KONKURENTNO PROGRAMIRANJE U JAVI I EIFFELU

SAŽETAK

U radu se prikazuju neke "stare" tehnike konkurentnog programiranja u Javi verzije 1 do 4, te neke novije tehnike, koje je omogućila verzija 5, a u verzijama 6 i 7 su još poboljšane. Nažalost, konkurentno programiranje teže je od uobičajenog sekvencijalnog programiranja, ali sa razvojem višejezgrenih procesora konkurentno programiranje postaje nužnost u svakodnevnom radu.

Postoje pokušaji (vrijeme će pokazati da li su uspješni) da se konkurentno programiranje učini jednostavnijim. Jedan drugačiji pristup u odnosu na onaj u Javi (a slično Javi ima i C#, pa i C++), koji bi trebao olakšati konkurentno programiranje, je Simple Concurrent Object Oriented Programming (SCOOP), model koji je realiziran u OOPL jeziku Eiffel, ali postoji šansa da se raširi i na druge OOPL jezike (npr. postoje i testne realizacije u Javi).

ABSTRACT

The paper presents some "old" concurrent programming techniques in Java, versions 1 to 4, and some newer techniques made possible by version 5, and further improved in versions 6 and 7. Unfortunately, concurrent programming is more difficult than usual sequential programming, but with the development of multi-core processors concurrent programming is becoming a necessity in daily work.

There are some attempts (time will show if they are successful) to make concurrent programming easier. A different approach in comparison to that in Java (and similar to Java are C# and even C++), which should facilitate concurrent programming, is the Simple Concurrent Object Oriented Programming (SCOOP), a model that has been implemented in OOPL Eiffel language, but there is a chance it could spread to other OOPL languages (e.g. there are some test implementations in Java.)

1. UVOD

Gotovo sva današnja računala imaju više CPU-a, manja računala u obliku višejezgrenih (engl. multi-core) procesora, a veća i snažnija računala obično sadrže više procesora (isto višejezgrenih). Prednost je takvih računala da mogu paralelno izvršavati dva ili više (u ovisnosti od broja CPU-a) nezavisnih programa, ali mogu paralelno izvršavati i dijelove istog programa. U potonjem slučaju, ako su dijelovi programa nezavisni jedan od drugoga, tada programeri i dalje mogu pisati kod kao da se on odvija u jednom dijelu.

No, najčešće su dijelovi programa međusobno zavisni, jer čitaju / pišu u isto memorijsko područje ili koriste neki drugi dijeljeni resurs, pa je moguće da dođe do tzv. *race condition,* gdje rezultat izračuna ovisi o redoslijedu izvršenja programskih instrukcija iz različitih dijelova programa. Zbog toga je potrebna sinkronizacija dijelova programa. Sinkronizacija najčešće traži posebne programske tehnike, koje svoje porijeklo imaju još u 60-tim godinama prošlog stoljeća. Te se programske tehnike obično zovu imenom *konkurentno programiranje*.

Postoje različite varijante konkurentnog programiranja, a najčešće se razlikuje deklarativno i imperativno konkurentno programiranje. Deklarativno konkurentno programiranje je jednostavnije. No, budući da najčešće radimo sa imperativnim programskim jezicima (to su npr. Java, C++, C#, ali možemo koristiti i ne-objektni imperativni jezik, npr. C), onda najčešće moramo koristiti imperativno konkurentno programiranje, u kojem postoje problemi zaključavanja, deadlocka (potpuni zastoj; u nastavku ćemo koristiti engleski termin) i drugi.

U radu se u 2. točki prikazuju neke "stare" tehnike konkurentnog programiranja u Javi verzije 1 do 4, a u 3. točki neke novije tehnike, koje je omogućila verzija 5, a u verzijama 6 i 7 su još poboljšane.

Postoje pokušaji da se i imperativno konkurentno programiranje učini jednostavnijim (tj. da se, na neki način, približi deklarativnom konkurentnom programiranju). Jedan drugačiji pristup imperativnom konkurentnom programiranju u odnosu na onaj u Javi (a slično Javi ima i C#, pa i C++), prikazan je u 5. točki – Simple Concurrent Object Oriented Programming (SCOOP), model koji je realiziran u OOPL jeziku Eiffel. Eiffel jezik ukratko je prikazan u prethodnoj (4.) točki.

2. KONKURENTNO PROGRAMIRANJE U JAVI VERZIJE 1 - 4

Kako navode autori u [4], iako mi sami možda nećemo kreirati dretve u našim Java programima, ipak ne možemo izbjeći višedretveni rad. Naime, svaka Java aplikacija koristi dretve. Kada se starta JVM, on kreira posebne dretve, npr. za GC (garbage collection), uz *main* dretvu. Kada koristimo Java AWT ili Swing framework, oni kreiraju posebnu dretvu za upravljanje korisničkim sučeljem. Kada koristimo servlete ili RMI, oni kreiraju pričuvu (pool) dretvi. Zato, kada koristimo te frameworke, moramo do neke mjere biti upoznati sa konkurentnošću u Javi. Naime, svaki takav framework uvodi u našu aplikaciju konkurentnost na implicitan način, te moramo znati napraviti da mješavina našeg koda i frameworkovog koda bude sigurna u višedretvenom radu.

Svaki Java program ima barem jednu dretvu, onu koja izvršava metodu *main()*. Kada želimo napraviti svoju dretvu, imamo u Javi dva načina; jedan način je da napravimo klasu koja nasljeđuje klasu *Thread* i da nadjačamo (override) metodu *run()*, kao što prikazuje sljedeći primjer [7], u kojem se kreiraju dvije klase, *Worker1* i *Worker2*:

```
class Worker1 extends Thread {
  public void run() {
    // implement doTask1() here
  }
}
class Worker2 extends Thread {
  public void run() {
    // implement doTask2() here
  }
}
```

Da bi se kreirale dretva, potrebno je kreirati objekt (instancu) klase (u ovom slučaju klase *Worker1* i/ili *Worker2*) i pozvati metodu *start()* nad tim objektom. Time će se automatski kreirati dretva i pozvati metoda *run()* u njoj. U nastavku se prikazuje implementacija metode *compute()* (iz neke treće klase čiji ostali detalji nisu prikazani), koja kreira dvije dretve, tako da dva posla (tasks) mogu biti izvedena paralelno (ako postoje dva slobodna procesora koja ih izvode):

```
void compute() {
  Worker1 worker1 = new Thread();
  Worker2 worker2 = new Thread();
  worker1.start();
  worker2.start();
}
```

Sada klase *Worker1* i *Worker2* proširimo sa sljedećim atributom i metodom:

```
private int result;
public void getResult() {
  return result;
}
```

Pretpostavimo da dretve snimaju rezultat izračuna u tu varijablu, a metodom *getResult()* ih možemo čitati. Sada želimo dobiti rezultat oba izračuna u metodi *compute()*:

```
return worker1.getResult() + worker2.getResult();
```

Očito, moramo čekati da obje dretve završe prije nego dobijemo taj rezultat. To se postiže naredbom *join()* koja, kad se poziva nad dretvom-izvršiteljem, uzrokuje da dretva-pozivatelj čeka dok dretva-izvršitelj ne završi. U ovom slučaju programski kod dretve-pozivatelja izgleda ovako:

```
int compute() {
  worker1.start();
  worker2.start();
  worker1.join();
  worker2.join();
  return worker1.getResult() + worker2.getResult();
}
```

Naravno, u ovom slučaju nije važno da li prvo pozivamo *join()* nad *worker1* ili *worker2*.

Kako je prije navedeno, postoji i drugi način za kreiranje dretvi u Javi. Prethodno prikazani način (kreiranje klase koja nasljeđuje klasu *Thread*), iako izgleda logičan, manje se koristi jer je manje fleksibilan. Problem je u tome što u Javi (za sada) postoji samo jednostruko nasljeđivanje ponašanja, tj. klasa (a ne samo sučelja). A, "Višestruko nasljeđivanje klasa je korisno, koliko god mi šutjeli o tome :)" (odnosno - koliko god nas pokušavali uvjeriti u suprotno). Uzgred, neki oblik višestrukog nasljeđivanja ponašanja uvodi se u Javi 8, na način da će sučelja imati implementaciju metoda. Bez višestrukog nasljeđivanja klasa, ako naša klasa "potroši" jednostruko nasljeđivanje za klasu *Thread*, ne može naslijediti od neke druge klase. Zbog toga se češće koristi drugi način kreiranja dretve. Prvo se napiše "pomoćna" klasa koja nasljeđuje sučelje (interface) *Runnable*, pri čemu mora implementirati metodu *run()*. Primjer [6]:

```
public class RunThread implements Runnable {
  String id;
  public RunThread(String id) {
    this.id = id;
  }
  public void run() {
    // do something when executed
    System.out.println("This is thread " + id);
  }
}
```

Onda se objekt te "pomoćne" klase šalje konstruktoru koji kreira objekt klase *Thread* (tj. dretvu) u kodu naše "glavne" klase:

```
Thread mt = new Thread(new RunThread("mt"));
mt.start(); ...
```

Do sada prikazani rad sa dretvama u Javi izgleda prilično jednostavno. No, problemi nastaju kada se dvije dretve (ili više njih) upliću jedna drugoj u posao, npr. tako da modificiraju isti objekt. To može stvoriti netočne rezultate i naziva se *race condition*, npr. u ovom slučaju [7]:

```
class Counter {
  private volatile int value = 0;
  public int getValue() {
    return value;
  }
  public void setValue(int someValue) {
    value = someValue;
  }
  public void increment() {
    value++; -- napomena: niti sama naredba value++ nije atomarna
  }
}
```

Pretpostavimo da neka metoda u nekoj drugoj klasi kreira objekt klase *Counter* i sadrži sljedeći kod:

```
x.setValue(0);
x.increment();
int i = x.getValue();
```

Pitanje je: koju vrijednost ima varijabla *i* na kraju ovih naredbi? Ako je riječ o jednodretvenom programu, onda je vrijednost 1. No u konkurentnom radu brojač može biti modificiran od drugih dretvi, tako da rezultat ovisi o ispreplitanju naredbi ove dretve sa naredbama neke druge dretve. Npr. ako druga dretva konkurentno izvodi naredbu

```
x.setValue(2);
```

varijabla *i* na kraju navedenih naredbi prve dretve može imati vrijednost 1, 2 ili 3, tj. rezultat ovisi o slučajnom redoslijedu izvođenja naredbi dvije dretve (zapravo, dodatni problem je i u tome što sama naredba *value*++ nije atomarna, već se u stvarnosti razlaže na tri interne naredbe: temp = value; temp = temp + 1; value = temp). Naravno, to nije ono što se želi. Taj se problem rješava pomoću sinkronizacije koja se zove *međusobno isključivanje* (*mutual exclusion*), za što Java ima jednostavno rješenje još od verzije Java 1. Svaki objekt u Javi ima lokot (lock; nasljeđuje se automatski od superklase *Object*), kojega istovremeno može držati (zaključati) samo jedna dretva.

