

Pasivni objekti

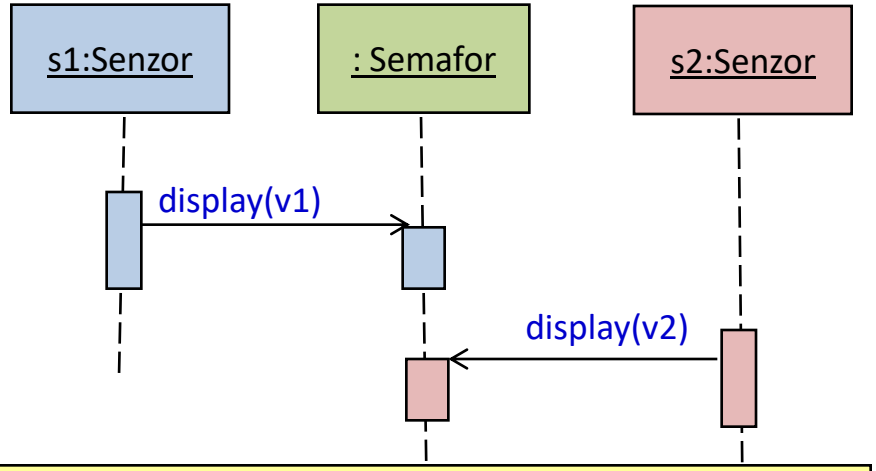
- objekti koji čekaju poruku od drugog objekta i izvršavaju operaciju na zahtjev
- nikad ne iniciraju akcije

Aktivni objekti

- objekti koji iniciraju akcije i koji se izvršavaju nezavisno od drugih objekata
- **još se nazivaju i konkurentni objekti (konkurentni procesi)**



Racun je pasivni objekat
Račun čeka poruku od transakcije, izvršava operaciju i vraća rezultat



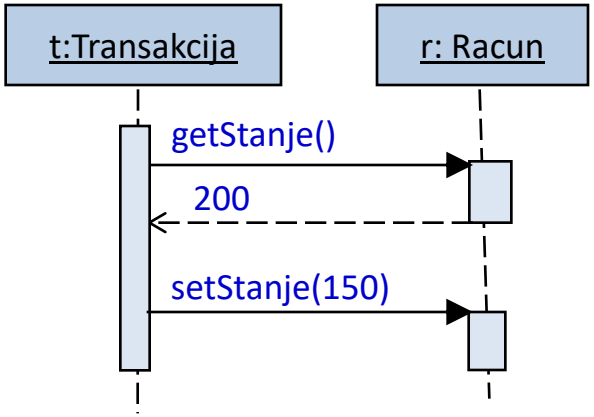
Senzori su aktivni objekti
Senzori mogu istovremeno, nezavisno jedan od drugog, da mjere vrijednosti i šalju semaforu

6. Konkurentnost

Sekvencijalne i konkurentne aplikacije

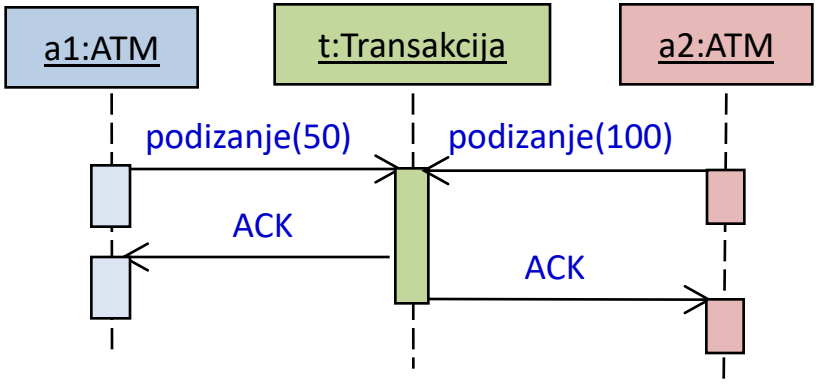
Sekvencijalne aplikacije

- sekvencijalni program čine **samo pasivni objekti**
- Kad neki objekat pozove operaciju na drugom objektu, kontrola se prenosi u pozvanu operaciju i pozivajući objekat čeka rezultat da bi nastavio izvršavanje
- U sekvencijalnim programima u komunikaciji se primjenjuje komunikacioni obrazac **sinhrona poruka sa odgovorom**



Konkurentne aplikacije

- konkurentni program ima više aktivnih (konkurentnih) objekata, pri čemu svaki aktivni objekat ima sopstveni kontrolni tok
- U konkurentnim programima **primjenjuje se asinhrona komunikacija**
- Konkurentni objekat šalje asinhronu poruku drugom objektu i nastavlja izvršavanje (odredišni objekat ima bafer u kojem drži sve primljene poruke, koje procesira kad bude raspoloživ)

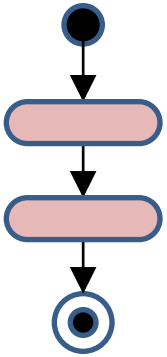


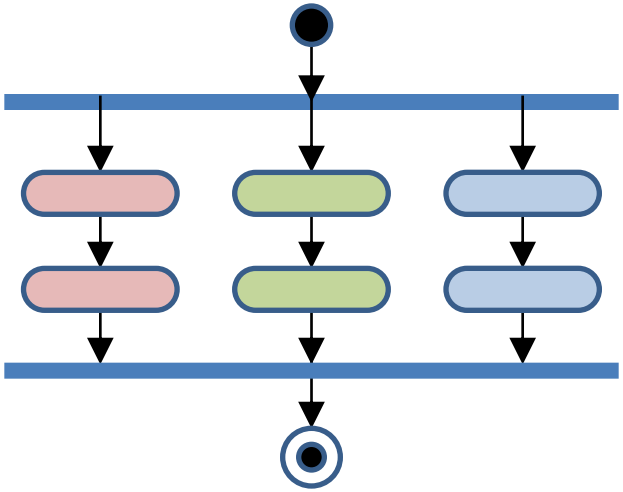
6. Konkurentnost

Konkurentni objekti

- Alternativno se nazivaju: **aktivni objekti, konkurentni procesi, konkurentni zadaci, niti**
- **Konkurentni objekat ima sopstvenu nit kontrole toka i može da se izvršava nezavisno od drugih objekata**
- **Pasivni objekti** nemaju sopstvenu nit kontrole toka, već se **izvršavaju unutar niti nekog aktivnog objekta**
- **Konkurentni objekat predstavlja izvršavanje jednog sekvencijalnog programa ili jedne sekvencijalne komponente konkurentnog programa**
- Svaki konkurentni objekat ima svoj sekvencijalni tok

sekvencijalni tok





konkurentni tokovi

6. Konkurentnost

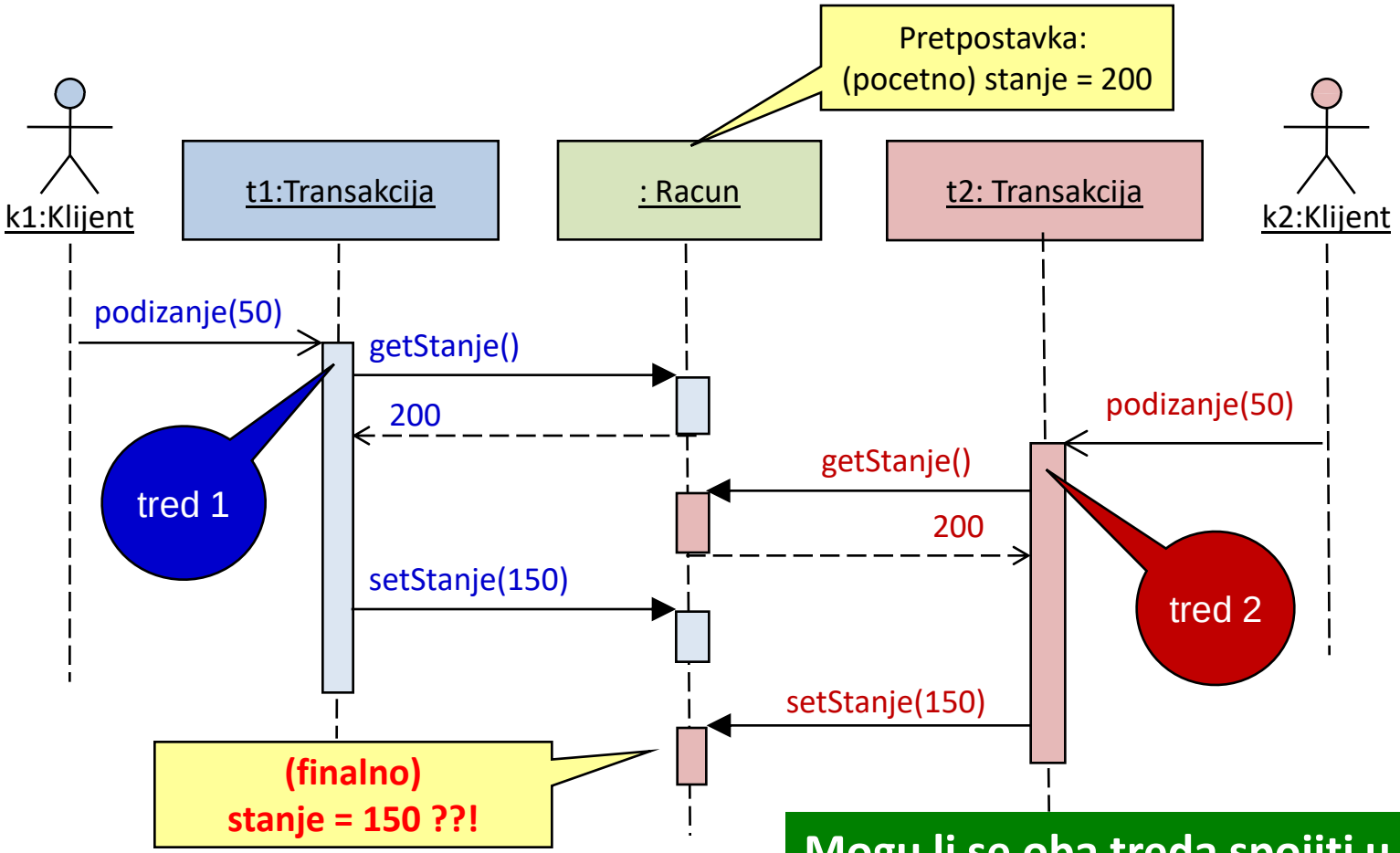
Problemi u komunikaciji konkurentnih procesa

- **Konkurentni procesi izvršavaju se asinhrono (različitim brzinama)**
- **Konkurentni objekti međusobno asinhrono komuniciraju** (s vremena na vrijeme) i moraju da se sinhronizuju
- U komunikaciji konkurentnih objekata **javljaju se problemi koji ne postoje u sekvencijalnim procesima:**
  - **uzajamna isključivost (*mutual exclusion*)**
    - **kad više konkurentnih objekata treba ekskluzivan pristup istom resursu** (npr. pristup istom dijeljenom podatku ili fizičkom uređaju)
  - **proizvođač-potrošač (*producer-consumer*)**
    - **kad jedan aktivni objekat šalje podatke drugom aktivnom objektu**
  - **sinhronizacija**
    - **kad izvršavanje više konkurentnih objekata treba da se sinhronizuje**

6. Konkurentnost

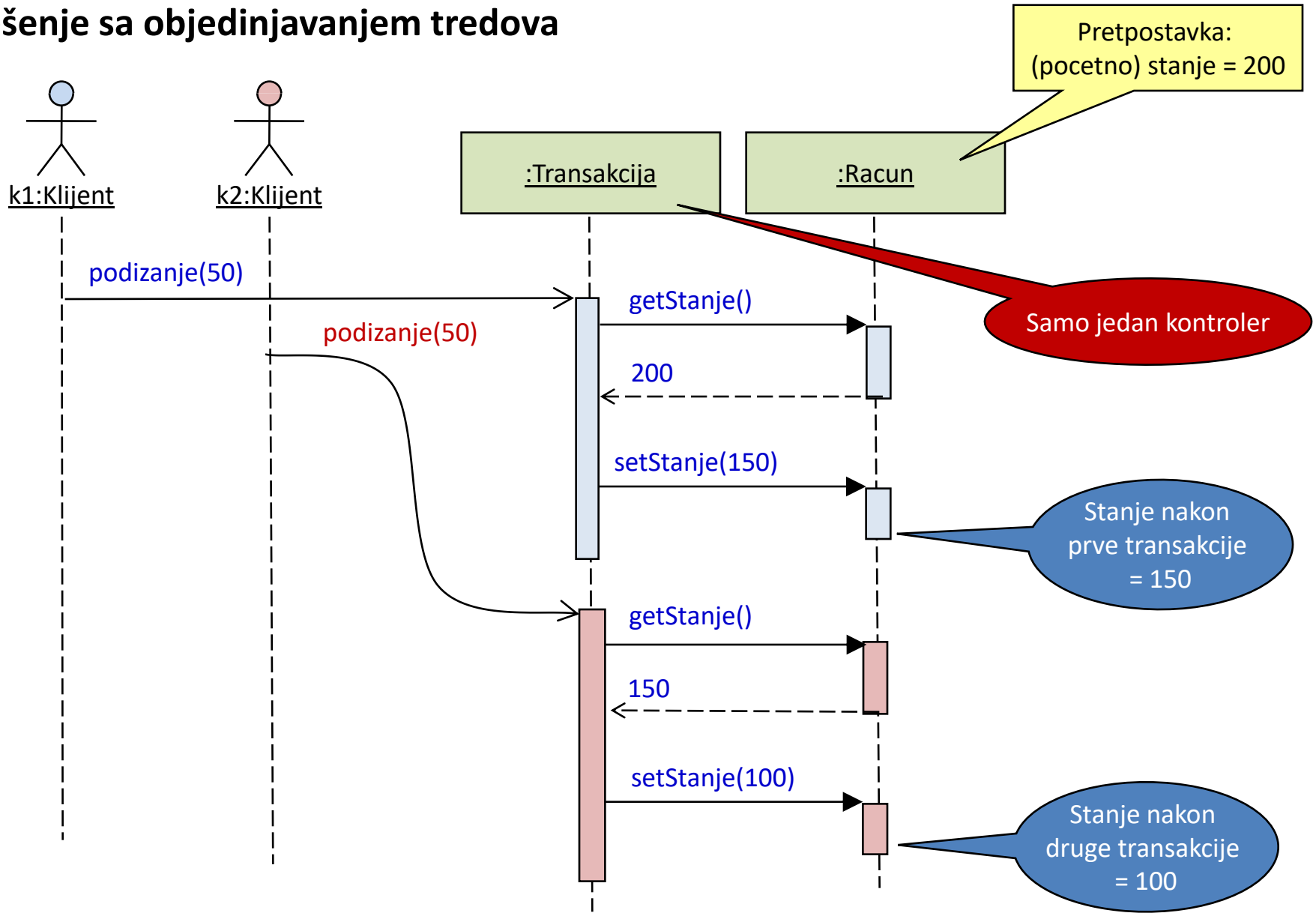
Uzajamna isključivost konkurentnih procesa