Objekt koji će služiti kao lokot može se kreirati npr. ovako:

```
Object lock = new Object();
```

Dretva koja će tražiti lokot (tj. zaključati lokot) to radi pomoću naredbe *synchronized*, koja označava početak tzv. *synchronized bloka* (inače *synchronized* i *volatile* su jedine Java ključne riječi vezane za konkurentnost; ostalo su metode klase Object ili metode klasa specijaliziranih za konkurentnost):

```
synchronized(lock) {
  // critical section
}
```

Kada dretva dođe do početka tog bloka, pokuša zaključati lokot objekta koji je naveden kao argument naredbe *synchronized.* Ako je lokot zaključan od neke druge dretve, polazna dretva čeka dok on ne postane otključan. Nakon toga ga polazna dretva drži zaključanim sve do kraja tog bloka. Problem iz prethodnog primjera mogli bismo riješiti pomoću *synchronized* npr. ovako (u prvoj, odnosno drugoj dretvi; naravno, moramo biti sigurni da varijable *lock* u oba koda referenciraju isti objekt):

```
// prva dretva
synchronized(lock) {
  x.setValue(0);
  x.increment();
  int i = x.getValue();
}

// druga dretva
synchronized(lock) {
  x.setValue(2);
}
```

Osim bloka, i cijela metoda (funkcija / procedura) može imati *synchronized* na početku:

```
synchronized type method(args) {
  // body
}
```

što je, zapravo, isto kao i ovo:

```
type method(args) {
  synchronized(this) {
    // body
  }
}
```

Prethodno je slično konceptu *monitora*, ali dizajneri Jave su napravili određena odstupanja od tog koncepta. Svaki objekt u Javi ima unutarnji (intrinsic) lokot i unutarnju kondiciju (condition). Ako je metoda deklarirana pomoću ključne riječi *synchronized,* ona djeluje kao monitor. Kondicijskim varijablama se pristupa pomoću naredbi *wait / notifyAll / notify*, što će biti prikazano u nastavku. Međutim, Java objekt se razlikuje od monitora [3] u tri važne stvari, kompromitirajući time sigurnost dretve (thread safety) :

- atributi ne moraju biti privatni;
- metode ne moraju biti sinkronizirane;
- unutarnji lokot je pristupačan klijentima.

Zbog toga je jedan od inventora monitora, Per Brinch Hansen, 1999. godine izjavio: "Zaprepašćuje me da je ovaj nesigurni Java paralelizam shvaćen tako ozbiljno od programske zajednice, četvrt stoljeća nakon invencije monitora i programskog jezika Concurrent Pascal." [Java's Insecure Parallelism, ACM SIGPLAN Notices 34:38–45, April 1999.]

Kao što vrijedi za monitor, tako vrijedi i za Java klase - zaštita pristupa djeljivim varijablama nije jedini razlog zašto dretve moraju biti međusobno sinkronizirane. Često puta treba odgoditi izvođenje metode (ili dijela metode) u nekoj dretvi, dok se ne zadovolji određeni uvjet (a taj uvjet nije samo otključavanje određenog lokota). To se zove *sinkronizacija na temelju uvjeta* (*condition synchronization*), koja se u Javi implementira pomoću naredbi *wait / notifyAll / notify,* koje se pozivaju nad sinkroniziranim objektima.

Jedan primjer problema koji traži sinkronizaciju na temelju uvjeta je tzv. *problem proizvođač-potrošač* (*producer-consumer problem*), koji je,u različitim varijantama, čest u praksi. Može se apstraktno opisati na ovaj način [7]:

- *Proizvođač*: u svakoj iteraciji (beskonačne) petlje, proizvodi podatke koje potrošač konzumira;
- *Potrošač*: u svakoj iteraciji (beskonačne) petlje, troši podatke koje je proizveo proizvođač.

Proizvođači i potrošači komuniciraju preko djeljivog međuspremnika (buffer) koji implementira red (queue), za koji smatramo (u ovom primjeru) da je neograničen, pa će nam biti važno samo da li je prazan. U primjeru [7] se prikazuje samo dio klase, a ne kompletan programski kod. Prvo pretpostavimo da imamo klasu *Buffer* (koja implementira neograničeni red) i da imamo jedan objekt te klase:

```
Buffer buffer = new Buffer();
```

Proizvođači dodaju podatke na kraj reda, koristeći metodu (reda) *void put(int item)*, a potrošači skidaju podatak sa reda koristeći metodu *int get()*. Broj podataka koji se nalaze u redu može se dobiti pomoću metode *int size()*. Pretpostavimo sada da u klasi *Consumer* imamo sljedeću metodu:

```
public void consume() {
  int value;
  synchronized(buffer) {
    value = buffer.get(); // incorrect: buffer could be empty
  }
}
```

Metoda nije dobra, jer ne provjerava da li je red (tj. međuspremnik) prazan, što može prouzročiti grešku kod izvođenja (runtime error). Dretva treba čekati dok red bude neprazan i tek tada pročitati podatak iz njega. Čekanje se u Javi može napraviti pomoću metode *wait()*, koja se može pozvati samo nad objektom koji je prethodno zaključan, tj. samo unutar *synchronized* bloka. *Wait()* tada blokira tekuću dretvu i otpušta lokot koji je dretva držala. Slijedi primjer [7]:

```
public void consume() throws InterruptedException {
  int value;
  synchronized(buffer) {
    while (buffer.size() == 0) {
      buffer.wait();
    }
    value = buffer.get();
  }
}
```

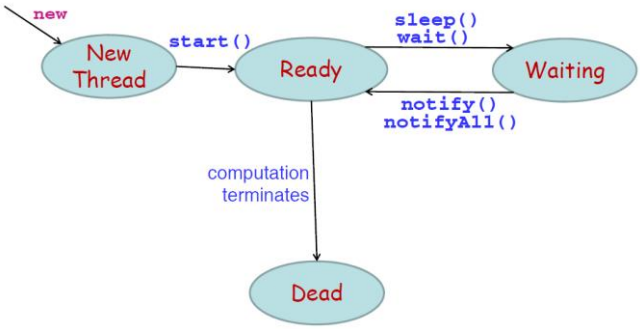
Sada lokot može preuzeti (zaključati) neka druga dretva, koja može mijenjati stanje navedene kondicije. Da obavijesti prvu dretvu o promjeni kondicije, druga dretva poziva *notify(),* što odblokira prvu dretvu, koja čeka na taj lokot, ali druga dretva ne otključa lokot odmah, već tek na kraju svog *synchronized bloka* (unutar kojega je i pozvala *notify()*). Kod monitora dva osnovna tzv. *pravila signalizacije* (*signaling disciplines*) mogu biti *Signal and Wait* ili *Signal and Continue* (Java koristi samo ovo drugo). Primjer u kojem proizvođač obavještava potrošača o promjeni kondicije;

```
public void produce() {
  int value = random.produceValue();
  synchronized(buffer) {
    buffer.put(value);
    buffer.notify();
  }
}
```

Proizvođač je proizveo neki slučajni broj, zaključao red, spremio podatak u njega i pozvao *notify()*, čime je odblokirao jednu (bilo koju!) dretvu koja je čekala na taj lokot. Na kraju *synchronized bloka* (što je u ovom slučaju bilo odmah iza) je i otključao taj lokot. No, odblokirana dretva ne može biti sigurna da je taj uvjet valjan i kada ona dođe na red, jer je u međuvremenu neka treća dretva mogla potrošiti podatak i red je možda opet prazan. Stoga je dretva potrošač morala ispitivati uvjet u *while* petlji. Važno je i to da *notify()* uvijek odblokira samo jednu (bilo koju!) dretvu koja čeka na određeni lokot. Stoga je u praksi puno sigurnije pozivati *notifyAll(),* koja odblokira sve dretve koje čekaju na određeni lokot.

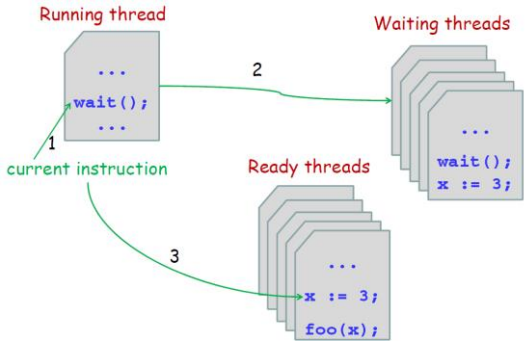
Metode *wait()*, *notify()* / *notifyAll()* rade interno sa tzv. *unutarnjim redovima kondicija* (*intrinsic condition queues*). Vidjet ćemo da u Javi 5 postoji njihova generalizacija, koja omogućava da programeri eksplicitno rade sa kondicijama.