– **više konkurentnih objekata treba ekskluzivan pristup istom resursu**
(npr. pristup istom dijeljenom podatku ili fizičkom uređaju)



6. Konkurentnost

Rješenje sa objedinjavanjem tredova



6. Konkurentnost

Sinhronizacija konkurentnih procesa

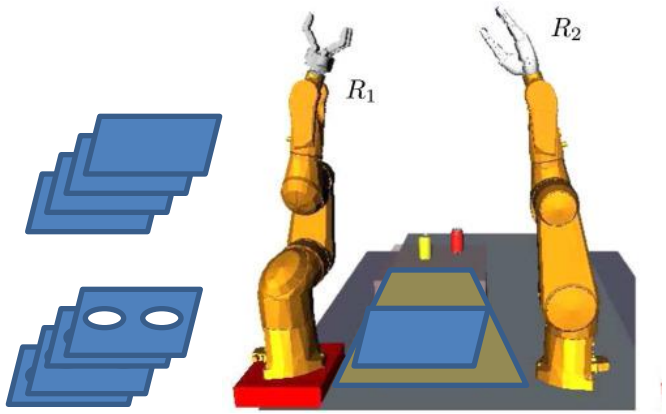
- **Sinhronizacija više konkurentnih procesa bez međusobne razmjene podataka**
- **Mehanizam za sinhronizaciju – signaliziranje događaja**
  - Odredišni objekat je u stanju čekanja signala (operacija wait)
  - Izvorni objekat signalizira događaj – šalje asinhronu poruka (signal, notify)

Primjer: sinhronizacija dva robota

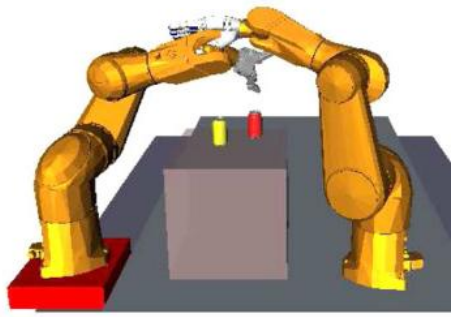
❶ robot R_1 postavlja objekat

❷ robot R_2 buši rupe

❸ robot R_1 odnosi objekat i postavlja novi



Potencijalni problem ako nema sinhronizacije, jer se radni prostor oba robota preklapa pa može da dođe do havarije



6. Konkurentnost

Primjer: sinhronizacija dva robota

❶ robot R_1 postavlja objekat

❷ robot R_2 buši rupe

❸ robot R_1 odnosi objekat i postavlja novi

Dva sinhronizaciona problema (kolizije):

- **R_1 mora da postavi objekat i da se izmakne prije nego što R_2 uđe u radni p.**
- **R_2 mora da završi bušenje i da se izmakne prije nego što R_1 uđe u radni p.**

Rješenje: signalizacija događaja

- R_1 postavi objekat, izmakne se i signalizira da je objekat spreman (*ObjectReady*)
- R_2 prima signal *ObjectReady* i iz stanja čekanja započinje manipulaciju
- R_2 završi bušenje, izmakne se i signalizira da je objekat završen (*ObjectCompleted*)
- R_1 prima signal *ObjectCompleted* i odnosi završeni objekat

Proces R_1

```
while (true)
{
   postaviNoviDio(),
   odmakniSe()
   signal(ObjectReady)
   wait(ObjectCompleted)
   odnesiZavrseniDio()
}
```

Proces R_2

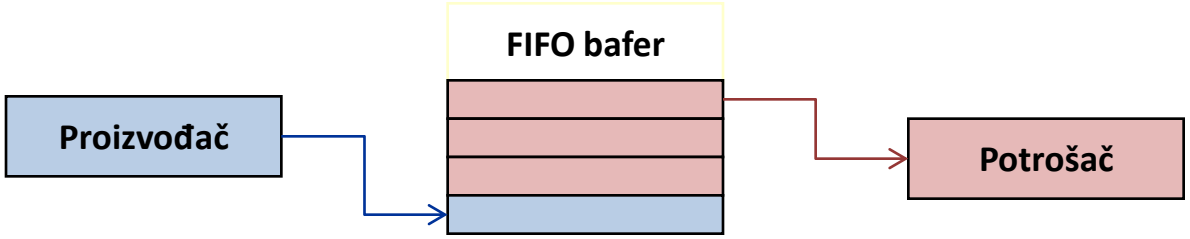
```
while (true)
{
   wait(ObjectReady)
   busenje(),
   odmakniSe()
   signal(ObjectCompleted)
}
```

6. Konkurentnost

Proizvođač-potrošač (*producer-consumer*)

- **Proizvođač generiše podatke koje prima potrošač**
- **Potrebna je odgovarajuća sinhronizacija, kako bi došlo do razmjene podataka**
  - proizvođač mora biti spreman da šalje podatke, a potrošač mora biti spreman da ih primi
  - ako je potrošač spreman za prijem, a proizvođač još ne može da ih pošalje, potrošač mora da čeka
  - ako proizvođač može da pošalje podatke prije nego što ih potrošač može primiti, tada proizvođač mora da čeka ili podaci moraju biti baferovani prije nego što ih potrošač preuzme

- **Tipična rješenja:**
  - **asinhrone poruke proizvođač → potrošač** (sa gubitkom informacija)
  - **FIFO bafer između proizvođača i potrošača** (bez gubitka informacija)



6. Konkurentnost

Identifikacija konkurentnosti u dinamičkom modelu sistema

- **Za dva objekta kažemo da su inherentno konkurentna ako istovremeno mogu da prime neku poruku ili signal, nezavisno jedan od drugog:**
  - to može da bude ista poruka/signal ili nezavisne poruke/signali
- **Osnov za identifikaciju konkurentnosti:**
  - objekti u dijagramu sekvence koji mogu simultano da primaju poruke/signale

Pitanja prilikom identifikacije konkurentnosti

- **Da bismo identifikovali konkurentne tredove treba postavljati sljedeća pitanja:**
  - Da li sistem omogućava (istovremeni) pristup većem broju korisnika?
  - Koji kontrolni objekti mogu da se izvršavaju nezavisno jedan od drugog?
  - Može li se neki upit prema sistemu dekomponovati u više upita i da li se ti upiti mogu rješavati u paraleli?
    - npr. pretraživanje u distribuiranoj bazi podataka
    - npr. segmentirano pretraživanje slika
    - ...

6. Konkurentnost

Mapiranje dinamičkog modela u tredove

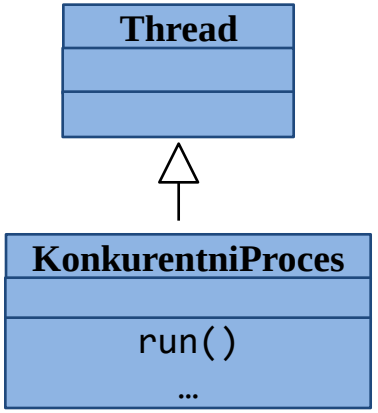
- **Inherentno konkurentni objekti tipično se raspoređuju u različite upravljačke niti (tredove)**
- **Objekti sa uzajamno isključivom (*mutual exclusive*) aktivnošću raspoređuju se u isti tred** (npr. pristup istom objektu koji ima za cilj promjenu stanja)
- **Tred je putanja kroz stanja u dijagramu stanja u kojima je neki objekat stalno aktivan:**
  - **Tred se zadržava** u nekom stanju sve dok objekat čeka:
    - rezultat poruke koju je poslao drugom objektu, ili
    - dok čeka neku poruku drugog objekta.
  - **Cijepanje treda**: kad objekat šalje ne-blokirajuću poruku (signal) drugom objektu

6. Konkurentnost

Implementacija konkurentnosti

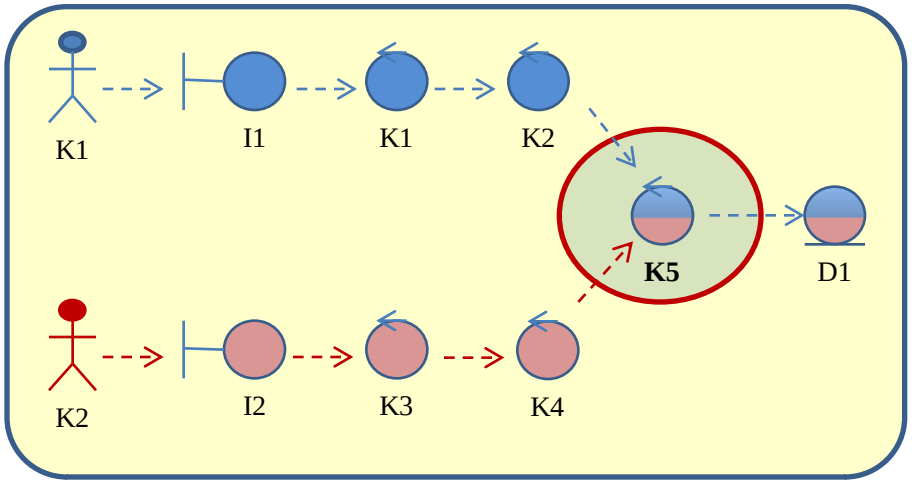
- Konkurentnost može da se realizuje u sistemima kod kojih je omogućena:
  - **fizička konkurentnost**
    - tredovi se raspoređuju na različite hardverske resurse
    - višeprocesorske mašine, klasteri, ...
  - **logička konkurentnost**
    - softverska realizacija tredova (neki jezici imaju mogućnost, npr. java)

```
public class KonkurentniProces extends Thread
      {
        public void run()
        {
           while (true)
           // task body
        }
      }
```



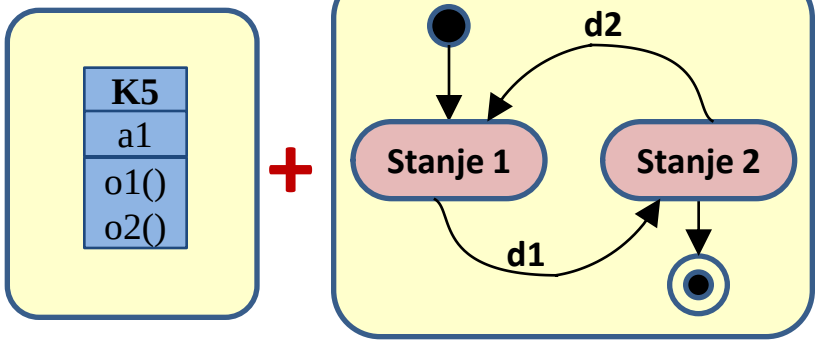

```
// instanciranje objekta i pokretanje niti
KonkurentniProces kp = new KonkurentniProces();
kp.start(); // poziva se run() metoda i sad imamo dvije niti
```

6. Konkurentnost



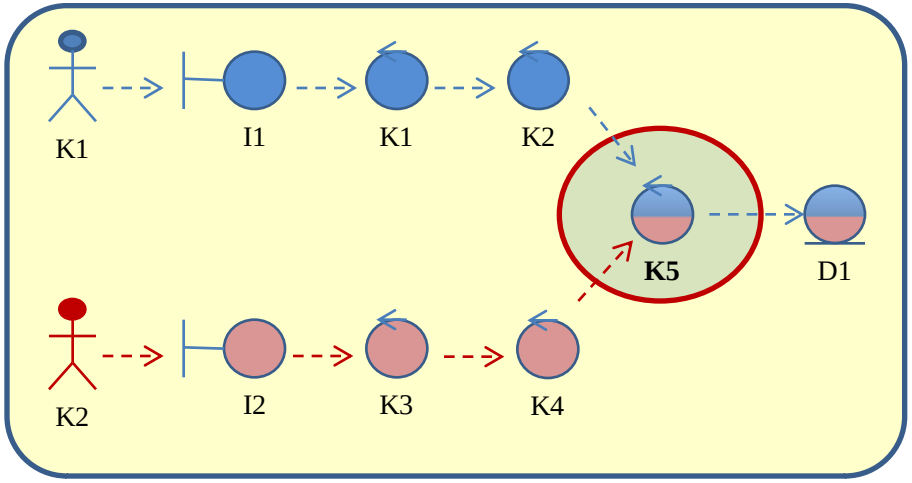
dijagram klasa

dijagram stanja




```
public class K5 extends Thread
{
  private int state=1;
  //...
  public void run()
  {
    while (true)
    {
      if (baf.get()==1 && state==1)
      {
        stanje=2;
        // ...
      }
      if (baf.get()==2 && state==2)
      {
        stanje=1;
        // ...
      }
    }
}
```