Na slici 2.1 prikazan je dijagram promjene stanja Java dretvi. Vidi se (i) da dretva prelazi iz stanja *Ready* (češće se to stanje naziva *Runnable*) u stanje *Waiting* nakon što pozove *wait()*, a obrnuti prijelaz se zbiva kada **druga dretva** (što se iz slike ne vidi) pozove *notify()* / *notifyAll()*:



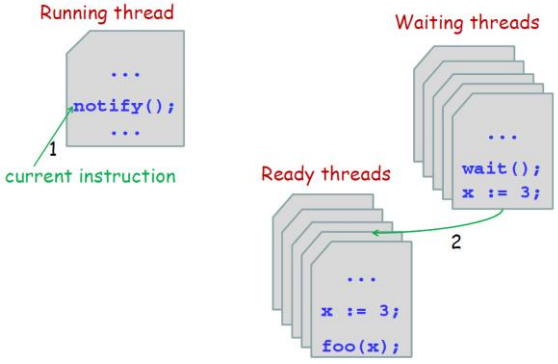
Slika 2.1: Dijagram promjene stanja Java dretvi; Izvor: [6]

Preciznije se zbivanje može pratiti kroz sljedeće dvije slike. Na slici 2.2 prikazana su tri koraka kod prijelaza dretve iz stanja *Running* (u kojem se dretva izvršava u procesoru) u stanje *Waiting*. Prvo dretva poziva metodu *wait()* (1), naravno, uvijek unutar *synchronized bloka*, čime otključa lokot i prelazi u stanje *Waiting* (2). Sada neka dretva koja je *Ready* može preći u stanje *Running*, jer se procesor oslobodio.



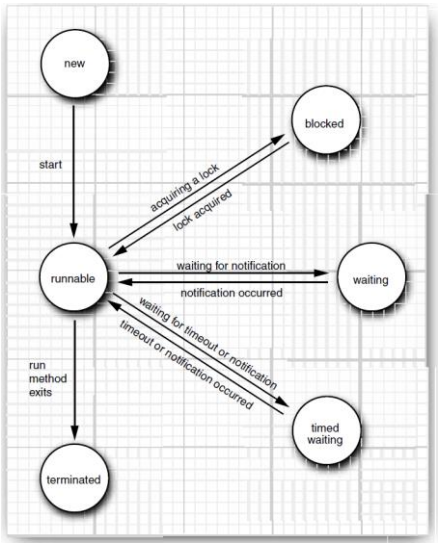
Slika 2.2: Promjena stanja nakon wait() operacije; Izvor: [6]

Slika 2.3 prikazuje nastavak zbivanja - poziva se *notify()* ili *notifyAll()*. Nakon što neka druga dretva pozove *notify()* ili *notifyAll()* (1) (naravno, uvijek unutar *synchronized bloka*), jedna dretva (ako je *notify()*) ili sve dretve (ako je *notifyAll()*) koje čekaju na lokot koji ima ta druga dretva, prelazi (2) iz stanja *Waiting* u Ready. Naravno, rekli smo da dretva koja poziva *notify()* / *notifyAll()* ne otključava lokot odmah, već tek kod izlaska iz *synchronized bloka* (to je signalizacijsko pravilo *Signal and Continue*). Mora se primijetiti da ova slika ne prikazuje metodu *notify()* u svim njenim detaljima.



Slika 2.3: Promjena stanja nakon notify() operacije; Izvor: [6]

Slika 2.4 prikazuje stanja Java dretvi na malo drugačiji način. Ovdje je stanje *Waiting* detaljnije razloženo u tri posebna stanja: *Blocked, Waiting* i *Timed waiting* (a *Ready* se naziva *Runnable*). No, niti ovdje zbivanja koja implicitno imaju naredbe *notify()* / *notifyAll()* nisu prikazana potpuno detaljno. Napomenimo da dretva prelazi u stanje *Waiting ne* samo nakon naredbe *wait()* (i *join()*), već i kod čekanja na objekte klase *Lock* ili *Condition* iz *java.util.concurrent* librarya, uvedenog u Java 5 verziji.



Slika 2.4: Dijagram promjene stanja Java dretvi; Izvor: [3]

Spomenimo da uz osnovnu varijantu metode *wait()* postoje i varijante *wait(long millis)* i *wait(long millis, int nanos)* koje uzrokuju da dretva čeka na obavijest od druge dretve ili na istek definiranog vremena. Java metode *wait()* / *notify* / *notifyAll* pripadaju klasi Object, od koje ih nasljeđuju sve druge klase. Inače, Java metode *notify* / *notifyAll* se u izvornoj terminologiji monitora zovu *signal* / *signal_all.*

Na kraju, prikažimo kako može nastati deadlock, kada se dvije dretve (ili grupa dretvi) blokiraju zauvijek, jer jedna dretva čeka lokot koji ima druga, i obrnuto. Primjer iz [7]:

```
public class C extends Thread {
  private Object a;
  private Object b;
  public C(Object x, Object y) {
    a = x;
    b = y;
  }
  public void run() {
    synchronized(a) {
      synchronized(b) {
        ...
      }
    }
  }
}
```

Pretpostavimo sada da se izvodi sljedeći kod, gdje su *a1* i *b1* objekti tipa *Object*:

```
C t1 = new C(a1, b1);
C t2 = new C(b1, a1);
t1.start();
t2.start();
```

Budući da su argumenti *a1* i *b1* međusobno permutirani kod kreiranja dretvi *t1* i *t2*, može doći do takve sekvence poziva u kojem dretva *t1* zaključa objekt *a1*, dretva *t2* zaključa objekt *b1*, i onda su obje dretve blokirane. Za razliku od sustava za upravljanje bazom podataka, Java sustav ovdje neće detektirati (i onda riješiti) deadlock. Zato programer mora paziti da do deadlocka ne dođe. U Javi 5 postoji mogućnost da se program lakše napiše na način da ne dođe do deadlocka.

3. KONKURENTNO PROGRAMIRANJE U JAVI VERZIJE 5, 6, 7

Konkurentnost u Javi od verzije 1 do verzije 4 praktički se nije mijenjala, osim što su se rješavali bugovi i sl. Suštinu "alata" za konkurentno programiranje činili su: klasa *Object*, tj. njen unutarnji (intrinsic) lokot (atribut te klase) i njene metode *wait(...)* / *notify* / *notifyAll*, Java ključne riječi *synchronized* i *volatile*, klasa *Thread* i sučelje *Runnable*.

U Javi verzije 5, koja se pojavila 2004. godine, uvedeno je dosta novina na drugim područjima (npr. generičke klase, bolje kolekcije i dr.), ali i na području konkurentnog programiranja. JSR 166, koji se odnosi na konkurentnost, temeljen je uglavnom na paketu *edu.oswego.cs.dl.util.concurrent*, kojega je napravio Doug Lea. Kroz novi paket *java.util.concurrent* uvedene su sljedeće nove mogućnosti:

- Executors (thread pools, scheduling);
- Futures;
- Concurrent Collections;
- Locks (*ReentrantLock*, *ReadWriteLock...*);
- Conditions;
- Synchronizers (Semaphores, Barriers...);
- Atomic variables;
- System enhancements.

U Java verziji 6, koja se pojavila godinu i pol nakon verzije 5, nije se pojavilo ništa revolucionarno, ali su se na području konkurentnog programiranja (kao i na drugim područjima) "iznutra" poboljšale biblioteke, bilo da su se riješili bugovi ili poboljšale performanse. U Java verziji 7, koja je izašla u ljeto prošle godine (2011.) u području konkurentnog programiranja novost je Fork/Join Framework.

U ovoj točki prikazat će se (ukratko) samo neke mogućnosti uvedene u Javi 5 (sa poboljšanjima u Javi 6), i to *ReentrantLock*, *ReadWriteLock*, *Conditions* i *Atomic variables*.

Do Java 5 verzije, jedini mehanizmi za koordinaciju pristupa djeljivim podacima bili su *synchronized* (koji koristi unutarnji lokot) i *volatile.* Java 5 donijela je i *ReentrantLock*, što je (klasa za) eksplicitan lokot. Kako navode autori u [4], on nije zamjena za implicitan, unutarnji lokot, već alternativa koju je bolje koristiti u nekim (ali ne svim) slučajevima. Klasa *ReentrantLock* implementira sučelje *Lock,* koje ima ove metode:

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)throws InterruptedException;
  void unlock();
  Condition newCondition();
}
```

Zaključavanja pomoću objekata klase *ReentrantLock* ima istu semantiku kao i *synchronized*, ali ima i dodatne mogućnosti. Najčešći oblik korištenja je sljedeći [4]:

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
  // update object state
  // catch exceptions and restore invariants if necessary
} finally {
  lock.unlock();
}
```

Za razliku od implicitnog zaključavanja i otključavanja kod *synchronized*, ovdje se zaključavanje i otključavanje mora raditi eksplicitno, a vrlo je važno da se otključavanje stavi u *finally* blok, inače lokot može ostati stalno zaključan. Moglo bi se reći da je to korak nazad u odnosu na *synchronized*, jer je sad moguće (dodatno) pogriješiti. No, mogućnost da se zaključavanje i otključavanje lokota napravi u različitim dijelovima koda ponekad je nužna (iako je takav kod manje čitljiv), a sa *synchronized* se to nije moglo izvesti. Preporuča se korištenje dosadašnje *synchronized* varijante ako nam ne treba ova mogućnost ili dodatne mogućnosti klase *ReentrantLock, koje su* navedene u nastavku.

ReentrantLock ima ove dodatne mogućnosti:

- pomoću metode *tryLock()* moguće je vidjeti da li je lokot slobodan; ako jeste, zaključa se, a ako nije, metoda odmah vraća exception; ovo je slično kao SQL naredba SELECT ... FOR UPDATE NOWAIT;

- pomoću metode *tryLock(long timeout,TimeUnit unit)* moguće je vidjeti da li je lokot slobodan; ako jeste, zaključa se, a ako nije, metoda vraća exception nakon isteka zadanog vremena; ovo je slično kao SQL naredba SELECT ... FOR UPDATE WAIT timeout;

- pomoću metode *lockInterruptibly()* dretva može (pokušati) zaključati lokot na taj način da aktivnost koja se može prekinuti (cancellable activities) može prekinuti dretvu dok čeka na lokot;

- *metoda newCondition()* omogućava definiranje *kondicija* (*condition*) uz lokot.