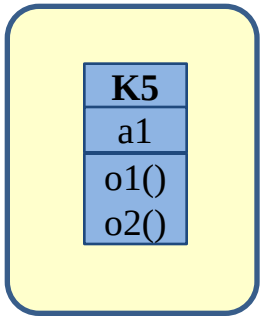
6. Konkurentnost



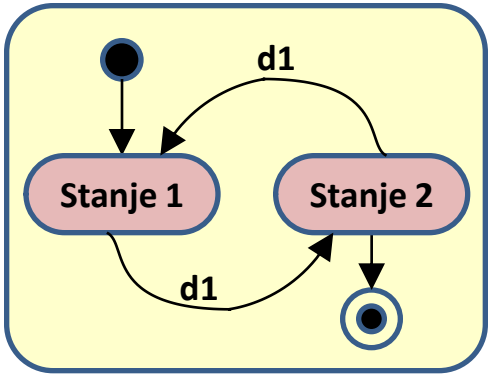
STATE
projektni obrazac

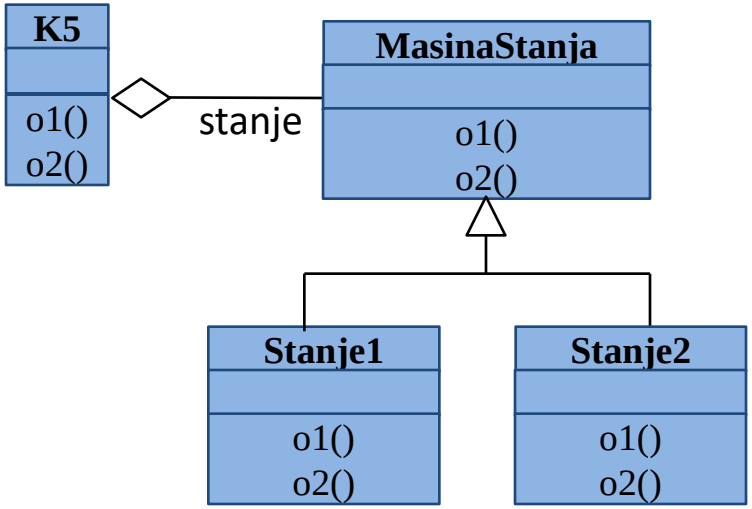
Detaljnije će biti prikazano kod projektnih obrazaca

dijagram klasa



dijagram stanja



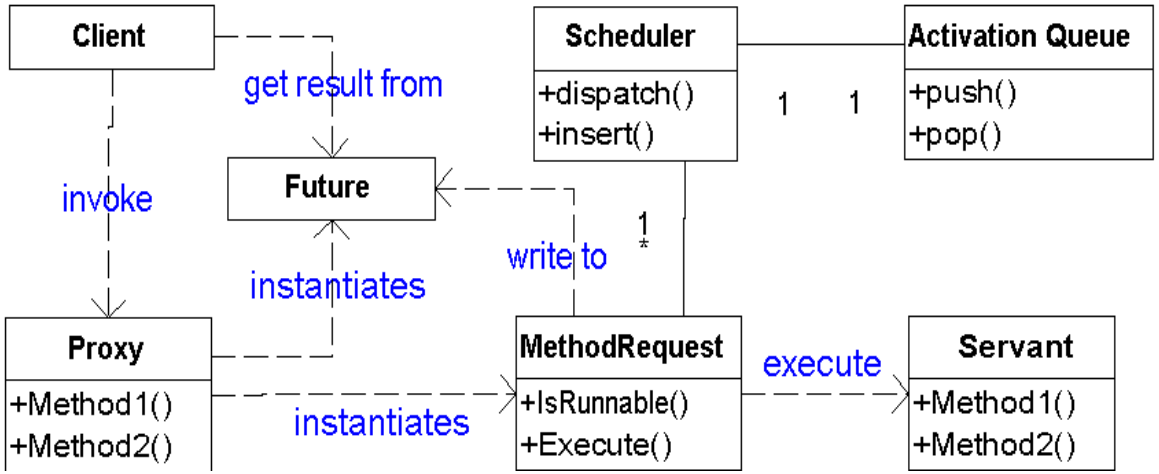


6. Konkurentnost

Projektovanje i implementacija aktivnih objekata

- **Projektni obrazac: *Active object* (*Actor Pattern*)**
  - uobičajena primjena u distribuiranim sistemima koji zahtijevaju višenitne servere
  - omogućava razdvajanje aplikativne logike za poziv i izvršavanje metode

struktura



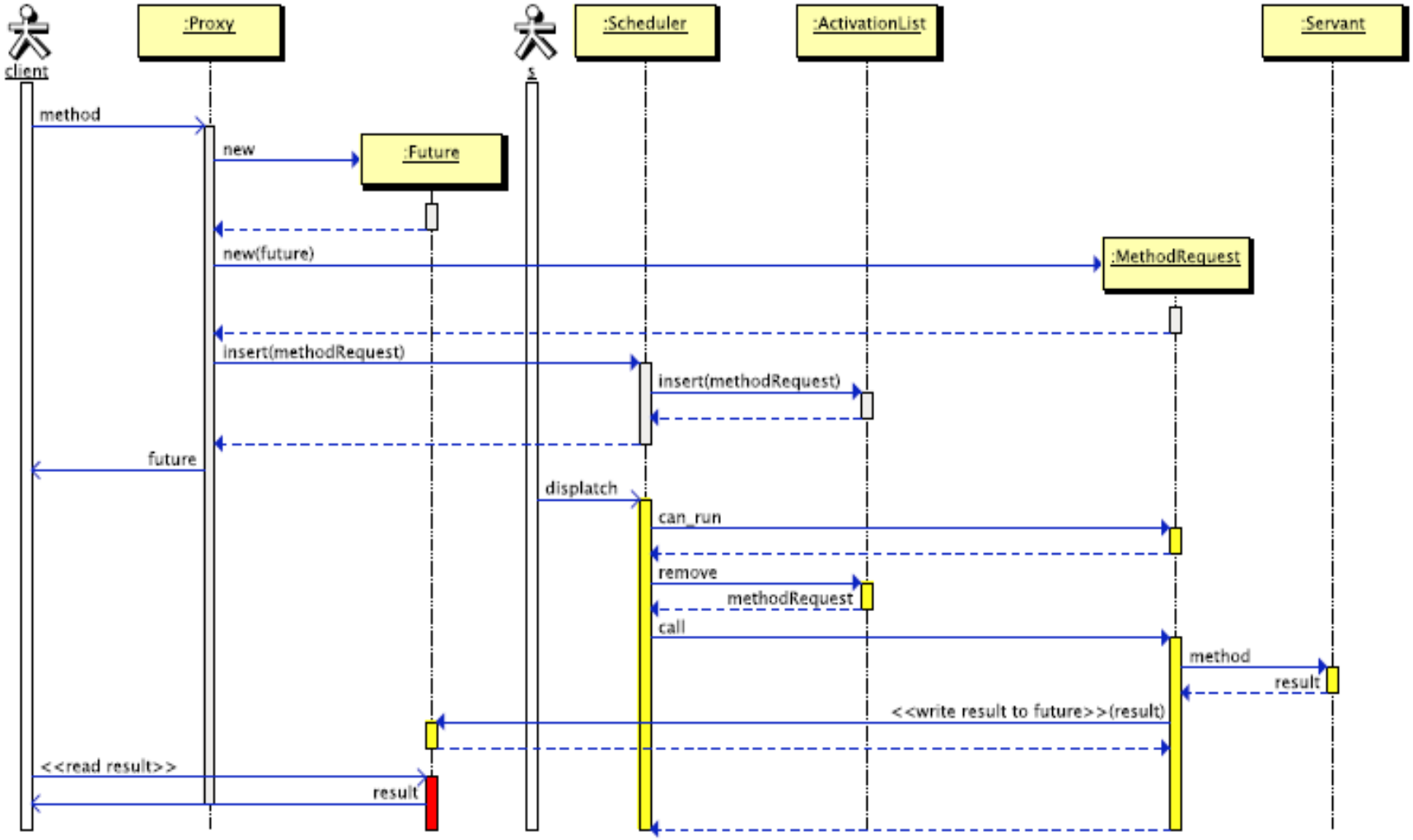
- **Proxy** pruža odgovarajući interfejs klijentu – omogućava poziv metoda
- **MethodRequest j**e objekat koji reprezentuje poziv metode (svaki poziv metode inkapsulira se u objekat)
- **Scheduler** upravlja izvršavanjem zahtjeva (raspoređuje ih na servanta kad su spremni za izvršavanje)
- **ActivationQueue** sadrži sve zahtjeve koji čekaju na izvršavanje
- **Servant** implementira metode
- **Future** omogućava klijentu da preuzme rezultat poziva

6. Konkurentnost

Projektovanje i implementacija aktivnih objekata

– **Projektni obrazac: *Active object* (*Actor Pattern*)**

Interakcija objekata

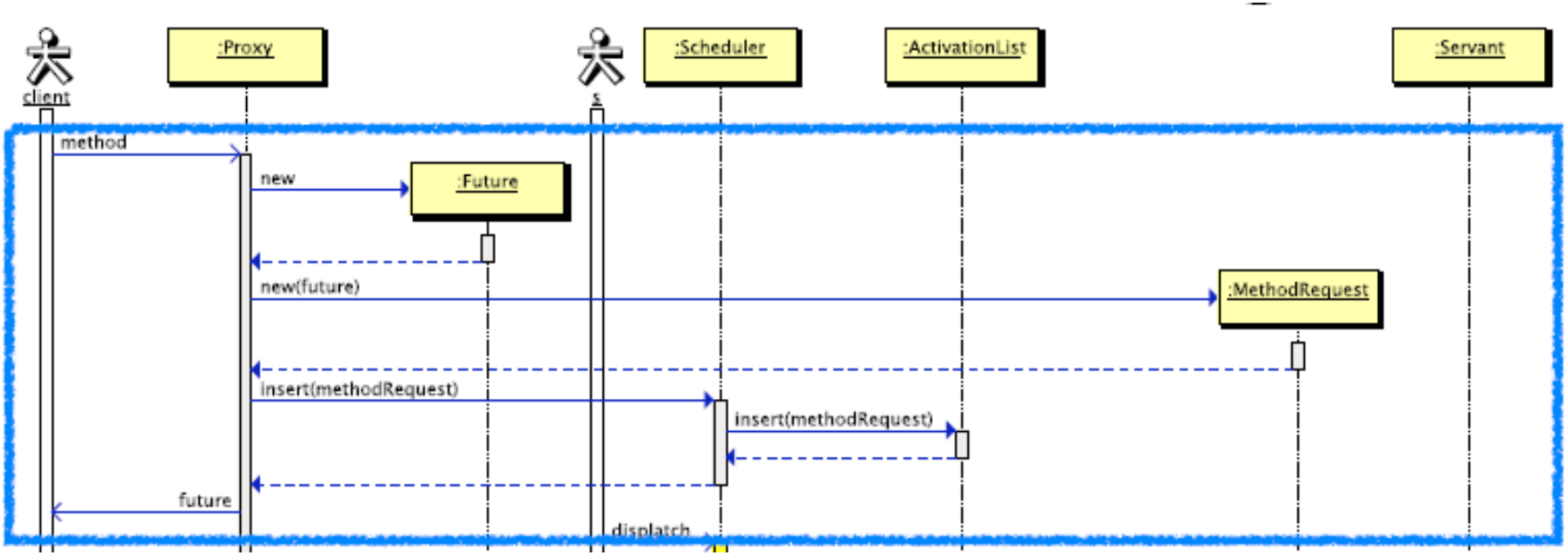


6. Konkurentnost

Projektovanje i implementacija aktivnih objekata

– **Projektni obrazac: *Active object* (*Actor Pattern*)**

Method scheduling



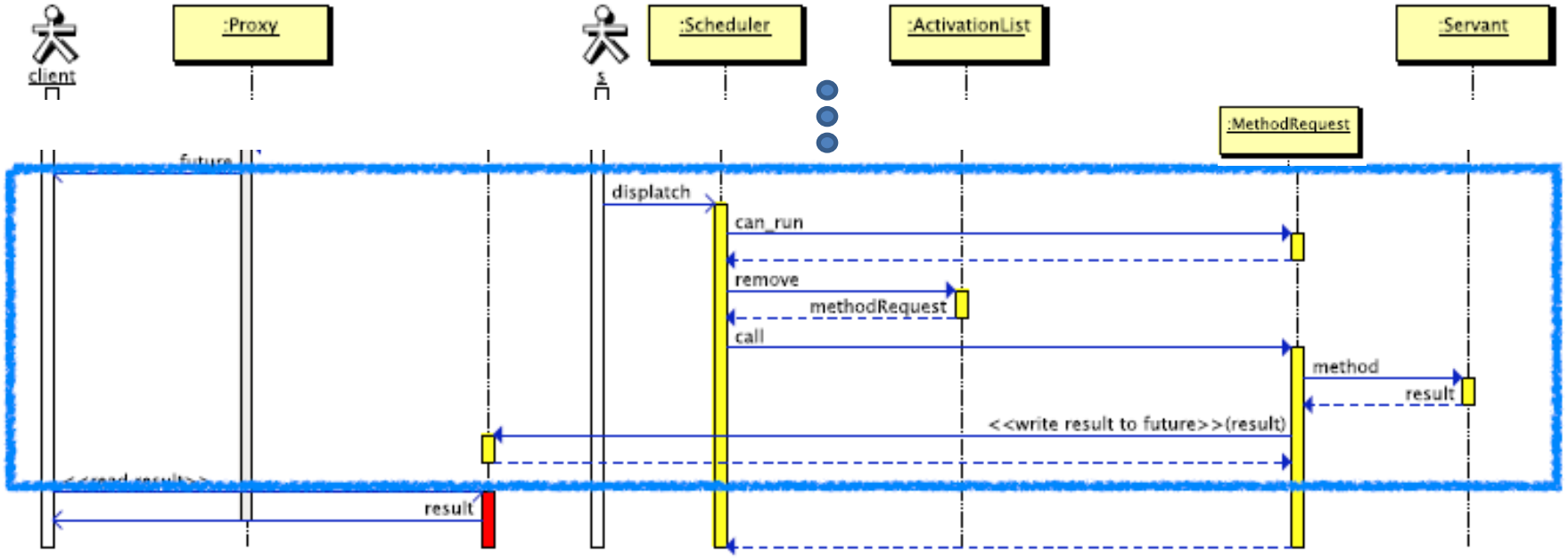
1. **Klijent poziva željenu metodu Proxy objekta**
2. **Proxy kreira Future objekat koji će vratiti rezultat klijentu (ako bude rezultata)**
3. **Proxy kreira MethodRequest objekat i šalje mu referencu na Future (kako bi se u Future kasnije mogao smjestiti rezultat)**
4. **Proxy šalje zahtjev Scheduler-u, koji smješta taj zahtjev u ActivationList (Queue)**
5. **Ako poziv metode treba da vrati rezultat, Future objekat se vraća klijentu**