Sljedeći primjer [4] prikazuje korištenje metode *tryLock(long timeout,TimeUnit unit)*:

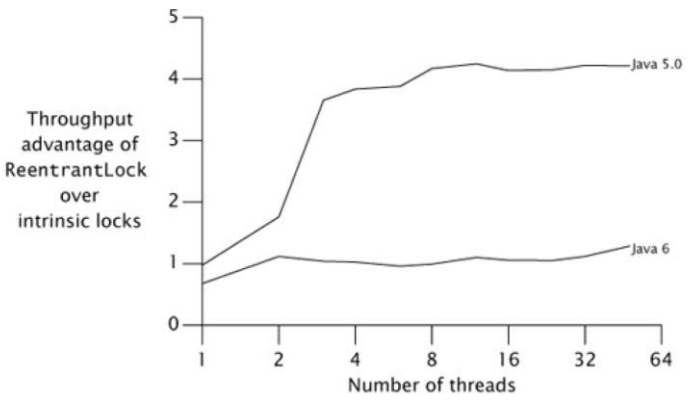
```
public boolean trySendOnSharedLine
  (String message, long timeout, TimeUnit unit)throws InterruptedException
{
  long nanosToLock = unit.toNanos(timeout)
  if (!lock.tryLock(nanosToLock, NANOSECONDS)) return false;
  try {
    return sendOnSharedLine(message);
  } finally {
    lock.unlock();
  }
}
```

Sljedeći primjer [4] prikazuje korištenje metode *lockInterruptibly()*:

```
public boolean sendOnSharedLine(String message)
  throws InterruptedException
{
  lock.lockInterruptibly();
  try {
    return cancellableSendOnSharedLine(message);
  } finally {
    lock.unlock();
  }
}
```

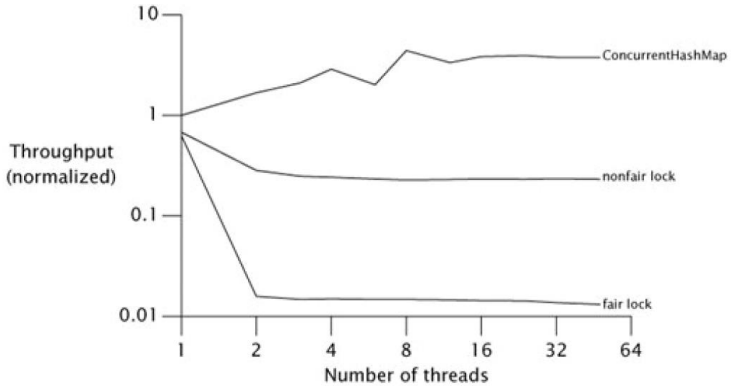
Treba napomenuti da zapravo i metoda *tryLock(long timeout,TimeUnit unit)* ima mogućnost prekidanja, a ne samo mogućnost čekanja (određeno vrijeme) da se lokot otključa. Prikazane metode omogućavaju programiranje na način da se izbjegne deadlock.

Slika 3.1 prikazuje kako je u Java 5 verziji *ReentrantLock* bio značajno bolje propusnosti (kod većeg broja dretvi) od *synchronized* varijante, ali je u verziji Java 6 to izjednačeno:



Slika 3.1: Propusnosti ReentrantLock-a u odnosu na propusnost unutarnjeg lokota; Izvor: [4]

Kod kreiranja *ReentrantLock* lokota mogu se definirati dvije varijante: fer i ne-fer lokot. Fer lokoti osiguravaju da dretve dobivaju lokote po redoslijedu prispijeća zahtjeva. Default su ne-fer lokoti, isto kao kod *synchronized* varijante. Ako nam nije nužno potreban fer lokot, bolje je koristiti ne-fer lokot, zbog puno veće propusnosti, što prikazuje slika 3.2:



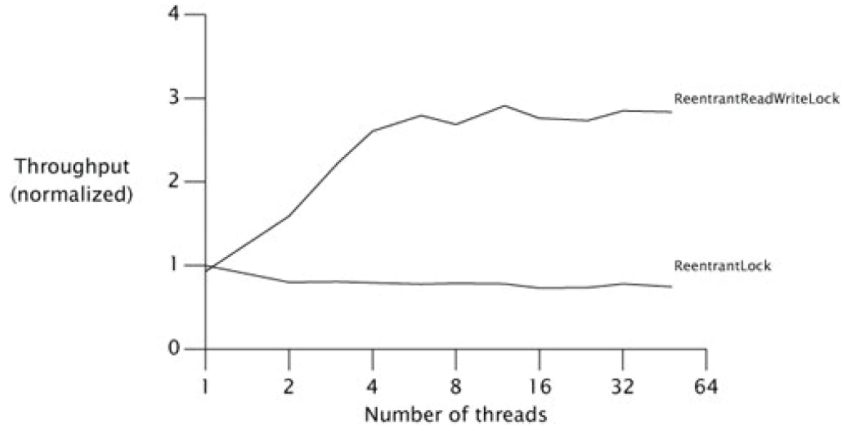
Slika 3.2: Propusnost fer i ne-fer ReentrantLock lokota; Izvor: [4]

Osim dvije krivulje za fer i ne-fer lokote, vidi se i krivulja koja prikazuje korištenje *ConcurrentHashMap* kolekcije. Uglavnom je bolje koristiti konkurentne kolekcije, koje često koriste *neblokirajuće algoritme* (tj. sinkronizaciju bez lokota), nego vlastita rješenja.

Osim *ReentrantLock* lokota, postoje i lokoti klase *ReentrantReadWriteLock*, koja implementira sučelje *ReadWriteLock*:

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

Strategija zaključavanja koju implementiraju ovakvi lokoti je: istovremeno može raditi više čitatelja koji blokiraju pisce, ali može raditi samo jedan pisac, koji blokira čitatelje i (druge) pisce. Slika 3.3 prikazuje propusnost koju ima *ReentrantReadWriteLock* u odnosu na *ReentrantLock*:



Slika 3.3: Propusnost "običnih" i read-write reentrant lokota; Izvor: [4]

U Java 5 verziji pojavili su se i *objekti-kondicije* (*condition objects*). Kao što su eksplicitni lokoti generalizacija unutarnjih lokota, tako su i objekti-kondicije generalizacija *unutarnjih redova kondicija* (*intrinsic condition queues*). *Kondicija* (*condition*) se povezuje sa *Lock* objektom na taj način da se pozove *Lock.newCondition* na već kreiranom lokotu (*Lock* objektu). Za razliku od unutarnjih lokota i njihovih redova kondicija, gdje je uz jedan unutarnji lokot vezan samo jedan red kondicija, kod eksplicitnih lokota može se vezati više kondicija za jedan lokot, ako postoji potreba. Kondicije imaju metode *await, signal, signalAll*, koje su ekvivalentne metodama *wait, notify, notifyAll* kod unutarnjih redova kondicija.

Ovako izgleda sučelje Condition:

```
public interface Condition {
  void await() throws InterruptedException;
  boolean await(long time, TimeUnit unit) throws InterruptedException;
  long awaitNanos(long nanosTimeout) throws InterruptedException;
  void awaitUninterruptibly();
  boolean awaitUntil(Date deadline) throws InterruptedException;
  void signal();
  void signalAll();
}
```

Primjer korištenja kondicija za implementaciju ograničenog međuspremnika (vidi se da se na jedan lokot vežu dvije kondicije, te da se koriste nove metode *await* i *signal*):

```
class BoundedBuffer {
  Lock lock = new ReentrantLock();
  Condition notFull = lock.newCondition();  // Povezivanje jednog lokota
  Condition notEmpty = lock.newCondition(); // i dvije kondicije
  Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x)throws IE {
    lock.lock();
    try {
      while (count == items.length) notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }

  public Object take() throws IE {
    lock.lock();
    try {
      while (count == 0) notEmpty.await();
      Object x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Vidjeli smo neka (a ima ih još dosta) poboljšanja u zaključavanju u Javi 5. No, kako kažu autori u [4], mnoge klase u paketu *java.util.concurrent*, kao što su *Semaphore* i *ConcurrentLinkedQueue*, pružaju bolje performanse i skalabilnost u odnosu na stare klase (koje su koristile *synchronized*), ne zato što koriste nove vrste lokota, već zato što uopće ne koriste lokote - koriste *atomarne varijable* (*atomic variables*) i *neblokirajuću sinkronizaciju* (sinkronizacija bez lokota). Neblokirajući algoritmi značajno su kompleksniji od blokirajućih, ali pružaju bolje performanse i skalabilnost, nemaju problema npr. sa deadlockom, pa su najčešći predmet novijih istraživanja na području konkurentnih algoritama.

No, kako naglašava autor u [1], pisanje neblokirajućih konkurentnih algoritama je posao za eksperte. Navodi da onaj tko misli da zna pisati takve algoritme treba proći tzv. *Goetzov test* (Brian Goetz je dugogodišnji stručnjak za konkurentno programiranje u Javi, trenutačno Java Language Architect u firmi Oracle): "Ako znate pisati JVM visokih performansi za moderne mikroprocesore, tada ste kvalificirani *da razmišljate o tome* da li smijete izbjeći lokote za sinkronizaciju (tj. pisati neblokirajuće algoritme)".

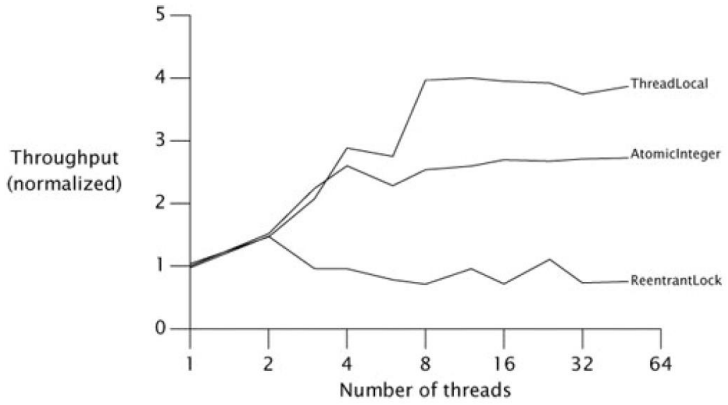
Atomarne varijable su na neki način generalizacija *volatilnih varijabli* (*volatile variables*) koje su postojale i prije Jave 5. Postoji dvanaest klasa atomarnih varijabli, podijeljenih u četiri grupe, a najvažnija je grupa *skalarnih varijabli* (*scalars*), koju čine klase *AtomicInteger, AtomicLong, AtomicBoolean* i *AtomicReference*. Atomarne varijable su, kao i lokoti u novim verzijama Jave, implementirane uglavnom uz pomoć CAS (compare-and-swap) operacije. Jedino kad određeni hardver ne podržava tu operaciju (što je rijetkost, jer svi procesori već desetak godina podržavaju CAS ili ekvivalent), onda JVM umjesto CAS obično koristi *spinn lock* (zaključavanje pomoću radnog čekanja). JVM-ovi su, kao i operacijski sustavi, koristili CAS (ako je postojao na određenom hardveru) i prije Jave 5, ali tek od Jave 5 mogu Java klase (uključujući i one koje mi pišemo) koristiti CAS operaciju.

U nastavku se prikazuju dvije realizacije [4] generatora pseudoslučajnih brojeva - pomoću klase *ReentrantLock* i pomoću klase *AtomicInteger* i operacije CAS [4]:

```
public class ReentrantLockPseudoRandom extends PseudoRandom {
  private final Lock lock = new ReentrantLock(false);
  private int seed;
  ReentrantLockPseudoRandom(int seed) {this.seed = seed;}
  public int nextInt(int n) {
    lock.lock();
    try {
      int s = seed;
      seed = calculateNext(s);
      int remainder = s % n;
      return remainder > 0 ? remainder : remainder + n;
    } finally {lock.unlock();}
  }
}

public class AtomicPseudoRandom extends PseudoRandom {
  private AtomicInteger seed;
  AtomicPseudoRandom(int seed) {this.seed = new AtomicInteger(seed);}
  public int nextInt(int n) {
    while (true) {
      int s = seed.get();
      int nextSeed = calculateNext(s);
      if (seed.compareAndSet(s, nextSeed)) {
        int remainder = s % n;
        return remainder > 0 ? remainder : remainder + n;
      }
    }
  }
}
```

Na slici 3.4 prikazuje se propusnost oba rješenja. Prikazana je i krivulja za tzv. *ThreadLocal* varijable (imaju najbolje performanse), a to su varijable koje imaju svoju posebnu instancu za svaku dretvu, na temelju *thread local storage* mehanizma - to je nešto slično kao kada u bazi podataka jedna sesija ne vidi mijenjane, ali nekomitirane, podatke druge sesije, jer čita svoju vlastitu verziju u UNDO tablespaceu.



Slika 3.4: Propusnost ReentrantLock i AtomicInteger kod srednjeg opterećenja; Izvor: [4]

4. EIFFEL OOPL

Eiffel je 1985. godine dizajnirao (a 1986. je napravljen prvi compiler) Bertrand Meyer, jedan od najvećih autoriteta na području OOPL-a. Prvo izdanje njegove knjige OOSC (1988.), značajno je djelo informatičke literature (aktualno je 2. izdanje iz 1997.). Ta je knjiga upoznala javnost sa OO jezikom Eiffel (tada verzije 2). Eiffel je od početka podržavao *višestruko nasljeđivanje*, *generičke klase*, *obradu iznimaka*, *garbage collection* i metodu *Design by Contract* (DBC), a kasnije su mu dodani *agenti* (nešto slično će dobiti Java 8 pod imenom *closures* ili *lambda expressions*; C# ih već ima), *nasljeđivanje implementacije* (uz nasljeđivanje tipa) i metoda za konkurentno programiranje *Simple Concurrent Object-Oriented Programming* (SCOOP). U široj je javnosti daleko manje poznat nego C++ i Java, ali ga mnogi autoriteti smatraju danas najboljim OOPL jezikom. Eiffel je od 2005. godine ECMA standardiziran, a od 2006. ISO standardiziran.

Slijedi primjer Eiffel klase. Napomenimo da konstruktor metoda u Eiffelu ne mora imati isto ime kao klasa u kojoj se nalazi, ali smo namjerno zadržali isto ime, kao što npr. mora biti u Javi:

```
class ZIVOTINJA
create zivotinja
feature {ANY}
   ime      : STRING
   visina_cm: DOUBLE
   zivotinja (p_ime: STRING; p_visina_cm: REAL) is
      do
         ime       := p_ime
         visina_cm := p_visina_cm
      end
   prikazi_podatke is
      do
         print ("  Ime: ");           print (ime)
         print (", Visina (cm): ");   print (visina_cm)
         print (", Visina (inch): "); print (visina_inch)
      end
   visina_inch: REAL is
      do
         Result := visina_cm / 2.54
      end
end
```

Vidimo da se u Eiffelu ne mora koristiti znak za odvajanje naredbi " ; ", ali je preporuka da se koristi ako imamo više naredbi u istom retku (zbog čitljivosti). Primijetimo da u funkciji ne postoji ključna riječ **return**, već Eiffel ima posebnu predefiniranu varijablu **Result**. Također, primijetimo da Eiffel omogućava tzv. uniformni pristup (uniform access), tj. poziv atributa ne razlikuje se od poziva funkcije bez parametara. Npr. u proceduri *prikazi_podatke* funkciju pozivamo sa *visina_inch*, dok npr. Java traži da se koriste zagrade i kad metoda nema parametara, pa se mora pisati *visina_inch()*. Uniformni pristup se smatra značajnom mogućnošću OOPL jezika.

Eiffel ima vrlo jednostavno i vrlo fleksibilno definiranje pristupa, jer za svaki atribut/metodu možemo definirati koja mu klasa može pristupati. Npr, ako ne želimo dati niti jednoj klasi pristup atributu *visina_cm* i funkciji *visina_inch*, pristup konstruktoru *zivotinja* želimo dozvoliti samo klasama TEST1 i TEST2, a pristup atributu *ime* i proceduri *prikazi_podatke* želimo dozvoliti svim klasama, napisat ćemo:

```
feature {NONE} visina_cm ...; visina_inch ...;
feature {TEST1, TEST2} zivotinja ...;
feature {ANY} ime ...; prikazi_podatke ...;
```

Važno je naglasiti da u Eiffelu dozvola pristupa atributu znači dozvolu čitanja atributa, ne i dozvolu pisanja. Vrijednosti atributa može mijenjati samo metoda iz klase u kojoj je atribut definiran (ili iz njene podklase), pomoću *set* procedura. Zbog toga u Eiffelu ne treba raditi *get* funkcije, dok Java mora imati *get* funkcije ako želimo atribute zaštiti od upisa izvan klase. Zanimljivo je da Eiffel može sakriti atribute nekog objekta čak i od drugih objekata (instanci) iste klase, pomoću {NONE} ili, što je isto, pomoću {}. Eiffel ne poznaje sakrivanje atributa/metoda od podklase (nema nešto kao Java **private**), jer Meyer drži da to nije u skladu sa OO modelom.

Eiffel podržava jednostruko i *višestruko nasljeđivanje* (klasa). No, postoje dvije vrste nasljeđivanja - *nasljeđivanja tipa* (zove se i *semantičko nasljeđivanje*, ili nasljeđivanje sučelja) i *nasljeđivanja implementacije* (zove se i *sintaktičko nasljeđivanje*, ili nasljeđivanje programskog koda). Prva vrsta nasljeđivanja je "pravo" nasljeđivanje, u kojem podklasa predstavlja podtip. Druga vrsta nasljeđivanja je "praktično" nasljeđivanje, gdje se želi iskoristiti neki postojeći programski kod, ali se ne želi koristiti polimorfizam. Eiffel je dobio nasljeđivanje implementacije naknadno, po uzoru na jezik C++.

Neki drže višestruko nasljeđivanje vrlo važnim svojstvom OOPL jezika. Npr. Meyer drži da je u mnogim konkretnim situacijama potrebno da klasa može naslijediti dvije ili više klasa i duhovito kaže da je odgovor na pitanje "Da li moja klasa C treba naslijediti klasu A ili klasu B (budući da moj jezik podržava samo jednostruko nasljeđivanje)?" često isto tako težak kao i odgovor na pitanje "Da li da izaberem mamu ili tatu?". Višestruko nasljeđivanje je naročito korisno u slučaju kada postoje dva (ili više) jednako važna kriterija za kreiranje hijerarhije klasa (jednostruko nasljeđivanje dopušta samo jednu hijerarhiju) i u slučaju kada podklasa od jedne nadklase nasljeđuje tip, a od druge nadklase nasljeđuje implementaciju, tzv. mix-in nasljeđivanje (primjer: klasa ARRAYED_STACK nasljeđuje od apstraktne klase STACK i klase ARRAY). Slijedi Eiffel (nepotpuni) primjer klase C koja nasljeđuje od klasa A i B, pri čemu klasa C nadjačava jedan atribut i jednu metodu iz klase A i jednu metodu iz klase B:

```
class C
inherit
   A redefine atribut_iz_a, metoda_iz_a end
   B redefine metoda_iz_b end
feature
   ...
end
```

Kod višestrukog nasljeđivanja najčešće se navode dva glavna problema. Jedan je problem kada roditeljske klase imaju atribute/metode istog imena, ali različitog značenja. Eiffel za to ima vrlo jednostavno rješenje – preimenovanje (barem jednog) atributa/metode pomoću **rename**. Drugi je problem kada imamo tzv. ponavljajuće (repeated) nasljeđivanje, npr. u primjeru gdje su klase B i C djeca klase A, a klasa D je dijete i od B i od C, pa izlazi da je klasa D dva puta (indirektno) dijete od klase A (to se naziva i dijamantnim nasljeđivanjem, zbog sličnosti crteža takvih klasa sa skicom dijamanta). Ako atributi/metode klase A nisu nadjačani u klasama B i C, Eiffel za to ima najjednostavnije moguće rješenje – ne treba ništa napraviti, jer duplih atributa/metoda niti nema. Ako su pak atributi/metode iz klase A redefinirani u klasi B i/ili C, tada postoje dva slučaja – da je kod nadjačavanja u klasama B/C zadržano isto ime atributa/metode kao u klasi A, ili da je ime promijenjeno. Prvi slučaj (ista imena) se najčešće svodi na drugi, preimenovanjem jednog atributa/metode (a rjeđe se koristi "eliminacija" jednog atributa/metode, pomoću **undefine**). U drugom slučaju (različita imena) Eiffel traži da se eksplicitno izabere (pomoću **select**) željeni atribut/metoda, što je potrebno da bi se razriješila dilema izbora prave metode kod dinamičkog pozivanja metoda (zbog polimorfičnog pridruživanja).

Eiffel šalje parametre referentnog tipa kao reference, a **expanded** parametre šalje kao vrijednosti. Bez obzira kako se parametri šalju, Eiffel ne dozvoljava da se oni u metodi mijenjaju, niti pomoću naredbe pridruživanja "parametar := nesto", niti pomoću naredbe kreiranja objekta "create parametar;". Ali, iako Eiffel metoda ne može mijenjati objekt predstavljen parametrom, može mijenjati njegove atribute (bilo direktno, bilo pozivom drugih metoda). Zanimljivo je pitanje *promjene signature*, tj. pitanje da li parametri u nadjačanoj metodi mogu imati drugačiji tip od parametara u metodi iz nadklase i (ako mogu) kakav mora biti taj tip. Postoje tri (glavne) mogućnosti:

1. tip parametra se ne može mijenjati - *no variance;*
2. može se mijenjati tako da bude podtip u odnosu na bazni – *covariance;*
3. može se mijenjati tako da bude nadtip - *contravariance*.

Eiffel podržava covariance i za parametre metoda i za povratnu (return) vrijednost funkcije i za atribute. Npr. Java je do verzije 1.4 imala u potpunosti no variance pristup, ali od verzije 1.5 (ili 5.0) podržava covariance za povratne vrijednosti funkcije (i PL/SQL se ponaša kao Java 1.5).

Eiffel je imao *generičke klase* od početka, dok ih Java dobila u verziji 1.5. Možemo reći da generičke klase zapravo nisu prave klase, nego predlošci za klase, jer imaju tzv. generičke parametre. Npr. u Eiffelu ovako izgleda generička klasa STACK [G] (generički parametar nazvan je G, a može se zvati i drugačije):

```
class STACK [G]
feature
   element_na_vrhu: G is
      do ... end
   ...
end
```

Generička klasa se koristi kao klijent neke (druge) klase, u kojoj se (drugoj klasi) atribut, varijabla ili parametar deklarira pomoću generičke klase, tako da se generički parametar zamijeni sa nekom (trećom) klasom. Npr. klasa STACK_KLIJENT može sadržavati dva atributa definirana na sljedeći način (generički parametar G zamijenjen je klasom ZIVOTINJA, odnosno klasom REAL):

```
atribut1: STACK [ZIVOTINJA]
atribut2: STACK [REAL]
```

Eiffel ima (kao i Java) i tzv. ograničenu generičnost (constrained genericity), gdje se generički parametar ograničava nekom određenom klasom, pomoću operatora ->. U slučaju ograničene generičnosti, klasa koja zamjenjuje generički parametar mora biti ili ista kao klasa koja ograničava generički parametar, ili podklasa te klase. Npr. ako bismo ovako definirali (ograničenu) generičku klasu STACK:

```
class STACK [G -> ZIVOTINJA] ... end
```

tada bismo imali sljedeće:

```
atribut1: STACK [BAKTERIJA]      -- greška, nije podklasa od ZIVOTINJA
atribut2: STACK [ZIVOTINJA]      -- OK, može biti ista klasa
atribut3: STACK [KUCNI_LJUBIMAC] -- OK, to je podklasa od ZIVOTINJA
```

Design By Contract (DBC) je metoda čiji je autor također Meyer. Stoga nije čudno da Eiffel u potpunosti podržava DBC. Za sada niti jedan drugi OOPL ne podržava DBC u potpunosti, barem ne na razini jezika. DBC je opširno opisan u [5]. Pojednostavljeno rečeno, DBC se zasniva na ideji da svaka metoda (procedura ili funkcija), uz "standardni" programski kod, treba imati još dva dodatna dijela - *pretkondiciju* (precondition) i *postkondiciju* (postcondition). Klasa treba imati još jedan dodatni dio - *invarijantu* (invariant). Ugovor (contract) se zasniva na tome da metoda "traži" od svog pozivatelja (neke druge metode) da zadovolji uvjete definirane u pretkondiciji (plus uvjete definirane u invarijanti), a ona (pozvana metoda) se tada "obvezuje" da će na kraju zadovoljiti uvjete definirane u postkondiciji (plus uvjete definirane u invarijanti). Ideja je na neki način upravo suprotna od tzv. defanzivnog programiranja, koje zagovora da se u svim mogućim trenucima pokušava što više toga provjeriti.

Eiffel za specifikaciju DBC elemenata koristi ključne riječi **require** (odnosno **require else** u nadjačanoj metodi, kod nasljeđivanja) za označavanje pretkondicije, **ensure** (odnosno **ensure then** u nadjačanoj metodi) za postkondicije i **invariant** za invarijante klasa. Navedene naredbe su najvažnije za DBC podršku, ali Eiffel ih ima još. Naredba **check** služi za provjeru nekog uvjeta u bilo kom trenutku. Za provjeru programskih petlji postoje dvije naredbe: **variant** provjerava da li se cjelobrojni izraz smanjuje kod svakog prolaza u petlji, a **invariant** (opet ista riječ kao za označavanje invarijante klase, ali je u kontekstu petlje značenje drugačije) provjerava da li je određeni uvjet u petlji zadovoljen u svakom prolazu.

U Eiffelu metoda ne može obraditi iznimku koja se desila zbog nezadovoljavanja pretkondicije – jednostavno, pozvana metoda nema što raditi ako se druga metoda (pozivatelj) ne drži ugovora! Dakle, za pojavu iznimke u vrijeme izvršavanja programskog koda pretkondicije "krivac" je metoda-pozivatelj, dok je za pojavu iznimke u vrijeme izvršavanja programskog koda postkondicije "krivac" metoda-izvršavatelj.

Nadjačane metode ne mogu imati bilo kakve pretkondicije ili postkondicije, već nadjačana metoda mora imati jednaku ili slabiju pretkondiciju (tj. može zahtijevati od metode koja ju je pozvala ili isto što i metoda nadklase, ili manje od toga) i mora imati jednaku ili jaču postkondiciju (tj. mora osigurati barem ono što je osiguravala metoda nadklase, ili više od toga). Zato se za definiranje pretkondicije u nadjačanoj metodi koristi oblik **require else** (umjesto **require**), a za definiranje postkondicije u nadjačanoj metodi koristi se **ensure then**.

Može se postaviti pitanje utjecaja izvršenja pretkondicija, postkondicija i invarijanti na brzinu izvođenja programa. Nažalost, utjecaj postoji, a naročito je skupa provjera invarijanti. Stoga se u Eiffel-u može odrediti nekoliko stupnjeva provjere. Najslabija provjera (ali sa najmanjim negativnim utjecajem na brzinu izvođenja) je provjera samo pretkondicija (ta se provjera obično ostavlja i nakon završetka faze testiranja, tj. ostaje u produkcijskom kodu). Sljedeći stupanj uključuje provjeru postkondicija, slijedi provjera invarijanti, zatim provjera petlji i na kraju provjera pomoću check naredbi. Najveći stupanj provjere se obično koristi samo kod testiranja, jer se takvi programi izvršavaju i do 2-3 puta sporije u odnosu na programe koji nemaju nikakvih provjera.

Treba naglasiti da su pretkondicije, postkondicije i invarijante korisne čak i kada nisu realizirane kao programski kod, nego samo kao komentar (najčešće zato što je nešto teško ili nemoguće izraziti – Eiffel nema operatore univerzalnog i egzistencijalnog kvantifikatora iz predikatnog računa), jer poboljšavaju dokumentiranost izvornog koda. Zbog DBC podrške, može se reći da je Eiffel i specifikacijski (a ne samo programski) jezik.

Slijedi Eiffel programski kod za realizaciju stoga ([5], str. 390.-391.). Zapravo, to nije kompletan programski kod, već samo tzv. kratki oblik klase, koji ne prikazuje izvršni kod metoda (zato to nije class, već class interface). Kratki oblik klase ne prikazuje niti skrivene implementacijske detalje, tako da ne vidimo da klasa ima jedan skriveni atribut tipa niz (implement: ARRAY [G]) koji služi za implementaciju stoga:

```
class interface STACK [G]

creation make

feature – inicijalizacija
-- stog od n elemenata
make (n: INTEGER) is
   require non_negative_capacity: n >= 0
   ensure  capacity_set: capacity = n
end

feature -- pristup
-- maksimalni broj elemenata
capacity: INTEGER
-- broj elemenata stoga
el_count: INTEGER
-- element na vrhu stoga

item: G is
  require not_empty: not empty
end

feature – statusi
-- da li je stog prazan?
empty: BOOLEAN is
   ensure empty_definition: Result = (el_count = 0)
end