6. Konkurentnost

Projektovanje i implementacija aktivnih objekata

– **Projektni obrazac: *Active object* (*Actor Pattern*)**

Method dispatch

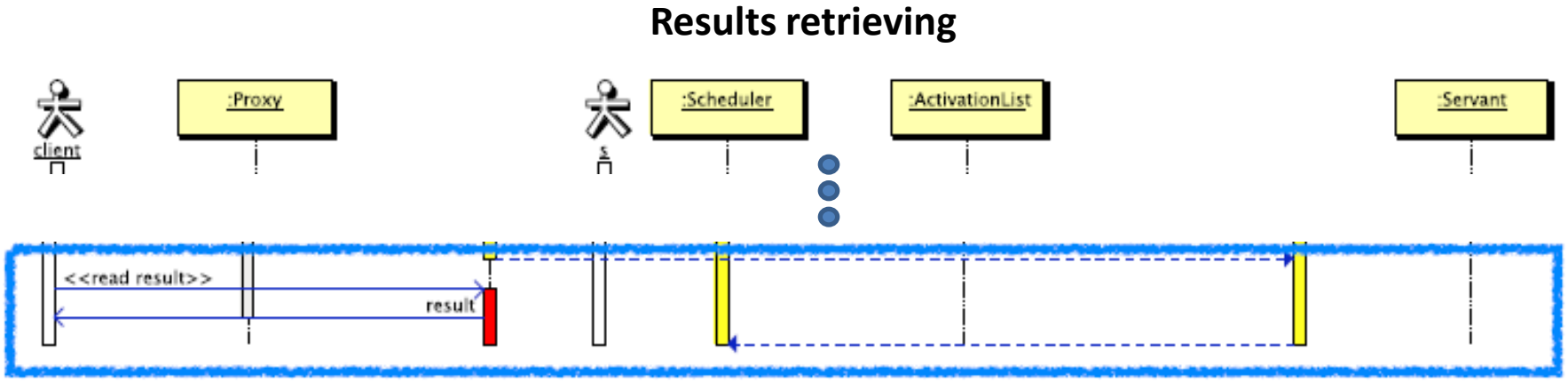


1. **Scheduler provjerava da li se može izvršiti zahtjev**
2. **Ako zahtjev može da se izvrši, zahtjev se uzima iz reda (brisanje iz reda) i pokreće izvršavanje**
3. **Zahtjev poziva korespondentnu metodu servanta**
4. **Ako servant vrati rezultat, MethodRequest upisuje rezultat u korespondentni Future objekat**

6. Konkurentnost

Projektovanje i implementacija aktivnih objekata

– **Projektni obrazac: *Active object* (*Actor Pattern*)**



1. **Nakon što je rezultat upisan u Future, klijent može da ga pročita**

Čitanje rezultata je asinhrono u odnosu na poziv metode

6. Konkurentnost

Implementacija konkurentnosti – definicija aktivnih objekata

– **Dvije vrste aktivnih objekata: Runnable i Callable**

Runnable – komanda

- **ne može da vrati rezultat**

```
public class Task implements Runnable
{
   // deklaracija atributa

   // definicija operacija

 @Override
   public void run()
   {
      try
      {
         // aplikativna logika
      }
      catch (InterruptedException e)
      {
         e.printStackTrace();
      }
   }
}
```

Callable

- **može da vrati rezultat**

```
import java.util.concurrent.Callable;
public class Task implements Callable<T>
{
   private T atr;

   public Task(T arg)
      { atr = arg; }

   @Override
   public T call() throws Exception
   {
      // izracunaj rezultat
      return rezultat;
   }
}
```

6. Konkurentnost

Primjer implementacije Producer-Consumer

```
// producer
class P implements Runnable
{
  private List<Integer> red;

  public P(List<Integer> r)
  {
    this.red = r;
  }

  @Override
  public void run()
  {
    int brojac = 0;
    while (true)
    {
       try { puni(brojac++); }
       catch (InterruptedException ex) {
          ex.printStackTrace();
       }
    }
  }
```

```
  private void puni(int i) throws
                InterruptedException
  {
    synchronized (red) // zakljucavanje reda
    {
      while (red.size() == 5)
      {
        System.out.println("Red je pun. " +
         Thread.currentThread().getName() +
         " ceka, size: " + red.size());
        red.wait(); // ceka dok je red pun
          // otkljucava red i "spava" dok ne
          // dobije notifikaciju da je
          // oslobodjen prostor
      }

      Thread.sleep(1000); // ceka 1s
      red.add(i);
      System.out.println("Upisano: " + i);
      red.notifyAll(); // budi sve tredove
              // koji su cekali
    }
  }
}
```

6. Konkurentnost

Primjer implementacije Producer-Consumer

```
// consumer
class C implements Runnable
{
  private List<Integer> red;

  public C(List<Integer> r)
  {
    this.red = r;
  }

  @Override
  public void run()
  {
    while (true)
    {
      try { prazni(); }
      catch (InterruptedException ex)
      {
        ex.printStackTrace();
      }
    }
  }
```

```
  private void prazni() throws
                InterruptedException
  {
    synchronized (red)  // zakljucava red
    {
      while (red.isEmpty())
      {
        System.out.println("Red je prazan. "
        + Thread.currentThread().getName()
        + " ceka, size: " + red.size());
        red.wait(); // ceka dok je prazan
          // otkljucava red i "spava" dok ne
          // dobije notifikaciju da je
          // nesto upisano u bafer
      }
      Thread.sleep(1000);
      int i = (Integer) red.remove(0);
      System.out.println("Procitano: " + i);
      red.notifyAll(); // budi sve tredove
              // koji su cekali
    }
  }
}
```

6. Konkurentnost

Primjer implementacije Producer-Consumer

```
// Demonstracija producer-consumer
public class PCdemo
{
   public static void main(String[] args)
   {

      List<Integer> red = new ArrayList<Integer>();

      System.out.println(“Start.“);

      Thread tP = new Thread(new P(red), "P");
      Thread tC = new Thread(new C(red), “C");
      tP.start();
      tC.start();
   }
}
```