-- da li je stog pun?
full: BOOLEAN is
  ensure full_definition: Result = (el_count = capacity)
end

feature – promjena
-- dodaj x na vrh stoga
put (x: G) is
  require not_full: not full
  ensure not_empty: not empty
         added_to_top: item = x
         one_more_item: el_count = old el_count + 1
end

-- makni element sa vrha
remove is
   require not_empty: not empty
   ensure not_full: not full
          one_fewer:  el_count = old el_count - 1
end

invariant
   count_non_negative: 0 <= el_count
   count_bounded: el_count <= capacity
   empty_if_no_elements: empty = (el_count = 0)

end -- class interface STACK [G]
```

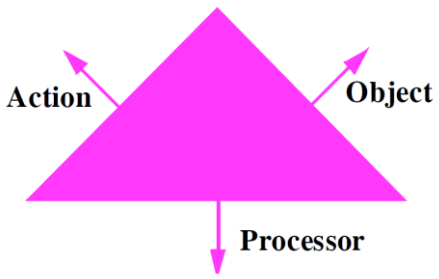
5. KONKURENTNO PROGRAMIRANJE U EIFFELU - SCOOP METODA

Metoda SCOOP (Simple Concurrent Object-Oriented Programming) ima prilično davne korijene. Njen kreator, Bertrand Meyer (koji je i kreator jezika Eiffel), prikazao je osnovne ideje te metode još 1990. godine na TOOLS Europe, a preradio ih je 1993. Detaljan prikaz dao je 1997. u [5]. Kasnije je SCOOP metoda eksperimentalno realizirana i poboljšavana na ETH Zurich, te se koristi naročito za nastavne i znanstvene potrebe. Formalni prikaz te metode zaokružio je 2007. godine P. Nienaltowski u doktorskoj disertaciji kod prof. Meyera. Prije oko godinu dana (lipanj 2011.) firma Eiffel Software (www.eiffel.com) je uključila SCOOP metodu u svoj proizvod EiffelStudio v.6.8. Moglo bi se reći da je SCOOP zaseban (mini) jezik za konkurentno programiranje, jer eksperimentalne implementacije postoje i izvan jezika Eiffel, npr. postoje i za jezik Java. SCOOP se i dalje razvija, npr. rade se istraživanja na sljedećim područjima:

- prevencija i detekcija deadlocka;
- uvođenje softverske transakcijske memorije;
- distribuirani SCOOP.

European Research Council (ERC) dodijelio je B.Meyeru "Advanced Investigator Grant" za petogodišnji (2012-2017) projekt "Concurrency Made Easy" koji se temelji na SCOOP metodi (http://se.inf.ethz.ch/research/scoop/CME.pdf).

Kod SCOOP metode vrlo je važan pojam *procesor*, ali se pod tim pojmom ne misli na fizički procesor, već na apstraktni procesor, koji može biti i fizički procesor (u nastavku će se za njega koristiti termin CPU), proces operacijskog sustava, dretva operacijskog sustava i sl. Da se smanji zabuna, u nastavku će se koristiti oblik (*apstraktni) procesor* (a ne samo *procesor*, kao što se koristi u [5], [6] i [7]). Za razliku od CPU-a, čiji je broj ograničen, možemo držati da je broj (apstraktnih) procesora praktički neograničen. Kako se navodi u [5], (apstraktni) procesor je jedna od tri sile računanja kod OOPL programiranja: neka akcija (ili metoda, tj. funkcija ili procedura) izvodi se na određenom (apstraktnom) procesoru nad određenim objektom (instancom klase), kako prikazuje slika 5.1:



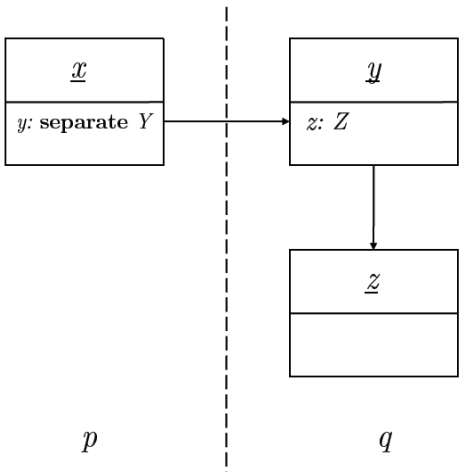
Slika 5.1: Tri sile kod računanja (The three forces of computation); Izvor: [5]

Temeljna ideja vezana za (apstraktne) procesore u SCOOP metodi je: svaki objekt je dodijeljen samo jednom (apstraktnom) procesoru, koji se naziva *rukovatelj (handler) objektom*. S druge strane, jedan (apstraktni) procesor može biti rukovatelj za više objekata. Kad se kreira novi objekt, runtime sistem (na temelju programskih uputa) odlučuje da li će mu se dodijeliti neki rukovatelj od postojećih, ili će se kreirati novi (apstraktni) procesor kao rukovatelj za novokreirani objekt. Ta veza ostaje do kraja - objekt je uvijek vezan za samo jednog rukovatelja, a to u konačnici znači da se nad istim objektom istovremeno može izvoditi samo jedna metoda, jer metode određenog objekta može izvoditi samo rukovatelj tog objekta.

Kada se u nekoj metodi, koja se izvodi nad nekim objektom, poziva metoda nad objektom kojim rukuje drugi rukovatelj (tj. koji nije isti kao rukovatelj prvog objekta), taj se poziv zove *asinkroni poziv* ili *odvojeni (separate) poziv*. U tom slučaju rukovatelj prvog objekta može nastaviti rad, ne čekajući da pozvana metoda završi. Za razliku od toga, poziv metode nad drugim objektom koji ima istog rukovatelja kao i prvi objekt, je *sinkroni poziv* ili *neodvojeni poziv* – to je standardna situacija iz sekvencijalnog programiranja, gdje rukovatelj mora čekati da jedna metoda završi prije nego nastavi izvršavati drugu (naravno, zanemarujemo fizičke detalje CPU-a, mogućnost instrukcijskog paralelizma i sl.).

Ostalo je otvoreno pitanje na koji način runtime sistem određuje kojeg će rukovatelja dodijeliti objektu. U odnosu na nekonkurentni Eiffel, SCOOP metoda uvodi samo još jednu ključnu riječ - **separate**. Uz uobičajenu deklaraciju varijable (napomena: Eiffel izvorno koristi termin *entitet* za atribute, lokalne varijable i argumente metoda) "x : X", koja označava da je varijabla *x* referenca na objekt tipa (tj. klase) *X*, sada se može pisati i "x : separate X", čime se izražava da kod izvođenja programa *x* može biti referenca na objekt koji ima drugog rukovatelja u odnosu na objekt u kojem se ta referenca nalazi. Takva se referenca zove *odvojena referenca*, a objekt na koji pokazuje zove se *odvojeni objekt*.

Slika 5.2 prikazuje tri objekta: *x*, *y* i *z*. Objektu *x* je rukovatelj (apstraktni) procesor *p*, a objektima *y* i *z* rukovatelj je (apstraktni) procesor *q*. Objekt *x* ima atribut *y*, koji predstavlja udaljenu referencu, jer ta referenca pokazuje na objekt *y*, koji ima drugog rukovatelja. Za razliku od toga, objekt *y* ima atribut *z* koji je ne-odvojena referenca na ne-odvojeni objekt *z*, jer objekti *y* i *z* imaju istog rukovatelja.



Slika 5.2: Odvojena referenca: u objektu *x*, atribut *y* referencira objekt na drugom (apstraktnom) procesoru; Izvor: [7]

Primjer [5] pojasnit će do sada navedeno. Prtpostavimo da u klasi *WORKER*, metoda *do_task* nešto radi i stavlja rezultat u atribut *output*:

```
class WORKER
  feature
    output: INTEGER
    do_task (input: INTEGER) do ... end
end
```

Klasa *MANAGER* ima dva atributa - reference tipa *WORKER,* ali jedna referenca je odvojena, a druga je ne-odvojena. U nekoj metodi te klase pozivaju se metode *task* nad odvojenim objektom *worker1* i nad ne-odvojenim objektom *worker2*, a onda se rezultati izvođenja zbrajaju:

```
class MANAGER
  feature
    worker1 : separate WORKER
    worker2 : WORKER

  -- in some routine:
    do
      . . .
      worker1.do_task (input1)
      worker2.do_task (input2)
      result := worker1.output + worker2.output
    end
end
```

Zbivanja kod izvođenja ovog programskog koda mogu se ovako opisati: procedura *do_task* nad odvojenim objektom *worker1* poziva se asinkrono, tj. program ne čeka da ona završi, već odmah prelazi na sljedeću proceduru *do_task* nad objektom *worker2*, koju izvodi sinkrono. Kada je ta druga procedura gotova, izvršava se sljedeća naredba, koja zbraja dva rezultata. Tek u ovom trenutku mora se čekati da procedura *do_task* nad objektom *worker1* završi, jer je sada potreban rezultat te procedure. To se čekanje obavlja automatski od strane SCOOP mehanizma, i zove se *wait-by-necessity* (čekanje po potrebi). U ovom primjeru je rad sa odvojenim objektima bio lagan, jer je sistem sam znao da li je poziv sinkron ili asinkron i kada treba čekati na rezultat.

U nastavku će biti prikazano kako se u SCOOP-u radi međusobno isključivanje (mutual exclusion) u slučajevima kada se odvojeni objekti međusobno upliću jedan drugome u rad, tj. kod tzv. *race condition*. Primjer rada sa brojačima je ekvivalentan Java primjeru iz 3. točke (oboje je iz [7]):

```
class COUNTER
feature
  value : INTEGER

  set_value (a_value: INTEGER)
    do
      value := a_value
    end

  increment
    do
      value := value + 1
    end
end
```

Pretpostavimo da postoji varijabla *x* deklarirana kao *separate COUNTER* i pratimo sljedeći niz naredbi:

```
x.set value (0)
x.increment
i := x.value
```

Istovremeno se u nekom odvojenom (apstraktnom) procesoru odvija naredba:

```
x.set value (2)
```

Kao i u ekvivalentnom Java primjeru u 3. točki, zbog mogućeg ispreplitanja ovih naredbi u paralelnom radu, vrijednost varijable *i* iz prvog koda može biti 1, 2 ili 3. U Javi se to moglo spriječiti sinkronizacijom pomoću (npr.) *synchronized*. U Eiffelu se koristi jednostavno pravilo, koje se zove *pravilo odvojenih argumenata* (*separate argument rule*): runtime sistem automatski zaključava (apstraktne) procesore koji rukuju odvojenim objektima koji su (objekti) poslani kao argumenti metode. Ako je (apstraktni) procesor zaključan, ne može ga se koristiti, pa se ne može pozvati niti jedna metoda nad objektom kojemu je rukovatelj zaključani (apstraktni) procesor. Izmijenimo prethodni kod (tri instrukcije) na sljedeći način, tako da ih stavimo u proceduru koja će imati odvojeni objekt kao argument:

```
compute (x: separate COUNTER)
  do
    x.set_value (0)
    x.increment
    i := x.value
  end
```

Pogledajmo sada poziv *compute (x)* i pretpostavimo da je (apstraktni) procesor *p* rukovatelj za *x*. Kako je prije rečeno, budući da je *x* odvojeni argument te procedure, procesor *p* mora biti zaključan. Tekući procesor, koji izvodi proceduru *compute*, automatski čeka dok runtime sistem ne zaključa procesor *p*. Kada je *p* zaključan, tijelo procedure *compute* može se izvršavati bez problema (ne postoje višestruki lokoti na jedan procesor), pa će varijabla *i* na kraju procedure uvijek ima vrijednost 1. SCOOP forsira takvo ponašanje, tj. svi pozivi nad odvojenim objektima moraju se uokviriti u procedure kojima se odvojeni objekt šalje kao argument.

Npr. ovo je neispravno (kompajler javlja grešku):

```
x : separate X
compute
  do x.f end
```

a ovo je ispravno:

```
x : separate X
compute (x1: separate X)
  do x1.f end
```

Prikažimo sada u SCOOP-u *sinkronizaciju na temelju uvjeta* (*condition synchronization*), koja se u Javi implementirala npr. pomoću naredbi *wait* / *notifyAll* / *notify*, na istom primjeru proizvođača i potrošača kao u 2. točki. Pretpostavimo da imamo generičku klasu *BUFFER[T]* koja implementira neograničeni red (unbounded queue):

buffer: separate BUFFER[INTEGER]

i sljedeću proceduru u klasi potrošača:

```
consume (a_buffer: separate BUFFER[INTEGER])
  require
    not (a_buffer.count = 0)
  local
    value: INTEGER
  do
    value := a_buffer.get
  end
```

Primijetimo da smo koristili pretkondiciju kako bismo osigurali da *buffer* nije prazan kada iz njega čitamo. Pitanje je što će se desiti ako je *buffer* prazan? U sekvencijalnom slučaju desila bi se iznimka (exception). Međutim, u konkurentnom radu pretkondicije na odvojene objekte imaju drugu semantiku - rukovatelj odvojenog objekta se otključa, čeka se dok se ne zadovolji zahtjev, a onda se rukovatelj odvojenog objekta opet zaključa. Pretkondicija se u konkurentnom radu pretvara u *uvjet za čekanje* (*wait condition*). To se ponašanje može iskazati kao *pravilo čekanja* (*wait rule*): "Poziv metode koja ima odvojene argumente izvršit će se onda kada su svi odgovarajući rukovatelji slobodni (nisu zaključani) i kada su zadovoljene sve pretkondicije. Rukovatelji su ekskluzivno zaključani za vrijeme trajanja metode."

Kao i u Javi, tako se i u SCOOP metodi može desiti deadlock. Primjer [5]:

```
class C
  creation
    make

  feature
    a : separate A
    b : separate A

    make (x : separate A, y : separate A)
      do
        a := x
        b := y
      end

    f do g (a) end
    g (x : separate A) do h (b) end
    h (y : separate A) do ... end
end
```

Pretpostavimo sada da se izvršava sljedeći kod, gdje su objekti *c1* i *c2* tipa *separate C*, objekti *a* i *b* su tipa *separate A*, objekt *a* ima rukovatelja *p*, objekt *b* ima rukovatelja *q*:

```
create c1.make (a, b)
create c2.make (b, a)
c1.f
c2.f
```

Budući da su kod inicijalizacija objekata *c1* i *c2* argumenti permutirani, moguća je sekvenca poziva kod koje je u jednom slučaju zaključan rukovatelj *p*, a čeka se da se oslobodi rukovatelj *q*, i obrnuto. Nastao je deadlock.

Deadlock se trenutačno ne može automatski detektirati u SCOOP-u, pa je programerova odgovornost da radi programski kod slobodan od deadlocka (deadlock-free). No, kako se navodi u [7], radi se na implementaciji sheme koja prevenira deadlock, a ona je temeljena na redoslijedu zaključavanja koji prevenira *cikličko zaključavanje* (*cyclical locking*).

6. ZAKLJUČAK

Nažalost, (imperativne) konkurentne programe nije lako pisati. Kako je naglasio autor u [1] (na kraju poglavlja o konkurentnosti): "Nakon rada kroz ovo poglavlje, možete primijetiti da rad sa dretvama u Javi izgleda vrlo kompleksno i teško za korektnu upotrebu. Dodatno, izgleda malo neproduktivno - iako dretve rade paralelno, morate investirati veliki trud da implementirate tehnike koje ih sprečavaju da na loš način utječu jedna na drugu. Ako ste ikada pisali programe u zbirnom jeziku (assembleru), kad pišete programe sa dretvama, imate sličan osjećaj: svaki mali detalj je važan, i nemate sigurnosnu mrežu u obliku provjere od strane kompajlera."

Slično govore i autori u [8]: "Concurrency in Java is complex to use and expensive in computational resources. Because of these difficulties, Java-taught programmers conclude that concurrency is a fundamentally complex and expensive concept. Program specifications are designed around the difficulties, often in a contorted way. But these difficulties are not fundamental at all. There are forms of concurrency that are quite useful and yet as easy to program with as sequential programs..."

Nije lako pisati tzv. blokirajuće algoritme, koji koriste različite vrste lokota za (blokirajuću) sinkronizaciju (zaključavanje). Najveću manu zaključavanja autori u [2] vide u tome što **"nitko stvarno ne zna kako organizirati i održavati veliki sustav temeljen na zaključavanju"**. Još je teže pisati neblokirajuće konkurentne algoritme, koji ne koriste zaključavanje.

Softverske i/ili hardverske transakcijske memorije (STM / HTM) mogle bi značajno pomoći. No, postoje i neka softverska rješenja (imperativnog) konkurentnog programiranja koja (za sada) ne koriste STM, ali izgledaju naprednija nego npr. rješenja u Javi.

Jedno takvo rješenje je Simple Concurrent Object Oriented Programming (SCOOP), model koji je realiziran u OOPL jeziku Eiffel (iako postoje eksperimentalne realizacije i u drugim jezicima, pa i u Javi). Suština te metode je da objektima ekskluzivno rukuju njihovi rukovatelji - apstraktni procesi, i ne zaključavaju se objekti, već procesi. Uobičajena semantika Eiffel pretkondicije se u SCOOP-u mijenja – neuspjeh zadovoljenja pretkondicije ne uzrokuje exception, već čekanje na zadovoljavanje uvjeta. European Research Council (ERC) dodijelio je autoru B.Meyeru "Advanced Investigator Grant" za petogodišnji (2012.-2017.) projekt "Concurrency Made Easy" koji se temelji na SCOOP metodi.

LITERATURA

1. Eckel B. (2006): Thinking in Java (4.izdanje), Prentice Hall

2. Herlihy, M., Shavit, N. (2008): The Art of Multiprocessor Programming, Elsevier / Morgan Kaufmann Publishers

3. Horstmann, C.S., Cornell, G. (2008): Core Java: Volume 1 - Fundamentals (8.izdanje), Prentice Hall / Sun Microsystems

4. Göetz B. i ostali (2006): Java Concurrency in Practice, Addison-Wesley

5. Meyer, B. (1997): Object-Oriented Software Construction, Prentice Hall

6. Meyer, B., Nanz S. (2010): Concepts of Concurrent Computation (materijali sa kolegija), ETH Zuerich - Chair of Software Engineering, http://se.inf.ethz.ch/old/teaching/2010-S/0268/index.html#lectures_slides (svibanj 2012.)

7. Nanz S. i dr. (2010): A Comparative Study of the Usability of Two Object-oriented Concurrent Programming Languages, ETH Zuerich & University of Toronto, http://se.inf.ethz.ch/people/nanz/publications/nanz-et-al_arxiv1011.6047.pdf (svibanj 2012.)

8. Van Roy P., Haridi S. (2004): Concepts, Techniques, and Models of Computer Programming, The MIT Press Cambridge, Massachusetts

Zlatko Sirotić, univ.spec.inf.
Istra informatički inženjering d.o.o., Pula
e-mail: zlatko.sirotic@iii.hr

Autor radi više od 25 godina na informatičkim poslovima. Najveći dio radnog vijeka proveo je u poduzeću Istra informatički inženjering d.o.o, Pula, gdje radi i sada. Oracle softverske alate (baza, Designer CASE, Forms 4GL, Reports, JDeveloper IDE, Java) koristi zadnjih 15 godina.

Objavljivao je stručne radove na kongresima "Hotelska kuća", na konferencijama CASE, KOM, HrOUG (Hrvatska udruga Oracle korisnika), u časopisima "InfoTrend" i "Ugostiteljstvo i turizam", a neka njegova programska rješenja objavljena su na web stranicama firmi Quest i Oracle.