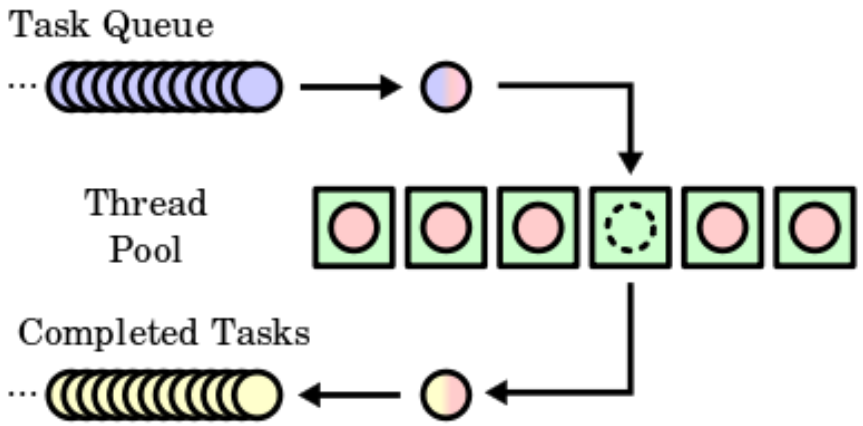
```
Start.
Upisano: 0
Procitano: 0
Red je prazan. C ceka, size: 0
Upisano: 1
Upisano: 2
Procitano: 1
Procitano: 2
Red je prazan. C ceka, size: 0
Upisano: 3
Upisano: 4
Procitano: 3
Upisano: 5
Procitano: 4
Upisano: 6
Procitano: 5
Procitano: 6
Red je prazan. C ceka, size: 0
Upisano: 7
Procitano: 7
Red je prazan. C ceka, size: 0

...
```

6. Konkurentnost

Implementacija konkurentnosti – upravljanje nitima u Javi

- **Executor framework**
  - Svako kreiranje nove niti je skupa operacija i (može da) degradira performanse
  - ***Executor framework*** **koristi** ***thread pool*** **za upravljanje konkurentnim taskovima**



Thread pool je kolekcija niti koju executor koristi za izvršavanje konkurentnih taskova

Thread pool može imati fiksnu ili promjenljivu veličinu

Ako ima više taskova nego niti, taskovi se smještaju u *Task Queue* (FIFO bafer) i čekaju na izvršavanje

Kad neka nit završi izvršavanje taska, iz bafera može da uzme i izvrši sljedeći task

Kad se izvrše svi taskovi, niti ostaju aktivne i čekaju sljedeće taskove

6. Konkurentnost

Implementacija konkurentnosti – upravljanje nitima u Javi

- **`ThreadPoolExecutor`** klasa (implementira `Executor` i `ExecutorService` interfejse)
- **`ThreadPoolExecutor` razdvaja kreiranje i izvršavanje konkurentnog objekta**
- **U korisničkoj aplikaciji treba:**
  - instancirati *thread pool*,
  - kreirati konkurentne objekte (tipa `Runnable` ili `Callable`)
  - poslati kreirane objekte u *thread pool*
  - na kraju zatvoriti *thread pool*
- **`ThreadPoolExecutor` vodi računa o performansama i opterećenju niti** (nije preporučljivo razvijati sopstveni sistem za upravljanje nitima i konkurentnim taskovima)

```
import
 java.util.concurrent.Executors;
import
 java.util.concurrent.ThreadPoolExecutor;

public class ThreadPoolExample
{
  public static void main(String[] args)
  {
     ThreadPoolExecutor executor =
        (ThreadPoolExecutor)
        Executors.newFixedThreadPool(2);

     Task t = new Task();
     executor.execute(t);

     executor.shutdown();
  }
}
```

6. Konkurentnost

Implementacija konkurentnosti – upravljanje nitima u Javi

- **Postoji pet vrsta `ThreadPoolExecutor` objekata:**
  - **Fixed thread pool executor**
    - fiksan broj niti – prekobrojni taskovi čekaju u redu

```
executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10);
```

  - **Cached thread pool executor**
    - broj niti nije fiksan – kreiraju se dodatne niti po potrebi (nije preporučljivo za vremenski zahtjevne taskove niti za mnogo taskova – može da naruši performanse)

```
executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();
```

  - **Scheduled thread pool executor**
    - omogućava odgođeno ili periodično izvršavanje

```
executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(10);
```

  - **Single thread pool executor**
    - samo jedna nit – kad imamo samo jedan konkurentni task

```
executor = (ThreadPoolExecutor) Executors.newSingleThreadPool();
```

  - **Work stealing thread pool executor**
    - Kreira dovoljno niti da obezbijedi projektovani nivo paralelizma (multiprocessing)

```
executor = (ThreadPoolExecutor) Executors.newWorkStealingThreadPool(4);
```

6. Konkurentnost

Implementacija konkurentnosti – definicija aktivnih objekata

Definicija aktivnog objekta koji se izvršava asinhrono i vraća rezultat

```
import java.util.concurrent.Callable;
public class AktObj implements Callable<T>
{
   private T atr;
   public AktObj(T arg)
      { atr = arg; }
   @Override
   public T call() throws Exception
   {
      // izracunaj rezultat
      return rezultat;
   }
}
```

Klasa `AktObj` implementira interfejs `Callable`.

`Callable` reprezentuje **asinhroni proces** koji vraća rezultat.

Klasa `AktObj` definiše metodu `call()` kojom implementira aplikativnu logiku.

6. Konkurentnost

Implementacija konkurentnosti

Poziv asinhronog procesa (Future)

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadPoolExecutor;

public class AktObjTest
{
  public static void main(String[] args)
    throws ExecutionException,
           InterruptedException
  {
    // ...
    ThreadPoolExecutor executor =
        (ThreadPoolExecutor)
        Executors.newFixedThreadPool(2);
    AktObj aobj = new AktObj(arg);
    Future<T> f = executor.submit(aobj);
    // ...
    T rez = f.get();
  }
}
```

Klasa `AktObjTest` ilustruje korištenje `AktObj` objekata.

`executor` upravlja životnim ciklusom asinhronih procesa.

Pomoću metode `submit()` executor prima i inicira izvršavanje `Callable` objekta.

`Future` (implementira `Runnable`) reprezentuje rezultat izvršavanja asinhronog procesa.

Objekat tipa `Future` dobija se odmah, a vrijednost rezultata biće naknadno pribavljena.

6. Konkurentnost

Primjer implementacije – ThreadPool & Future

```
public class Fact implements Callable<int>
{
  private int n;
  public Fact(int n) { this.n = n; }
  @Override
  public int call() throws Exception
  {
    int f = 1;
    if (n>1) for (int i=2; i<=n; f*=i++);
    System.out.println(n + "! -> " + f);
    return f;
  }
}
```

```
public class CallableDemo
{
  public static void main(String[] args)
  {
    ThreadPoolExecutor executor =
        (ThreadPoolExecutor)
        Executors.newFixedThreadPool(2);
    List<Future<int>> fl=new ArrayList<>();
    Random random = new Random();
    for (int i=0; i<4; i++) {
      int n=random.nextInt(10);
      Fact f = new Fact(n);
      Future<int> r=executor.submit(f);
      fl.add(r);
    }
    for(Future<int> f:fl) {
      try {
        System.out.println("Future: " +
        f.get()+" Task done:" + f.isDone());
      }
      catch (...) { e.printStackTrace(); }
    }
    executor.shutdown();
  }
}
```

```
4! -> 24
6! -> 720
Future: 720 Task done: is true
Future: 24 Task done: is true
2! -> 2
6! -> 720
Future: 720 Task done: is true
Future: 2 Task done: is true
```